Optimistic Stack Allocation For Java-Like Languages

Erik Corry
Esmertec AG
ecorry@esmertec.com

Abstract

Stack allocation of objects offers more efficient use of cache memories on modern computers, but finding objects that can be safely stack allocated is difficult, as interprocedural escape analysis is imprecise in the presence of virtual method dispatch and dynamic class loading. We present a new technique for doing optimistic stack allocation of objects. Our technique does not require interprocedural analysis and is effective in the presence of dynamic class loading, reflection and exception handling. Moreover, we usually achieve higher proportions of stack allocated data than static strategies. In this paper we present optimistic stack-allocation for Java-like languages. For experiments we use traces of running Java programs to drive simulations of various garbage collection and allocation strategies.

Categories and Subject Descriptors D.3.4. [*Programming languages*]: Processors – Memory management (garbage collection)

General Terms languages, performance

Keywords garbage collection, Java, stack allocation

1. Introduction

Object-oriented languages such as Java, BETA or Smalltalk require allocation and deallocation of objects in an ordering that is not always last-in-first-out. This means that allocation cannot in general occur on a stack. It is however often the case, that a considerable subset of object allocations and deallocations are in last-in-first-out order or some useful approximation hereof. This opens up the possibility of performing these allocations on a stack-like structure. Stack allocation holds the promise of improved execution performance through efficient use of deep cache hierarchies.

In this paper we present an effective allocation system where memory allocations occur in stacked regions, but can be moved to a heap if necessary. This potentially combines the simplicity of universal heap allocation of objects with the performance of systems where stack allocatable objects have been identified by static analysis. Our approach is based on loops rather than method invocations and therefore requires simple *intra*procedural escape analysis, which we describe and implement. *Pretenuring* heuristics are examined and evaluated. Results are presented from simulating our system on programs from the SPECjvm98 benchmark suite. Our experiments indicate that we in most cases achieve higher proportions of stack-allocated data than alternative strategies.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006…$5.00.

1.1 The Performance Argument for Stack Allocation

Stack allocation generally means that memory is on average reclaimed very fast (on method termination), and reused immediately (often on the next method call). This reused memory is very likely to be in the fastest, smallest cache, since it was very recently used. Garbage collected heaps have difficulty in concentrating their most frequently used objects in an area as small as that covered by the level 1 (fastest) cache and even making good use of the slightly slower and larger level 2 cache requires some care.

Stack allocation also has the desirable property of making good use of caches regardless of their size and configuration. In this sense stack allocation is *cache oblivious* [Pro99]. Stack allocation is, however, not in general asymptotically optimal under any particular assumption of cache behaviour, a meaning that is sometimes also associated with the term.

1.2 The Cache Benefits of a Write-allocate Cache

Despite the number of recent papers detailing stack allocation techniques for Java, the benefits of stack allocation are not entirely uncontroversial. Several papers, including [AS94], [Rei94] and [DTM93] argue that a *write-allocate* cache can reduce the costs of heap allocation to the point (4-5%) where other optimisations made possible by heap allocation can more than compensate.

Unfortunately, it appears that modern processors are not implemented in this way. According to "The IA-32 Intel Architecture Software Developer's Manual" [Inta] Section 10.2 most memory in a modern Pentium 4 or Xeon system is handled in "write back mode" (fetch-on-write mode) and no mode corresponding to write-allocate mode is in practice available to the systems programmer. The situation for embedded CPUs is not much better, with the popular Xscale RISC CPU also implementing a fetch-on-write policy. See Section 6.2.3.3 in "Intel XScale Microarchitecture for the PXA250 and PXA210 Applications Processors" [Intb]. While many recent CPUs include *prefetch* machine instructions [Int00], our measurements indicate that they cannot replace write-allocate caches in terms of performance. See for details our M.Sc. thesis [Cor04].

1.3 A Review of Static Escape Analysis in Java

Much effort has in the last few years gone into static analysis of Java in order to determine which objects may be stack allocated without altering program semantics.

The following papers describe escape analysis algorithms for Java. The analyses identify objects whose lifetime is delimited by the lifetime of a method invocation. This allows the objects to be allocated in the stack frame of the relevant method invocation and deallocated automatically when the method returns and its stack frame is removed from the stack.

Blanchet [Bla99], Choi et al. [CGS+99] and Reid et al. [RMB+99] implement their static escape analysis for the static subset of Java (without dynamic loading). Gay and Steensgaard [GS00] implement their escape analysis for Marmot, a static compiler that compiles a language that is almost Java.

Changes to the allocation strategy are always in danger of changing the space complexity of the algorithm they seek to implement. If a stack allocation analysis changes any program that uses $O(n)$ non-collectable memory for an input of size n into a new program that uses $O(n^2)$ non-collectable memory then that is unlikely to be an acceptable 'optimisation'. Blanchet tries to avoid changes in space complexity due to loops by reusing space that is provably dead when the next loop iteration starts. This does not completely avoid space complexity issues associated with loops since it is limited by the ability of the stack allocation algorithm to automatically prove things about object lifetimes. Therefore Blanchet also runs his benchmarks in a safe configuration, where the system does not do stack allocation at all in loops unless it can reuse the space in the stack frame by using liveness analysis. Even in the safe mode, space complexity is not preserved in the presence of recursive method calls. The approach to space complexity issues in Gay and Steensgaard's paper is similar to Blanchet's safe configuration, while Choi et al.'s proposal aggressively stack allocates objects without regard for space complexity issues. The stack allocation techniques in Reid et al.'s paper will also potentially cause adverse changes in space complexity. Though Reid et al. acknowledge the problem they give no systematic way to avoid it.

All the Java analyses mentioned above are *whole program* analyses. That is, the analysis relies for correctness on knowledge of the whole program rather than working on individual classes or methods. This means the analysis must be rebuilt if the program dynamically loads new classes during program execution. Some of the schemes in theory allow previously calculated data to be reused when the new analysis is generated, but as far as we can ascertain this feature is not used in the sample implementations.

Reid et al.'s proposal includes *deep stack allocation* where an object can be allocated in the stack frame of a method on the current call stack other than the currently active one. They describe a rotating stack system where n different stacks are used in rotation, allowing simple allocation on any of the last n stack frames. Gay and Steensgaard also support deep stack allocation, though only one level deep.

Some objects may be subject to *scalar replacement*. That is, they are reduced to a collection of local variables by inlining all methods that act on the object. This optimisation is implemented by Gay and Steensgaard.

In order to increase the opportunities for stack allocation, methods are inlined in Blanchet's proposal and in Gay and Steensgaard's proposal. Inlining of method calls (where the code in a method is copied into the calling method instead of being called) can increase the number of objects that fulfill the necessary criteria for reduction. It is of course only possible where the called method can be determined statically by the compiler. Another effect of inlining is that it can convert a deep stack allocation into a normal stack allocation, by unifying the allocating method and the method from which an object does not escape.

The results in Choi et al.'s paper are encouraging, showing that 1-70% of objects can be stack allocated. Unfortunately, none of the results appear comparable with other papers due to choice of benchmark programs. Results for the other papers are presented in Section 2.

The emphasis in the literature on whole-program analyses is a problem in the light of the trend within the Java world towards more dynamic class loading. In addition, some of the papers have unresolved issues around space complexity.

2. Optimistic Stack Allocation According to Baker

Since the analysis of 'stackability' is of necessity imprecise, it would be nice to be able to optimistically stack allocate objects where the compiler was in doubt, and then move these objects to the heap at a later time if the object turned out not to be stackable after all. A scheme to achieve this was proposed by Henry Baker in "CONS should not CONS its Arguments, or a Lazy Alloc is a Smart Alloc" [Bak92]. We believe it suggests a productive line of inquiry.

The scheme will here be described in terms of a downwards growing stack. It is based on the idea of a simple write barrier used for pointer writes, combined with a read barrier, which we will show in the next section can be avoided. In Baker's scheme, the stack is placed in the lowest part of memory, followed by the youngest generation of garbage collected heap, followed by the other generations if any. This arrangement is critical, since it makes a very fast inlined write barrier possible. It corresponds to an arrangement as in the left side of figure 1. All objects are initially allocated on the stack in the stack frame of the currently executing method invocation.

