# Stack Allocation of Objects in the Cacao Virtual Machine

Peter Molnar
Theobroma Systems Design und Consulting GmbH
Wien, Austria
peter.molnar@theobroma-systems.com

Andreas Krall Florian Brandner
Institut für Computersprachen
Technische Universität Wien
{andi,brandner}@complang.tuwien.ac.at

## Abstract

Stack allocation of objects reduces the cost of object allocation and garbage collection and can thus lead to large reductions in runtime. Escape analysis can statically determine which objects are eligible to stack allocation by examining the escape behavior of allocation sites. If objects created at a particular allocation site do not escape, i.e., are guaranteed not to leave the scope of the allocation site, stack allocation instead of expensive heap allocation can be applied.

We have implemented a lightweight and fast escape analysis within the CACAO Java Virtual Machine to enable stack allocation. The analysis proceeds in two stages: an intraprocedural analysis computes escape information for each allocation site within a single method and builds call-context agnostic summary information for the method. The summary information is then used during interprocedural analysis to capture the escape behavior of method arguments. The computed escape information is finally used to allocate a subset of thread-local Java objects on the call stack.

The implementation has been evaluated using the SPEC JVM98 and the dacapo benchmark suites. For the SPEC benchmarks up to 90% of all objects allocated at runtime can be allocated on the call stack, leading to a speed up of up to 69%. The more complex dacapo benchmarks still show speedups of up to 10%, with up to 20% of all objects being allocated on the stack.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Memory management

***General Terms*** Algorithms, Languages

***Keywords*** object allocation on stack, escape analysis, just-in-time compiler

## 1. Introduction

Object allocation on the heap and garbage collection can contribute a significant part of the runtime of Java programs. Stack allocation of objects is an effective optimization technique to reduce the allocation and deallocation costs. Unfortunately not all objects can be stack allocated: if the lifetime of an object exceeds the lifetime of its creation site, it must not be allocated on the stack. *Escape analysis* is necessary to determine which objects can be safely stack allocated.



*PPPJ '09,* August 27–28, 2009, Calgary, Alberta, Canada.


Escape analysis in object oriented languages studies the lifetime of objects, more precisely whether the lifetime of an object is bounded by the lifetime of its creating site, be it the creating method or the creating thread. The information gained through *escape analysis* can be used to implement several optimizations in the domain of synchronization and memory management. The algorithm presented in this work is based on an algorithm by Kotzmann [9, 10, 11].

CACAO [12] is an open source Java virtual machine. Its development started as a research project at the Vienna University of Technology. CACAO follows a compile-only approach: all bytecode gets compiled *just-in-time* the first time a method is executed. CACAO initially targeted DEC's 64bit Alpha architecture, but was soon ported to MIPS and PowerPC, later to IA32, x86_64, ARM, SUN Sparc, PowerPC64, IBM S390, and Coldfire architectures.

An additional higher optimizing compiler is under development. Frequently executed methods are recompiled with more expensive optimizations. The recompilation framework which supports on-stack replacement of methods, de-optimization and its use with adaptive inling has been presented in [15]. In this work we present results for additional optimizations for the optimizing compiler.

The main contributions presented in this work are:

- A detailed evaluation of the potential of escape analysis and stack allocation using instrumentation within the CACAO JVM.
- An implementation of a lightweight escape analysis for CACAO following the approach by Kotzmann [9, 11].
- A detailed evaluation of the results of the escape analysis and the related stack allocation of objects using the SpecJVM98 and dacapo benchmark suites that shows that our approach achieves considerable speedups by eliminating memory allocation and garbage collection overhead.

The optimization potential of stack allocation is evaluated in section 2. In section 3 we present our escape analysis algorithm and in section 4 we give an experimental evaluation of our implementation in the CACAO VM. Section 5 discusses related work.

## 2. Escape Behavior

Static escape analysis algorithms are inherently conservative: they are not able to determine the exact escape behavior of objects, but they approximate the behavior in a way that guarantees that no escaping object will be falsely identified as non-escaping. Traditionally, escape analysis algorithms are evaluated by comparing the number of objects they identify as non-escaping to the number of total objects allocated in the program. This ratio can be calculated from either *static* numbers that are collected during compilation or using *dynamic* numbers that are collected at runtime. Static results only relate the total number of allocation sites to the number of allocation sites recognized as non-escaping, and is thus generally not suited to predict possible gains in runtime, e.g., by eliminat-

ing overhead of heap allocation using stack allocation. Dynamic numbers are much better suited for this purpose. However, even dynamic numbers only allow the relative comparison of different escape analysis algorithms, and can not give insights into the actual potential of a given benchmark.

Because of its conservative nature an escape analysis usually takes the most pessimistic assumptions possible. During intraprocedural analysis, it is assumed that every branch will be taken and during interprocedural analysis, it assumes a pessimistic call context. These pessimistic assumptions, which in practice occur rather rare, increase the gap between the results of the analysis and the real escape behavior significantly. In order to evaluate the accuracy of the analysis and to get a better understanding of the interprocedural escape behavior, more accurate and realistic escape data is desirable.

We have extended the CACAO JVM to trace object references at runtime using code instrumentation. This not only allows to collect information on the reachability of objects, but also to collect data on the escape state of objects and even detailed information where objects actually escape to.

Objects are grouped into non-overlapping *regions* based on reachability information that is collected during program execution. The heap is modeled using a global region (or heap region) that contains all objects that possibly escape. In addition, every active stack frame is modeled using a separate region. Each region is associated with a *lifetime*. In the case of the global region this lifetime is infinite, while stack regions have a shorter lifetime that is derived from the negated stack depth of the corresponding stack frame. Note, that we are not interrested in the absolute lifetime of a region, but rather in the relative lifetime, the negated stack depth is thus a well suited approximation. Initially newly allocated objects are assigned to the region of the current stack frame. If an object becomes reachable from a different region with a longer lifetime, the object is moved into that region. It is important that objects may only be moved from regions with short lifetime into a region with longer lifetime. In addition, recursively all objects reachable from the original object need to be moved to the new region as well.

