FAKULTÄT FÜR !NFORMATIK

Escape analysis and stack allocation of Java objects in the CACAO VM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Peter Molnár
Matrikelnummer 0226327

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Ao.Univ.Prof Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 09.02.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Eidesstattliche Erklärung

Peter Molnár
SK-Tranvská 188, 90027 Bernolákovo

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 09.02.2009

_____

Abstract

Escape analysis is a static code analysis that determines, whether the lifetime of objects exceeds the lifetime of their creation site. The additional knowledge gained through escape analysis can be used to optimize memory management and synchronization in a virtual machine.

In the context of this thesis, escape analysis has been implemented for the CACAO virtual machine. The analysis proceeds in two stages: intraprocedural analysis computes escape information for allocation sites within a single method and call-context agnostic summary information, that is used for interprocedural analysis. Escape information is used to allocate a subset of thread-local Java objects on the call stack. The implementation and the modifications of the virtual machine are described in detail.

Finally, the implementation is evaluated by benchmarking using the Spec JVM98 and the Dacapo benchmark suites. In selected Spec benchmarks, 50% to 90% of all objects get allocated on the call stack leading to an execution time reduction of up to 40%.

Kurzfassung

Escapeanalyse ist eine statische Analyse, welche feststellt, ob Objekte länger leben als deren Erzeuger. Die durch Escapeanalyse zusätzlich gewonnenen Informationen können zur Optimierung der Speicherverwaltung und Synchronisation in einer virtuellen Maschine genutzt werden.

Im Kontext dieser Arbeit wurde Escapeanalyse für die freie virtuelle Maschine CACAO implementiert. Die Analyse arbeitet in zwei Schritten. Die intraprozedurale Analyse berechnet Escapeinformation für einzelne Allokationspunkte in einer Methode und Information für die gesamte Methode, welche im Rahmen von einer interprozeduralen Analyse in verschiedenen Aufrufkontexten wiederverwendet werden kann. Die berechnete Escapeinformation wird verwendet um einen Teil von threadlokalen Java Objekten auf dem Stack zu allozieren. Die Implementierung und die notwendigen Änderungen der virtuellen Maschine werden im Detail beschrieben.

Anschließend wird die Implementierung durch Benchmarken evaluiert. In einigen Spec-Benchmarks werden 50% bis 90% von Java Objekten auf dem Stack alloziert, was zu einer Verbesserung der Laufzeit von bis zu 40% führt.

Contents

Chapter 1

Introduction

1.1 The Java programming language

Java is a programming language developed by SUN microsystems [11]. The language has several design goals: object-orientation, simplicity, security, high performance and portability as expressed in its tagline *write once, run everywhere!*.

To achieve the portability goal, Java programs are not directly compiled to machine code, but to an architecture-neutral intermediate representation called *byte-code* targeting the *Java virtual Machine* (JVM) [22].

The interpreted nature of the first virtual machines caused Java programs to run significantly slower then their equivalents written in languages targeting machine code. One option to address this problem would have been to translate the *byte-code* representation into machine code *ahead-of-time*, like in conventional C or C++ compilers, but the dynamic nature of the Java language poses several difficulties: for example the JVM specification mandates the possibility to generate or load additional program code at run-time, what makes complete *ahead-of-time* compilation challengig.

So translation to machine code has been moved to run-time. Java methods were translated to machine code immediately prior to execution by the virtual machine, leading to the notion of a *just-in-time compiler*. Although conventional compiler construction was a well researched area at that time, *just-in-time* compilers brought a set of new challenges, the main being extremely low compile times, as these account to the overall run-time of the program.

It was soon discovered, that placing the compiler into the *run-time* environment opened a high potential for new optimization that are impossible to realize *ahead-of-time*. First, it is feasible to compile only methods that get executed frequently enough to amortize the compilation time, the so called *hot methods*. This leads to a *mixed-mode* where code is both interpreted and compiled. Second, the run-time environment provides valuable profiling

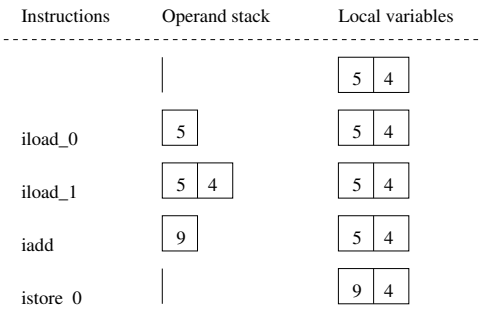


Figure 1.1: Java bytecode example

data and information that can be continuously used to improve the quality of the generated machine code.

1.2 Java bytecode

The binary, portable format to store Java byte code is the *.class* file. It contains a *constant pool* and executable code for all methods of a single Java class. The executable code targets a stack-oriented instruction set: instead of explicit operands, bytecode instructions operate on an *operand stack*. The operand stack is untyped and consists of 32-bit slots. A method has access to a fixed-size array of 32-bit local variables. Instructions are provided for arithmetic operations, local variable access, type conversions, object creation and manipulation, operand stack management, control transfer, method invocation and return, exception throwing and synchronization via monitors.

To illustrate Java byte code, a simple example is shown in figure 1.1: two local variables contain the values 5 and 4 respectively. These values are loaded on the operand stack, then added and stored into local variable 0.

A detailed specification of Java byte code can by found in [22].

1.3 The CACAO virtual machine

CACAO [18] is a virtual machine executing Java bytecode licenced under the terms of the GNU GPL. Its development started as a research project at the Vienna University of Technology. CACAO follows a compile-only approach: all byte code gets compiled *just-in-time*. CACAO initially targeted DEC's Alpha architecture, but was soon ported to MIPS and PowerPC and later to the IA32, x86_64, ARM, SUN Sparc, PowerPC64, IBM S390 and Coldfire architectures.

The CACAO *just-in-time compiler* is triggered by the first execution of a Java method. In the long term CACAO will provide two compilers: the *baseline compiler* is designed for very short compile times and consists of a parse pass that translates the bytecode into an IR, a stack analysis pass, optional code verification, fast register allocation and machine code generation. An optimizing compiler and an optimization framework are currently in development. The goal is to recompile hot methods with aggressive optimizations, which include inlining and linear scan register allocation.

A very simplified view of the core data structures of the JIT compiler for the purpose of this thesis can be seen in figure 1.2: the state of the JIT compiler for one method is represented by a `jitdata` structure. It contains a pointer to a `methodinfo` structure, which is the VMs run-time representation of a Java method. This in turn contains a pointer to a `classinfo` structure - the run-time representation of a Java class. The intermediate representation (IR) consists of a linked list of basic blocks, each pointing to an array of IR instructions. The IR instructions are quadruple code with an opcode and explicit typed operands - numbered IR variables. Source operands are named `s1`, `s2`, `s3`, the destination operand `dst`. Each variable number is an index into the `var` array of variable descriptors of the JIT state. Symbolic names for IR opcodes are by convention prefixed with `ICMD_`. A `codegendata` structure holds the state of the machine code generation. For more details on the CACAO IR refer to [26].

The JVM specification does not mandate any particular object layout. In CACAO, the memory representation of a Java object as seen in figure 1.3 consists of an *object header*, represented by a `java_object_t` followed by the object's instance fields, meeting the target architectures alignment requirements. The object header itself consists of 3 machine words: a *virtual function table pointer* pointing to the object's virtual function table and run-time type information, a *lock word* for the purpose of synchronization and a word for storage of various flags.

1.4 Static single assignment form

Static single assignment form (SSA) [6] is an intermediate program representation where every variable is assigned a value exactly once. Such a representation is usually linear to the size of the original program and thus efficiently represents the relation between uses and definitions of a variable. If makes data flow explicit and thus simplifies data flow analyses. SSA can be used in combination with flow-insensitive analyzes to obtain results comparable to the more expensive flow-sensitive analyses.

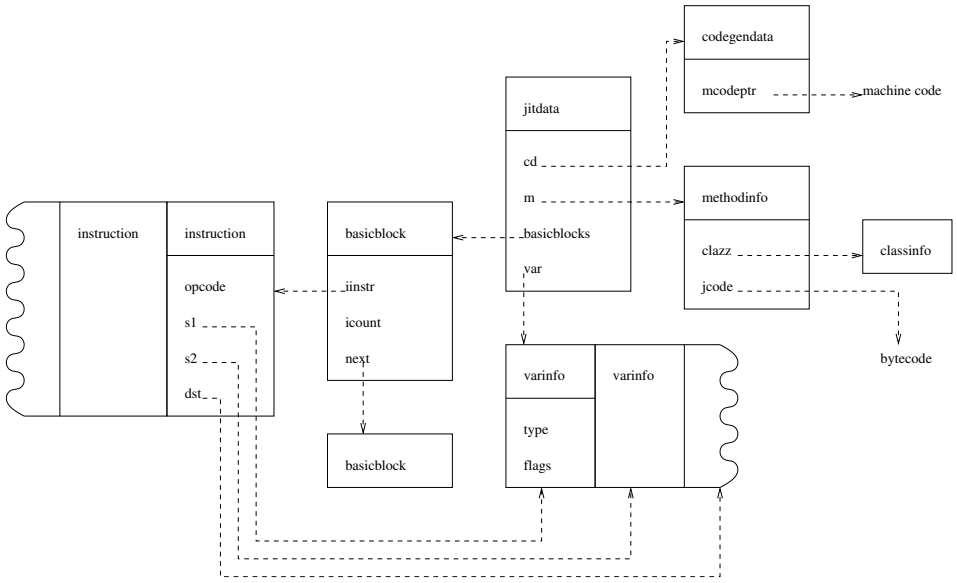


Figure 1.2: CACAO JIT compiler data structures

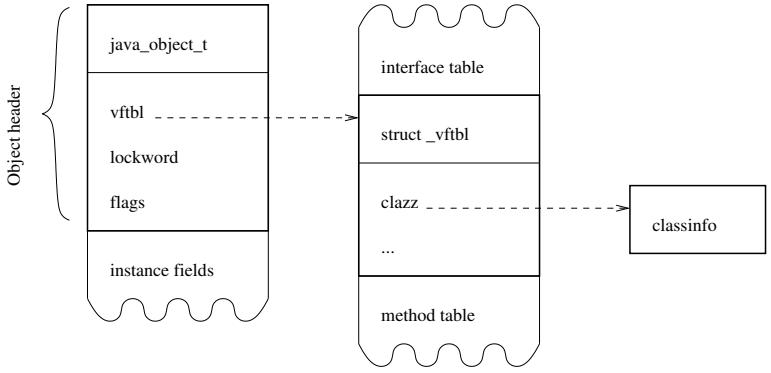


Figure 1.3: CACAO object layout

1.5 Escape analysis

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.

1.6 Goals

The goal of this thesis is to implement escape analysis for the CACAO virtual machine. Next, optimization possibilities given by escape analysis are to be considered and one optimization is chosen for implementation. Finally the selected optimization is to be evaluated.

1.7 Overview

The reminder of this work is structured as follows: Chapter 2 introduces the opportunities given by escape analysis and motivates the possible optimizations. In chapter 3, real-world Java programs are studied in order to understand the escape behavior of typical programs. In chapter 4 the implementation of escape analysis and the chosen optimization - stack allocation of objects is explained in detail. Finally, chapter 5 gives an empiric evaluation of the implementation.

Chapter 2

Optimization opportunities

Escape analysis is no optimization on its own, but only provides information about the scope in which objects are accessed [16]. The additional knowledge gained through escape analysis can be further used to optimize services provided by some subsystems of the virtual machine and to generate better code in the JIT compiler. In this chapter, first the subsystems of interest for escape analysis and their run-time costs are introduced and described, then the potential for optimization using escape analysis is discussed.

2.1 Memory management

Languages like C [15] or C++ [27] offer the programmer the possibility of allocating memory for objects in a dynamically sized region called heap. To allocate a block of memory, the programmer uses a `new()` primitive and passes it either the size of the memory requested or the type of the object to be allocated, which implies the size. The call returns a typed or untyped pointer to the block of memory. The memory is live until the programmer deallocates it explicitly using a `delete()` primitive. This feature, although necessary and useful, often introduced some families of programming errors:

Memory leak The programmer forgets to `delete()` a block of memory previously allocated with `new()`. Memory leaks in long running programs or in loops cause the maximum available memory to the program to be exhausted which leads to an abortion of the program.

Double free By mistake, the programmer calls `delete()` more than once on a block of memory he allocated previously. Allocated blocks usually include some metadata, that is used later when they are recycled. If the block of memory has already been recycled and reused, invalid

metadata will be read by `delete()` and its usage will lead to memory corruption.

Dangling pointer The programmer `delete()`s a block of memory, but keeps a pointer to it. By using the pointer later, recycled and potentially reused memory will be used which can lead to memory corruption.

To decrease the likehood of these programming errors, some memory management strategies have been developed for these languages:

Avoiding dynamic memory Some programming languages offer alternate memory management schemes. In C++ objects can be allocated on the stack and are then automatically freed. Classes can define a destructor with well defined semantics. When using an appropriate programming style, it is possible to greatly reduce usage of heap objects or to isolate it into few well engineered classes. This strategy addresses memory leaks and double frees, but dangling pointers are still possible, as the programmer can create pointers to stack objects which can outlive the respective objects.

Region based memory management Instead of performing manual memory management on the granularity of objects, a coarser granularity is chosen. If the program performs a well defined task, during which it dynamically allocates objects, and it is known that none of these objects will outlive the task, a region is associated with this task. All objects allocated are implicitly allocated in the region. They are freed all at once as soon as the given task is finished. Such a strategy decreases the likehood of memory leaks and dangling pointers. It improves performance as well, as less time is spent in freeing objects.

These strategies however are not satisfying, as they only decrease the likehood of the errors above, but they do not eliminate them completely.

To address these issues, some programming languages, including Java, introduced fully automatic memory management schemes, where the programmer only allocates objects, and freeing them is the responsibility of the runtime system. Widely used schemes include:

Reference counting There is a counter associated with every allocated object. Whenever a reference to the object is created, the counter is incremented. Whenever a reference to the object is disposed, the counter is decremented. If after decrementing the counter becomes zero, the object can be freed, as there are no more references to it. There are two fundamental problems with reference counting:

1. If copying references is a frequent operation, then maintaining the reference counters can require much resources.
2. If there is a chain of objects referencing each other in a cyclic manner, they can never be freed using the basic algorithm, as their reference count will never become zero.

Garbage collection Objects are allocated on a heap, until the heap gets exhausted, or some run-time system defined condition occurs. The run-time system then starts identifying the set of reachable objects on the heap, in order to reclaim memory. It does this by starting with the set of so called root pointers: pointers to heap objects located at well defined locations like CPU registers, the call stack and global variables. By recursively following references from these objects, the set of all reachable objects is determined. The space of the objects left is reclaimed for reuse. Garbage collection avoids double `delete()`s and dangling pointers: as long as there exists a (reachable) pointer to an object, the object will conservatively be kept. Contrary to popular belief, memory leaks are possible when using garbage collection [20]. If the programmer by mistake accumulates references to objects in the root set, the respective objects won't get reclaimed because of the conservative nature of garbage collection.

Most Java virtual machines use garbage collection for memory management. A notable exception are virtual machines adhering to the real time Java specification [10]. In real-time applications, garbage collection is inadequate because of its temporal indeterminism. Instead region based memory management is used. At the time of this writing, CACAO uses the conservative Boehm-Demers-Weiser garbage collector, but a new precise garbage collector is being implemented.

2.2 Costs of garbage collection

In the Java language, an object allocation is represented by a single statement of the form `T o = new T();`. A Java compiler translates such a statement into 2 bytecode instructions. First a `new T` instruction, which is responsible for allocation and initialization of a memory block of adequate size. Second, an `invokespecial T.<init>` instruction is used to invoke the statically bound constructor of the class `T`.

In CACAO, a `new` bytecode instruction is transformed to a `BUILTIN` intermediate instruction, which in turn is translated into the invocation of a builtin function `builtin_new` provided by the virtual machine written in C language. This function performs several steps:

1. If not already done, the class is initialized and linked.

2. `heap_alloc` is used to get an initialized block of heap memory:
   (a) It calls the allocation routine `GC_MALLOC` provided by the garbage collector to get a block of heap memory.
   (b) It overwrites the memory with zeros to satisfy the requirement of default initialization of instance variables to zero.
3. The virtual function table pointer of the object is initialized to point to the class' virtual function table.
4. The lock word in the object header is initialized to 0, meaning that the object is unlocked.

The garbage collector provides it's own 2-level memory allocator [12]. Free lists for several large objects sizes are maintained and an approximate best-fit algorithm is used to find a free block of appropriate size. Small objects are allocated in chunks sized in the order of magnitude of the page size: the chunk is split and then used exclusively for allocation of same-sized objects. Free lists are maintained for different object sizes. In a multi-threaded environment, the data structures of the allocator must be accessed in a mutually exclusive way.

In the current implementation, the costs of object allocation are mostly determined by:

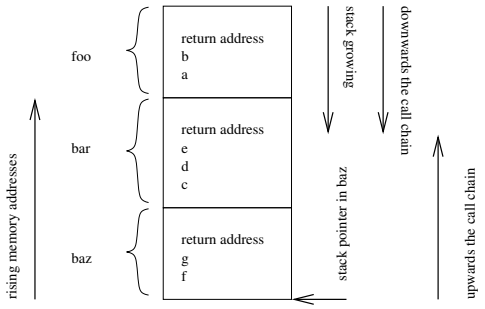
1. The call overhead into a builtin C function.
2. Mutual exclusion on the allocators data structures.
3. The call overhead into the garbage collector's allocation routine.
4. Searching for free space in free lists.
5. Overwriting the allocated memory with zeros.