We now set up the "Baker constraint": That no pointer in memory may point *downwards*, that is that no pointer may have the addresser above the addressee. For calculating the *height* of a pointer we divide memory into one *region* per stack frame, and one for the heap. It is critical to note that within each region, the *height* of all objects is equal, and a pointer is only *downwards* if it points to an object with a lower *height*.

In order to conservatively detect potential violations of this constraint, it is enough to compare the addresser with the addressee – if the pointer points to a numerically higher address, then it cannot violate the constraint. If it points to a numerically lower address then the runtime system needs to investigate further. It may be that the two objects are in the same region and thus have the same *height*, or it may be that a potential violation of the constraint has occurred.

This suggests a simple write barrier (reinvented in [Ste99]), such that (in pseudo-C) setting the 4th word in the object p to point to the object q is code generated in the following way:

```
if (p > q)       // Perform in-line fast-path test
    slow_path(); // Call out-of-line slow path check
p[4] = q;        // Pointer write. Not part of barrier
```

This code is intended to be very fast for the common case, namely the one where the pointer points upwards, or remains within the same stack frame. The "greater-than" test is also compact in most instruction sets. It will be noted that the same fast common case applies to the write barrier needed to record pointers from an older generation to a newer one (not illustrated here). Thus the write barriers for both purposes can be combined. The first task of the slow path code is to determine whether the two pointers are in fact in different regions. If not, we will term this a *false alarm*. Note that the 'slow path' need not be particularly slow. It is likely to look rather like a regular write barrier, having a fast and a slow part itself, and it may even be partially inlined.

If the pointers are in different regions, then our write barrier has detected a violation of the Baker constraint (see the left of figure 1, where the new pointer from B to W is a violation). In this case the slow path code should move (*evacuate*) the pointed-to objects (in this case W) to the heap in order to preserve the constraint, leaving behind a *forwarding pointer* for the read barrier to find. In figure 1 the object X was pointed to by the evacuated object W, and therefore this object was also evacuated. The resolution according to Baker is illustrated in the centre of figure 1, where the old copy of W is replaced with a forwarding pointer to the new location.

When reading a reference from an object, Baker's scheme requires the program to use a read barrier. The job of the *read barrier* is to check the reference in order to ascertain whether the pointed-to object is still there, or whether it has been moved to a new location, leaving behind a forwarding pointer.

The implementation of the read barrier is likely to be very expensive in terms of code size and execution resources. A 20 percent slowdown combined with a 100 percent code size increase is mentioned by Zorn [Zor90]. While and Field [WF] manage to get the overhead down to about 10 percent, but by using techniques that cannot be applied to object-oriented languages with public member variables. Other proposals for fast read barriers [BH04] rely on all data being tagged, which is not normally the case in Java implementations. As will be seen in the results section a read barrier would be invoked on the order of once for every byte allocated by a Java program.

What the Baker constraint buys us is a guarantee that, when a method returns, and its stack frame is deallocated by raising the stack pointer there can be no live pointers to the data in that stack frame. To see why, consider that since no pointers may point downwards, and since the stack frame being deallocated is by definition the lowest object area in the system, there can be no pointers into the stack frame being deallocated. It is therefore garbage, and may be deallocated.

Registers may contain pointers, but register operations are not checked for Baker constraint violations. Registers are few and cannot themselves be referenced since they have no address. It is sufficient to check the registers for references to a stack frame when it is deallocated. Since in Baker's proposal stack frames are deallocated at method return most registers can simply be zeroed.

It is of course the case that the eviction of an object to a higher object area is itself a write operation and is itself subject to the constraint and therefore must be protected by the write barrier. As the pointers in the evicted object move to a higher area, they may thus end up pointing downwards. This can result in new recursive evacuations which are however bound to terminate since no new objects are being created by the process, and there is a finite number of objects on the stack that can be evicted. This case is illustrated in the centre of figure 1, where the eviction of object W also results in the eviction of object X.

3. Optimistic Stack-Allocation for Java-like languages

3.1 How to Remove the Read Barrier

The most obvious problem with Baker's approach, and in our opinion the main reason why it has (to our knowledge) never been implemented lies in the read barrier and the performance problems associated with it (see above).

The read barrier of Baker's proposal is necessary in order to catch pointers that point to the old location and redirect them to the new location of the object. Instead we propose to find and fix such pointers at eviction time. These pointers can only reside in the same or lower stack frames as the object being evicted (or they would have triggered the eviction earlier). It turns out that evacuations can be made very rare, and most evacuations will happen to objects near the tip of the stack (the newest, or lowest end of the stack), so that only a small number of stack frames need be scanned and fixed. See Section 5.6. The right of figure 1 shows the part of the stack that needs scanning for pointers that need fixing.

3.2 Heap Allocating Selected Objects Immediately

We can use heuristics to identify some objects that are likely to be evicted. They can then be heap allocated right away. This is known as *pretenuring* and is used in other garbage collection schemes to choose the generation an object should be allocated in [CHL98]. Our stack allocation scheme is safe in both directions. If we allocate an object on the stack that should have been on the heap or vice versa then the only consequence is a loss of performance; the program will still execute correctly. This gives us a lot of leeway in choosing heuristics for pretenuring when compared with traditional stack allocation that relies on a necessarily pessimistic analysis to decide which objects belong on the stack.

Heap allocating objects immediately because they are likely to be evicted soon can cause extra eviction of objects, even if we were correct in saying that our object would end on the heap. To understand why, consider the following scenario: An object, *A*, is stack allocated. A pointer to object *B*, also stack allocated, is written into *A*. The pointer to *B* is overwritten with a pointer to a third object, *C*. If *A* is now evicted, *B* may stay on the stack, since there is no longer a pointer from *A* to *B* that would violate the Baker constraint. However, if we had been 'clever' and immediately heap allocated *A*, knowing that it was likely to be evicted later then *B* would have been evicted when the pointer to it was written into *A*. In practice this effect is very small as we shall see in Section 5.3.

3.3 Locking of Stack Allocated Objects

Choi et al. [CGS$^+$99], Blanchet [Bla99] and Ruf [Ruf00] cite significant performance gains from elimination of unnecessary locking operations in Java through static escape analysis. For our variation of Baker's scheme something similar is possible. Stack allocated objects in Java are not accessible from other threads (there is of course a stack per thread).

3.4 Loop-Based Allocation and Deallocation on an Object Stack

Conventionally, when stack allocating data, the data is allocated on the same stack as the method invocation data. We propose having separate stacks. One stack holds the method invocations as before. The second stack (the *object stack*) holds stack allocated objects. This arrangement gives us more freedom in determining when to deallocate the stack allocated objects, a subject we delve into in this subsection. We divide the object stack into regions (or frames). It must support the following operations:

- Starting a new region
- Allocating a new object in the current region
- Evicting a given object from any region to the heap
- Scanning the objects in a region and the region's successors for pointers to evicted objects.
- Deallocating a region and all objects in it

Note that our regions do not support intra-region garbage collection and compaction, which, though possible, is probably not necessary, and adds considerable extra complication to an implementation.

Traditionally, the starting of a new frame (region) and the deallocation of a frame have been done on method entry and exit. This is not always a good match for object lifetime: Not all objects allocated in a given method are garbage on method exit; conversely other objects should be made into garbage long before exit in order to avoid space complexity bugs. Therefore various variations have been made to the frame management regime, e.g. deep allocation, inlining of methods to avoid new frame creation, heap allocation of all objects allocated from within loops, etc. (see Section 1.3ff).