When the program terminates the global region contains only objects that are considered globally escaping. Objects that are left in regions associated with a stack frame at the time of its destruction are considered thread-local, as they become unreachable after the destruction.

During JIT compilation of a Java method, the following intermediate representation constructs are instrumented: method entry, method exit, exception throw, object allocation, field assignment, array store, and global variable assignment.

When compared to a static escape analysis algorithm, this algorithm determines the root set of escaping objects in an analogous conservative way: objects reachable from global variables, thrown as exceptions and passed to native methods are globally escaping. The difference is that a static escape analysis identifies all objects that *might* get reachable from this root set as globally escaping, while this algorithm identifies only objects that *are* reachable from this root set. A static analysis assumes that every possible control flow path of a program could be taken, while the data collected during instrumentation only accounts for paths that are actually executed.

In the first series of measurements, the total number of available thread-local objects was determined (see Figure 1). The numbers were counted as follows: upon method exit, after the return value has been moved into the callers region, all objects that remained in the method's region were counted as thread-local.

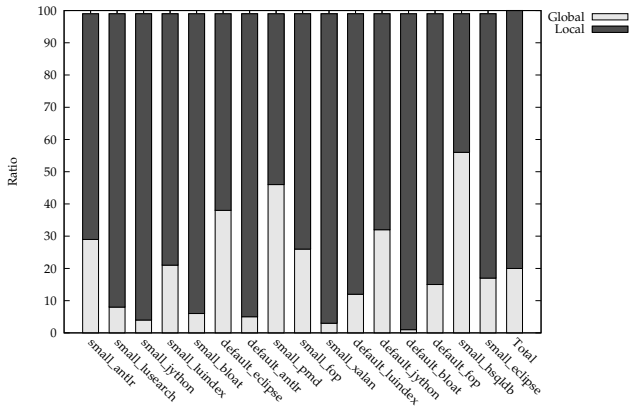



**Figure 1.** Number of thread-local objects.

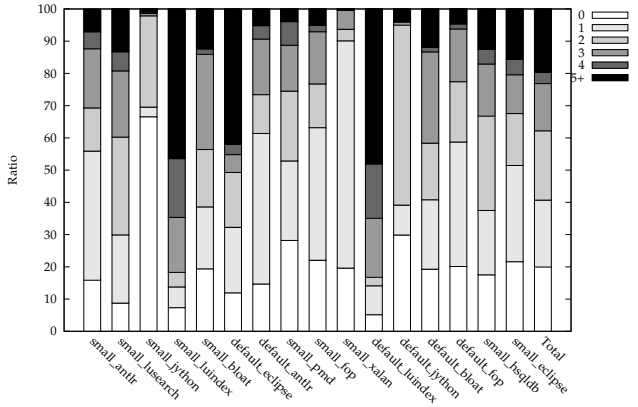



**Figure 2.** Number of stack frames objects passed towards the caller.

Finally, we have investigated how objects that are considered eligible for stack allocation are passed to other methods. Figure 2 shows the number of call frames objects are passed towards the caller i.e., returned from methods. It can be seen that a minority of thread-local objects, around 10 - 20 %, does not move towards the caller at all. The rest of thread-local objects however does. But for most benchmarks thread-local objects are typically not returned more than three stack frames towards the caller.

When designing an intraprocedural analysis with stack allocation or regions in mind, there are basically three options on handling thread-local objects that are returned towards the caller:

1. An object that is returned from a method is considered always globally escaping.
2. If a method can return an object up to 1 stack frame upwards, the caller allocates memory in its stack frame. A specialized version of the callee is compiled that accepts a pointer to the reserved memory as an additional parameter.

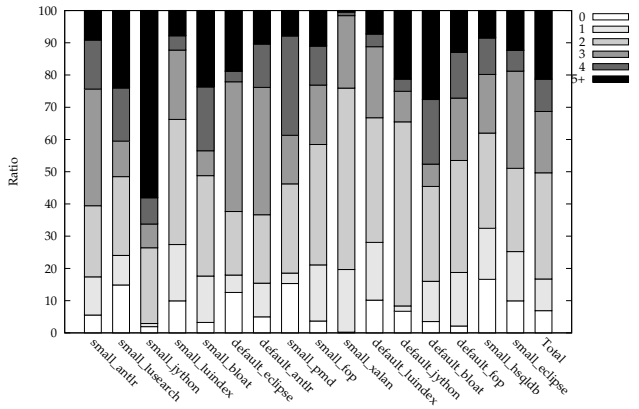



**Figure 3.** Number of frames objects are passed towards the callee.

3. If possible and feasible, the method is inlined into the caller. This way the returned object gets trapped in the caller and is not returned.

Considering the results obtained by the experiment in this section we can predict how effective each of these options can be in practice using perfect escape information:

1. If objects returned from a method escape globally, only 20% of thread-local objects can be considered non-escaping.
2. If thread-local objects returned to a caller can be handled that do not escaping further, 26% of thread-local objects can be saved from escaping.
3. Good inlining decisions up to a depth of up to three levels can save a lot of objects from escaping, usually more than 50% of thread-local variables.

Similarly, Figure 3 shows the number of call frames objects are passed down, i.e., passed as arguments to callees. The figure shows that in most benchmarks the majority of thread-local objects is not passed more than four stack frames down the call chain. The results indicate that it is sufficient to traverse only up to four levels of the call graph in order to achieve good results for an interprocedural escape analysis in practice.