2.3 Thread-local allocation

Long-running massively multithreaded server applications tend to require very large heaps. Most garbage collection techniques require a *stop-the-world* phase: all application threads must be suspended and their state - the stack and CPU registers - is captured and scanned for root pointers. Unless a conservative garbage collector is used, the machine code is annotated with stack maps, which for a particular program position describe the locations holding pointers to heap objects. Usually, stack maps are not available for every program point, but only for discrete *GC points*. So during the *stop-the-world* phase all application threads first must reach a *GC point* before collection can be started. The costs of this synchronization prior to

```
void foo() {
    int a, b;
    bar(a, b);
}
void bar(int a1, int a2) {
    int c, d, e;
    baz(c, d);
}
void baz(int a1, int a2) {
    int f, g;
}
```

(a) Source code



(b) Call stack

Figure 2.1: Call stack

garbage collection are proportional to the number of threads and can take considerable time: in the VolanoMark client benchmark up to 23% of the total garbage collection time is spent in this synchronization [14].

In presence of escape information, objects, that are referenced exclusively by their creating thread can be identified. These objects can be allocated in a *thread specific region* of the heap. If a thread runs out of memory, its *heap region* can be collected independently of other threads. This means, that no global rendezvous of all application threads, no synchronization and no locking are required.

2.4 Stack allocation

The call stack is a data structure which holds local data of the active methods in a program. Figure 2.1 shows an example call stack at the point where method `baz()` was called. There exists one call stack for every thread of execution. When a method is invoked, a new stack frame for this method is pushed onto the call stack, when it returns, the stack frame is poped. In CACAO, the size of a stack frame is of a fixed size determined for every method at compile time. Every architecture that CACAO supports provides a *stack pointer* register that always points to the start of the current stackframe. Local variables are accessed relative to the stack pointer.

In the current CACAO VM implementation, only primitive types can be allocated in the stack frame: integral types and reference types. Objects and arrays are always allocated on the heap. The stack frame of the caller always outlives the stack frame of its callees. So if escape analysis determines, that an object does not escape the creating thread and it does not

escape the creating method towards the caller, the object can be safely allocated in the stack frame of the creating method. Stack allocated objects bring some benefits for the generated code and for the run-time system:

1. A stack frame is very likely created for every method. The costs of stack frame creation are that of one instruction, which decrements the stack pointer by a fixed offset. Memory for a stack object is reserved at compile time and affects only the size of the stack frame. The allocation of memory for a stack object therefore comes at no cost.

2. Disposal of a stack object comes at no cost. It is disposed implicitly when the stack frame of the creating method is poped. The costs of poping a stack frame are independent of the size of a stack frame too and correspond to the costs of an instruction that increments the stack pointer.

3. For heap objects, a word of storage has to be reserved to hold the address of the object. In the creating method, the address of a stack object is always known to be the address of the stack pointer plus a fixed offset.

4. Fields of stack objects can be accessed relative to the stack pointer. They are located at a fixed offset from the stack pointer.

5. Stack allocation is much more cache friendly than heap allocation. The top of the stack in contrast to the top of the heap will usually be cached.

6. Heap allocation has has an especially bad cache behavior. Whenever an heap object gets allocated, the memory will initially be filled with random values and likely not be cached. The memory must be brought into the cache in order to be zero filled causing even more memory activity.

Stack allocated objects bring an indirect benefit for garbage collection. The run time of garbage collection is a function of the heap size. The less objects are allocated on the heap, the less time is spent in garbage collection. Massively multithreaded applications also benefit from stack objects, because of the *stop-the-world* phase occurs less frequently.

2.5 Object elimination

If escape analysis determines that an object does not escape the allocating method at all, further optimization is possible. In unoptimized code, this can apply only to arrays. Because of the two-step object allocation in the

JVM - allocation of a block of memory followed by a call into the class' constructor, every object escapes at least into its constructor. However, if the call of the class' constructor can be inlined into the creating method, the object stays method local. In this case the object's allocation can be eliminated completely and its fields can be replaced by properly initialized scalar variables. This brings all benefits of stack objects: i.e. zero cost allocation and disposal, improved garbage collection times. On architectures featuring large register files, the object's fields can be allocated to registers. Additionally, if an object gets completely eliminated, because of the eliminated memory dependence, more traditional optimizations are possible.

2.6 Synchronization

Multithreaded programs consist of multiple threads of execution sharing resources. By concurrent use of the shared resources without proper synchronization, the resources might get corrupted. To prevent an inconsistent program state, the programing environment usually provides some kind of synchronization primitives. The Java programming language was designed from the ground up to support multithreaded programming, so synchronization primitives are part of the core language.

The Java programing language supports monitors [13] to synchronize two threads accessing a shared resource via the `synchronized` statement. A monitor can be entered on every object, in one of two ways:

1. Putting a portion of source code into a `synchronized` block causes the executing thread to enter a monitor during execution of the code block. A Java compiler translates the block into a pair of `monitorenter` and `monitorexit` bytecode instructions on the block entry and exit point respectively.

2. Calling a method that is declared `synchronized` causes the monitor of the receiver of the call to be entered during the execution of the method by the executing thread. On bytecode level, a synchronized method has the `ACC_SYNCHRONIZED` flag set. When entering or leaving such method, the virtual machine must carry out the appropriate implicit monitor operations.

Synchronization is pervasive in Java system libraries, to ensure a correct behavior if objects are shared between threads. As these classes are used quite frequently, an efficient monitor implementation is crucial for good overall performance.

2.7 Costs of synchronization

To implement monitors, the virtual machine associates an intrinsic lock with every object. Entering and leaving a monitor corresponds to acquiring and releasing that lock respectively.

In CACAO, the VM reserves a machine word in the header of every Java object, called the *lock-word* for implementation of this intrinsic lock.

Empiric evidence shows, that if a thread tries to acquire a lock on an object, it will often find the object unlocked. This scenario is called *uncontested locking*. The locking algorithm implemented in CACAO is a variation of the *tasuki lock* [24]. It is very fast at uncontested locking, requiring only one atomic *compare and swap* instruction on the lock word. In the case of contested locking, the algorithm falls back to a slow builtin function of the VM. Inlining of the fast path into JIT code is currently being implemented to make the fast path even faster.

2.8 Synchronization elimination

If escape analysis is available, objects that are local to their creating thread can be identified. Synchronization on such objects can be safely eliminated, as it can be proven that no other thread will ever access these objects.

Even if uncontested locking of objects is fast, it still requires an expensive atomic instruction. Especially in the advent of multi-core machines, atomic instructions become more expensive. The benefit of synchronization elimination lies in the elimination of these expensive atomic primitives.

Chapter 3

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 through the ratio of the number of objects it identifies as non-escaping to the number of total objects allocated in the program. This ratio can be calculated from two different sets of numbers. Static numbers examine the number of sites allocating objects while dynamic numbers examine the number of objects allocated at run-time by the program. If the aim of an escape analysis algorithm is to gain a speedup in a compiled program, then static numbers have rather limited use for an evaluation: eliminating a lot of allocation or synchronization sites does not result in a significant speedup, if these sites get not executed frequently enough, or even never at all. As every eliminated allocation brings a defined speedup, the dynamic numbers can be used to express the gain in execution time.

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 significantly the gap between the results of the analysis and the real escape behavior. 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.

3.1 The algorithm

A simple code instrumentation is used to trace and determine reachability of objects and to determine whether they escape and where they escape to. Objects are grouped into non-overlapping *regions* based on reachability in-

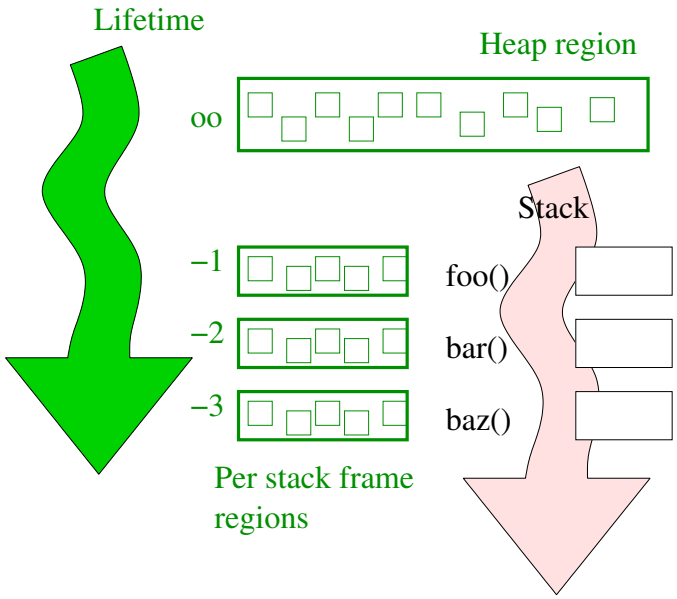


Figure 3.1: Escape behaviour algorithm

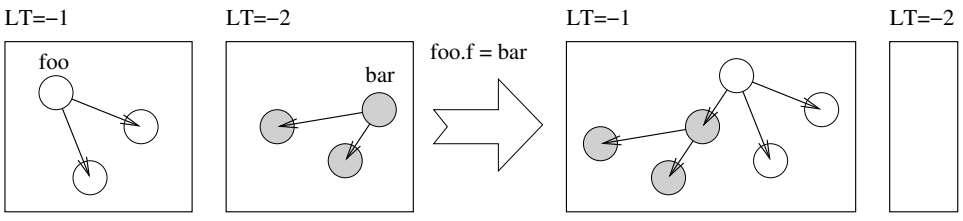


Figure 3.2: An object being moved into a region with a longer lifetime

formation. A region for every stack frame and an additional global region (also called heap region) are created as seen in figure 3.1. With every region, a *lifetime* is associated: the heap region has an infinite lifetime. Regions associated with stack frames have a shorter lifetime the deeper they are in the call chain, so their lifetime is defined as the negated stack depth. When a new object is allocated, it is put into the region of the current stack frame. When an object becomes reachable from an object contained in a different region, it might be moved into that region, as seen in figure 3.2. Whether the move will be performed depends on the lifetime of the destination region: an object is always moved into a region with a longer lifetime. The move operation is performed recursively: all objects referenced via fields or array elements of the moved object are moved too.

At the end, the heap region contains 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 IR constructs are instrumented. The instrumentation is performed by generating a call into C functions implemented in the VM.

Method entry: a region is allocated and pushed on the region stack. As native methods can't be further analyzed, upon entry into a native method, all arguments are moved into the global region.

Method exit: if the method returns an object reference, the return value is first moved into region of the caller. The return value of a native method must be handled separately: it is always moved into the global region because references to the returned object might have been kept at unknown locations. Then all objects left in the region associated with the stack frame are identified as not escaping the current method and not escaping the current thread. The region is then destroyed.

Exception throwing: if an object is thrown as exception, it is conservatively moved into the heap region. Exception handling is considered to be an exceptional situation and the thrown object is not optimized. If the exception leaves the method, the same processing as upon method exit is done.

Object allocation: if an object is newly allocated, it is moved into the region associated with the activation record of the allocating method. For the purpose of statistics, it is further tagged with the current stack depth and a pointer to the currently active method's descriptor. Precaution is necessary for objects that have finalizers: they are moved into the heap region, as their finalizers will be called asynchronously at an undefined time.

```
for (int i = 0; i < 1000000; ++i) {
    T o = new T();
    if (i == fortyTwo()) {
        T.staticVar = o;
    }
}
```

Listing 3.1: Example program for static vs. dynamic algorithm comparison

Field assignment: when an object *bar* is assigned to a field of an object *foo*, then *bar* is moved into the region of *foo*. The rationale is that if *foo* is reachable from a region, then *bar* is reachable from that region too.

Array store: the store of an object into an array is handled the same way as if the object was assigned to a field.

Global variable assignment: if an object is assigned to a global (static) variable, it is reachable from any location in the program. It is therefore moved into the heap region.

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 globaly 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 will be taken, while the dynamic algorithms takes into account only paths, that are really taken. The difference can be seen in the example program in listing 3.1: a static algorithm would identify all instances of `T` as escaping, while this algorithm identifies only a single instance as escaping.

An example for the algorithm is shown in listing 3.2 namely method `foo()` being invoked. First, a region is created for `foo()`, with a lifetime set to 0. Inside this region, 2 objects are allocated: `o1` and `o2`. Then `o2` is assigned to a field of `o1`. As both `o1` and `o2` are located in the same region, no change occurs. Next, `o1` is the receiver of a call of `bar()` which causes a region to be created for `bar()`, with a lifetime set to -1. Inside this region, `o3` is first allocated and later assigned to a field of the receiver, which corresponds to `o1`. As `o1`'s region has a longer lifetime than `o3`'s, `o3` is moved into this region, which is the one of `foo()`. Then, `o1` is the receiver of a call to `baz()`, which causes the creation of a region with a lifetime of -2, where several objects `o4` $\cdots$ `o8` are allocated. `o4` is assigned to a global variable, which causes it to be moved into the *heap* region. `o5` is passed to a native

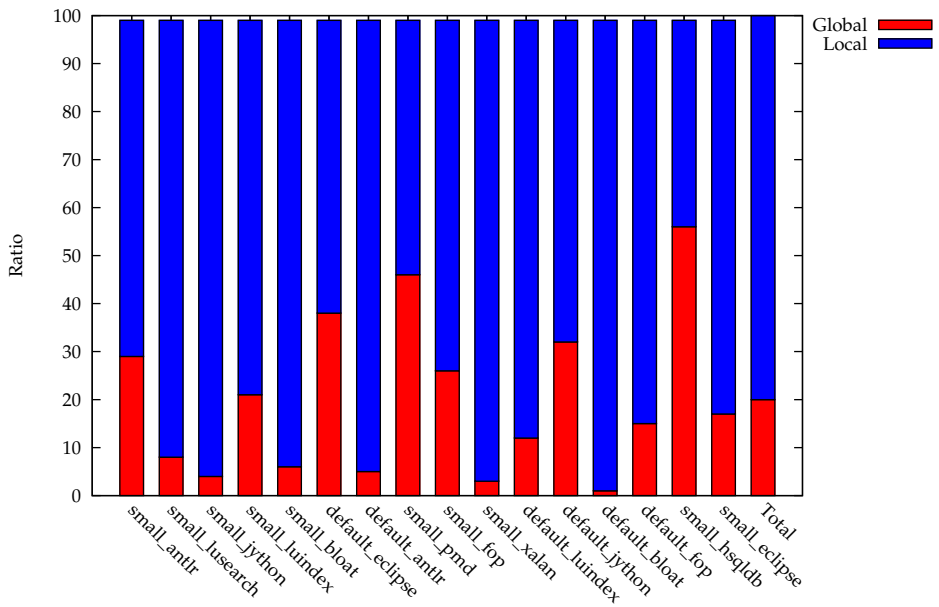


Figure 3.3: Number of thread local objects

method `nat()` and is therefore moved into the *heap* region too. The return value of `nat()` is moved to the *heap* region as well, because it comes from a native method. `o8` is moved to the heap region as soon as it is allocated, because it implements the `Runnable` interface and is therefore conservatively treated as a thread object. `o6` is returned from `baz()` and therefore moved into the region of the caller: `bar()`. Upon return from `bar()`, its region is destroyed. At this time, it contains only `o7`, which is now identified as unreachable. Back in `bar()`, `o6`, the return value of `baz()` is assigned to a field of `o1`. This causes it to be moved into the region of `foo()` with a longer lifetime. Upon return from `bar()`, its region is empty so there are no thread-local objects identified. Back in `foo()`, the region contains the objects `o1`, `o2`, `o3`, `o6`. These become unreachable after `foo()` returns and its region is destroyed.

The modified VM is used to run benchmarks of the dacapo benchmark suite with benchmark sizes default and small.

3.2 Number of thread local objects

In the first series of measurements, the total number of available thread local objects was determined. Figures 3.3 and 3.4 display the ratio of thread local objects to global objects: figure 3.3 examines the number of objects,

```
class T {
    Object field1, field2, field3;
    static Object global;

    static void foo() {
        T o1 = new T();
        Object o2 = new Object;
        o1.field1 = o2;
        o1.bar();
    }

    void bar() {
        Object o3 = new Object();
        this.field2 = o3;
        this.field3 = this.baz();
    }

    Object baz() {
        Object o4 = new Object();
        Object o5 = new Object();
        Object o6 = new Object();
        Object o7 = new Object();
        Thread o8 = new Thread();
        T.global = o4;
        nat(o5);
        o8.start();
        return o6;
    }

    static native Object nat(Object o);
}
```

Listing 3.2: Example program for escape behavior evaluation

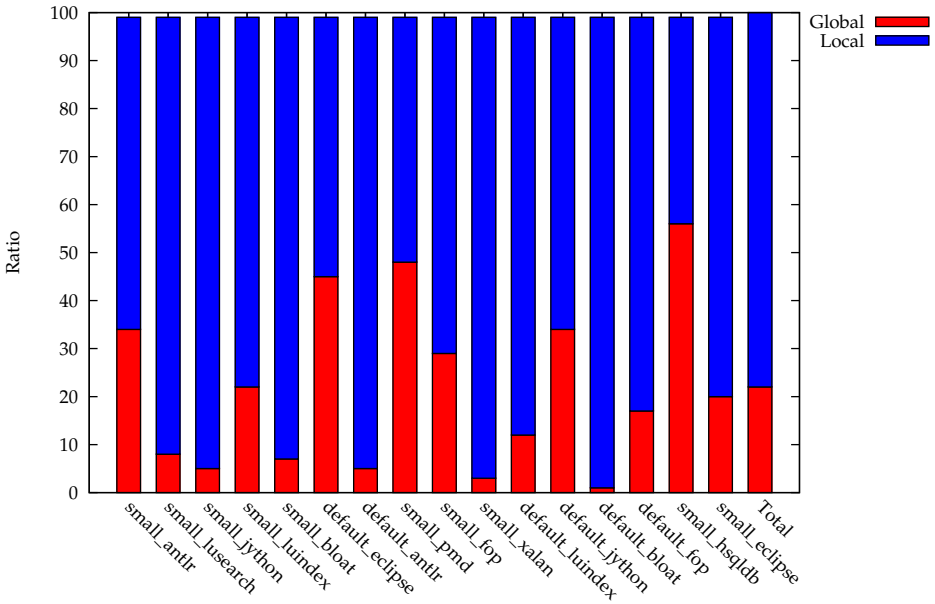


Figure 3.4: Size of thread local objects

while figure 3.4 examines the total sizes of the two object families. 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.