We propose a more radical change: Starting a new stack region on entry to a loop, and deallocating the region on leaving the loop. No allocation or deallocation on the object stack occurs at method invocation or termination. On each iteration of a loop, the region should be emptied (equivalently it can be deallocated and a new

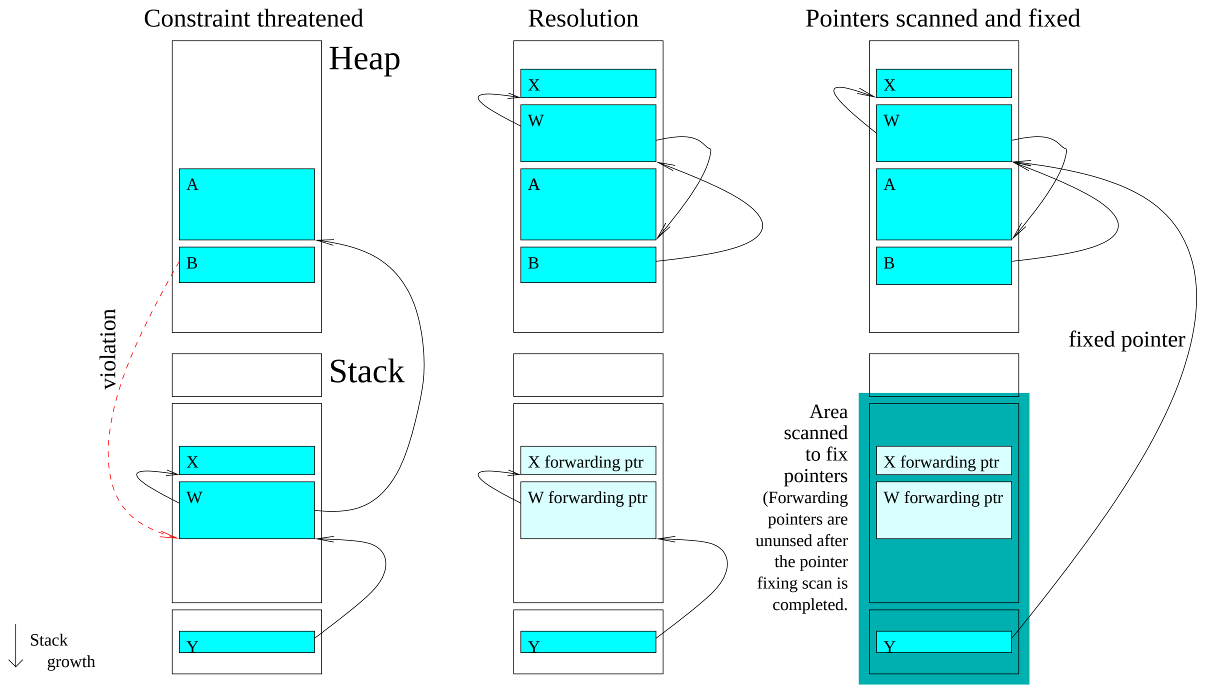
Constraint threatened
Resolution
Pointers scanned and fixed
Heap
Stack
A
B
X
W
Y
violation
Stack growth
X forwarding ptr
W forwarding ptr
Area scanned to fix pointers (Forwarding pointers are unsed after the pointer fixing scan is completed.
fixed pointer

Figure 1. *On the left the write barrier has detected a potential violation of the constraint, as a pointer to W was about to be written into B. In the centre, W has been evicted to the heap, and has pulled X out with it (forwarding pointers are not drawn for clarity). This is as detailed in [Bak92]. On the right we see the modified scheme where the tip of the stack is scanned and the pointer in Y is fixed, removing the need for the read barrier.*

region created in the same place). With this scheme the method call stack and the object stack are decoupled. This scheme is illustrated by figure 2. The advantages of this scheme are:

- Factory methods (any method that allocates a new object and then returns it to its caller) are now not a problem. Since no new frame is started when calling the factory method it will allocate the new object in the same frame as its caller. This frame is not deallocated when the factory method returns.
- Constructors that create new objects and embed references to them in the object they are constructing are unproblematic for the same reasons.
- All allocation can take place in the latest frame. Since there is no deep allocation the frames can be stacked in a linear manner, reducing frame management overhead to a minimum.
- Objects allocated in a loop can still be stack allocated and yet do not cause unlimited expansion of a stack frame.
- There is no need to inline method bodies in their callers merely in order to optimise their allocation behaviour. This is both a simplification and also avoids the code size increase associated with inlining.

Based on these points it is our opinion that loops form a much more natural boundary for creating and destroying regions than method invocations. As far as we know the idea has not previously been pursued.

A potential disadvantage of the scheme in its pure form is that a method that has a recursive call outside a loop and recurses deeply would use a large amount of memory in this scheme. The scheme shares this flaw with all static escape analyses that are powerful enough to stack allocate in recursive methods.

Deeply recursive method calls in most object-oriented languages result in memory usage at least proportional to the recursion depth, since most of them lack tail call optimisation. Therefore it is difficult to construct a credible code example that suffers from excessive memory use due to this effect.

In any case, the problem can be neatly avoided in our scheme by one of two means. In an architecture that does not support tail call recursion (this includes all known Java implementations) the system can use stack allocation heuristics: It would be trivial to put a limit on the object stack size. If the limit is exceeded we can fall back to doing only heap allocation. The infrastructure is already in place to evacuate the stack regions. We can evacuate the stack regions completely and switch to a pretenuring heuristic that only uses the heap subsequently, collecting it as normal (see Section 5.2 on pretenuring heuristics).

In a system that supports tail recursion elimination we have the alternative possibility of treating a tail recursion in a very similar way to a loop iteration. At the point where a method is called in a tail recursive way we can evacuate all objects from the top region, resetting its size to zero. This will of course reduce the oportunities for stack allocation. The primary object oriented platform with tail recursion elimination is Microsoft CLR. Here, a tail recursive call must be specially marked and is associated with a performance penalty [BSS04], so it can be expected that only recursive tail calls will be so marked.

In summary, it should be relatively simple to modify our scheme to be safe for space complexity.

Having two stacks (the region stack and the call stack) means that two stack pointers need to be maintained. This may increase register pressure, which can be a problem on architectures where registers are in short supply. In fact, a frame pointer is also required for the region stack, in order to make it simple to test whether a pointer is contained in the top region and in order to speed up the

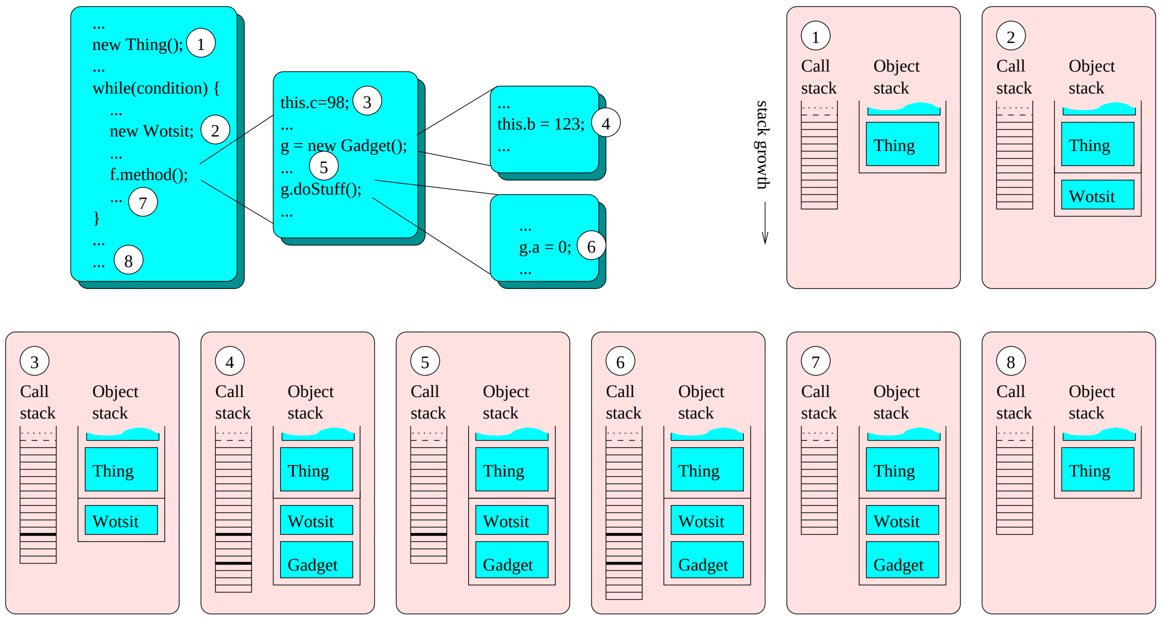

Figure 2. *Separate call and object stacks. At ① the new* `Thing` *object is allocated on the object stack (the* `Thing` *constructor is omitted for clarity). Immediately afterwards, we enter the while loop, and a new region is created on the object stack. This new region is used to allocate the* `Wotsit` *object at ②. We then call the* `method` *method and in ③ we can see the new frame on the call stack, but no change to the object stack. The new* `Gadget` *object is allocated in the same region as the* `Wotsit` *as can be seen in ④ (inside the* `Gadget` *constructor). Various calls and returns take place in ⑤-⑦; they are reflected in the call stack, but have no effect on the object stack since no loops are involved. After ⑦ the loop may restart, in which case the new region on the object stack is emptied, or it may terminate, in which case the new region is deallocated ⑧. At this point the objects in the new region have been either evicted to the heap (evictions are not shown in this diagram) or are garbage.*

operation of clearing the top region by resetting the region stack pointer to the start of the region. Our results show that many regions remain empty at all times, making the region frame pointer more frequently used than the region stack pointer and therefore a better candidate for register allocation.