Also note that, because of Java's two step object creation process: (1) allocation of a block of memory followed by (2) a call to the class' constructor, every Java object is passed at least one stack frame downwards as an argument to the constructor. Arrays constitute an exception to this rule, as they do not have a constructor and are initialized directly by the VM. Consequently, objects that are not passed to any callee depicted in Figure 3 are exclusively arrays.

## 3. Escape Analysis Algorithm

The *just-in-time* compiler of CACAO is organized in passes. In the context of this work, several passes were designed, implemented and adapted as described in this chapter.

First a *control flow graph* representing regular and exceptional control flow is created for the compiled method. Then the intermediate representation of the method is transformed into *static single assignment* form for the purpose of performing *escape analysis* of the method. This pass computes escape information for static allocation sites along with summary information for each method that is subsequently used during interprocedural analysis to capture the escape behavior of method arguments and return values. The information computed by this pass is finally used to perform stack allocation. Special care has to be taken when applying stack allocation inside loops, because the size of stack frames is not allowed to changed dynamically in CACAO. Therefore, a static loop analysis is performed in order to handle stack allocation in these cases. We will discus details of each of these passes in more detail in the subsequent sections. In addition, requirements to the runtime system and the VM for the implemented optimizations are discussed.

### 3.1 Control flow graph

To model intraprocedural control flow in a compiled method, the JIT compiler constructs a *control flow graph* (CFG): a directed graph with nodes consisting of *basic blocks* and edges representing control flow transitions between them. Intraprocedural control flow of a Java method is categorized into two classes:

**Regular control flow** originates from the execution of a conditional or unconditional branch instruction and transfers control to a different basic block.

**Exception control flow** originates from a *potential exception-throwing instruction* (PEI) in response to an exceptional condition and transfers control to an exception handler.

To augment the regular CFG with exceptional control flow, basic blocks need to be split at every PEI, and an edge from the basic block fragment to every potential exception handler needs to be added, leading to a CFG greater in size. A space-efficient representation of exceptional control flow can be achieved using the *factored control flow graph* (FCFG) [3]. In addition to regular CFG edges it contains *factored edges*. A *factored edge* $BB \xrightarrow{T} EH$ from a basic block $BB$ to an exception handler $EH$ annotated with an exception type $T$ represents all control flow transitions from PEIs throwing an exception of type $T$ to the respective exception handler. In CACAO, a variation of the FCFG has been implemented, exceptional control flow is represented using factored edges, but they are not annotated with an exception type. Instead a factored edge connects a basic block containing at least one PEI to every exception handler for that basic block. This leads to a less precise, but still valid, FCFG.

The reason for this simplification is that for a correct match of the exception thrown by a PEI and the exception caught by an exception handler a subtype check is required. This in turn requires the two exception classes in question to be *linked*, i.e. fully initialized in the runtime system. This is not necessarily the case at compile time, even not at recompilation time, as classes are linked *lazily* the first time they are used at runtime.

### 3.2 Static-single assignment form

The intermediate representation (IR) of a Java method is transformed into *static-single assignment form* using the algorithm proposed in [14], which is based on abstract interpretation of the IR. According to Mössenböck, it is better suited than the mainstream algorithm proposed in [4] for an input program that is in an IR rather than in the form of source code.

The algorithm traverses the IR once, basic block by basic block, maintaining a per basic block *state array*. The *state array* contains for every IR variable the definition flowing out of the basic block. State arrays are *merged* at join points, which results in $\phi$-functions.

In presence of loops, definitions flow back from the loop body into the loop header via backward branches and the state array has to be recomputed. To prevent this, $\phi$-functions for every variable are created in loop headers in advance, and are eliminated later if they turn out to be redundant.

### 3.3 Escape state

The aim of the escape analysis algorithm is to annotate static object allocation sites, which represent runtime objects, with an *escape state*. The state can have one of the following values:

**ESCAPE_NONE:** the object is accessible only from its creating method. Such an object can be eliminated and replaced with scalars.

**ESCAPE_METHOD:** the object escapes its creating method, but does not escape the creating thread. This happens if the object is passed to a callee, but does not escape further. Such an object can be allocated on the stack, in addition, synchronization can be eliminated for the particular object.

**ESCAPE_METHOD_RETURN:** the object escapes its creating method via a return to the caller. Such objects are not further optimized, however, method inlining can reduce the number of objects that are returned beforehand.

**ESCAPE_GLOBAL:** the object escapes its creating method and even its creating thread, i.e., the object becomes accessible via a static variable or is passed to native code or code that is not yet analyzed. No optimizations are possible on such an object.

An ordering on the escape state values is defined with `ESCAPE_NONE` and `ESCAPE_GLOBAL` being the lowest and highest values respectively.

### 3.4 Intraprocedural escape analysis

The first step of the actual escape analysis operates on the intermediate instructions of a method one by one. Intermediate instructions in CACAO have the form of quadruple code, $dst = OP(s_1, ..., s_n)$ where $dst$ and $s_i, i \in \{1, \ldots, n\}$ are intermediate variable indexes. References to run time objects are stored in intermediate variables that are annotated with an escape state during intraprocedural escape analysis. The escape information of variables is then back-propagated to allocation sites and can be used to replace the original allocation construct with stack allocation. The escape state of variables is initialized to `ESCAPE_NONE` and is further refined during the analysis such that the escape state increases with respect to the ordering defined in Section 3.3.

Variables are grouped into *equi-escape sets* (EES) where all members share the same escape state. The escape state of variables is adjusted on copy operations $dst = src$ or $\phi$-functions $dst = \phi(s_1, ..., s_n)$. The analysis is thus a flow-insensitive *Steensgaard* analysis. For brevity, adjusting the escape states of variables denotes merging the two EESs of the variables. The escape state of the newly formed ESS corresponds to the largest escape state of the variables ESSs with repsect to the previously defined ordering.