It is interesting to examine the distribution of the classes of thread local objects. Table 3.1 shows for every benchmark the 5 most frequent classes of thread local objects. The results were rather surprising: in all benchmarks, a great fraction of all thread local objects are instances of rather few classes. Popular classes include:

- character and byte arrays
- String
- StringBuffer

Class	Instances	Percent
small_antlr		
char[]	57027	48 %
java/lang/String	19493	16 %
java/lang/StringBuffer	11684	10 %
antlr/Lookahead	4839	4 %
long[]	4322	3 %
others	19327	16 %
small_lusearch		
char[]	281898	15 %
java/lang/String	176629	9 %
int[]	165380	9 %
java/lang/Object[]	103199	5 %
gnu/java/util/WeakIdentityHashMap$WeakBucket$WeakEntry	98238	5 %
others	981979	54 %
small_jython		
byte[]	2952821	58 %
gnu/java/util/WeakIdentityHashMap$WeakBucket$WeakEntry	986344	19 %
java/lang/Class$MethodKey	296152	5 %
java/util/HashMap$HashEntry	288183	5 %
char[]	108055	2 %
others	389522	7 %
small_luindex		
char[]	63067	20 %
java/lang/String	57436	18 %
org/apache/lucene/analysis/Token	43856	14 %
org/apache/lucene/analysis/standard/Token	43455	14 %
int[]	28688	9 %
others	69539	22 %
small_bloat		
char[]	1127478	34 %
java/lang/String	706977	21 %
java/lang/StringBuffer	474120	14 %
java/util/HashMap$HashIterator	138448	4 %
EDU/purdue/cs/bloat/util/Graph$1	129310	3 %
others	684152	20 %
default_eclipse		
java/util/Hashtable$HashEntry	6363162	28 %
char[]	5869379	26 %
char[][]	2018369	9 %
int[]	1969670	8 %
java/lang/String	942384	4 %
others	4919336	22 %
default_antlr		
char[]	1582900	54 %
java/lang/String	438540	15 %
java/lang/StringBuffer	314166	10 %
antlr/Lookahead	184504	6 %
long[]	92850	3 %
others	299364	10 %

Class	Instances	Percent
small_pmd		
char[]	7796	9 %
org/jaxen/expr/IdentitySet$IdentityWrapper	7285	9 %
net/sourceforge/pmd/jaxen/DocumentNavigator$1	7258	9 %
java/lang/String	7095	9 %
int[]	5700	7 %
others	43004	55 %
small_fop		
char[]	140260	46 %
java/lang/String	50847	16 %
java/lang/StringBuffer	25860	8 %
byte[]	11641	3 %
java/util/LinkedList$LinkedListItr	9173	3 %
others	61532	20 %
small_xalan		
char[]	3898803	87 %
java/util/LinkedList$LinkedListItr	209836	4 %
java/lang/String	107354	2 %
javax/xml/namespace/QName	67404	1 %
java/util/AbstractList$1	59318	1 %
others	106824	2 %
default_luindex		
char[]	2145469	20 %
java/lang/String	1893023	18 %
org/apache/lucene/analysis/Token	1386335	13 %
org/apache/lucene/analysis/standard/Token	1382143	13 %
int[]	770638	7 %
others	2786108	26 %
default_jython		
gnu/java/util/WeakIdentityHashMap$WeakBucket$WeakEntry	7721679	50 %
byte[]	3083831	20 %
org/python/core/PyObject[]	1534532	10 %
char[]	665510	4 %
java/lang/String	347820	2 %
others	1904191	12 %
default_bloat		
char[]	10237021	33 %
java/lang/String	6719925	21 %
java/lang/StringBuffer	4380955	14 %
EDU/purdue/cs/bloat/util/Graph$1	1684948	5 %
java/util/HashMap$HashIterator	1394831	4 %
others	6520880	21 %
default_fop		
char[]	805374	45 %
java/lang/String	332844	18 %
java/lang/StringBuffer	182403	10 %
byte[]	68334	3 %
java/nio/ByteBufferImpl	66029	3 %
others	317162	17 %

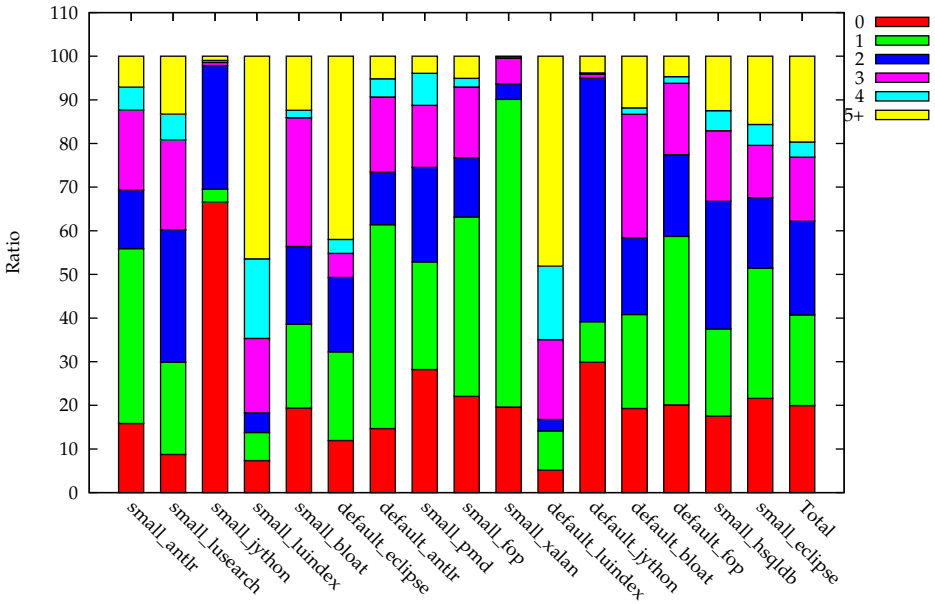


Figure 3.5: Number of stack frames objects are passed upwards the call chain

Class	Instances	Percent
small_hsqldb		
char[]	12022	25 %
java/lang/String	9439	19 %
int[]	3255	6 %
java/lang/StringBuffer	2865	6 %
java/lang/Object[]	2612	5 %
others	17180	36 %
small_eclipse		
char[]	1939387	40 %
java/lang/String	724099	15 %
int[]	230790	4 %
java/lang/String[]	196324	4 %
org/eclipse/core/runtime/Path	180506	3 %
others	1526372	31 %

Table 3.1: Distribution of classes among thread-local objects

3.3 Passing objects upwards the call chain

The next question of interest is the interprocedural escape behavior of objects. Namely, to determine what is the chance of thread local objects to

move upwards or downwards the call stack. For this purpose, upon allocation, every object is annotated with the current stack depth.

When an object becomes unreachable the difference of the stack depth at creation time and the current stack depth is used to calculate how far the object has been returned. There are two possible ways for objects to be returned: as return value or through assignment to a field of an argument. These numbers are displayed in figure 3.5.

It can be seen, that a minority of thread local objects, around 10 - 20 %, does not move upwards the call chain at all. The rest of thread local objects however does. But for most benchmarks, most thread local objects are not returned farther than 3 stack frames upwards the call chain.

When designing an intraprocedural analysis with stack allocation or regions in mind, there are basically 3 attempts on handling thread-local objects that are returned upwards the call chain:

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 his stack frame. A specialized version of the callee is compiled, which accepts a pointer to the reserved memory as an additional parameter.

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 upward the call chain.

The results determined in this section play an important role on how efficient these attempts can become in the best case:

1. If objects returned from a method escape globally, only 20% of thread local objects can be kept non-escaping.

2. If thread-local objects returned to a caller and not further escaping that caller can be handled, 26% objects can be saved from escaping.

3. Good inlining decisions to a depth up to 3 levels can save a lot of objects from escaping.

3.4 Passing objects downwards the call chain

The visualization in figure 3.6 shows, how deep objects are passed through method invocation. For this purpose, every object is annotated with a field `maxdepth` which is the depth of the deepest stack frame, the object was

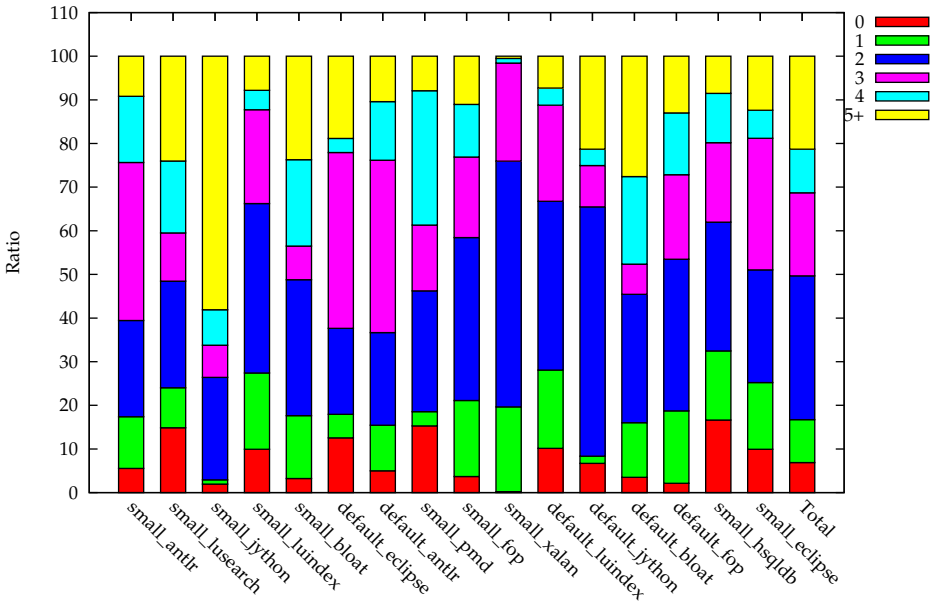


Figure 3.6: Number of frames objects are passed downwards the call chain

reachable from. `maxdepth` is initialized to the current stack depth upon object creation. Upon a method entry, all parameters of a reference type are inspected and their `maxdepth` field is adjusted to the maximum of its current value and the current stackdepth. The figure shows, that in most benchmarks, the majority of thread local objects is not passed deeper than 4 stack frames down the call chain.

This information can be used to determine, till what depth it is feasible to recurse into callees when doing interprocedural escape analysis.

Because of Java's two step object creation process: allocation of a block of memory followed by a call to the class' constructor, every Java object is passed as argument to the constructor and is thus passed at least one stack frame downwards. Arrays constitute an exception to this rule, as they have no constructor and are initialized directly by the VM. So the fraction of objects passed 0 frames downwards in figure 3.6 consists exclusively of arrays.

3.5 Region properties

Because the lifetime of a thread local object is bounded by the lifetime of the last region in which it is contained, in a region based memory management scheme, the object could be allocated in the corresponding region.

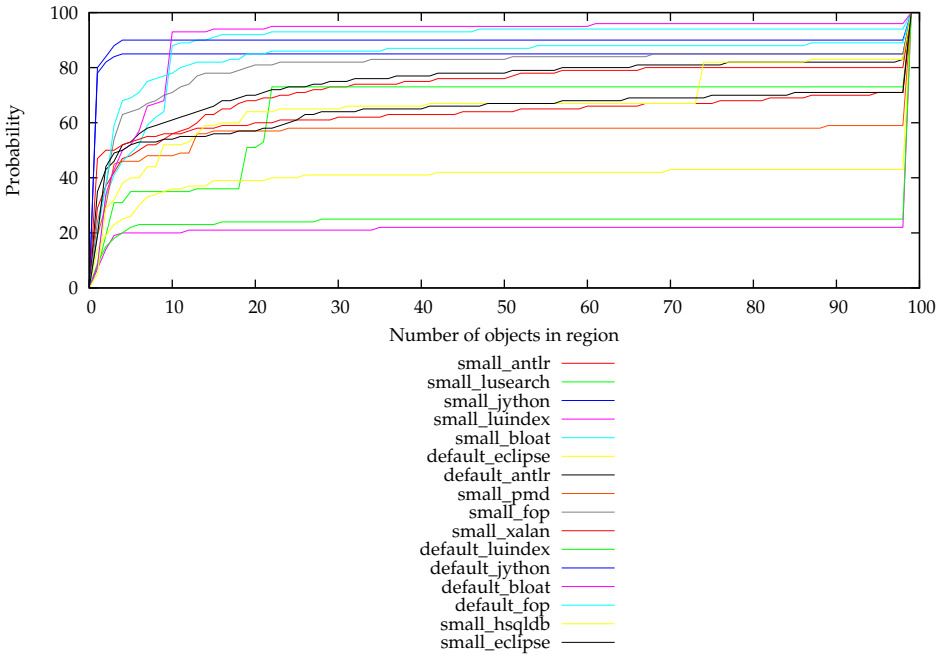


Figure 3.7: Distribution of region sizes of non-empty regions

The question of interest in such a scheme is: what is the expected size of a region? For this purpose, the distribution of region sizes is examined. Figure 3.7 visualizes the distribution of thread local objects among regions of a given size, up to the size of 100 objects. The distribution shows, that most thread local object could be allocated in rather small regions: in average 59% of all thread local objects would reside in a region that would contain less than 10 objects. On the other hand, a significant amount of thread local objects resides in rather large regions: in average 27% of thread locals objects reside in regions of a size greater than 99 objects.

This means that a lot of thread local objects reside in rather small regions, while quite some ones reside in huge regions. The implications of this observation to a potential memory management schema are

1. Objects located inside rather small regions are well suited for either stack allocation or for a region based memory management scheme.

2. Objects located in rather huge regions are well suited for allocation on a thread local heap. Region management would bring a rather small benefit, because these objects are clustered in rather few and large regions.

The implementation of some compilers requires stack frames to have

a statically known size. Under this restriction, allocation sites located in loops can't be considered for stack allocation. This rises the question, how much thread local objects become non-stack-allocable under this restriction. In the methods producing the high number of small regions, thread-local objects are probably not created inside loops.

Another very important observation is that 99.2023% of all regions are empty upon destruction. This constitutes an implication for a possible memory management scheme: in a region based memory management scheme, there should be no overhead for the management of empty regions, because there are so many of them.

Chapter 4

Implementation

The *just-in-time* compiler of CACAO is organized in passes. In the context of this thesis, 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 produces escape information for static allocation sites, used for stack allocation. Escape information for arguments and the return value is computed for the purpose of interprocedural analysis. Stack allocation in loops must be handled separately, so loops are detected and analyzed in methods allocating objects on the stack. Finally, requirements on the run-time system and the VM for the implemented optimizations are discussed.

4.1 Control flow graph

To model interprocedural 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. Basic blocks are represented using a `struct basicblock` and are created early during compilation: during the *parse* pass. A basic block contains a sequence of intermediate instructions with the property, that only the last effective[1] one may cause a regular control flow transition. The JIT compiler stores basic blocks in a linked list via the `next` member.

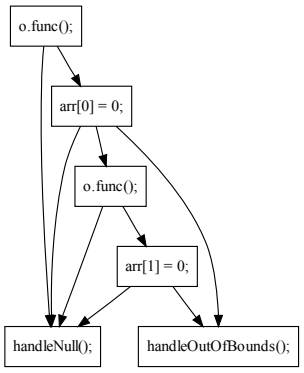
Interprocedural control flow of a Java method can be categorized into two classes:

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

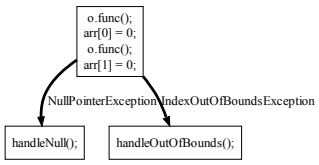
[1] A non-NOP instruction

```
try {
    o.func();
    arr[0] = 0;
    o.func();
    arr[1] = 0;
} catch (NullPointerException e) {
    handleNull();
} catch (IndexOutOfBoundsException) {
    handleOutOfBounds();
}
```

(a) Source code



(b) Control flow graph



(c) Factored control flow graph

Figure 4.1: Exceptional control flow

block.

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

To augment the regular CFG with exception control flow, basic blocks need to be split at every PEI, and an edge from the basicblock fragment to every potential exception handler needs to be added. This would pose no problem if PEIs were rare, but unfortunately, they are quite frequent in Java [5]: field accesses, array accesses, method calls and object allocation can all throw exceptions and thus cause control flow transfers. Splitting basicblocks at every PEI leads to CFGs of a much greater size. An example CFG with basicblocks split at PEIs is shown in figure 4.1(b).

A more space-efficient representation of exceptional control flow can be achieved using the *factored control flow graph* (FCFG) [5]. 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 a PEI throwing an exception of type T to the respective exception handler. An example FCFG is shown in figure 4.1(c).

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. A factored edge is connecting 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 CFG.

The reason for this simplification is that for a correct match of PEIs to exception handlers a subtype check is required, which in turn requires the two classes in question to be *linked*, i.e. fully initialized in the run-time 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 run-time.