On the other hand, on the call stack we may be able to save a register. A call stack often needs both a frame pointer and a stack pointer in order to be able to stack alllocate variable length objects. However, if stack allocation occurs on a different stack there is never any need to allocate variable length objects on the call stack, and so the call stack frame pointer can be omitted.

The performance consequences of these implementation choices are a subject for future work.

3.5 Intraprocedural Escape Analysis

One consequence of creating and deallocating stack frames around loops is that we need to do escape analysis on method-local variables and registers. These analyses allow us to determine which method-local variables and registers need to be checked for *dangling* pointers when the loop iterates or terminates and its associated stack region is respectively cleared or destroyed. Dangling pointers are pointers into the imminently empty stack region. If dangling pointers are discovered we either evict the objects they refer to or, if we can determine that the pointers will not be used again, null the pointers.

In the following we describe the analysis for the Java virtual machine because the JVM is simple and well defined. In addition, we implemented it for the byte code language in the JVM. An equivalent analysis can be done on the registers and spill slots of compiled code from Java or other garbage collected languages.

The local variables and the positions on the operand stack are referred to collectively as the *slots*. The slots in a method can transport references from inside a loop to outside it, and they are not subject to the "Baker constraint" with regard to pointing "down" the stack. (This is for the same reasons that registers are not subject to the constraint in the original Baker scheme, see Section 2).

The *intra*procedural analysis is rather simpler than the *interprocedural* analysis needed for traditional stack allocation. No reference can point to a slot, and so there are no aliasing issues: Indirect slot manipulations that affect unknown slot(s). Also, there is no ambiguous control flow. For every instruction there is a limited number of instructions that may follow it. In contrast, for interprocedural escape analysis, the destination of method calls is unknown, both due to polymorphism and dynamic class loading.

In addition, dynamic class loading is no problem at all for intraprocedural analysis. There is no global aspect to our analysis, and no data that needs rebuilding when new classes with new methods are loaded. In fact we perform the entire analysis on the system classes ahead of time and can reuse the analysis for every program that uses the system classes. For the same reason reflection and reflective invocation of methods poses no problem for our analysis. (Our analysis will not consider the complications caused by JVM jsr and ret instructions, since they can be eliminated by code copying and are no longer generated by Java compilers).

The escape analysis needs to identify points where the control flow enters, iterates or exits a loop. At the iteration or exit points we need to identify those references that may still contain a reference to the (object) stack frame being emptied or deleted.

3.5.1 Loop Analysis

We construct a flow graph for the method. In addition to the obvious edges caused by branches, conditional branches and switch instructions we also need edges caused by exception handlers.

We can model these edges conservatively, by ignoring the type of the exception and looking only at the range of the exception handler. Any instruction that can throw an exception is given an edge to each of the handlers that cover its range. The edge is split in two by adding a synthetic node because our flow graph analysis attaches data to nodes rather than edges.

If an exception is thrown for which no handler exists then the current method is terminated, the stack is unwound, and the exception is 're-thrown' by the instruction that called the terminated method. Under some circumstances we add a catch-all exception handler that cleans up stack regions and then leaves the method by re-throwing the exception.

Methods compiled by a Java compiler will only contain reducible flow graphs. This very convenient property (which applies even in the presence of exception handlers) will not necessarily hold for other languages that can be compiled for the Java virtual machine. If a method has a non-reducible flow graph, we can fall back on a less efficient strategy: All allocations in this method take place on the heap, and we start a new object stack frame just before all method calls, and terminate it after they return. This is suboptimal, but safe in terms of space complexity.

After identifying the loops, we can categorise each edge in the graph with respect to the loop into one of three types: the edges that enter the loop, the back edges that indicate loop iteration and the edges that leave the loop. These three types of edges are split into two with the addition of new nodes. These nodes will contain the code needed to handle the results of our intraprocedural escape analysis.

We designate loops in which no allocation and no method calls occur as *trivial*, and delete them from our analysis. If we were to create stack regions for such loops they would always remain empty and so would cause needless overhead. Allocations and method calls that occur within an *inner* loop cannot prevent the *outer* loop from being designated as trivial, for the same reason.

3.5.2 Reference Copy Propagation Analysis

For the region number analysis in the next section it is useful to know which slots (local variables and operand stack positions) can be said to contain the same reference. Most Java virtual machine instructions can operate only on the top stack positions, so values are copied there in order to work on them.

We construct a data flow analysis to determine reference propagation. The analysis is conservative in the sense that it determines slots that are *guaranteed* to contain the same reference. There is no attempt to trace reference copies written into or read from objects/arrays.

3.5.3 Region Analysis

Each instruction is given a *home* stack region number. This tells us the currently active object stack region (see figure 2) for that instruction. It is also the loop nesting depth. This stack region number tells us the region (relative to the active region at method entry) that will be used for a stack allocation at that point. The region at method entry is numbered 0, and the regions created on entering loops are numbered 1, 2 etc. depending on how deeply the loop is nested.

We perform a data-flow analysis to determine for each slot which stack region its referred object is in. There is no attempt to track the stack region number for references that are written into objects or arrays, then read back out, though the Baker constraint does allows us to make some guarantees about references in objects even without tracking.

As is traditional for data-flow analyses this proceeds by generating a function on a lattice for each instruction, then finding a fixed point for the lattice. For most instructions the function is fairly straightforward. For method call instructions that return a reference the returned reference is given the stack region number corresponding to the home stack region number of the instruction. Allocation instructions result in a new reference which is either given the home stack region number of the allocating instruction, or the stack region number zero. Zero would be the correct stack region number if the instruction were *known* to be allocating the object in the heap for some reason (e.g. a directive from the programmer or as a result of data from a training run of the program). In our implementation (see below) we do not make use of this possibility, since it adds little power to the analysis.

Reference store instructions (`aastore`, `putfield` and `putstatic`) are a little more interesting. Due to the Baker constraint we can be sure that after the instruction completes (including possible evictions) the referenced object has a stack region number that is less than or equal that of the referring object. We can use the results of the reference copy analysis above to extend that information to copies of the referenced object reference in other slots. Conversely, the Baker constraint ensures that reference load instructions (`aaload`, `getfield`, `getstatic`) yield a new reference that has a stack region number less than or equal that of the reference used to load it. We conservatively assume the loaded reference has the same stack region number as the reference from which it is loaded.

For the newly inserted loop exit and loop iteration nodes we can assume in our analysis that, after the execution of the new nodes, slots will have been checked and zeroed so that none of them refer to the now defunct stack region.

3.5.4 Using the Analyses

We use the results of the above analyses to generate code for the new nodes in the flow graph: Loop entry, loop iteration and loop exit. On entry to a nontrivial loop we create a new stack region and arrange for subsequent allocations to take place in it.

On loop iteration or exit we must check slots for references that point to the current region number. The check is rather fast since the slots to be checked are usually in registers in the case of JITed code. The check itself is simply a greater-than comparison relative to the start of the stack region. The slots that need checking are those that our static intraprocedural analysis determines are references, live *and* have the correct region number (the region number equal to the loop nesting depth).

If the runtime check finds references that point to the stack region being destroyed, the corresponding objects are evicted. In addition, we need to take dead slots into account if they reference the destroyed stack frame. Although they are dead and thus will never be used by the program itself they are still visible to the conventional garbage collector, which uses the slots as roots (but not the object stack). Therefore we need to zero slots that are references *and* dead *and* have the correct region number.

This operation is rather cheap (in JITed code it merely involves zeroing a register) and generally beneficial even in the absence of stack allocation, since it reduces memory leaks.