If an object $s2$ is assigned to a field of an object $s1$, and $s1$ escapes later, $s2$ escapes as well. If however $s2$ escapes at a later point, $s1$ won't, as it is not necessarily accessible via $s1$. This dependency can't be modelled using an EES, as it is unidirectional. For this purpose, every variable maintains a list of (variable, field identifier) pairs it possibly references via fields - the *dependency list*. If the ESSs of two variables are merged, their dependency lists are merged as well, because the new set may reference any object from the union of the two dependency lists. If the escape state of a variable changes, the escape state of all elements of the dependency list must be adjusted as well.

The IR is traversed once to construct the EESs. The following IR constructs are considered:

**Method prologue** The IR variables containing arguments are initialized as not escaping.

**ICMD_NEW, ICMD_...NEWARRAY** An object is newly allocated. The source operand for the instruction is a variable, containing a pointer to a class descriptor previously loaded with an `ICMD_ACONST` instruction. This instruction is looked up to determine the class of the allocated object. If the class has a finalizer method, the object escapes globally, because its finalizer might be called at an undefined time. Otherwise, the destination is marked as `ESCAPE_NONE`. The class descriptor loaded by `ICMD_ACONST` might be unresolved, i.e., it is unknown whether the class has a finalizer. Unresolved classes are generally treated conservatively, we thus assume that the class defines a finalizer and that the object escapes globally.

**ICMD_PUTSTATIC** If an object is stored into a static (global) variable, it becomes accessible from different threads and is thus marked as globally escaping.

**ICMD_GETSTATIC** If an object is loaded from a static variable, there is no information available about its allocation site, thus it must be marked as globally escaping.

**ICMD_PUTFIELD (s1.f = s2)** If $s2$ is assigned to an instance field of $s1$, it inherits the escape state of $s1$. In particular, if $s1$ is reachable from a different thread, $s2$ will become reachable as well. Further, $s2$ is added to the *dependency list* of $s1$. This is necessary, in order to update the escape state of $s2$, if at a later point $s1$ is found to escape. If $s1$ contains a method argument, $s2$ always escapes globally, because it will get accessible from the caller method and thus escapes the method.

**ICMD_GETFIELD (dst = s1.f)** If $s1$ contains a method argument, $dst$ is marked as being a globally escaping object, as there is nothing known about its allocation site. Otherwise $dst$ is initially marked as not escaping and the instruction is added to a list of getfield instructions for later processing.

**ICMD_AASTORE** Is handled in analogy to `ICMD_PUTFIELD`.

**ICMD_AALOAD** Is handled in analogy to `ICMD_GETFIELD`.

**ICMD_IF_ACMP...** If an object reference is compared against a different object reference, the object must not have been eliminated and must exist at least on the stack. The compared objects are thus marked as escaping the method.

**ICMD_IF...NULL, ICMD_CHECKNULL** If an object reference is compared against the `null` constant, the same applies as for `ICMD_IF_ACMP...`. Although not done in CACAO, some comparisons against `null` could be evaluated statically. In that case, adjustment of the escape state is not necessary.

**ICMD_CHECKCAST, ICMD_INSTANCEOF** A checked cast must not be eliminated, unless the compiler can statically determine whether it always succeeds. To perform the cast, the object must exist at least on the stack, it is thus marked as escaping the method.

**ICMD_INVOKESTATIC, ICMD_INVOKESPECIAL** For a method invocation that can be statically bound, i.e. if the called method is statically known, the escape summaries of interprocedural analysis are used to adjust the escape state of arguments and the return value. Interprocedural analysis also yields, which arguments are possibly returned from the callee. The summary information is then merged with the intraprocedural information. If interprocedural analysis data is not available for the callee, the arguments and the return value must conservatively be marked as globally escaping.

If the callee is unresolved and the caller has been already executed *often*, it is assumed that it won't be resolved at all, and the instruction is ignored. This assumption must be recorded with

the deoptimization framework, and the generated code must be invalidated once this assumption is found to be invalid.

If the callee is a native method, it can't be further analyzed. The arguments and the return value must conservatively be treated as globally escaping.

**ICMD_INVOKEVIRTUAL, ICMD_INVOKEINTERFACE** These instructions are processed similar to statically bound method calls, with the notable difference that all possible target methods have to be considered. Class hierarchy analysis is used to determine possible target methods. The escape state of the arguments and return value is again adjusted using interprocedural summary information. If the escape summary is missing for a single target method, it must conservatively be assumed that all arguments and the return value escape globally.

The set of target methods is again subject to change, if additional classes are loaded during the execution of the program. This optimization again has to be registered with the deoptimization framework and the code has to be invalidated accordingly.

**ICMD_ARETURN** The escape state of the source operand is adjusted to `ESCAPE_METHOD_RETURN`. The instruction is further added to a list of all return instructions for post-processing.

**ICMD_ATHROW, ICMD_GETEXCEPTION** Objects that are thrown as exception are not further tracked and are always marked as globally escaping.

**ICMD_COPY** A copy of a reference variable of the form $dst = src$ is treated by merging the ESS of the two variables.

**ICMD_PHI** $\phi$-functions correspond to a set of copy operations, for example, the $\phi$-function $dst = \phi(s1, s2)$ can be treated as two copy operations $dst = s1$ and $dst = s2$. The ESSs are merged accordingly.