To facilitate analyses based on the CFG, the root basic block is required to have no predecessors.

4.1.1 Implementation

Root basic block

The requirement of the root basic block having no predecessors is not always fulfilled by a Java method, consider a method where the first basic block is a loop header. Therefore an artificial root basic block is always added, containing no instructions and an implicit fall through branch to the actual root basic block. This is implemented in the function `cfg_add_root`.

Regular control flow

The nodes of the regular CFG are represented by the `struct basicblock`. The edges of the CFG are represented using the following members:

- `successors` - an array of pointers to successor nodes in the CFG.
- `predecessors` - an array of pointers to predecessor nodes in the CFG.

The regular CFG is built in the function `cfg_build` in two traversals: the first one counts the number of predecessors and successors of every basic block, the second one allocates and fills in the respective arrays. In theory, a single traversal would be sufficient to build the CFG. The second traversal is an implementational detail and is needed for a time and space efficient allocation of the arrays. In every traversal all basic blocks must be considered, a depth first search is not sufficient. The reason is that because

of exceptional control flow, the regular control flow graph can consist of disconnected components. For every block, NOPs at the end of the basic block are skipped in order to to find the last effective instruction, which falls into one of the following categories:

Unconditional branch : the target basic block is considered as successor.

Conditional branch : both the target basic block (conditional path), and the next basic block (fall through path) are considered as successors.

Switch statement : the basicblocks that are targets of the branches are considered as successors.

Exceptional control flow

The exceptional part of the CFG is represented in analogy to the regular part: by double linking basic blocks via the `exhandlers` and `expredecessors` members of `struct basicblock`.

It is built in the function `cfg_add_exceptional_edges`. In one pass, the number of PEIs is counted in every basicblock and stored in the `exouts` member of `struct basicblock` as it determines the number of exceptional exits from that block. The compiler represents the relation of `try` to `catch` blocks in a linked list of `exception_entry`. Each entry contains:

- A `start` and `end` basicblock. The sequence of basic blocks `[start, end)` contains all blocks of the `try` block.
- A `handler` block which is the exception handler.
- `catchtype` is the type of the exception to handle.

The list of exception entries is traversed twice: first the predecessor count for every exception handler and the number of successors for every guarded block is calculated. In the case that a guarded block does not contain any PEIs, no edge is connected to its handler blocks, as no control flow transition is possible. In the second pass the blocks are double linked via the `exhandlers` and `expredecessors` fields. The arrays are allocated and filled in.

4.2 Static single assignment form

Static single assignment form (SSA) is an intermediate representation of a program where every variable is assigned exactly once [6] used for static program analysis. The term static refers to the fact that although there is only one definition of a variable in the IR, the portion of the IR might be executed more than once at run-time.

```
i = 0;
i += 1;
if (cond) {
    i += 2;
} else {
    i += 3;
}
j = i;
```

(a) Original program

```
i_0 = 0;
i_1 = i_0 + 1;
if (cond) {
    i_2 = i_1 + 2;
} else {
    i_3 = i_1 + 3;
}
j_0 = phi(i_2, i_3);
```

(b) Program in SSA form

Figure 4.2: Example program transformed into SSA form

A usual imperative program won't be in SSA form and has to be transformed into it. This is done by subscripting every program variable v, creating new variable *instances* in a way satisfying the above requirement.

Within a basic block, the rules for subscripting are quite straight forward.

1. Every definition of a variable v defines a new variable v_i, with a new unique subscript.

2. Every use of a variable v is replaced with its most recent definition v_i.

However, in the presence of branches, a basic block can have several predecessors, each defining a different instance of a variable. At such join points it is not clear which definition is the most recent one. This problem is solved by introducing a notational fiction called ϕ function. A ϕ function for a variable v is placed at the beginning of a basic block B where control flow joins and is considered to be executed prior to any other statement in B. The number of arguments corresponds to the number of predecessor blocks and argument j is the definition of v reaching B from its jth predecessor. The ϕ function itself is a definition of v, namely it evaluates to the definition of v reaching the basic block.

Figure 4.2 shows an example Java program transformed into SSA form.

When translating a program in SSA form into machine code the ϕ functions are implemented as follows: for a function $v_k = \phi(v_i, v_j)$ in a basic block B with predecessors P_1 and P_2 a move $v_k \leftarrow v_i$ is emitted at the control flow edge $P_1 \rightarrow B$ and a move $v_k \leftarrow v_j$ at the control flow edge $P_2 \rightarrow B$. The resulting code is not in SSA form any more, therefore this pass is commonly called *leaving* SSA form. The example program from figure 4.2 is shown in figure 4.3 after leaving SSA form.

```
i_0 = 0;
i_1 = i_0 + 1;
if (cond) {
    i_2 = i_1 + 2;
    j_0 = i_2; // from j_0 = phi(i_2, i_3);
} else {
    i_3 = i_1 + 3;
    j_0 = i_3; // from j_0 = phi(i_2, i_3);
}
```

Figure 4.3: Program after leaving SSA form

4.2.1 IR variables

In the *bytecode* representation of a Java method there are two kinds of variables:

Local variables are untyped of the size 4 bytes and are live during the lifetime of the method.

Operand stack Bytecode instructions operate on an operand stack. An instruction pops source operands from the operand stack and pushes the result on the operand stack.

The stack analysis pass of CACAO transforms the stack-oriented representation with implicit operands to quadruple code with explicit operands. The operands are typed, numbered *IR variables* of one of the following categories:

Local variables correspond to Java local variables and are live during the whole method.

Temporary variables correspond to Java operand stack slots. Their liveness is limited to the basic block they are defined in. Temporary variables are not reused, every temporary is defined only once.

Preallocated variables have the same properties as temporary variables and are used for passing arguments and returning values from method calls. They are allocated before register allocation to match the calling conventions.

In-Variables and Out-Variables (inout variables) correspond to Java operand stack slots that are live across basic block boundaries.

The first IR variable indices are reserved for local variables. Thus a local variable can be recognized by falling into the range `[0, maxlocals)`, `maxlocals` being a member of the `jitdata` state.

Temporary and preallocated variables are already in SSA form, only locals and inouts must be transformed. They however need to be renamed: if the SSA transformation produces new instances of local variables, they require reserved indexes that can be already allocated to temporaries.

4.2.2 Translating into SSA form

A naive approach of translating a program into SSA form is to split every variable at every basic block boundary and to put a ϕ function for every variable into every basic block. Although this produces a program in valid SSA form, a representation with the minimal number of ϕ functions is desired: redundant ϕ functions are wasteful and unnecessary [1] and some optimizations perform well only operating on a minimal SSA form [3].

An often cited algorithm for transforming into SSA form is [6]. For CACAO a simpler algorithm based on abstract interpretation of the IR proposed by [23] was chosen. According to the author, the algorithm is better applicable when compiling from bytecode instead of source code.

The algorithm traverses the IR once, basicblock by basicblock, maintaining a per-basicblock *state array*. The *state array* contains for every variable the definition flowing out of the basic block. At join points, state arrays are *merged* producing ϕ 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, ϕ functions for every variable are created in loop headers in advance, and are eliminated later if they turn out to be redundant.

The algorithm first determines loop headers, then traverses the CFG once to create ϕ functions and to rename variables, producing the SSA form. In the SSA form, variables are not numbered sequentially as mandated by the CACAO IR, so in an additional step they are renumbered and the IR is normalized. After performing optimizations on the IR, it is ready for register allocation and code generation.

4.2.3 Loop headers

The first pass of the SSA transformation determines loop headers. Loop headers are blocks which have incoming backward branches. To find loop headers, the CFG is traversed in a depth-first search. As soon as a block, which was already visited, but which is still active on the stack is visited by traversing an edge, that edge is a backward branch and the block is marked as loop header. The pseudo code can be seen in algorithm 1.

Algorithm 1 mark_loops

Input: basicblock b, num_branches (defaults to 1)

```
1:  if not b.visited then
2:    b.visited ← true
3:    b.active ← true
4:    for all successors s of b do
5:      // there is 1 CF edge b → s
6:      mark_loops(s, 1)
7:    end for
8:    for all exception handlers e of b do
9:      // there is an CF edge b → e for every PEI in b
10:     mark_loops(e, number of PEIs in b)
11:   end for
12:   b.active ← false
13: else if b.active then
14:   b.backward_branches += num_branches
15: end if
```

4.2.4 State array

The aim of processing a basic block is to rename local and inout variables so that they adhere to the requirements of the SSA. To keep track of the most recent value assigned to a variable a per basic block mapping called *state array* is used. It maps a category dependent variable number (which is the IR index for locals and the stack depth for inout variables) to the instruction that most recently defined that variable. The destination operand of the instruction implies the IR variable holding that definition.

At the time a basic block gets processed, it already has an input state array, created during processing of its predecessors. The block is then processed instruction by instruction. On each definition of a variable a fresh variable is created and the *state array* is updated. Each use of a variable is replaced by the most recent definition recorded in the *state array*. The procedure is illustrated in figure 4.4.

4.2.5 CFG traversal

Basic blocks are traversed in an order such that a basic block is processed after all predecessor blocks from which there is a forward branch to that block were already processed. The pseudo code of the traversal is shown in algorithm 2.

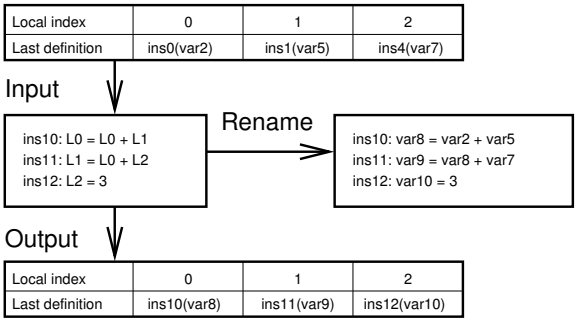


Figure 4.4: State array

Algorithm 2 traverse

Input: basicblock b

```
1:  process(b)
2:  for all successors s of b do
3:      merge b into s
4:      s.complete_predecessors += 1
5:      if s.complete_predecessors == s.incoming_forward_branches then
6:          traverse(s)
7:      end if
8:  end for
9:  for all exception handlers e of b do
10:     merge b into e
11:     e.complete_predecessors += b.num_peis
12:     if e.complete_predecessors == s.incoming_forward_branches then
13:         traverse(e)
14:     end if
15: end for
```

4.2.6 Merging state arrays

The fundamental operation during the traversal of the CFG is the *merge* operation, implemented in `ssa_enter_merge`, which merges an output state array of a basic block into its successors. This operation is responsible for the creation of ϕ functions, as illustrated in figure 4.5. As merging always occurs along a control flow graph edge, in the direction of that edge, the two participating basicblocks will be called *predecessor* and *successor*.

A special IR opcode has been introduced to CACAO, `ICMD_PHI` to represent a ϕ function uniformly with other IR instructions using a `struct instruction`.

When *merge* is called for the first time for a successor block, no *state array* exists yet for the successor, so a new one is allocated. If the successor is not a loop header, the state array is initialized to a copy of the predecessor's one and the *merge* operation terminates. For loop headers, an empty state array is created and populated entirely with ϕ functions, i.e. a ϕ function is created for every variable. The ϕ arguments are left undefined and merging continues as if a state array was found.

When a *merge* finds an already existing *state array* in the successor basic block, the state array is processed element by element. For every variable, one of the following cases occurs:

- The *state arrays* of the predecessor and successor map the variable to the same definition. In this case no action is necessary.
- The *state arrays* of the predecessor and successor map the variable to different definitions.
  - If the definition in the successor block is a ϕ function, the argument flowing in from the predecessor is set to the definition found in its *state array*.
  - If the definition in the successor block is no ϕ function, it is replaced by a newly created ϕ function. All arguments are first initialized to the old definition, as that value might have already flown into the successor from several predecessors. The argument flowing in from the predecessor is then set to the definition found in its *state array*.

4.2.7 IR properties

For the purpose of simplified processing with few corner cases in compiler passes it is desirable for the IR to adhere to some invariants. One of these invariants is, that every value is created by an IR instruction, thus no values are produced by means of side effects. This invariant is especially useful

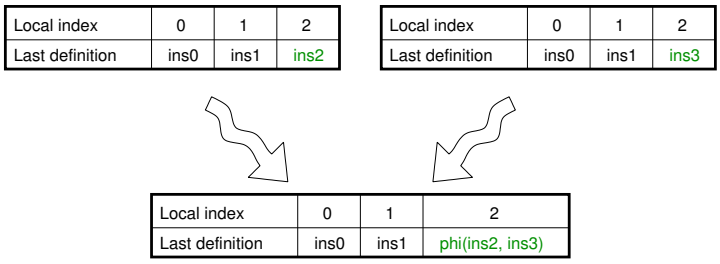


Figure 4.5: Merge of state arrays creating a ϕ function

for an IR in SSA form, as a program in SSA form directly corresponds to a functional program [2]. If every value is produced by an IR instruction, the IR instruction can be used to represent that value.

In the current CACAO implementation this invariant is violated because of the following constructs:

1. Arguments passed to a method are not produced by any IR instruction. They are located in special IR variables. Identifying these special variables is cumbersome as the mapping of positional argument numbers to IR variables is non-trivial.

2. In an exception handler block, the passed exception object is not produced by any IR instruction. It is implicitly contained in a special IR variable, namely invar 0, corresponding to the single operand stack slot.

The first violation could be solved by introducing an `ICMD_GETARGUMENT` IR generator instruction that would explicitly associate the positional argument with an IR variable. Such an instruction would even make the code generator backends simpler. Currently every backend implements the logic of loading arguments into the IR variables they were allocated to in the method prologue, leading to a lot of duplicated code. In presence of an `ICMD_GETARGUMENT` IR instruction, a backend only needs to implement this IR instruction as a copy. In the context of this thesis, this instruction was not implemented and instead the special variables containing arguments were handled specially in the root basic block.

The second violation has more severe consequences: although rare in bytecode generated from Java source, it is legal to reach an exception handler through both regular and exceptional control flow. If the semantics of an exception handler block dictate, that it always implicitly loads the current exception object into a special variable, such mixed control flow can't be represented at all. The ramifications of this violation are even more severe: the backend generates invalid machine code constituting a security vulnerability, for which a highly reliable exploit has been proposed in [21].

To workaround this problem, a generator IR instruction `ICMD_GETEXCEPTION` was introduced to represent a load of the current exception object explicitly. In a separate pass prior to computation of the CFG all exception handlers in the exception table are replaced by a special prologue block which consists of an `ICMD_GETEXCEPTION` instruction loading the current exception object into outvar 1 followed by a jump to the actual exception handler. This solves both problems: the creation of the exception object is made explicit and a direct jump to the exception handler does not result in the value contained in invar 1 to be killed.

4.2.8 Leaving SSA form

The SSA transformation was written for the purpose of performing escape analysis. Prior to performing code generation, it is necessary for the SSA form to be left. As escape analysis does not modify the IR at all, leaving SSA form is rather straight forward: the original IR is simply reconstructed.

Although a real *leave SSA* pass is not needed in the scope of escape analysis, there are several motivations for implementing it. First, in the context of future work it is supposed that optimizations will be implemented that modify the IR. The second, most important one is that transforming the generated SSA form to executable machine code can be used to validate the *enter SSA* pass.

The SSA transformation has the property of generating a lot of intermediate variables. To generate reasonable machine code, at a copy coalescation and elimination pass is required [1]. As the motivation for the *leave SSA* pass was validation of the algorithm, not performance of the generated machine code, the pass is implemented with the goal to transform the IR into a form that is valid input for the following passes: register allocation and code generation. Copy elimination and coalescing is not desirable for validation of the SSA form, because it causes some instances of variables to be eliminated and thus never to be tested.

The current `simplereg` register allocator relies on the javac compiler to coalesce uses of Java local variables and thus does not coalesce them further. So all IR variables are transformed to local variables prior to register allocation. This guarantees that every single variable instance will be allocated to a different physical location and provides a better test coverage.

ϕ functions at the beginning of a block must be translated into copy instructions placed at incoming control flow edges into that block. This is done separately for regular and for exceptional control flow.

For regular control flow, the list of all basic blocks is traversed. In every block the last effective instruction is determined in an analogous way than in CFG construction (see section 4.1). If this instruction is a jump or switch statement, every target block is replaced with a newly generated *transition* block filled with copy instructions and terminated with a jump to

the real destination block. Copy instructions for the fall through path of a conditional jump are appended at the end of the basic block.

For exceptional control, leaving SSA form is more complicated as every PEI has the potential for a control flow transition. In a traversal of the exception table, every guarded block is split at every PEI. For the guarded block fragment a new exception table entry and a new exception handler *prologue* are created. The generated prologue block has a single outvar for the purpose of passing the exception object to the actual exception handler, which expects the exception object to be passed via its invar 1. The prologue block starts with an `ICMD_GETEXCEPTION` instruction into the single outvar followed by copy instructions for the control flow edge from the PEI to the corresponding exception handler and is terminated with a jump to the actual exception handler.

4.2.9 Example

The passes of the SSA transformation are presented in a complete example. Consider the method `foo` in the source code listing 4.1. The single important variable in this example is the local variable `i`, corresponding to the IR variable 0. It gets different values on different control flow paths and is finally returned. The method gets compiled by the `javac` compiler into the IR shown in listing 4.2. After the SSA transformation, ϕ functions are created and the resulting IR is presented in listing 4.3. This form is suitable for optimization passes. Finally, ϕ functions get eliminated leading to the IR shown in listing 4.4.

For clarity, NOPs and information of no relevance for the SSA transformation have been removed from the IR representation.