4. Measuring Opportunities for Stack Allocation

As a first step to evaluate our proposed stack allocation scheme we perform some detailed simulations. This involves implementing the intraprocedural analysis and implementing a simulated system containing our stack allocation together with a conventional garbage collector. This enables us to verify that the proposal works and to evaluate what opportunities for stack allocation are present, how to decide whether to stack allocate a particular object and the effectiveness of loop oriented creation and deallocation of regions on the stack.

All the suggested modifications to Baker's scheme from the previous section (except synchronisation avoidance) are combined in our simulation in order to test how they work in practice:

- Stack scanning instead of read barriers
- Two stacks, one for method invocations, the other for stack allocating objects in stacked regions
- Stack region creation and deletion based on loop entry/exit/iteration with associated intraprocedural escape analysis
- Heuristics to decide when to pretenure objects to the heap

The simulation framework is in four stages – see Figure 3. The first is an intraprocedural analysis very similar to that described in the previous section. This is implemented as a byte-code analyser that reads Java class files. For the system classes we produce a file that summarises the results of the analysis. This file can be used for all applications since the system classes do not change.

When the intraprocedural analysis has been completed, the second stage of the simulation can proceed: We annotate the class files with new instructions designed to generate a trace of execution. The analysis and the annotation can occur both before-hand and at run time as the classes are loaded into the Java virtual machine.

The third stage is to trace some runs of medium sized Java applications. A high level of detail is needed in the trace files.

From these trace files we can run the actual simulations of allocation and garbage collection and collect statistics on the results. This is the fourth stage of the simulation. The architecture is illustrated in figure 3. An added potential advantage is that we can use interactive programs without running into problems of repeatability – the traces can be replayed several times.

In order to generate these trace files it is not necessary to modify the Java virtual machine at all. Instead we modify the class files to log information on allocations, loop iterations and writes of pointers to stack and object locations. This is detailed below.

4.1 Intraprocedural Analysis for Stack Allocation Around Loops

The intraprocedural analysis needed to generate trace files of memory allocation activity is almost identical to the analysis that would be needed for a virtual-machine-internal implementation of the stack allocation scheme proposed. Therefore the analyses detailed in the previous section are implemented: loop analysis (Section 3.5.1), reference copy propagation analysis (Section 3.5.2), and region analysis (Section 3.5.3).

4.2 Using the Analyses – Rewriting the Byte Codes

In this section we will describe how the system and application classes were modified to output a trace of their activity that could be used by our garbage collection simulator.

The BCEL framework comes with a subclass of Java's ClassLoader class. We use this mechanism to run an entire application under the control of our byte code modifier, inserting trace-generating code into all methods in the classes comprising an application. There are some tricky aspects to this technique: static initializers and finalizers.

Static initializers in Java are methods that are called when the class is initialized. The procedure is rather involved, and is described in [GJS96] section 12.4.2. The order in which static initializers are invoked is poorly defined and easy to perturb by byte-code rewriting. This triggers various hard-to-diagnose bugs in the Java system while it is starting up. Therefore, following [Bur01], we do not add trace-generating code to static initializers in the standard libraries. The result is that some objects appear in the trace files without a preceding allocation event. During the simulation, these objects are assumed to be on the heap on the rare occasions that they are encountered. This is a worst-case assumption that we do not expect to cause significant distortion. Trace-generating code is added to to static initializers of (non-system) application classes without encountering problems.

Java provides for a *finalizer* method to be called on an object immediately before it is garbage collected. In Sun's JVM implementation the finalizers are all run in a separate thread. We chose not to trace this thread, since doing so caused crashes in the VM. The benchmarks we ran in our system did not make heavy use of finalization and we did not observe any distortion due to this decision. For an implementation of our stack allocation scheme, finalizers would not be a serious problem. They are typically implemented using a linked list to prevent premature garbage collection of objects with nontrivial finalizers. This would cause finalizable objects to be evicted to the heap immediately.

4.3 The Simulated Machine

We built a simulated Java virtual machine (JVM) with an allocation and garbage collection subsystem to run the trace files on. The simulated system uses 4 bytes per pointer. Each simulated object is the size of its member variables plus an 8 byte header to simulate virtual machine overhead. Arrays have a 12 byte header. A real implementation might add 4 bytes to store extra bookkeeping in stack allocated objects, though Qian and Hendren [QH03] showed an alternative implementation that reuses the lock field for the purpose.

The simulated JVM has one invocation stack and one object stack per thread. The heap is split into a nursery which is collected by a semispace copying collector and an older generation. In the current implementation the older generation is not collected. Though ideally this generation should be collected too, since no references exist from mature space to the stacks (by design) the addition of a mature space collector is not expected to make a big difference to the important measurements (on the stack regions).

Since the heap is two-generation we implemented a table that records pointers from the old generation to the nursery. This table is maintained by the write barrier.

Objects in the simulated JVM are represented by objects of type `GCSObject` in the simulator. This class represents both real first class objects, stack frames on the call stack and the static data (class variables) associated with classes. The latter are of course always in the oldest generation and stack frames are not in a specific region. All other objects are placed in a region object, which manages their position, performs garbage collection etc. Each object in the simulator also has a reference to its current region.

References in the simulated JVM are represented by *fat pointers* in the simulator. These consist of a reference to the `GCSObject` pointed to (or null) and a region reference that indicates where the object is. When objects are moved to another region in the simulated JVM it is important that pointers to them are updated to refer to the new address (failure to do so would indicate a bug in the algorithm). In the simulator this is represented by updating the region reference in the fat pointer. If a bug in the algorithm leaves us with a dangling pointer in the simulated JVM we risk following a pointer to an object that is no longer there. In the simulator this is represented by an *assertion* inserted wherever we use the fat pointer. The assertion checks that the object is in the same region as the fat pointer indicates it is.

When the memory associated with objects in the simulated JVM is deallocated, the memory is reused. We simulate this in the simulator by marking the `GCSObject` object as dead. Whenever we use an object, an assertion checks that it is not dead.

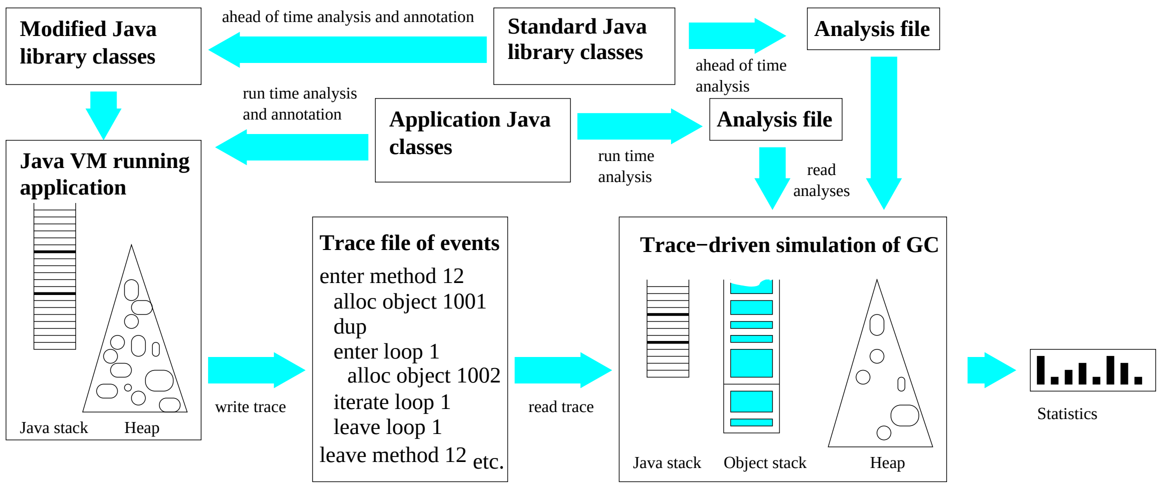

Figure 3. *The architecture for simulation of optimistic stack allocation*

The simulator starts by reading in two files with the result of our intraprocedural analysis. One of these contains the results for the standard classes, while the other has the results for the program run we are simulating. See Figure 3. We then read in the trace file itself, dispatching the events from the trace file to the correct parts of the simulator. In the trace file, objects are referred to by a 64-bit unique identifier (UID).