### 3.5 Interprocedural escape analysis

The intraprocedural escape analysis computes escape states for all reference variables of a method. In particular incoming arguments of the method are included in this analysis. The information gathered during this analysis step can thus be used to construct summary information for a method that can be reused later in the caller context to adjust the escape state of actual arguments. The summary information contains:

- The escape state for parameters of reference type. This information is derived from the escape state of the local variable that is associated with the parameter within the method body.
- For every parameter of reference type, whether this parameter can be returned from the method. However, the escape state stored in the summary information is not `ESCAPE_METHOD_RETURN`, but `ESCAPE_METHOD` instead, to reflect the view of the caller context.
- The escape state of the method's return value. This is computed as the maximum escape state of the source operands of all return statements in the method. If this is `ESCAPE_METHOD_RETURN`, `ESCAPE_METHOD` is set in the summary information, again to reflect the view of the caller.

### 3.6 Native methods

Objects passed to native methods have to be marked as globally escaping. Empiric evidence has shown that many objects are passed to native methods, so applying this rule consequently leads to a large number of escaping objects – often unnecessarily. This is confirmed by tables 1 and 2, which show the dynamic ratio of method-local objects for selected SpecJVM98 and dacapo benchmarks. The results in the first column were obtained using the pessimistic assumption that objects passed to native methods escape globally. The results in the second column were obtained using the optimistic assumption that objects passed to native methods never escape globally. We thus use hardcoded method summaries for some important methods of the Java runtime library where the escape behavior of arguments is known beforehand. Some important examples are listed in the following:

| Benchmark | Pessimistic | Optimistic |
|---|---|---|
| _202_jess | 0.07% | 26.54% |
| _228_jack | 48.16% | 69.18% |

**Table 1.** Dynamic ratio of method-local objects for different assumptions about native methods (SpecJVM98).

| Benchmark | Pessimistic | Optimistic |
|---|---|---|
| eclipse | 3.37% | 3.62% |
| pmd | 0.21% | 11.87% |
| xalan | 2.47% | 3.73% |

**Table 2.** Dynamic ratio of method-local objects for different assumptions about native methods (dacapo).

- java.lang.System.identityHashCode
- java.lang.Object.getClass
- java.lang.Object.clone
- java.lang.System.arraycopy

### 3.7 Stack-allocation

Once escape analysis has identified method-local allocation sites, i.e., allocation sites with an escape state less than or equal to `ESCAPE_METHOD`, these sites are selected for stack allocation. Requirements for stack space are calculated right after escape analysis. Although stack space could be reused for stack objects with non-overlapping lifetimes, we do not perform such an optimization, because the number of stack allocation sites in a method tends to be rather low.

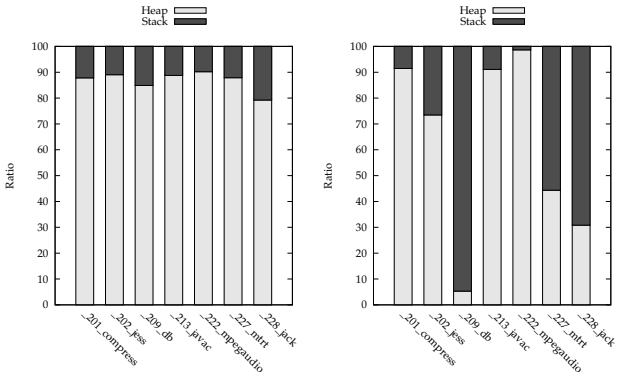
For every method-local allocation site a fixed amount of stack space is reserved that can be addressed relative to the methods stack pointer using a unique offset. The original `BUILTIN new` IR instructions of the allocation sites are rewritten into dedicated `STACK_NEW` IR instructions, which take the offset of the reserved stack space and a class descriptor as arguments.

The generated machine code for the `STACK_NEW` instruction is rather trivial: the address of the stack object is calculated by adding an offset to the stack pointer. Then the object header of the object is initialized by setting its virtual function table pointer and initializing the lockword to 0. Finally, all data fields of the object are zeroed as required by the JVM specification. An example of code generated for the x86 architecture for stack allocation of a `StringBuilder` object with a destination operand `%esi` is show in figure 4.

Although it is possible to dynamically grow the stack frame of an active method, similar to the `alloca` function in C, this mechanism introduces additional runtime overhead. For example, due to maintaining an additional `frame pointer` register, additional overhad for subroutine calls as well as method entry and exit. To avoid this overhead CACAO uses stack frames of fixed size, consequently stack space required for object allocation has to be known at compile time too. The number of stack allocated objects thus needs to be known statically, which poses a problem for non-escaping objects inside loops.

```
  ; store object pointer into %esi
  lea  0x80(%esp),%esi
  ; set VFTBL pointer
  movl $0x854186c,(%esi)
  ; set lockword to 0
  movl $0x0,0x4(%esi)
  ; zero data fields
  ; initialize loop counter
  mov  $0x4,%ecx
loop:
  ; zero 1 data word
  movl $0x0,0x14(%esi,%ecx,1)
  ; decrement counter
  sub  $0x4,%ecx
  ; loop
  jge  loop
```

**Figure 4.** Machine code generated for a `STACK_NEW` IR instruction.




(a) Static (b) Dynamic

**Figure 5.** Ratio of stack allocated objects for SPECjvm98 relative to static allocation sites and objects allocated at runtime.

Within loops objects can be allocated on the stack only if the space for the object can be reused on every loop iteration [9]. This condition is considered holding, if the allocated object is not live-in in the loop header, i.e. if its definition does not flow back into the loop along backward edges in the CFG.

If the allocation site is located inside a loop, the alias set of the allocated object is constructed, by following forward branches that do not leave the loop. The only way for the object to reach the loop header is via $\phi$-functions. The object can not be allocated on the stack, if an element of the alias set is an argument of a live $\phi$-function at the loop head. If the object in question is stored into a static field, instance field or an array inside the loop it is heap allocated, thus there is no need to track memory dependencies separately. The algorithms described in [1] are used to first detect all loops in *reducible control flow graphs*, then determine their nesting level, organize them in a loop hierarchy, and finally store the information in a way that supports efficient testing for loop membership and efficient traversal of loops and nested loops. The code generated by today's Java compilers almost always results in reducible flow graphs, our loop analysis is thus not restrained by this assumption. However, we verify whether the CFG in question is reducible, and disable all optimizations for the particular function if not.