4.3 Escape analysis

The aim of escape analysis is to determine whether an object escapes its creating site. As a result of the analysis, static object allocation sites and reference variables are annotated with an *escape state* which 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, which doesn't let it escape further. Such an object can be allocated on the stack and no synchronization needs to be performed on it.

```
1  class Test {
2
3      static int foo(int i, boolean doLoop) {
4          try {
5              while (doLoop) {
6                  i = 10;
7                  mayThrow();
8                  i = 20;
9                  mayThrow();
10                 i = 30;
11             }
12         } catch (Exception e) {
13         }
14         return i;
15     }
16
17     static void mayThrow() throws Exception {
18     }
19
20 };
```

Listing 4.1: Java source code of SSA example

```
1  Exceptions:
2      L001 ... L003  = L006  (catchtype: java/lang/Exception)
3
4  ======== L000 ======== (type: STD)
5  ======== L001 ======== (type: STD)
6     5:   1:  ILOAD           L1 => Li1
7     5:   2:  IFEQ            Li1 0 (0x00000000) --> L003
8  ======== L002 ======== (type: STD)
9     6:   4:  ICONST          10 (0x0000000a) => Li0
10    6:   5:  ISTORE          Li0 => L0
11    7:   6:  INVOKESTATIC    Test.mayThrow()V
12    8:   7:  ICONST          20 (0x00000014) => Li0
13    8:   8:  ISTORE          Li0 => L0
14    9:   9:  INVOKESTATIC    Test.mayThrow()V
15   10:  10:  ICONST          30 (0x0000001e) => Li0
16   10:  11:  ISTORE          Li0 => L0
17   10:  12:  GOTO            --> L001
18 ======== L003 ======== (type: STD)
19   13:  14:  GOTO            --> L005
20 ======== L004 ======== (type: STD)
21 IN:  [Ia6]
22   12:  16:  ASTORE          Ia6 => L2
23 ======== L005 ======== (type: STD)
24   14:  18:  ILOAD           L0 => Li0
25   14:  19:  IRETURN         Li0
26 ======== L006 ======== (type: EXH)
27    0:   0:  GETEXCEPTION    => Ia10
28    0:   0:  GOTO            --> L004
29 OUT: [Ia10]
```

Listing 4.2: IR of SSA example prior to SSA transformation

```
1  Exceptions:
2      L001 ... L003  = L006  (catchtype: java/lang/Exception)
3
4  ======== L000 ======== (type: STD)
5  ======== L001 ======== (type: STD)
6     0:   0:  PHI             [ Li13 Li0 ] => Li3 used
7     0:   0:  PHI             [ Li6 Li1 ] => Li4 used
8     0:   0:  PHI             [ La5 La2 ] => La5
9     5:   1:  ILOAD           L4 => Li6
10    5:   2:  IFEQ            Li6 0 (0x00000000) --> L003
11 ======== L002 ======== (type: STD)
12    6:   4:  ICONST          10 (0x0000000a) => Li7
13    6:   5:  ISTORE          Li7 => L8  (javaindex 0)
14    7:   6:  INVOKESTATIC    Test.mayThrow()V
15    8:   7:  ICONST          20 (0x00000014) => Li9
16    8:   8:  ISTORE          Li9 => L10  (javaindex 0)
17    9:   9:  INVOKESTATIC    Test.mayThrow()V
18   10:  10:  ICONST          30 (0x0000001e) => Li12
19   10:  11:  ISTORE          Li12 => L13  (javaindex 0)
20   10:  12:  GOTO            --> L001
21 ======== L003 ======== (type: STD)
22   13:  14:  GOTO            --> L005
23 ======== L004 ======== (type: STD)
24   12:  16:  ASTORE          La18 => L14  (javaindex 2)
25 ======== L005 ======== (type: STD)
26    0:   0:  PHI             [ Li3 Li11 ] => Li15 used
27    0:   0:  PHI             [ La5 La14 ] => La16
28   14:  18:  ILOAD           L15 => Li17
29   14:  19:  IRETURN         Li17
30 ======== L006 ======== (type: EXH)
31    0:   0:  PHI             [ Li8 Li10 ] => Li11 used
32    0:   0:  GETEXCEPTION    => La18
33    0:   0:  GOTO            --> L004
```

Listing 4.3: IR of SSA example after SSA transformation

```
1  Exceptions:
2      L002.a ... L002.b  = L011  (catchtype: java/lang/Exception)
3      L002.b ... L002.c  = L012  (catchtype: java/lang/Exception)
4
5  ======== L000 ======== (type: STD)
6     0:   0:  COPY            Li0 => Li3
7     0:   0:  COPY            Li1 => Li4
8     0:   0:  COPY            La2 => La5
9  ======== L001 ======== (type: STD)
10    5:   1:  ILOAD           L4 => Li6
11    5:   2:  IFEQ            Li6 0 (0x00000000) --> L007
12 ======== L002.a ====== (type: STD)
13    6:   4:  ICONST          10 (0x0000000a) => Li7
14    6:   5:  ISTORE          Li7 => L8
15    7:   6:  INVOKESTATIC    Test.mayThrow()V
16 ======== L002.b ====== (type: STD)
17    8:   7:  ICONST          20 (0x00000014) => Li9
18    8:   8:  ISTORE          Li9 => L10
19    9:   9:  INVOKESTATIC    Test.mayThrow()V STATIC
20 ======== L002.c ====== (type: STD)
21   10:  10:  ICONST          30 (0x0000001e) => Li12
22   10:  11:  ISTORE          Li12 => L13
23   10:  12:  GOTO            --> L008
24 ======== L003 ======== (type: STD)
25   13:  14:  GOTO            --> L009
26 ======== L004 ======== (type: STD)
27   12:  16:  ASTORE          La18 => L14
28    0:   0:  COPY            Li11 => Li15
29    0:   0:  COPY            La14 => La16
30 ======== L005 ======== (type: STD)
31   14:  18:  ILOAD           L15 => Li17
32   14:  19:  IRETURN         Li17
33 ======== L007 ======== (type: STD)
34    0:   0:  GOTO            --> L003
35 ======== L008 ======== (type: STD)
36    0:   0:  COPY            Li13 => Li3
37    0:   0:  COPY            Li6 => Li4
38    0:   0:  GOTO            --> L001
39 ======== L009 ======== (type: STD)
40    0:   0:  COPY            Li3 => Li15
41    0:   0:  COPY            La5 => La16
42    0:   0:  GOTO            --> L005
43 ======== L010 ======== (type: STD)
44    0:   0:  GOTO            --> L004
45 ======== L011 ======== (type: EXH)
46    0:   0:  GETEXCEPTION    => La18
47    0:   0:  COPY            Li8 => Li11
48    0:   0:  GOTO            --> L010
49 ======== L012 ======== (type: EXH)
50    0:   0:  GETEXCEPTION    => La18
51    0:   0:  COPY            Li10 => Li11
52    0:   0:  GOTO            --> L010
```

Listing 4.4: IR of SSA example after ϕ function elimination

ESCAPE_METHOD_RETURN: the object escapes its creating method by being returned to the caller. Such objects are not further optimized.

ESCAPE_GLOBAL: the object escapes its creating method and even its creating thread. This happens if the object becomes accessible via a static variable or if it is passed to native or unanalyzed code. No optimization 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.

The algorithm that was implemented for CACAO is based on the algorithm presented in [16].

4.3.1 Intraprocedural analysis

Intermediate instructions in CACAO have the form of quadruple code of the form $dst = OP(s_1, ..., s_n)$ where dst and s_i are intermediate variable indexes. At the point where escape analysis is performed they are in SSA form. During intraprocedural analysis IR instructions are processed one at a time.

References to run time objects are stored in intermediate variables. These variables are annotated with an escape state which expresses the escape state of the objects they reference.

During escape analysis, the escape state of a variable can only take a higher value. Thus *adjusting* the escape state to some value corresponds to setting it to the maximum of its current value and of the new value. This is implemented in the function `escape_analysis_ensure_state`.

Variables are merged into *equi-escape sets* (EES) where all members share the same escape state. Variables get merged upon a copy operation $dst = src$ or upon a phi function $dst = \phi(s_1, ..., s_n)$, the analysis is thus a *Steensgaard*, flow-insensitive analysis. For brevity, "merging" two variables denotes merging the two EESs the two variables are part of. Merging is implemented in the function `escape_analysis_merge`.

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 necessary 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 two variables are merged, their dependency lists are merged as well as the merged set of variables 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 `classinfo *` 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 `ICMD_ACONST` might be unresolved: in that case it is unknown, whether the class has a finalizer and it must be conservatively assumed that it has one.

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 and must thus 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 $s1$: if $s1$ is reachable from a different thread, $s2$ will be reachable as well. $s2$ is further added to the *dependency list* of $s1$. This is necessary, because if $s1$ escapes at a later point, the escape state of $s2$ needs to be adjusted as well. If $s1$ contains an 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 an 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 agains `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 and is thus marked as escaping the method.

ICMD_INVOKESTATIC, ICMD_INVOKESPECIAL In a method invocation that can be statically bound, i.e. the callee method is statically known the results of interprocedural analysis are used to adjust the escape state of the arguments and the return value. Interprocedural analysis also yields, which arguments can be returned from the callee. They are all merged with the return value. If results of interprocedural analysis are 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. The assumption must be recorded with the deoptimization framework, and the generated code must be invalidated once this assumption gets violated.

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 The instruction is processed in analogy to a statically bound method call, but all possible target methods must be considered. Class hierarchy analysis is used to determine all possible target methods and the escape state of the arguments and return value is adjusted. If the results are missing for a single possible target, it must be conservatively assumed that all arguments and the return value escape globally.

Also, the assumption about the possible targets of the given invocation must be registered with the deoptimization framework. In case the assumption does not hold anymore, the method's code must be invalidated.

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.

```
1 void foo() {
2     T o1 = new T();
3     o1.f = new NullPointerException();
4     throw o1.f;
5 }
```

Listing 4.5: Object field as alias to object

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

ICMD_PHI As a phi function of the form $dst = \phi(s1, s2)$ corresponds to one of the following copy operations $dst = s1$ or $dst = s2$, it is handled as these copy operations.

Postprocessing getfield instructions

Object fields pose an problem to escape analysis, because they introduce additional aliases. Consider the source code in listing 4.5. An object is first stored in the field of a method local object, and thus does not escape. At a later point it is loaded from the same field and thrown as exception. The algorithm must be able to mark it correctly as globally escaping.

To determine such objects, all `ICMD_GETFIELD` instructions of the form $dst = s1.f$ are processed once again, after the EESs have been constructed, and thus aliases are known. The variable dst has been initially marked as non-escaping. The dependency list of $s1$ is traversed, considering each item of the form (f, dep) and the escape state of dep is adjusted to the one of dst.

Further, it might have become evident, that the EES of $s1$ can contain an argument and thus dst was loaded from an argument. In this case, the escape state of dst is adjusted to `ESCAPE_GLOBAL`.

4.3.2 Interprocedural analysis

As escape analysis computes an escape state for every variable of reference type, and thus for arguments as well, this information is used to construct summary information for a method, that can be used in caller contexts to adjust the escape state of actual arguments. The summary information is represented using a `class ParamEscape` and stored with the method descriptor (`methodinfo`) via the `paramescape` member. The summary information contains:

- The escape state for every argument of a reference type. This is computed as the escape state of the EES of the variable the argument is passed in.

- For every argument of a reference type, whether this argument can be returned from the method. This is true if its escape state is `ESCAPE_METHOD_RETURN`. However, the escape state `ESCAPE_METHOD_RETURN` is transformed to `ESCAPE_METHOD` in the summary information, to reflect the view of a caller: if an argument is passed to a callee and gets returned back, it escapes only the method from the perspective of the caller.
- The escape state of the 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, to reflect the view of a caller.

4.3.3 Implementational notes

Native methods

Objects passed to native methods are marked as escaping globally. Empiric evidence however has shown, that many objects are passed to native methods, so applying this rule consequently leads to a lot of objects escaping unnecessarily.

This evidence is confirmed by tables 4.1 and 4.2, 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.

As the escape behavior of some methods of the Java runtime library that call into native methods is known, hardcoded method summaries can be used for these methods. A good choice of methods is:

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

Inlining

Thread-local objects returned from a method escape globally and thus can't be considered for optimization. However, if the method in question gets inlined, the return value does not escape the caller. Thus inlining can improve the impact of escape analysis a lot. The canonical example for this scenario is the `iterator()` method of container classes which is usually implemented as an one liner returning a temporary iterator object that almost

Benchmark	Pessimistic	Optimistic
_200_check	7.06%	7.32%
_202_jess	0.07%	26.54%
_228_jack	48.16%	69.18%

Table 4.1: 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 4.2: Dynamic ratio of method-local objects for different assumptions about native methods (Dacapo)

never escapes the calling method. Inlining such methods leads in practice to high dynamic numbers of non-escaping objects.

The more methods are inlined, the greater the chance to keep objects non-escaping. [16] even suggest inlining more aggressively, if there is a chance to capture more objects.

4.3.4 Example

The implemented algorithm will now be applied to an example method and examined in detail. Consider the Java function `foo` in listing 4.6. The function allocates 3 objects of type `T` and conditionally links them via the `f` field. The object allocated at line 12 is assigned to the field of the argument `arg` and thus escapes the method. The two other objects allocated at lines 10 and 11 obviously don't escape their creating method.

At compilation time for the sample method it is supposed, that the constructor `T.<init>` and the method `T.test` have already been analyzed, and thus interprocedural escape information is available. The trivial implicit constructor obviously does not let the `this` argument escape, and the `T.test` method does not even use the implicit `this` argument.

The representation of the method in a simplified CACAO IR is shown in listing 4.7. Each statement is followed by a representation of the relevant data structure it affects: an *eqi-escape set*. The following notational convention is used to represent an *eqi-escape set*:

$$\{ESCAPE\_STATE : v_1, \ldots, v_n, [(d_1, f_1), \ldots, (d_n, f_n)]\}$$

Where `ESCAPE_STATE` denotes the escape state of the set, v_i are IR variables contained in that set and (d_i, f_i) are elements of the dependency list:

d_i being an IR variable, and f_i the identifier of the field. The escape state might be followed by the flag ARG denoting that the set contains an argument.

Analysis phase

In the method header, all variables corresponding to arguments are marked as containing an argument. This can be observed at line 2.

At line 4, the instruction sequence for object allocation is shown, consisting of a `ACONST` and a `BUILTIN new`. The newly allocated object, stored in variable `a18` starts as non-escaping and constitutes a single element EES. The `COPY` instruction at line 8 causes `a19` to be added into the EES of `a18`. At line 10, the newly allocated object is passed to the class' constructor. At this position, the escape state of the variable must be adjusted to at least `ESCAPE_METHOD`. The method summary for the constructor yields, that the constructor does not let the object escape further, the escape state is not further changed.

The instruction sequence is then repeated for every of the three initial allocations.

At line 34, an object is stored in a field of an argument. This is evident because the EES containing the reference `a6` bears the `ARG` flag. In this special case, the escape state of the stored object is adjusted to global. The stored object is then added to the dependency list of `a6`.

At line 38, the reference variable `a7` is passed to the virtual method `T.test` and its escape state must be adjusted to `ESCAPE_METHOD`. Class hierarchy analysis is used to determine, that no other class overrides the method, and thus the method summary of only `T.test` is used to adjust the escape state of `a7`, which is left unchanged, as the implicit `this` reference does not escape `T.test`.

Basic blocks 2 and 3 contain only copy instructions and cause the escape sets to grow.

At line 56, 3 ϕ functions are processed: the arguments are merged with the destination operand and the respective EESs grow. The source code level variable `o1` flows into block 2 as `a7` and `o2` into block 3 as `a5` respectively. The ϕ function at line 56 corresponds to the source code level variable `o`, setting it to either `o1` or `o2` depending on the incoming path.

At line 66, the field `f` of `a15` is set to `a16`. As the EES of `a15` contains only method local objects, `a16` does not escape at this point, but is added to the dependency list of `a15`, for the case it would escape later.

Determining the results

After the analysis phase has been completed, the question of interest is, what the escape state of objects allocated inside the analyzed method is.

```
1  class T {
2      T f;
3
4      boolean test() {
5          return false;
6      }
7
8      static void foo(T arg) {
9          T o;
10         T o1 = new T();
11         T o2 = new T();
12         arg.f = new T();
13
14         if (o1.test()) {
15             o = o1;
16         } else {
17             o = o2;
18         }
19
20         o.f = o1;
21     }
22 }
```

Listing 4.6: Example method for escape analysis

For this, we consider all `BUILTIN new` instructions and examine the escape state of the EESs their destination operands are contained in.

For the first two allocation sites, at line 10 and 11 at source code level, the allocated objects escape the method, for the allocation site at line 12, the object escapes globally.

4.4 Stack allocation

Results of escape analysis are used to allocate method-local objects on the call stack, as described in section 2.4.

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 chosen for allocation on the call stack. Requirements for stack space are calculated right after escape analysis. Although stack space could be reused for stack objects with non-overlapping lifetimes, we don't perform such an optimization, because in chapter 3 it was determined, that the number of stack local objects in a method tends to be rather low.