4.4 Garbage Collecting the Semispaces

The semispace garbage collection proceeds much as a standard semispace collector, with some small differences. The simple, but unsatisfactory, way to collect the semispaces is to treat all references in objects in the stack regions as roots and collect as normal. The problem with this is that not all objects in the stack regions are reachable. This might keep potentially unbounded memory uncollectable merely because it is linked from unreachable objects on the region stack. Therefore in our simulation we treat only the invocation stack as a source of roots and trace *through* the region stacks to find reachable references to the semispaces. We can do this with a traditional marking phase much like the one used in the first stage of a mark-sweep collector. The algorithm is:

```
add invocation stack roots to work list
add intergeneration pointers to work list
for references in work list
  if reference points to stack region object
    if stack region object is not marked
      mark stack region object and add it to work list
  if reference points to semispace object
    if semispace object is young
      move it to new semispace and update reference
    else
      move it to old generation and add to intergen. table
```

As in a traditional semispace collector the objects moved to the new semispace are also added to the worklist. This is implemented by alternating between scanning objects on the work-list itself and scanning newly moved objects in the new semispace until both sources of new references are empty.

Garbage collecting the semispaces does show up one interesting aspect of the Baker constraint: It does not apply during semispace collection. The reason for this is that semispace collections occur during multi-object evictions. This exception to the Baker constraint is not a problem since no stacked region is deallocated during a semispace garbage collection, and the constraint is restored after the semispace collection and the eviction have completed.

5. Results

5.1 Example Programs

We chose to run four of the benchmarks from the popular SPECjvm98 benchmark suite. These benchmark programs have been used in several of the papers already discussed and are intended to be representative of typical Java workloads while performing an interesting amount of allocation and deallocation. They are provided with input data in three variations, designed to run for 1%, 10% or 100% of the standard time. We chose to use the 10% data sets. In this configuration the applications generate trace files with between 32 million and 121 million events in them. See Table 1 for details.

In Table 1 we have a column named “read barriers”. Our proposal does not include read barriers, but we have counted how many times a reference is read from an object. This would be the number of read barrier operations needed under the reasonable assumption that references on the stack are fixed at eviction time, but references in the stacked regions are not fixed until they are next used. It can be seen that read barriers are of the order of one per byte allocated. It seems unlikely that a read barrier that is so frequently invoked can be implemented efficiently enough to be an effective alternative to our data scanning proposal.

It can be seen from Table 1 that the example programs have far fewer loop iterations than they have method invocations. This is because trivial loop iterations (those without allocation or method invocation in the body of the loop) are not counted. Almost all allocations are associated with at least one method invocation in Java: The constructor. The relative frequencies of loop iteration and method invocation help explain why our loop-based stack allocation is able to find more stack-allocatable data than invocation-based stack allocators.

5.2 The Zero Tolerance Policy (ZT)

We experimented with different *pretenuring heuristics*. These are systems for making the decision on whether to stack allocate or heap allocate.

Our first heuristic is the *zero tolerance* (ZT) policy. This heuristic is the relatively simple one that maintains a ‘reputation’ for each point in the code where an object can be allocated. All allocation

Program	Memory allocated (kbyte)	Objects allocated	Read barriers	Write barriers	Method invocations	Stack regions destroyed	Non-empty stack regions	Av. references checked per region	Av. refs zeroed per region
202 jess	3876	101,592	15,620,216	485,506	5,867,568	4,296,007	35,344	0.0006	0.2074
209 db	3694	113,273	5,439,867	503,179	1,216,964	498,671	53,757	0.0933	1.5869
213 javac	7424	214,734	3,566,420	517,346	2,982,147	733,094	34,447	0.0427	0.1667
228 jack	19,534	694,988	7,671,859	1,277,803	7,119,896	1,764,054	301,992	0.0500	0.1778

Table 1. *Characteristics of the example programs.*

Program	Zero Tolerance Section 5.2	Calling Method Section 5.4	Blanchet (safe) [Bla99]	Gay & Steensgaard [GS00]	Hendren & Qian Section 6.1
202 jess	8.4%	12.3%	21%	10.5%/5.3%	6.50%
209 db	8.5%	8.7%			0.74%
213 javac	21.1%	34.0%	13%		8.80%
228 jack	27.1%	60.6%	20%		22.87%

Table 2. *The simple* Zero Tolerance *heuristic and the more complex* Calling Method *heuristic when compared with other papers. The table shows the percentage of allocated bytes that were allocated and deallocated on the stack.*

points start with a good reputation, and are therefore allowed to stack allocate. If an object is evicted, that immediately blackens the reputation of the allocation point that allocated that object, causing it to heap allocate.

The ZT heuristic is relatively easy to support. Each object needs to carry a note of its allocation point with it so that we can identify the reputation when eviction occurs, but this note is only needed in the stack regions. One option would be to reserve a few extra bytes for the purpose when the object is in a stack region, and not to copy them onto the heap. We shall see that the stack regions contain fairly few objects at any one time so the space needed may not be so large.

Using this heuristic and the CM heuristic we will compare the amount of stack allocation achieved (in bytes) with some of the other papers we discussed in this article. Unfortunately the other papers with which we are comparing ourselves did not run their benchmarks with one of the standard data sets provided by SPEC. Therefore, we are not sure of the data sets and in some cases program versions used by the other researchers and there are other reasons why the results may not be comparable. Nevertheless, Table 2 gives a hint at the relative efficacy of the competing schemes. Blanchet uses `javacc` which is a later version of the `jack` program found in SPECjvm98. Gay and Steensgaard run `jess` with two different data sets, giving the two different results shown.

One advantage that some static analyses have is that inlining methods increases the number of distinct allocation sites, which can improve the results. Our optimistic stack allocation scheme could be combined with selective inlining to achieve the same effect.

5.3 Oracle

The *oracle* pretenuring heuristic is a ‘magic’ heuristic in that it can see into the future. It is of course very unrealistic, but it gives an upper limit for the effectiveness of the other heuristics. If a heuristic gets almost as good results as the oracle then it is close to optimal. The oracle heuristic stack allocates an object if and only if that *particular object* would escape eviction in a program run where all objects were stack allocated. It works by using the data from a previous test run of the program with the exact same input.

5.4 Calling Method (CM)

The calling method (CM) heuristic maintains a reputation for each combination of calling method and allocation point. It is an open question how this could be implemented efficiently; possibilities include generating a hash using an allocation point id and the return address on the stack, or passing an explicit token when calling a method.

It is good Java style to have multiple overloaded methods with the same name but with different parameters. These methods will often call each other in sequence. Therefore the immediate caller of a method is very often another method in the same class, which tells us very little about where we are in the program. For this heuristic we therefore use the identity of the most recent method on the call stack that is from a different class than the current one.

The CM heuristic works well, though there is wide variation depending on the benchmark in question. On the `jack` benchmark it doubles the amount of stack allocation, reaching near oracular 60.6% stack allocation.

5.5 Feedback Heuristics (ZTF and CMF)

The feedback heuristics (ZTF and CMF) work in the same way as the non-feedback heuristics with the corresponding names (ZT and CM), but they initialize their table of reputations from data collected during a test run. We ran the SPEC programs on the 1% datasets in order to produce reputation data for the allocation sites used in the 10% test runs shown in Table 3.

For the Jess and DB benchmarks the input data set for the 1% training run is disjoint from the input data set for the 10% measured run. For the Javac benchmark we think this is also the case. For the Jack benchmark, the input data appears to be identical for the 1% and 10% runs. No source code is provided for Jack and Javac.

Using feedback data makes almost no difference to the amount of stack allocation, but it can make a big difference to the amount of data that is evicted from the stack. This supports our intuition that most evictions occur at the start of the program run.

5.6 Assessment of the Simulation Results

For our proposal to be viable it is important that the overhead associated with it is low enough that the cache benefits from stack allocation are not more than outweighed by the extra work that must be performed by the runtime system. The extra work consists of (1) invocations of the slow path code from the write barrier (see Section 2), (2) overhead for the pretenuring heuristics, (3) the overhead of pointer checks and slot zeroing resulting from the intraprocedural analysis and finally work associated with evictions of objects. The eviction work can be divided up into (4) the actual copying of evicted objects and (5) the scanning of the stacks needed to fix pointers to the old locations.