### 3.8 Optimization framework

The optimizations implemented in this work target CACAOs second-level compiler and optimization framework which are currently in development. As this framework is not available at the time of this writing, the optimizations have been integrated into CACAO using an ad-hoc recompilation framework.

Initially all code gets compiled with the baseline compiler. Once enough classes are loaded and thus enough information for optimization is available, recompilation of all methods with single-static assignment form, escape analysis and stack allocation enabled is triggered interactively. Finally, after all methods were successfully recompiled, runtime statistics are gathered.

Because all methods get compiled at once, a recompilation order that is favorable for interprocedural escape analysis can be chosen, i.e., the call graph is traversed in a depth first order starting from leaf method. This ensures that methods get recompiled before call sites that invoke them.

Usually, escape analysis and stack allocation require the optimization framework to provide support for deoptimization. However, CACAOs optimization framework is still in development and does not yet provide all features required to revert stack allocation decisions. For example, during interprocedural analysis, it is assumed that a virtual call targets only those implementations of the method that are already loaded into the VM. The escaping behavior can however change, if an additional implementation is loaded during the execution of the program. The optimized code needs to be invalidated and recompiled in this case, and all objects that currently reside on the stack need to be promoted to the heap in order to ensure correctness.

## 4. Empirical Evaluation

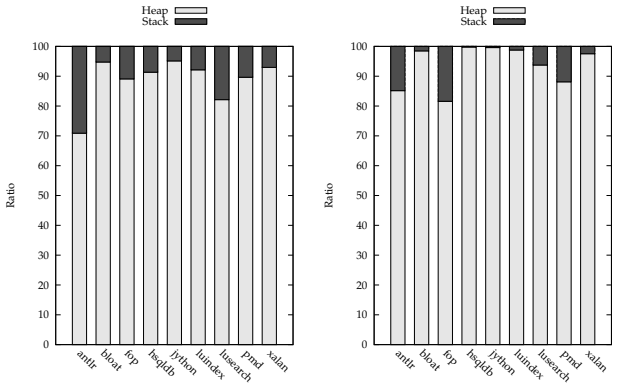
The SPECjvm98 and dacapo benchmark suites were executed on a Lenovo Thinkpad X60s system in order to evaluate the escape analysis algorithm and the corresponding stack allocation. The machine was equipped with an Intel Core Duo L2400 CPU at 1.66GHz and 512 MB of RAM. Debian GNU/Linux 2.6.24-1-686 served as an operating system with libc6 version 2.7-5. A development snapshot of CACAO was compiled using gcc version 4.1.3 20070718 (prerelease) (Debian 4.1.2-14), and configured to use a development snapshot of the GNU Classpath Java runtime library.

Each benchmark was executed at least 3 times. The first run was performed without optimizations in order to load enough classes before the actual measurements. Then, in the second run, when all classes have been loaded, recompilation with escape analysis and stack allocation was performed. Finally, during the third run, the optimized code was executed and final performance results collected.

### 4.1 Stack allocated objects

During the first run of the benchmarks, statistics on escaping objects and stack allocation were collected in order to evaluate the quality of the escape analysis. Figure 5 and Figure 6 show the ratio of objects that our analysis identified as not escaping their creating method, and thus can be allocated on the stack. Static numbers refer to the ratio of method-local allocation sites, while the dynamic numbers show the ratio of objects allocated on the stack relative to the overall number of objects allocated at runtime.

We have further compared the results of the escape analysis against the actual escaping behavior of the benchmarks observed at runtime. Note, that this is merely a theoretical bound of the results of the escape analysis, as objects might still escape along (exceptional) paths of the program that were not executed during the measurements. For this experiment CACAO was extended to track objects throughout their lifetime. Arrays are not considered here because they are never allocated on the stack. The results of this




(a) Static (b) Dynamic

**Figure 6.** Ratio of stack allocated objects for dacapo relative to static allocation sites and objects allocated at runtime.

| Benchmark | Inlining 1 | Inlining 2 | Stack allocated |
|---|---|---|---|
| _201_compress | 30.87 % | 44.84 % | 8.56 % |
| _202_jess | 64.27 % | 70.09 % | 26.54 % |
| _209_db | 94.73 % | 99.32 % | 94.68 % |
| _213_javac | 40.13 % | 41.45 % | 8.93 % |
| _222_mpegaudio | 16.87 % | 24.44 % | 1.33 % |
| _227_mtrt | 93.30 % | 94.18 % | 55.63 % |
| _228_jack | 64.88 % | 90.59 % | 69.18 % |

**Table 3.** Number of stack allocated objects compared to number of objects eligible for stack allocation (SpecJVM98).

| Benchmark | Inlining 1 | Inlining 2 | Stack allocated |
|---|---|---|---|
| antlr | 60.53 % | 78.58 % | 14.84 % |
| bloat | 61.05 % | 78.05 % | 1.50 % |
| fop | 59.77 % | 79.96 % | 18.42 % |
| hsqldb | 21.48 % | 31.80 % | 0.22 % |
| jython | 7.59 % | 60.09 % | 0.40 % |
| luindex | 16.23 % | 18.33 % | 1.22 % |
| lusearch | 36.34 % | 66.74 % | 6.25 % |
| pmd | 56.21 % | 74.79 % | 11.87 % |
| xalan | 32.21 % | 63.56 % | 2.46 % |

**Table 4.** Number of stack allocated objects compared to number of objects eligible for stack allocation (dacapo).

comparison are shown in Table 3 and Table 4 for the SPECjvm98 and dacapo benchmark suites respectively.