For every method-local allocation site, space is reserved in an area for object allocation on the stack. This space is identified by an offset into this area. For efficient inline zeroing of the space from JIT code, the space reserved for an object is always rounded to the machine word size. The total

```
=== BB 0 ===
    {NONE, ARG: a0}
=== BB 1 ===
10: ACONST          class T => a17
    {NONE: a17}
10: BUILTIN         new (a17) => a18
    {NONE: a18}
10: COPY            a18 => a19
    {NONE: a18, a19}
10: INVOKESPECIAL   T.<init> (a19)
    {METHOD: a18, a19}
10: ASTORE          a18 => a4
    {METHOD: a4, a18, a19}
11: ACONST          class T => a20
    {NONE: a20}
11: BUILTIN         new (a20) => a21
    {NONE: a20}, {NONE: a21}
11: COPY            a21 => a22
    {NONE: a21, a22}
11: INVOKESPECIAL   T.<init> (a22)
    {METHOD: a21, a22}
11: ASTORE          a21 => a5
    {METHOD: a5, a21, a22}
12: ALOAD           a0 => a6
    {NONE, ARG: a0, a6}
12: ACONST          class T => a23
    {NONE: a23}
12: BUILTIN         new (a23) => a24
    {NONE: a23}, {NONE: a24}
12: COPY            a24 => a25
    {NONE: a24, a25}
12: INVOKESPECIAL   T.<init> (a25)
    {METHOD: a24, a25}
12: PUTFIELD        T.f a6 a24
    {GLOBAL: a24, a25}, {NONE, ARG: a0, a6, [(T.f, a24)]}
14: ALOAD           a4 => a7
    {METHOD: a4, a7, a18, a19}
14: INVOKEVIRTUAL   T.test (a7) => i26
    {METHOD: a4, a7, a18, a19}
14: IFEQ            i26 0x0 --> L003
15: NOP
=== BB 2 ===
15: ALOAD           a7 => a10
    {METHOD: a4, a7, a10, a18, a19}
15: ASTORE          a10 => a11
    {METHOD: a4, a7, a10, a11, a18, a19}
15: GOTO            --> L004
17: NOP
=== BB 3 ===
17: ALOAD           a5 => a8
    {METHOD: a5, a8, a21, a22}
17: ASTORE          a8 => a9
    {METHOD: a5, a8, a9, a21, a22}
20: NOP
=== BB 4 ===
0:  PHI           [ a11 a9 ] => a12
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a18, a19, a21, a22}
0:  PHI           [ a10 a7 ] => a13
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a13, a18, a19, a21, a22}
0:  PHI           [ a5 a8 ] => a14 (unused)
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a13, a14, a18, a19, a21, a22}
20: ALOAD           a12 => a15
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a13, a14, a15, a18, a19, a21, a22}
20: ALOAD           a13 => a16
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a18, a19, a21, a22}
20: PUTFIELD        T.f a15 a16
    {METHOD: a4, a5, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a18, a19, a21, a22, [(T.f, a16)]}
21: RETURN
=== BB 5 ==
```

Listing 4.7: Escape analysis performed on the CACAO IR

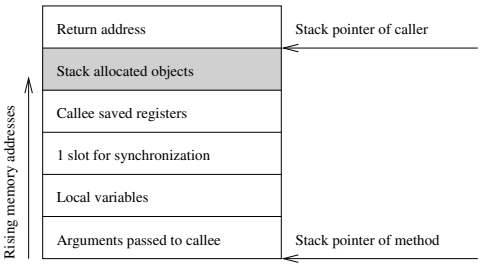


Figure 4.6: Layout of stack frame with highlighted area for stack allocation

size of the reserved space is stored in the `jitdata` state in the member `stackobjectsize`.

Allocation sites represented by a `BUILTIN new` IR instruction are transformed to a newly introduced `STACK_NEW` instruction which contains the offset into the space for stack allocation in the s1 operand field and the `classinfo *` of the allocated class.

Allocation of stack objects has been implemented on the x86 architecture. When the code generator generates the method prologue, it lays out the fixed-size stack frame. If it finds the `stackobjectsize` member of the `jitdata` state non-zero, it reserves space in the stack frame, at an offset `stackobjstart`. The layout of the stack frame and the location of this area in the stack frame can be seen in figure 4.4.

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. Then all data fields of the object are zeroed as required by the JVM specification.

An example of code generated on x86 for stack allocation of a `StringBuilder` object with a destination operand `%esi` is show in listing 4.8.

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

Listing 4.8: Machine code generated for a `STACK_NEW` IR instruction

4.4.1 Allocation in loops

Although it is possible to dynamically grow the stack frame of an active method, like in the C language via the `alloca` function, this mechanism introduces additionnal run-time overhead. For example the maintanance of a `frame pointer` register or additionnal overhad for subroutine calls. For this reason CACAO uses stack frames of a static size. Thus space in the stack frame can be reserved only for a fixed number of objects at compile time. This poses a problem if non-escaping objects are allocated in a loop.

In such a situation, the objects can be allocated on the stack, only if the space for the object can be reused on every loop iteration [16]. 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 via backward branches.

To achieve this, if an allocation site is considered for stack allocation, loop analysis (see section 4.5) is peformed to detect loops. If the allocation site is located inside a loop, the alias set of the allocated object obtained by following forward branches but not leaving the loop is constructed. The only way for the object to reenter the loop is via ϕ functions of the loop header. If any element of the alias set is an argument of a live ϕ function of the loop header, the object must not be allocated on the stack. Memory dependencies are not tracked: if the object in question is stored into an static field, instance field or an array in the loop, it is heap allocated. This algorithm is implemented in the `class LoopEscapeAnalysis`.

4.5 Loop analysis

The stack allocation optimization requires to determine, whether an allocation site is located inside a loop. If so, it needs to traverse the container loop. In case of nested loops, the container loop and its nested loops need to be traversed.

To fulfill this requirement, the algorithms described in [1] are used to first detect all loops, then determine their nesting level and 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 a loop and its nested loops.

The algorithm described in this section detects loops only in *reducible control flow graphs*. Java compiler generated code consists almost exclusively of such control flow graphs, so the approach taken is to test, whether the CFG in question is reducible and if not, give up optimization of the particular function.

The code is implemented in the `class LoopHierararchy`.

4.5.1 Dominator tree

The algorithm to detect loops makes use of the dominator to identify backward branches, which in turn identify a loop header and are used to induce the loop body.

A node D in the CFG *dominates* a node N, if every directed path of edges from the root node to N leads through D. In a connected CFG, if two nodes D_1 and D_2 both dominate a node N, then it can be proved [1] that either D_1 dominates D_2 or vice versa. This in turn leads to the theorem, that for every node N, except the root node there is an *immediate dominator* $idom(N)$ different from N such that $idom(N)$ dominates N, but does not dominate any other dominator of N.

The *dominator tree* contains all nodes of the CFG and edges $idom(N) \rightarrow N$, linking every CFG node to its immediate dominator.

The algorithm used to construct the dominator tree is the algorithm of Lengauer-Tarjan [19] and it was originally implemented as a perquisite for Cytrons algorithm [6] of constructing the SSA form, which was implemented as part of this thesis but discontinued in favor of the algorithm presented in 4.2.

When taking into account exceptional control flow, exceptional summary edges are treated as ordinal CFG edges to successor nodes. Consider the CFG in figure 4.7(a) with a summary exceptional edge $BB1 \rightarrow EXH$. Splitting $BB1$ at PEIs and expanding the summary edge results in the CFG in figure 4.7(b). The single-entry, multiple-exit extended basic block BB_1 is treated as opaque and not further structured by the analysis, but its internal structure must be considered if making use of the analysis results.

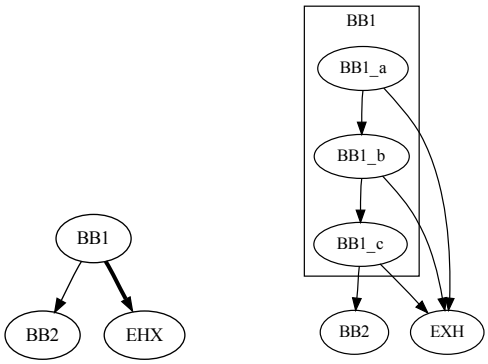
4.5.2 Loop detection

In a *reducible control flow graph* there is only a single entry point into a loop, the *loop header*. It has the property that it *dominates* all nodes in the loop. A *backward edge* is an CFG edge $N \rightarrow H$ from a node N to a node H such that H dominates N. Every backward edge identifies a loop, more precisely a loop header, which is H.

This property is used to test every CFG edge $N \rightarrow H$ whether it is a *backward edge*. If so, a loop header was found. All reverse CFG paths starting at N are then followed, terminating when H is reached. The nodes collected underway are added to the loop identified by its header H.

4.5.3 Loop hierarchy

Let H_{outer} and H_{inner} be two headers identifying loops. The loop induced by H_{inner} is a loop nested inside the loop induced by H_{outer} if H_{inner} is contained withing the loop induced by H_{outer}.



(a) With summary exceptional edge

(b) With expanded summary edge

Figure 4.7: Example CFG

After detecting loops, every loop header is tested whether it is contained in the set of nodes induced by some different loop header. If so, all nodes of the inner loop are removed from the outer loop, including the header and the header is inserted into the set of child loops of the outer loop.

To efficiently store loop membership and loop nesting of the CFG, tree numbering is used [29], which was also used for representing the class hierarchy in earlier releases of CACAO. Every basicblock is annotated with a `LoopInfo` class showed in listing 4.9. The tree of loop headers is traversed in a depth first search and a counter is incremented each time some loop header is processed. The value of the counter at the point a loop header was entered and left is stored in the `_nr` and `_to` fields respectively of the `LoopInfo` of all member nodes of the loop.

This representation enables quick testing, whether two nodes are inside the same loop. This is true, if the `_nr` values of their loop infos are the same, as seen in the `containsStrict` method. To test, whether some node N_{inner} is contained in the same or inside some nested loop of a node N_{outer}, it must be tested, whether `_nr` of N_{inner} is in the range `[_nr, _to]` of the loop info of N_{outer}. This test is implemented in the `contains` method.

An example of the loop hierarchy encoding is shown in figure 4.8. It contains a loop $H1$ with two inner loops $H2$ and $H3$ and two more inner loops $H4$ and $H5$ of $H3$. Unlabeled nodes correspond to basic blocks contained immediately in a loop. The two numbers in the nodes correspond to the `_nr` and `_to` values.

```
1  class LoopInfo {
2      public:
3          bool containsStrict(const LoopInfo& other) const {
4              return _nr == other._nr;
5          }
6          bool contains(const LoopInfo& other) const {
7              return _nr <= other._nr && other._nr <= _to;
8          }
9      private:
10         unsigned _nr;
11         unsigned _to;
12         basicblock *_header;
13 };
```

Listing 4.9: Loop information associated with every basic block

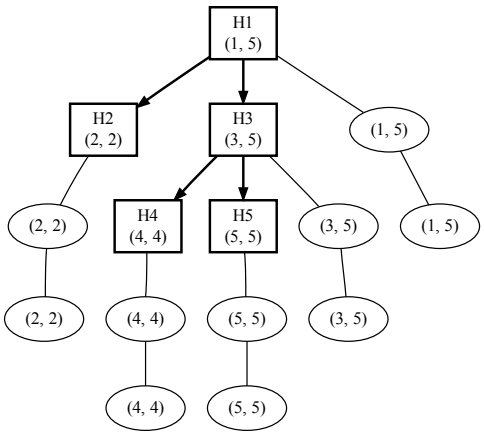


Figure 4.8: Loop hierarchy represented using tree numbering

4.6 Optimization framework

The optimizations implemented in this thesis 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.

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, when all methods were recompiled, gathering of statistics is triggered interactively.

Because all methods get compiled at once, a recompilation order that is favorable for interprocedural escape analysis can be chosen. This is one where every callee gets recompiled before the caller. This is determined by constructing a program call graph. A virtual method call is modelled as sequential calls to all possible targets of the call. The program call graph is then traversed in a depth-first traversal. Every time a method is left, it is queued for recompilation.

Escape analysis and stack allocation require the optimization framework to provide support for deoptimization.

In interprocedural analysis, it is assumed that a virtual call targets only those implementations of the method that are loaded into the VM. If an additional implementation gets loaded, the optimized code relying on that assumption must be invalidated and recompiled.

An invalidated method might already have allocated objects on the stack, which as a result of the invalidation suddenly become escaping and thus should have never been allocated in the stack frame. In such a situation, the deoptimization framework must be able to move the objects in question to the heap and rewrite all references pointing to it. Fortunately, such references reside at well known locations only, namely in registers and in the stack frame downwards the call chain.

The current ad-hoc implementation does not support these requirements and assumes, that once recompilation with stack allocation gets triggered, no further classes will be loaded, and thus no methods invalidated.

In a realistic optimization framework, methods won't get recompiled in an order favorable for escape analysis and the quality of interprocedural analysis will thus significantly degrade, because of missing information in intraprocedural analysis. Further research is necessary here to support interprocedural analysis well.

Chapter 5

Evaluation

To evaluate the implemented escape analysis algorithm and the related stack allocation optimization, the SPECjvm98 and dacapo benchmark suites were used.

All benchmarks were executed on a Lenovo Thinkpad X60s system equipped with an Intel Core Duo L2400 CPU running at 1.66GHz, 512 MB of RAM and a Debian GNU/Linux operating system with a kernel version 2.6.24-1-686, a libc6 version 2.7-5 and a gcc version version 4.1.3 20070718 (prerelease) (Debian 4.1.2-14).

The CACAO version used for the benchmarks is a development version that will be submitted with slight changes into the mercurial repository, in the branch *pm-escape* targeting the i386 architecture. The Java run-time library is a GNU Classpath development snapshot[1] from CVS updated on 2008-02-29 12:06 CET.

The benchmarks are always run several times, at least 3 times. The first run is performed without optimization, in order for enough classes to be loaded. Then, in the second run, it is assumed that all classes the benchmark will ever use are loaded and recompilation with escape analysis and stack allocation is triggered. The third run executes already optimized code and can be used to collect results.

5.1 Stack allocated objects

In the first run of benchmarks, the number of stack allocated objects is considered. This evaluates in the first place the quality of the escape analysis. Table 5.1, table 5.2, figure 5.1 and figure 5.2 show the number of objects that escape analysis identified as not escaping their creating method. Static numbers refer to the number of method-local allocation sites, while dy-

[1] At the time of development the CACAO development branch did not work with any classpath release

	Static			Dynamic		
Benchmark	Stack	Heap	Ratio	Stack	Heap	Ratio
_200_check	172	325	34.61 %	141	1784	7.32 %
_201_compress	52	375	12.18 %	165	1763	8.56 %
_202_jess	80	651	10.94 %	1412924	3910570	26.54 %
_205_raytrace	65	472	12.10 %	2943812	2161302	57.66 %
_209_db	71	399	15.11 %	2907508	163267	94.68 %
_213_javac	121	955	11.25 %	353919	3608786	8.93 %
_222_mpegaudio	51	470	9.79 %	45	3329	1.33 %
_227_mtrt	65	471	12.13 %	2956974	2358143	55.63 %
_228_jack	144	550	20.75 %	4410417	1964647	69.18 %

Table 5.1: Number of stack allocated objects (SPECjvm98)

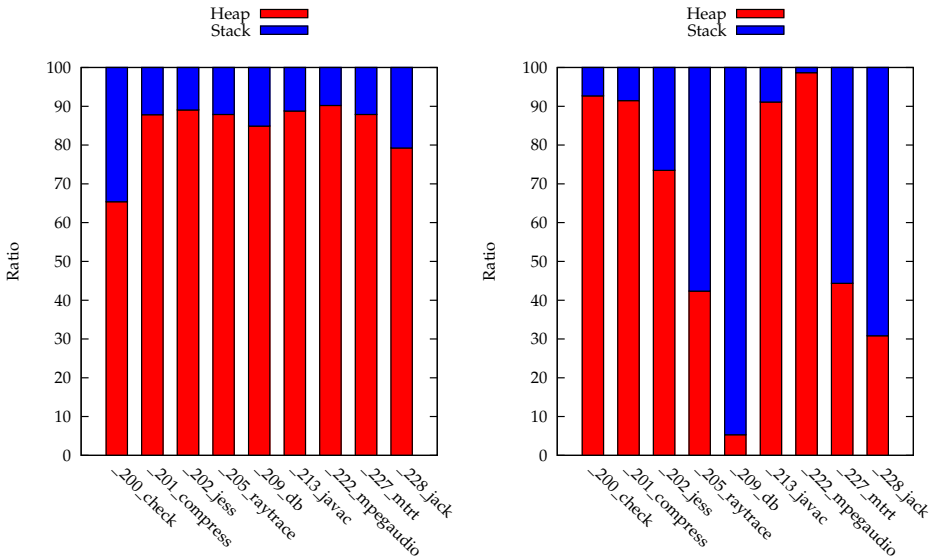
	Static			Dynamic		
Benchmark	Stack	Heap	Ratio	Stack	Heap	Ratio
antlr	854	2080	29.11 %	220661	1265972	14.84 %
bloat	142	2571	5.23 %	303141	19931783	1.50 %
fop	226	1845	10.91 %	160563	711345	18.42 %
hsqldb	132	1394	8.65 %	509	231406	0.22 %
jython	195	3801	4.88 %	80823	20337599	0.40 %
luindex	121	1421	7.85 %	114201	9213760	1.22 %
lusearch	217	997	17.87 %	703365	10550607	6.25 %
pmd	197	1709	10.34 %	3449114	25608569	11.87 %
xalan	163	2150	7.05 %	69621	2757033	2.46 %

Table 5.2: Number of stack allocated objects (Dacapo)

namic numbers refer to the number of method-local objects allocated at run-time.