1. The fast case of the write barrier works very well everywhere except in the nursery. For example for `jess` with the ZT heuristic the write barrier tests around half a million pointer pairs (Table

	ZT	CM	ZTF	CMF	Oracle
	Allocated and deallocated on stack				
202.jess	8.39%	12.34%	8.34%	12.16%	55.58%
209.db	8.48%	8.65%	8.32%	8.48%	8.55%
213.javac	21.09%	34.02%	21.00%	33.91%	64.73%
228.jack	27.05%	60.61%	27.02%	60.57%	62.99%
	Evicted from stack				
202.jess	0.42%	1.09%	0.03%	0.06%	0.18%
209.db	0.20%	0.61%	0.00%	0.01%	0.16%
213.javac	0.54%	0.91%	0.18%	0.31%	0.16%
228.jack	0.06%	0.16%	0.00%	0.00%	0.07%
	Data scanned to fix pointers				
202.jess	4.80%	6.72%	0.06%	0.22%	0.33%
209.db	0.92%	1.53%	0.01%	0.02%	0.07%
213.javac	0.85%	1.96%	0.31%	1.15%	0.72%
228.jack	3.33%	3.34%	0.00%	0.05%	0.11%
	Stack scanned for semispace GC				
202.jess	3.61%	3.51%	3.59%	3.51%	2.11%
209.db	1.37%	1.37%	1.37%	1.37%	1.37%
213.javac	2.43%	2.16%	2.43%	2.20%	1.50%
228.jack	0.87%	0.50%	0.85%	0.51%	0.47%
	Maximum region stack size				
202.jess	15k	46k	3k	5k	11k
209.db	16k	31k	3k	3k	9k
213.javac	27k	27k	14k	14k	22k
228.jack	66k	92k	51k	62k	68k
	Average regions scanned per eviction				
202.jess	1.052	1.048	1.000	1.029	1.000
209.db	1.000	1.000	1.000	1.000	1.000
213.javac	0.946	0.977	0.897	0.963	1.000
228.jack	0.985	1.018	1.000	1.000	1.000

Table 3. *Evaluation of competing pretenuring heuristics. All percentages are calculated as a proportion of total memory allocated. GC stack scanning data is shown for comparison.*

1). Of these, only 12% fail the fast case less-than test so that the conventional write barrier code must be called. Of those that fail the initial test 94.5% are false alarms caused by two objects both in the nursery, a case that the ‘slow path’ can easily be optimised to handle quickly. Almost no write barrier false alarms occur in the region stacks (0.2%), and when they do happen they are invariably in the most recent region, which is the easiest to test membership of. More data on the effectiveness of the write barrier may be found in our M.Sc thesis [Cor04].

2. The ZT heuristic is simple to implement with low overhead and is able to find substantial amounts of stack-allocatable data (see Table 2). The CM heuristic is considerably more difficult to implement with low overhead but also considerably better at finding stack-allocatable data.

3. Our intraprocedural analysis results in almost no slots needing to be checked when a region is deallocated (Table 1). We quite often need to zero a single slot when a region is deallocated or emptied. This is, however, a very fast operation (zeroing a register).

4. From Table 3 we can see that very little data is evicted from the stack regions. The copying is rarely more than 1% of the memory allocated, often much less.

5. Some heuristics cause rather a lot of scanning of the invocation stack and stack regions. However, it is never more than about twice as much scanning as the system would be doing anyway in order to find roots when garbage collecting the semispaces and it is sometimes much less. As can be seen from Table 3, the feedback-aided heuristics do much less stack scanning than the non-feedback aided heuristics. The reason for this is probably that almost all evictions affect only the bottom stack region and the bottom few invocation stack frames. For systems where feedback-aided heuristics are not realistic, an alternative implementation that combined the loop-oriented, heuristic-controlled optimistic stack allocation with a highly optimised read barrier could be an interesting alternative.

In summary all the possible sources of overhead are small or can be made so. Against these overheads we must consider the benefits, i.e. how well the stacks make use of the cache. One indication of this is their size, since level 1 caches are usually between 16kbyte and 64kbyte large. Region stacks are usually very small and can be expected to make very good use of the cache. The ZTF heuristic on `javac`, for example allocates 21% of objects in a space maximally 14kbyte large, putting a very small upper bound on the cache footprint of the region stack.

In conclusion, our results are positive, showing that the strategies we propose have encouraging tradeoffs between extra work the runtime system must perform to implement them and the results they achieve in terms of stack allocating data.

6. Related Work

Much of the related work has been described earlier.

6.1 An Adaptive, Region-Based Allocator for Java

“An Adaptive, Region-Based Allocator for Java” by Qian and Hendren [QH03] contains some of the same ideas as our proposal. Like our proposal they optimistically stack allocate all objects and use a write barrier to detect pointers into these regions. They also use pretenuring heuristics. Like in our proposal stack allocated objects are placed on a different stack from the method invocations, and this allows them to decouple the two to a certain extent. Stack region lifetimes in [QH03] are still based roughly on method invocations (rather than loop iterations), but a method that returns a reference will always share a stack region with its caller.

The most significant difference is the system’s reaction to a problematic pointer into a stacked region. In this case the system declares the stack region to be escaped, and it conceptually becomes part of the heap. The region is, however, not moved to a different address. The consequence of this is that their stacked regions cannot be arranged contiguously in memory, and finding space for a new region can be very costly in terms of execution time. Unlike our proposal Qian and Hendren’s scheme allows for garbage collection within the stacked regions. In effect the garbage collector is a multi-generational collector that treats the regions as a generation. Heuristics are used to determine when objects should be promoted to other generations. It is not clear whether this avoids the space complexity problems around loops and deep recursion.

Another consequence of the more free placement of stack regions is that the simple write barrier due to Baker (Section 2) cannot be used. Instead memory is arranged in *chunks* and each chunk allows a relatively fast lookup to determine the region it belongs to. Their write barrier still appears rather more complicated than ours, and the chunk system causes some waste of memory since the chunks have a fixed size and are not always filled up. The memory wasted by this is termed *froth* in the paper and varies from less than 1% to over 600% depending on the benchmark involved.

The proposal in the paper has been implemented for the Jikes RVM. At the time of the paper no performance improvements could be attested. The overhead of allocating and deallocating the regions and the overhead of the write barrier was more than the gain from stack allocation. As can be seen from Table 2 our scheme stack allocates more data and likely with less overhead.

6.2 Region Inference

In a series of articles starting 1992 and summarised in 2004 [TBEH04] Mads Tofte et al. describe an automatic transformation for ML progams that annotates them with region information. All allocation takes place in a stack of regions and functions take zero or more region parameters in addition to those parameters they already took before the transformation. In the pure form of the system, deallocation can only take place by deallocating a whole region at once. The system was implemented in the ML-Kit [BTV96].

The ML Kit region based system is proved safe in terms of object and region lifetimes. Thus there is no need to test at runtime for dangerous references that refer to an object in a region closer to the top of the stack. On the other hand, the system is not safe for space complexity. Certain constructs will cause memory use to grow asymptotically larger than it would be with garbage collection. For this reason garbage collection within the regions was implemented (see [HET02]). Garbage collection never moves objects from one region to another, it simply shrinks the regions by removing unreachable objects.

Though the analysis is in principle local, some of the optional analyses are whole-program in nature. An example is the storage-mode analysis, which attempts to prevent tail recursive function invocations from using memory proportional to the number of recursions (see “From Region Inference to von Neumann Machines via Region Representation Inference” [BTV96]).

The transformation can be done on individual modules, but it is always done by making use of information about the way the module is used. Thus the code for a module may need to be regenerated with different numbers of region parameters to functions if the module is used in a different way (see Martin Elsman’s Ph.D. thesis page 189 [Els99]). This work is performed at link time, but if the scheme was implemented in a language with dynamic loading of code (for example Java) the analysis and code regeneration would complicate the run-time system.

In the case where a program can be tuned to use only region allocations without region size explosions the system is likely to be superior to our proposal in terms of speed, space and real-time response.