### 4.2 Execution times

The ultimate goal of stack allocation is to reduce the execution time by eliminating the cost of allocating and reclaiming memory on the heap at runtime. To evaluate the impact of our approach for this purpose, the execution scheme was extended to run every benchmark 5 times without escape analysis and 5 times with escape analysis. From these runs the one with the shortest execution time was taken into account. Note, that all runs were performed after recompilation with the ad-hoc optimization framework. The results are listed in Table 5 and Table 6

### 4.3 Discussion

Considering the data from Figure 5 and 6 we see that the escape analysis is quite successful in identifying allocation sites where

| Benchmark | With EA | Without EA | Speedup |
|---|---|---|---|
| _201_compress | 8.51 s | 8.50 s | -0.12 % |
| _202_jess | 46.01 s | 55.81 s | 21.29 % |
| _209_db | 28.52 s | 42.71 s | 49.75 % |
| _213_javac | 50.91 s | 53.56 s | 5.20 % |
| _222_mpegaudio | 13.02 s | 13.12 s | 0.77 % |
| _227_mtrt | 20.82 s | 35.23 s | 69.21 % |
| _228_jack | 43.69 s | 64.06 s | 64.62 % |

**Table 5.** Execution times with and without escape analysis (SPECjvm98).

| Benchmark | With EA | Without EA | Speedup |
|---|---|---|---|
| antlr | 34.16 s | 37.52 s | 9.84 % |
| bloat | 219.34 s | 216.95 s | -1.09 % |
| fop | 11.95 s | 12.64 s | 5.77 % |
| hsqldb | 2.77 s | 2.90 s | 4.69 % |
| jython | 274.30 s | 274.54 s | 0.09 % |
| luindex | 96.78 s | 96.80 s | 0.02 % |
| lusearch | 247.07 s | 261.70 s | 5.92 % |
| pmd | 178.63 s | 197.12 s | 10.35 % |
| xalan | 73.19 s | 73.24 s | 0.07 % |

**Table 6.** Execution times with and without escape analysis (dacapo).

objects do not escape statically. However, for our final goal these results are only secondary because not all of these allocation sites are contributing to the overall execution time equally. For example, we found that a large number of non-escaping allocation sites can be attributed to exceptional paths that build error messages using the `StringBuilder` class.

Especially for the SPECjvm98 benchmarks the good static results were directly reflected by the dynamic behavior. For `jess`, `raytrace`, `db`, `mtrt` the number of stack allocated objects is extremely high at runtime, which is coupled to a single class. These extreme allocation sites were already observed in previous work [9]: In `mtrt`, temporary `Vector` objects are allocated in a loop, in `db`, two temporary `Enumeration` objects are used to compare a pair of database records. However, the high speedups of more than 69% are not only due to stack allocation. The current memory management implementation in CACAO performs rather poorly: CACAO uses a slow conservative garbage collector and the `new` instruction is not inlined but rather implemented in a C function which in turn is wrapped into a builtin stub. Another explanation for the extreme speedup is the simple nature of the SPECjvm98 benchmarks.

The behavior of the dacapo benchmarks is more moderate, which consequently results in much lower gains of up to 10.35% in execution time – `bloat` even experiences a slight slowdown. The more complex dacapo benchmarks show the limits of our current whole-program interprocedural analysis. The large program call graphs lead to unfavorable traversals, such that callees are not always visited before the respective call sites. This in turn leads to very conservative results of the escape analysis. Investigation reveals that for the dacapo benchmarks with very low numbers of stack allocated objects even `StringBuffers` are heap allocated exactly because of this problem.

Many benchmarks profit from stack allocation of `StringBuffer` and `StringBuilder` objects that are used to implement Java string concatenation. They hardly escape the creating method and can often be stack allocated. Another family of popular stack objects consist of `Enumeration` and `Iterator` objects used in conjunction with container classes. They usually get allocated when iterating

over a container class and never escape the creating method. These objects significantly contribute to the number of stack objects in the benchmarks `db` (almost all) and `jack`. Finally, extremely short-lived objects are chosen for stack allocation, this is particularly true for the dacapo `pmd` and `bloat` benchmarks.

## 5. Related Work

An early study by Charles McDowell [13] already tried to investigate the potential of stack allocation using code instrumentation. However, only four benchmarks were considered, consequently, it is hard to generalize their measurements to a wider range of programs. In their study up to 56% of the objects were eligible for stack allocation, in the general case however only between 5 and 15%.

Escape analysis algorithms can be categorized [8] into two groups: A *Steensgaard* analysis merges both sides of an assignment, computing the same solution for each side. An *Anderson* analysis in contrast passes a value from the right-hand side of an assignment to the left-hand side, offering greater precision at the cost of a higher computational effort. *Flow-sensitive* analysis take the order of statements in a program into account, while *flow-insensitive* analysis merges results along different control flow paths into a single summary. Similarly, *context-sensitive* analysis, in contrast to *context-insensitive*, distinguishes between the calling context, i.e., the different call sites of a given function or method within a program.

A prominent algorithm for escape analysis in Java, which is adopted by the *Java HotSpot™ server compiler*, was proposed by Choi et. al [5]. The algorithm is context-sensitive and allows for both a flow-sensitive and a flow-insensitive variant. They propose a novel program representation, the *connection graph* (CG) that is used to represent the escape state of objects allocated within a method. Later, the nodes in the CG are collapsed into summary information that can be used in interprocedural analysis at call sites. The analysis distinguishes between the following escape states: *NoEscape*, *GlobalEscape* – the object escapes the thread, *Local Escape* – the object escapes only the currently considered method, and *ArgEscape* – the object escapes the method via arguments. Nodes that are only reachable from *NoEscape* nodes are eligible for stack allocation.