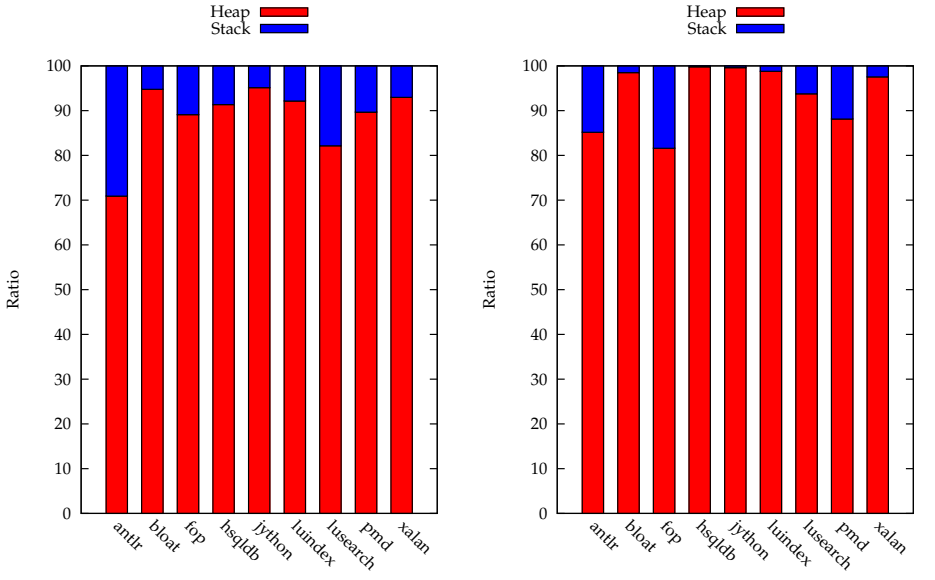
5.2 Stack allocated classes

In the next set of benchmark runs, the distribution of the dynamic number of stack allocated objects among Java classes is considered. In tables 5.3 and 5.4, the 5 most frequent classes of stack objects for every benchmark are shown.



(a) Static (b) Dynamic

Figure 5.1: Number of stack allocated objects (SPECjvm98)



(a) Static (b) Dynamic

Figure 5.2: Number of stack allocated objects (Dacapo)

Class	Instances	Percent
_200_check		
java/lang/StringBuffer	82	57.75 %
java/lang/String	45	31.69 %
java/lang/Double	2	1.41 %
spec/benchmarks/_200_check/superClass	2	1.41 %
java/lang/Integer	1	0.70 %
others	9	6.34 %
_201_compress		
java/lang/String	59	35.54 %
java/lang/StringBuffer	53	31.93 %
java/io/File	26	15.66 %
spec/io/File	26	15.66 %
java/util/Vector$1	1	0.60 %
others	0	0.00 %
_202_jess		
java/lang/Integer	1409151	99.73 %
java/lang/StringBuffer	3642	0.26 %
java/lang/String	111	0.01 %
spec/benchmarks/_202_jess/jess/Deftemplate	9	0.00 %
java/io/File	2	0.00 %
others	9	0.00 %
_205_raytrace		
spec/benchmarks/_205_raytrace/Vector	2739890	93.07 %
spec/benchmarks/_205_raytrace/Color	102417	3.48 %
spec/benchmarks/_205_raytrace/IntersectPt	62417	2.12 %
spec/benchmarks/_205_raytrace/Ray	25474	0.87 %
java/lang/StringBuilder	12810	0.44 %
others	804	0.03 %
_209_db		
java/util/Vector$1	2904257	99.89 %
java/lang/StringBuilder	2855	0.10 %
java/lang/String	255	0.01 %
spec/benchmarks/_209_db/Entry	80	0.00 %
java/lang/StringBuffer	52	0.00 %
others	9	0.00 %
_213_javac		
java/lang/StringBuffer	229436	64.83 %
spec/benchmarks/_213_javac/Context	48592	13.73 %
java/util/Vector$1	46593	13.16 %
java/lang/StringBuilder	10832	3.06 %
spec/benchmarks/_213_javac/Environment	8868	2.51 %
others	9598	2.71 %
_222_mpegaudio		
java/lang/StringBuffer	28	60.87 %
java/lang/String	14	30.43 %
spec/io/File	1	2.17 %
java/io/File	1	2.17 %
java/util/Vector$1	1	2.17 %
others	0	0.00 %

Class	Instances	Percent
_227_mtrt		
spec/benchmarks/_205_raytrace/Vector	2739890	92.65 %
spec/benchmarks/_205_raytrace/Color	102417	3.46 %
spec/benchmarks/_205_raytrace/IntersectPt	62417	2.11 %
java/lang/StringBuilder	25620	0.87 %
spec/benchmarks/_205_raytrace/Ray	25475	0.86 %
others	1338	0.05 %
_228_jack		
java/lang/String	1409614	31.96 %
java/util/Hashtable$EntryEnumerator	1321869	29.97 %
java/util/Hashtable$KeyEnumerator	1321869	29.97 %
java/lang/StringBuffer	347236	7.87 %
java/util/Vector$1	6104	0.14 %
others	3725	0.08 %

Table 5.3: Distribution of classes among stack objects (dynamic)

Class	Instances	Percent
antlr		
java/lang/StringBuffer	216645	98.18 %
java/io/BufferedReader	1596	0.72 %
java/io/File	1150	0.52 %
java/lang/String	409	0.19 %
java/util/Hashtable$EntryEnumerator	252	0.11 %
others	609	0.28 %
bloat		
EDU/purdue/cs/bloat/trans/NodeComparator$1	260112	85.81 %
EDU/purdue/cs/bloat/ssa/SSAGraph$6	14931	4.93 %
EDU/purdue/cs/bloat/trans/SSAPRE$13	9408	3.10 %
EDU/purdue/cs/bloat/trans/NodeComparator$2	6247	2.06 %
EDU/purdue/cs/bloat/diva/InductionVarAnalyzer$2	5046	1.66 %
others	7397	2.44 %
fop		
java/lang/StringBuffer	136734	85.16 %
java/util/StringTokenizer	13143	8.19 %
java/lang/Double	9976	6.21 %
java/util/ArrayList	419	0.26 %
java/lang/Character	256	0.16 %
others	35	0.02 %
hsqldb		
java/lang/StringBuffer	205	40.20 %
org/hsqldb/GroupedResult	200	39.22 %
java/util/Hashtable$EntryEnumerator	41	8.04 %
java/util/Hashtable$KeyEnumerator	41	8.04 %
java/lang/String	18	3.53 %
others	4	0.78 %
jython		
org/python/core/ArgParser	55322	68.45 %
java/util/ArrayList	22504	27.84 %
org/python/core/ReflectedCallData	1974	2.44 %
java/io/DataOutputStream	306	0.38 %
java/lang/StringBuffer	227	0.28 %
others	490	0.61 %
luindex		
java/util/ArrayList	33318	29.17 %
gnu/java/util/regex/RE$CharUnit	33318	29.17 %
java/lang/StringBuffer	27062	23.70 %
java/io/File	4995	4.37 %
java/util/Hashtable$EntryEnumerator	2606	2.28 %
others	12902	11.30 %
lusearch		
java/lang/StringBuffer	428089	60.86 %
java/util/Vector	262144	37.27 %
java/lang/StringBuilder	12663	1.80 %
java/io/File	424	0.06 %
org/apache/lucene/index/IndexReader$1	32	0.00 %
others	75	0.01 %

Benchmark	With EA	Without EA	Speedup
_200_check	0.05 s	0.03 s	-67.86 %
_201_compress	8.51 s	8.50 s	-0.19 %
_202_jess	46.01 s	55.81 s	17.56 %
_205_raytrace	20.32 s	34.32 s	40.80 %
_209_db	28.52 s	42.71 s	33.23 %
_213_javac	50.91 s	53.56 s	4.94 %
_222_mpegaudio	13.02 s	13.12 s	0.75 %
_227_mtrt	20.82 s	35.23 s	40.91 %
_228_jack	43.69 s	64.06 s	31.80 %

Table 5.5: Execution times with and without escape analysis (SPECjvm98)

Class	Instances	Percent
pmd		
org/jaxen/expr/IdentitySet$IdentityWrapper	3386708	98.19 %
net/sourceforge/pmd/symboltable/ImageFinderFunction	40464	1.17 %
java/lang/StringBuffer	19714	0.57 %
java/lang/StringBuilder	958	0.03 %
java/util/ArrayList	400	0.01 %
others	870	0.03 %
xalan		
java/util/StringTokenizer	27800	39.93 %
org/apache/xalan/templates/ElemNumber$NumberFormatStringTokenizer	16100	23.12 %
java/lang/StringBuffer	3605	5.18 %
java/util/Vector$1	3400	4.88 %
org/apache/xerces/parsers/SecuritySupport$4	3400	4.88 %
others	15318	22.00 %

Table 5.4: Distribution of classes among stack objects (dynamic)

5.3 Execution times

In the last set of benchmark runs, the impact of escape analysis and stack allocation on the run-times of the benchmarks is measured. For this purpose, the execution scheme was extended to run every benchmark 5 times without escape analysis and 5 times with escape analysis. All runs were performed after recompilation with the ad-hoc optimization framework. The run with the shortest execution time was taken into account. The results are listed in table 5.5, table 5.6, figure 5.3 and figure 5.4.

5.4 Comparison with dynamic algorithm

The dynamic nubers of stack allocated objects are compared with numbers obtained with the escape behaviour algorithm from chapter 3. The purpose

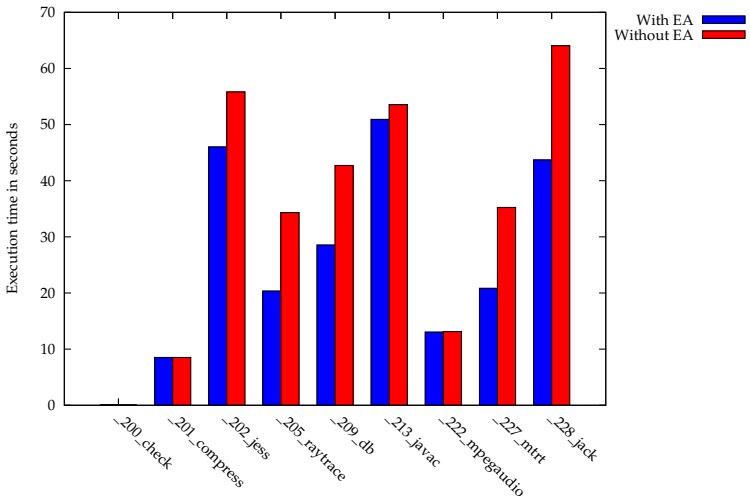


Figure 5.3: Execution times with and without escape analysis (SPECjvm98)

Benchmark	With EA	Without EA	Speedup
antlr	34.16 s	37.52 s	8.97 %
bloat	219.34 s	216.95 s	-1.10 %
fop	11.95 s	12.64 s	5.50 %
hsqldb	2.77 s	2.90 s	4.62 %
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	5.59 %
pmd	178.63 s	197.12 s	9.38 %
xalan	73.19 s	73.24 s	0.07 %

Table 5.6: Execution times with and without escape analysis (Dacapo)

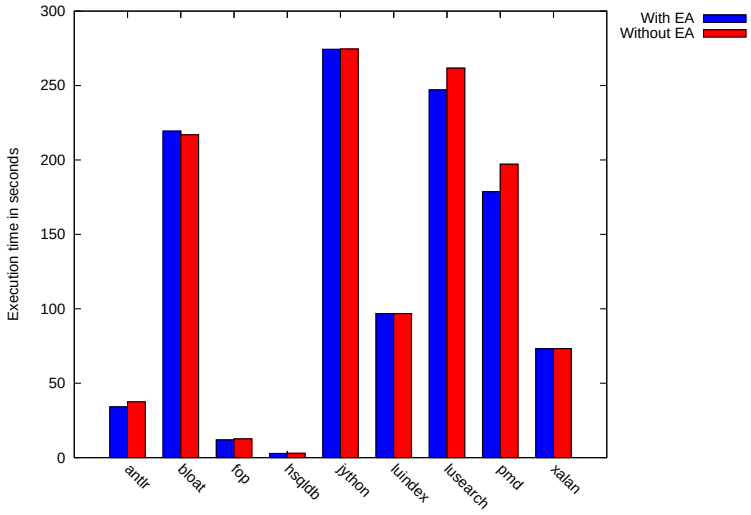


Figure 5.4: Execution times with and without escape analysis (Dacapo)

Benchmark	Inlining 1	Inlining 2	Stack allocated
_200_check	28.97 %	44.27 %	7.32 %
_201_compress	30.87 %	44.84 %	8.56 %
_202_jess	64.27 %	70.09 %	26.54 %
_205_raytrace	96.46 %	96.94 %	57.66 %
_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 5.7: Number of stack allocated objects compared to number of objects eligible for stack allocation (SpecJVM98)

of this comparison is to approximate, what the ratio of the number of stack-allocated objects is compared to the number of objects eligible for stack allocation. This comparison also expresses the loss of stack allocated objects caused by the conservative nature of static escape analysis.

The algorithm was adapted to gather statistics about objects only. Arrays are not counted, as in this implementation they are never allocated on the stack. Objects considered eligible for stack allocation are passed at most 1 or 2 frames upwards the stack. This very rougly simulates effects of inlining, where objects passed upwards the stack from inlined methods are captured by the caller.

The comparison can be seen in table 5.4 and table 5.4 for the SPECjvm98 and Dacapo benchmark suites respectively.

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 5.8: Number of stack allocated objects compared to number of objects eligible for stack allocation (Dacapo)

5.5 Discussion

The static count of stack allocations is not a metric of much interest. In the empiric evaluation, it was determined that there are much `StringBuilder` objects constructed at code paths, where error messages are composed. In practice these code paths get executed rather infrequently.

The expected dynamic number of stack allocated objects was around 10% in the context of chapter 3 and in the context of related work. In some SPECjvm98 benchmarks (`jess`, `raytrayce`, `db`, `mrtr`) the number of stack allocated objects is extremely high, coupled to a single class and leads to an extreme speedup. These extreme allocation sites were already observed in [16]: in `mrtr`, temporary `Vector` objects are allocated in a loop, in `db`, 2 temporary `Enumeration` objects are used to compare a pair of database records.

Much benchmarks profit of stack allocation of `StringBuffer` and `StringBuilder` objects. These objects are used to implement Java string concatenation. They almost never escape the creating method and can thus 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 especially visible in the dacapo `pmd` and `bloat` benchmarks.

In the SPECjvm98 benchmarks that allocate a huge ratio of stack objects, the execution time gains are significant. This is not only due to the benefits of stack allocation, but because the current memory management implementation in CACAO performs rather poorly: CACAO uses a conser-

vative garbage collector, the `new` instruction is not inlined but rather implemented in a C function which in turn is wrapped in a builtin stub. Another explanation for the extreme speedup is the simple nature of the SPECjvm98 benchmarks. In the more complex dacapo benchmarks, the speedups are more moderate, even with significant numbers of stack-allocated objects.

The more complex dacapo benchmarks show the limits of the current whole-program interprocedural analysis. The greater the program call graph becomes, the more difficult it becomes to find a favorable traversal, that visits every callee before its caller. Investigation reveals, that in the dacapo benchmarks with very low numbers of stack allocated objects, even `StringBuffer`s get heap allocated, exactly because of that problem.

Chapter 6

Related work

6.1 Static single-assignment form

The mainstream algorithm for computing SSA form was proposed by Cytron in [6]. It places the minimal number of ϕ functions exactly into basic blocks requiring them. It is based on the observation that SSA form is related to *dominance*: in an SSA form a definition always dominates its use. If at a program point a use of a variable is dominated by a single definition, the use can be linked to that definition. A point, where a definition *stops* dominating, may be reached by a different definition and thus exactly at that point a ϕ function is needed. This location is thus called *dominance frontier*.

The algorithm starts by computing a CFG and then builds a dominator tree. The dominator tree links the *immediate dominator* $idom(n)$ of a CFG node n to the node n. $idom(n)$ is the single dominator of n that does not dominate any other dominator of n. The dominator tree is used to determine for every node n its *dominance frontier* $DF[n]$. It is defined as the set of nodes w such that n dominates a predecessor of w, but does not strictly dominate w. Next, for every variable a ϕ functions are placed in a worklist algorithm. The worklist is initially populated by all nodes containing a definition of a. The worklist is iterated, in every step a node n is picked and all nodes Y in $DF[n]$ get a ϕ function for a, if they don't have already one. As the ϕ function is itself a definition, Y must be added to the worklist.

Once ϕ functions are placed, the variables can be renamed. The dominator tree is traversed node by node. In every node n, all instructions are processed: every use of some variable v is replaced by the most recent definition - the closest definition v_i that is above n in the dominator tree. If a definition of v is encountered it is renamed to v_j with j being a newly allocated variable index. To keep track of the most recent definition of a variable v a separate stack is maintained during the traversal: when visiting a node, every definition v_j of v is pushed onto the stack. After the node has been processed and the algorithm has recursed into children of the domina-

tor tree, the definitions are popped. The top of stack thus always contains the most recent definition of v. ϕ functions are treated separately: once processing of a node n is complete, the argument of every ϕ function corresponding to n in every successor following the CFG is adjusted to the most recent definition found in n: the one on the top of the stack.

Aycock and Horspool propose another algorithm [3] which places the minimal set of ϕ functions for reducible control flow graphs and a correct set of ϕ functions for non-reducible ones. It operates in two phases. In a *RC phase* every variable is split at every basic block boundary and a ϕ function for every variable is placed at every basic block. In a *minimization phase*, redundant ϕ functions are eliminated. They can be recognized by having one of the following forms:

1. $v_i = \phi(v_i, v_i, \ldots, v_i)$
2. $v_i = \phi(v_{x_1}, v_{x_2}, \ldots, v_{x_k})$ where $x_1, \ldots, x_k \in \{i, j\}$

The first form can be safely removed because it corresponds to the assignment $v_i \leftarrow v_i$ on all incoming edges which does not change program state in any way. The second form corresponds to the assignment $v_i \leftarrow v_i$ with no effect on some incoming edges and to the assignment $v_i \leftarrow v_j$ on other incoming edges. v_i can thus be safely replaced by v_j.