The set of objects that can be stack allocated and deallocated for the ML Kit region scheme is neither a superset or a subset of those our scheme allocates. No program analysis can find all dynamically available opportunities for stack allocation. This is both due to inevitable insufficiencies in the analysis algorithm and due to data-driven program behaviour. For example, a parser that is only presented with well-formed inputs has a lot of code that is ’unreachable’ in the sense that it is never executed. However, no correct program analysis can ever determine this, since the program analysis cannot know that the inputs are well-formed. Any references that only escape by means of this ’unreachable code’ can be stack allocated in our scheme, but must be allocated in a deep region by the ML Kit. This region is likely to need garbage collection.

On the other hand, our scheme can only allocate in the top region or on the heap, while the ML Kit scheme is free to allocate in any region, giving more opportunities for placement of objects in a region whose lifetime corresponds well with the object’s lifetime.

Our decision only to allow allocation in the top region helps reduce space waste and simplifies allocation and deallocation of regions. Our regions do not need to be based on a whole number of linked memory pages, but can be arranged contiguously in a memory area.

Cherem and Rugina [CR04] have developed a similar program transformation for Java. The result of the transformation is a region annotated program that runs on a region-aware virtual machine. Unlike the ML Kit, regions are not lexically scoped, but can be associated with loops. Like the ML Kit, the resulting program is safe and needs no run-time checks for pointers into deallocated regions. The analysis is whole-program and thus only targets the static subset of Java. The transformation can eliminate conventional garbage collection completely, but in this case the transformation is not in general safe for space complexity. No results are presented for SpecJVM programs.

The Cherem and Rugina program transformation has many of the same strengths and weaknesses relative to our proposal as the ML Kit and for broadly the same reasons: Firstly, deep allocation is allowed and secondly, the analysis is static rather than dynamic.

7. Conclusion and Further Work

Our stack allocation proposal shows promise. Our simulation of the full proposal (including loop-based rather than invocation-based allocation, pretenuring heuristics and stack scanning to fix pointers) indicates that the overhead from the sources we identified can be expected to be low and that a relatively large proportion of memory can be allocated on small stacks. These small stacks can be expected to make good use of cache memory. Despite its efficacy the analysis behind our scheme is simple to implement and robust in the face of popular Java technologies such as dynamic class loading, exception handling and reflection.

There are many avenues that can be explored relating to new allocation heuristics that may be more effective or easier to implement than the ones already measured. In addition, is is worth exploring whether objects that are evicted to the heap at the moment a region is deallocated could in some circumstances be added to the previous region instead.

The next stage is to implement our scheme in a Java Virtual machine. The Jikes RVM is a suitable target since it has some infrastructure for running with different garbage collectors. Since our proposal necessitates some code rewriting (insertion of exception handlers to deallocate stack regions, checking of local variables on loop iteration and loop exiting) it will not be possible to implement our scheme using only the pluggable garbage collection interface provided by the standard Jikes RVM. Changes will also be needed to the core VM. Such an implementation would provide opportunities for more detailed measurements.

Acknowledgments

I wish to thank Ole Lehrmann Madsen for supervising my M.Sc. thesis on which this article is based [Cor04]. Thanks to Ulrik Pagh Schultz, Erik Ernst and Mads Torgersen for encouragement and detailed comments on an earlier version of the paper. Thanks to Flemming Gram Christensen and Kresten Krab Thorup for stimulating discussions on the subjects of stack allocation and garbage collection. Thanks to the anonymous reviewers for useful criticism and to Greg Morissett for his help in the final shepherding of this article. Finally, many thanks to my wife, Aino Corry, for love, support, and looking after the family in my busy periods.

References

[AS94] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS–TR–450–94, 1994. Available from: `http://citeseer.nj.nec.com/appel94empirical.html`.

[Bak92] Henry G. Baker. CONS should not CONS its arguments, or a lazy alloc is a smart alloc. *ACM SIGPLAN Notices*, 27(3), March 1992. Available from: `http://home.pipeline.com/~hbaker1/LazyAlloc.ps.Z`.

[BH04] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM ’04: Proc. of the 4th intern. symp.*

on Memory management. ACM Press, 2004. Available from: http://cs.anu.edu.au/~Steve.Blackburn/pubs/papers/wb-ismm-2004.ps.gz.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999. Available from: http://citeseer.nj.nec.com/blanchet99escape.html.

[BSS04] Yannis Bres, Bernard P. Serpette, and Manuel Serrano. Bigloo.NET: compiling Scheme to .NET CLR. *Journal of Object Technology*, 3(9):71–94, 2004. Available from: http://www.jot.fm/issues/issue_2004_10/article4.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996. Available from: http://citeseer.ist.psu.edu/birkedal96from.html.

[Bur01] Vidal Burgoa. Implementation of CPU resource accounting for Java (M.Sc, U. of Geneva). ftp://cui.unige.ch/pub/tios/papers/Diploma-Vidal.pdf, 2001.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *Proc. of the Conf. on Object-Oriented Prog. Sys., Lang., and Applications (OOPSLA)*, pages 1–19, 1999. Available from: http://citeseer.nj.nec.com/choi99escape.html.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profi le-driven pretenuring. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–173, 1998. Available from: citeseer.ist.psu.edu/cheng98generational.html.

[Cor04] Erik Corry. Stack allocation for object-oriented languages (M.Sc. thesis, Department of Computer Science - Daimi, University of Aarhus). http://www.daimi.au.dk/~corry/corrythesis.pdf, 2004.

[CR04] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 85–96, New York, NY, USA, 2004. ACM Press. Available from: http://www.cs.cornell.edu/~rugina/papers/ismm04.ps.

[DTM93] Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs with intensive heap allocation. Technical Report CMU-CS-93-227, 1993. Available from: http://citeseer.nj.nec.com/diwan93memory.html.

[Els99] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen, January 1999. Available from: http://www.it.edu/people/mael/mypapers/phd.ps.gz.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specifi cation*. Addison-Wesley Longman Publishing Co., Inc., 1996. Available from: http://java.sun.com/docs/books/jls/.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Intern. Conf. on Compiler Construction (CC'2000)*, volume 1781. Springer, 2000. Available from: http://citeseer.nj.nec.com/gay00fast.html.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152, New York, NY, USA, 2002. ACM Press. Available from: http://www.itu.dk/~mael/mypapers/pldi2002.ps.gz.

[Inta] Intel. The IA-32 Intel architecture software developer's manual, volume 3: System programming guide. Available from: http://www.intel.com/design/pentium4/manuals/253668.htm.

[Intb] Intel. Intel XScale microarchitecture for the PXA250 and PXA210 applications processors. Available from: http://developer.intel.com/design/pca/prodbref/pxa250.pdf.

[Int00] Intel. IA-32 Intel architecture optimization reference manual, 2000. Available from: http://www.intel.com/design/pentium4/manuals/248966.htm.

[Pro99] H. Prokop. Cache-Oblivious Algorithms, June 1999. Master's thesis, MIT Department of Electrical Engineering and Computer Science. Available from: http://citeseer.ist.psu.edu/prokop99cacheobliviou.html.

[QH03] Feng Qian and Laurie Hendren. An adaptive, Region-Based Allocator for Java. *SIGPLAN Not.*, 38(2 supplement):127–138, 2003. Available from: http://doi.acm.org/10.1145/773039.512446.

[Rei94] Mark B. Reinhold. Cache performance of garbage-collected programs. In *SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 206–217, 1994. Available from: http://citeseer.nj.nec.com/reinhold94cache.html.

[RMB+99] Alastair Reid, John McCorquodale, Jason Baker, Wilson Hsieh, and Joseph Zachary. The need for predictable garbage collection. In *Proceedings of the ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. Available from: http://www.cs.utah.edu/flux/papers/gc-wcsss99.ps.gz.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proc. of the ACM SIGPLAN 2000 conf. on Prog. lang. design and impl.*, pages 208–218. ACM Press, 2000.

[Ste99] Darko Stefanovic. *Properties of age-based automatic memory reclamation algorithms*. PhD thesis, 1999. Director-J. Eliot Moss. Available from: ftp://ftp.cs.umass.edu/pub/osl/papers/stefanovic_dissertation.ps.gz.

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, 2004. Available from: http://www.itu.dk/~birkedal/papers/regmmp.ps.gz.

[WF] R.L. While and A.J. Field. The Non-stop Spineless Tagless G-machine. Available from: http://citeseer.nj.nec.com/while96nonstop.html.

[Zor90] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, 1990. Available from: http://citeseer.nj.nec.com/zorn90barrier.html.