Gay and Steensgaard have implemented a whole program interprocedural escape analysis for Java programs [7] in an SSA based IR. An object is considered to escape if it is returned from a method, thrown as an exception or assigned to a class or instance field. These rules are encoded as constraints on a type system, which can be solved in linear time and space in the number of constraints, e.g., using Rapid Type Analysis. *Fresh methods* return a newly allocated object, similarly *fresh variables* refer to newly allocated objects that are either created by a *new* or returned from a fresh method. For each *fresh variable* it must be determined whether that assigned value may escape. At a method invocation site all possible target methods are considered and constraints are added. If a parameter can be returned from the callee, it is handled as an assignment of the parameter to the left-hand side of the variable.

A different analysis technique based on abstract interpretation was developed by Blanchet [2]. He directly operates on Java bytecode instructions and thus models the Java runtime stack for the abstract representation. Information is propagated both forwards and backwards. Forward propagation occurs when instructions are analyzed following the normal flow of control – as in the approach of Choi et al. [5]. Backwards propagation is performed by interpretively executing the bytecode instructions along the reverse of control flow paths. The combination of forwards and backwards analysis passes enables much more precise results to be obtained. Blanchet uses a quite different domain of values to represent the escape state of objects. He represents each class type by an integer, which is the *context* for what may escape from an instance of that class. His abstract values are equations (or context transformers) which map from the contexts of the arguments and result of a method to the escape contexts of concrete values. Instead of manipulating a collection of graphs, as Choi et al. do, Blanchet manipulates sets of equations.

There has not been a qualitative comparison between the two approaches of Choi et al. and Blanchet. Blanchet states that Choi's approach is more time consuming, he also claims bigger speedups for his set of sample programs. However, Blanchet performs extensive inlining of small methods. As shown in Section 2 inlining already up to a level of three can drastically improve the results of escape analysis.

In [9, 10, 11], Kotzmann presents an escape analysis algorithm tailored for a virtual machine performing adaptive optimizations. The results of the analysis are used to peform synchronization removal on thread-local objects, stack allocation of thread-local objects and elimination of method-local objects. Intraprocedural and interprocedural analysis use the same principles as the algorithm presented in this work. In contrast to our work, which is a whole-program analysis, his approach proceeds incrementally: the compiler gathers information method by method and uses the collected data to improve the machine code. The analysis is supported by a mature optimization framework, i.e., only hot methods are recompiled with escape analysis. Summary escape information for methods that were not yet compiled is approximated through a fast and conservative abstract interpretation of the Java bytecode. A complete deoptimization framework is able to rematerialize eliminated objects and undo synchronization elimination. Inlining is used to improve the impact of escape analysis.

In [6] a memory allocator supporting *thread-local heaps* was developed. The aim is to partition the global heap among threads. Each thread can then allocate memory in its own partition without synchronization. Also garbage collection can be applied to each partition separately. An analysis is required that assignes objects to the proper partition, i.e., thread-local objects are alloacted within thread-local heaps, while all other objects are alloacted on the shared global heap. Static escape analysis is able to perform such a partitioning, however, the authors prefer runtime monitoring of heap allocated objects over static analysis. All objects carry a *global* bit that indicates, if the object belongs to the global heap or not. Most objects are created on the thread-local heap and thus their *global* bit is initially cleared. The bits are automatically updated on stores into static or member fields. If the store causes a local object to become reachable from the global heap or a static field, the *global* bit is set accordingly. Similarly, all objects reachable from that particular object are migrated into the global heap.

## 6. Conclusion

Stack allocation of objects is very effective in reducing the cost of object allocation and garbage collection. We have presented a detailed evaluation of the potential of stack allocation. We found that most applications show high potential for stack allocation in their dynamic behavior. However, escape analysis is not always able to prove this behavior statically, because (1) interprocedural analysis is too costly on a large scale and thus requires conservative assumptions, (2) the observed runtime behavior may not always execute all possible paths, e.g., exception handler, that need to be considered during static analysis. Our study also shows that inlining, even for a moderate call depth of three, has a very high potential to improve the results of static escape analysis.

Finally, we have implemented a fast and lightweight static escape analysis within the CACAO JVM that conservatively determines which objects can safely be allocated on the stack. Up to

95% of the potentially stack allocatable objects are recognized by the conservative analysis leading to speedups of up to 69% for the SpecJVM98 benchmark suite. Even for the more complex dacapo benchmarks an improvement of up to 10% can be observed.

## References


[1] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[2] Bruno Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 20–34, Denver, November 1999. ACM.

[3] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *SIGSOFT Softw. Eng. Notes*, 24(5):21–31, 1999.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–19, Denver, November 1999. ACM Press.

[6] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. *SIGPLAN Not.*, 38(2 supplement):76–87, 2003.

[7] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93, London, UK, 2000. Springer-Verlag.

[8] Richard Jones and Andy C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Thomas Kotzmann. *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. PhD thesis, Johannes Kepler University Linz, 2005.

[10] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments*, pages 111–120, New York, NY, USA, 2005. ACM.

[11] Thomas Kotzmann and Hanspeter Mössenböck. Run-time support for optimizations based on escape analysis. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, October 1998. IFIP,ACM,IEEE, North-Holland.

[13] Charles Edward McDowell. Reducing garbage in Java. *SIGPLAN Notices*, 33(9):84–86, 1998.

[14] Hanspeter Mössenböck. Adding static single assignment form and a graph coloring register allocator to the Java HotSpot™ client compiler. Technical report, Johannes Kepler University Linz, 2000.

[15] Edwin Steiner, Andreas Krall, and Christian Thalinger. Adaptive inlining and on-stack replacement in the CACAO virtual machine. In *International Conference on Principles and Practice of Programming in Java*, pages 221–226, Monte de Caparica/Lisbon, Portugal, September 2007.