In order for the algorithm to perform well, some improvements are proposed. In the RC phase, when placing ϕ functions in a basic block, all its predecessor blocks must have already been processed. Information flowing into a block through back edges requires back patching. If however the variables are numbered in a way, that the last definition of v in a block B_i is numbered v_i, all definitions flowing into a block are known regardless of the order the blocks are traversed. In the minimization phase, literally replacing variable instances would be inefficient. The replacement relation is encoded in a mapping table through which every access to a variable instance is filtered through. A third improvement proposes to omit creation of ϕ functions at blocks with no predecessors, as they don't constitute join points.

The authors claim that their algorithm is significantly simpler than Cytron's algorithm. They have implemented it in a Modula compiler as a drop-in replacement. The algorithm was competitive with Cytron's one. They further argue, that entering SSA form takes such insignificant fraction of the total compile time, that a simpler algorithm is justified even if performing slightly worse.

Brandis and Mössenböck propose an algorithm [4] for generating SSA for structured languages. In contrast to the previous algorithms, their algorithm does not transform an IR into SSA form, but it generates an IR which is already in SSA form ready for optimization. The benefits are obvious: an intermediate step, time and memory are saved.

A structured language does not consist of arbitrary control flow, but the control flow is dictated by the patterns given by the control statements (if-then-else, repeat-while, for) of the compiled language. For every control statement the join node for that statement is known in advance. As control statements can be nested, they define the notion of the *current join node* - the one in the innermost statement. An assignment statement leading to a new definition for a variable causes a ϕ function to be created for that variable in the *current join node*.

Sreedhar and Gao [25] propose a linear time algorithm for placing ϕ functions which is based on a novel program representation called *DJ-Graph*. It consists of all nodes of the CFG and two classes of edges: D edges are all edges from the dominator tree, while J edges are edges from the original CFG which lead to potential *join nodes* where data flow information is merged. Thanks to the properties of the DJ-Graph, the iterated dominance frontier of a set of sparse nodes - nodes containing definitions of some variable - can be computed in time linear in the size of the input CFG, and this can in turn be used to place ϕ functions.

In [2], Appel shows a direct correspondence between SSA and a functional language. He shows, that a program in SSA form can be transformed into a set of mutually recursive functions. He then transforms the program in SSA form into a simpler form, using the concept of nested scopes found in languages like Scheme, ML and Haskell. This form corresponds directly to the SSA form with the minimal number of ϕ functions. He then argues, that the benefit of functional programming consists of *equational reasoning*, which a compiler can make good use of.

SSA can be found in current state of the art compilers.

The *GNU Compiler Collection* (GCC) as of version 4 makes use of SSA. The frontends generate GENERIC code, which is translated into the SSA based GIMPLE representation and processed by language independent optimizations. The back-end translates the GIMPLE representation into a low level RTL representation and finally into assembly code.

The *Low level virtual machine* (LLVM) is an infrastructure for building compilers and virtual machines. It features its own language-independent instruction set and its own type system. The instructions are in SSA form: every *typed register* can be assigned only once. SSA form is left only late in the compilation process.

The *Java HotSpot^{TM}client compiler for Java 6* [17] includes a Just-In-Time compiler to compile Java bytecode into native code. The compilation by translating bytecode into an *high-level* intermediate representation (HIR) via abstract interpretation. In the HIR, every value is represented by the instruction generating that value. A use of the value is represented by a pointer to the corresponding HIR instruction. Such a representation eliminates the need for an additional level of indirection via variable numbers and makes elimination of dead instructions especially easy: a dead instruc-

tion is transformed into a link to the instruction replacing it. The HIR is optimized, for example using constant folding, local value numbering, method inclining, null check elimination, conditional expression elimination. Next, the HIR is transformed into a low-level intermediate representation which is not in SSA form anymore. LIR operations use explicit virtual registers as operands in contrast to references to prior instruction. The virtual registers in the LIR are allocated using a linear scan register allocator and machine code is generated.

6.2 Escape analysis

[14] categories escape analysis algorithm 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.

The algorithms can be further categorized by their sensitiveness. A *flow-sensitive* analysis in contrast to a *flow-insensitive* analysis takes into account the order of statements in a program. A *context-sensitive* analysis takes into account the calling context when analyzing a callee site.

A prominent algorithm for escape analysis in Java, which is implemented in *Java HotSpot^{TM}server compiler for Java* is the one proposed by Choi et. al in [7]. With the algorithm a novel program representation, the *connection graph* (CG) is introduced.

The algorithm is context-sensitive and exists in both a flow-sensitive and a flow-insensitive variant. In an intraprocedural analysis the CG for a method is computed. The connection graph is used to calculate the escape state of objects created in the method. It is then collapsed into summary information that can be used in interprocedural analysis at call sites to that method.

The nodes of the CG represent objects and reference variables (locals, formals, class fields, instance fields). Special *phantom nodes* are used to represent objects that flow into the method from outside, like arguments, and are used as hooks for interprocedural analysis.

Nodes are annotated with an *escape state* which is one of *GlobalEscape* (globally escaping), *Local Escape* (not escaping the method) and *ArgEscape* (escaping the method via arguments, but not escaping the thread). The initial *escape state* for nodes is *LocalEscape*. For reference nodes representing static variables, it is initialized to *GlobalEscape* and for nodes representing arguments to *ArgEscape*.

To connect nodes of the CG, several classes of edges are used:

A points-to edge $p \xrightarrow{P} O$ connects a reference variable to an object and

indicates that p might point to O.

A deferred edge $p \xrightarrow{D} q$ connects two reference variables and corresponds to a copy of a reference variable. It indicates that p might point to any object that q points to. It *defers* computation and thus graph updates. A deferred edge $p \xrightarrow{D} q \xrightarrow{P} O$ can always be eliminated by *bypassing* q and creating a direct points-to edge $p \xrightarrow{P} O$.

A field edge $O \xrightarrow{F} f$ connects an object node with its instance field node f.

The CG can be computed separately for control flow paths. A CG at a join point results from the merge of the incoming CGs, which is simply defined as the union of the two CGs. The following statements are evaluated to update the CG:

`S: p = new T():` An object node for the statement S representing the newly created object and a reference node for p are created and connected using a *points-to* edge $p \xrightarrow{P} S$.

`S: p = q;` p and q are connected using a *deferred* edge $p \xrightarrow{D} q$.

`S: p.f = q;` The set $U = PointsTo(p)$ of all object nodes O that p might point to is determined. It is found by following paths $p \xrightarrow{+P} O$ consisting of *deferred* edges and one *points-to* edge at the end. For every node $O \in U$ a field node f, a field edge $O \xrightarrow{F} f$ and a *deferred edge* $f \xrightarrow{F} q$ are created.

`S: p = q.f;` Again, $U = PointsTo(P)$ is determined and all field nodes f such that there is a path $O \xrightarrow{+F} f$ for $O \in U$ are found. For every f a *deferred* edge $p \xrightarrow{D} f$ is added to the graph.

In the flow-sensitive variant whenever a statement of the form `p = q` is encountered, p is *bypassed* in the CG prior to creating the *deferred* edge $p \xrightarrow{D} q$: p's incoming *deferred* edges are redirected to its successor nodes.

At the method exit point, escape analysis is reduced to reachability analysis over the CG. Nodes reachable only from *NoEscape* nodes constitute the *LocalGraph* and are eligible for stack allocation. All nodes reachable from a *GlobalEscape* node are marked *GlobalEscape* and collapsed to a single bottom node. This is combined with the subgraph of nodes reachable from *ArgEscape* nodes to form the *NonLocalGraph* which will serve as summary information for the method in interprocedural analysis.

In interprocedural analysis, at a call site the callees summary information is mapped into the caller. The callees summary information contains

phantom nodes for the formal arguments and the return value. Argument passing is modeled as consisting of assignments of the actual arguments to the formal arguments of the callee and vice versa for the return value. The callees summary information is then mapped into the callers CG.

Gay and Steensgaard have implemented a whole program interprocedural escape analysis for Java programs [9] in an SSA based IR. An object is considered to escape if it is returned from a method, thrown as exception or assigned to a class or instance field. These rules are encoded as constraints on a type system, which can be solved linear time and space linear in the number of constraints for example by Rapid Type Analysis. They define a *fresh method* as one that returns a newly allocated object and a *fresh variable* as one that holds a directly (new) or indirectly (*fresh method*) newly created object. For each *fresh variable* it must be determined whether that assigned value may escape.

For every local variable v, two boolean properties are calculated: $escaped(v)$ determines whether v holds potentially escaping references, $returned(v)$ determines whether v hold references that potentially escape from the creating method by being returned. The property $vfresh(v)$ of a variable is a Java class C if v has assigned a *fresh variable* and references an object of exactly the type C, it is $\top$ if the variable is definitely not fresh and $\bot$ if the freshness is unknown. The property $mfresh(m)$ is a Java class C if the method is *fresh* and returns an object of exactly the type C, otherwise it is $\top$ or $\bot$.

Java statements may impose constraints on these properties, which usually have the form of an implication. For example:

- $v = new\ T()$ imposes that $T \leq vfresh(v)$
- $throw\ v$ imposes $true \Rightarrow escaped(v)$
- $v_0 = \phi(v_1 \ldots v_n)$ imposes $T \leq vfresh(v_0)$ and for every $i > 0\ escaped(v_0) \Rightarrow escaped(v_i)$, $returned(v_0) \Rightarrow returned(v_i)$

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. The value $escaped$ property of actual arguments is adjusted to the value of the respective formal arguments of the callee.

6.3 Memory management

In [8] a memory allocator supporting *thread local heaps* is developed. The aim is to partition the global heap into threads. A thread can then allocate in its own partition without synchronization and its partition can be

collected independently of other threads. For this purpose, objects need to be categorized into thread-local and global objects, which is traditionally achieved by static escape analysis.

The authors preferred dynamically monitoring heap objects to a static analysis for determining this categorization. They explain several drawbacks of escape analysis:

1. Conservative nature. An object es treating as globally escaping if it escapes on any path.

2. Objects are considered only by their allocation site. If an allocation sites produces mostly local objects, they are all identified as global.

3. If an object becomes global after a long lifetime, it is considered being global from the point of its creation.

To monitor objects, a *global* bit is associated with every object. Most objects start as local and thus their *global* bit is cleared initially. Some Java specific objects are global by nature, for example classes and threads. They are created with the *global* bit set.

Every time a new value is written into a field of reference type, a write barrier causes the *global* bit of the two objects participating on the write to be examined. If the write causes a local object to become reachable from a global one, the *global* bit is set in the local object and all its descendants using a depth-first search. For this procedure, no synchronization is required, as all the objects having the *global* bit cleared are reachable only from the current thread.

In [28], Tomsich evaluated scalability of Java memory management on high-performance multiprocessor systems. He gives a motivating example of a typical transaction processing system which spawns a separate thread for every transaction to be processed. In every transaction, temporary short-lived objects are allocated frequently to hold transaction state. In an example real-life application running on a very large multiprocessor machine he observed a degradation in response time starting with 50 clients and a sudden jump in response time for 70 and more clients. An investigation showed, that one third of execution time was spent trying to acquire the shared heap lock during object allocation.

To solve this scalability problem, he proposes two approaches:

First, to reserve a fragment of the heap for exclusive allocation for every thread, a *thread-local allocation buffer*, where the thread can allocate without the need for the global heap lock. This simple idea however causes additional fragmentation of the heap and may cause the VM to run out of memory, even if there is enough space in the areas exclusively reserved for thread local allocation.

Second, in an append-only heap implementation, allocation is implemented as an atomic read-increment-store sequence of the top-of-heap pointer. He examines the implementation of this sequence on the MIPS architecture in details and determines, that under heavy load a phenomenon he calls *cache line contention* can be observed: several processors acquire concurrently the cache-line containing the top of heap in a shared state during the read and then all but one loose it once a write to the cache line is performed, generating a lot of cache invalidations. He then fine-tunes the operation to force the cache logic to mark the cache line in a dirty-exclusive state prior to any atomic operation and let the cache coherency logic resolve the conflict, leading to much better scalability.

Chapter 7

Summary

In this thesis, the potential for optimizations using escape analysis has been explored. The subsystems targeted by escape analysis are memory management and synchronization. Once objects, that don't escape their creating thread are identified, they can be allocated in a thread specific heap, which does not require synchronization for allocation and which does not require a global stop-the-world for collection. If objects are bounded by the lifetime of a method, they can be allocated in a region associated with the given method or even in its stack frame, with very low allocation costs and zero collection costs. If objects stay method local they can be eliminated and their fields can be replaced with scalars.

Escape behavior in complex real-life Java programs has been studied and it was determined, that in fact much objects stay thread-local. Once objects are allocated, they are not passed much around stack frames: most thread-local objects are not returned farther than 3 stack frames upwards the call chain, and most thread-local objects are not passed deeper than 4 stack frames down the call chain. Only a minority of 10% - 20% of thread local objects is not passed at all upwards the call chain. This implies the best possible performance of an escape analysis algorithm which does not optimize objects returned from a method and justifies the benefit of inlining for escape analysis. It was further determined, that the size of regions associated with activation records tends to be rather small and has a bounded size, what favors stack allocation and a memory management scheme with stacked fixed sized regions.

Static-single assignment form has been added to CACAO for the purpose of implementing a flow-insensitive *Steensgaard*-style escape analysis. The analysis produces escape information for object allocation sites and summary escape information for method arguments and the return value which is reusable in interprocedural analysis in different call contexts. An experimental optimization framework to support escape analysis has been implemented. Escape information is used to implement allocation of ob-

jects in stack frames. As the size of a stack frame is statically determined, objects allocated in loops can be optimized only if the space they occupy can be reused in every loop iteration. To check this condition a loop analysis algorithm has been implemented.

The analysis and optimization are finally evaluated using the SPECjvm98 benchmark suite. The analysis identifies around 10% of method local objects in some benchmarks, and numbers as high as $> 90\%$ in some special benchmarks, which are chosen for stack allocation. The objects in question are mainly of the classes `String`, `StringBuilder`, iterator classes and extremely short-lived domain specific objects. A high speed-up is gained in the benchmarks that allocate an extremely high number of stack objects, which can be explained by the rather suboptimal implementation of the `new` primitive in the current CACAO codebase. An evaluation using the more complex dacapo benchmark suite leads to more realistic numbers: 6% to 18% of method-local objects coupled with speedups of of 5% to 9% in some benchmarks.

Bibliography

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

[2] APPEL, A. W. SSA is functional programming. *SIGPLAN Not. 33*, 4 (1998), 17–20.

[3] AYCOCK, J., AND HORSPOOL, R. N. Simple generation of static single-assignment form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction* (London, UK, 2000), Springer-Verlag, pp. 110–124.

[4] BRANDIS, M. M., AND MÖSSENBÖCK, H. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1684–1698.

[5] CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. Efficient and precise modeling of exceptions for the analysis of Java programs. *SIGSOFT Softw. Eng. Notes 24*, 5 (1999), 21–31.

[6] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (1991), 451–490.

[7] DEOK CHOI, J., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. Escape analysis for Java. In *In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (1999), ACM Press, pp. 1–19.

[8] DOMANI, T., GOLDSHTEIN, G., KOLODNER, E. K., LEWIS, E., PETRANK, E., AND SHEINWALD, D. Thread-local heaps for Java. *SIGPLAN Not. 38*, 2 supplement (2003), 76–87.

[9] GAY, D., AND STEENSGAARD, B. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction* (London, UK, 2000), Springer-Verlag, pp. 82–93.

[10] GOSLING, J., AND BOLLELLA, G. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java™ Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[12] HANS BOEHM. A garbage collector for C and C++, 2008. [Online; accessed 2008-10-23].

[13] HOARE, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM 17*, 10 (1974), 549–557.

[14] JONES, R., AND KING, A. C. 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* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 129–138.

[15] KERNIGHAN, B. W., RITCHIE, D., AND RITCHIE, D. M. *C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.

[16] KOTZMANN, T. *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. PhD thesis, Johannes Kepler University Linz, 2005.

[17] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008), 1–32.

[18] KRALL, A., AND GRAFL, R. CACAO – a 64 bit JavaVM just-in-time compiler. In *Workshop on Java for Science and Engineering Computation* (Las Vegas, June 1997), G. C. Fox and W. Li, Eds., ACM.

[19] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (1979), 121–141.

[20] LEZUO, R. R. Porting the CACAO virtual machine to POWERPC64 and Coldfire. Master's thesis, Technische Universität Wien, 2007.

[21] LEZUO, R. R., AND MOLNAR, P. Just in time compilers - breaking a VM. In *24C3 Tagungsband. Volldampf vorraus!* (2007), pp. 97–111.

[22] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[23] MÖSSENBÖCK, H. Adding static single assignment form and a graph coloring register allocator to the Java HotSpot™ client compiler. Tech. rep., Johannes Kepler University Linz, 2000.

[24] ONODERA, T., AND KAWACHIYA, K. A study of locking objects with bimodal fields. *SIGPLAN Not. 34*, 10 (1999), 223–237.

[25] SREEDHAR, V. C., AND GAO, G. R. A linear time algorithm for placing ϕ-nodes. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1995), ACM, pp. 62–73.

[26] STEINER, E. Adaptive inlining via on-stack replacement. Master's thesis, Technische Universität Wien, 2007.

[27] STROUSTRUP, B. *The C++ Programming Language*, third ed. Addison-Wesley Professional, February 2000.

[28] TOMSICH, P. R. *Implementation of Java virtual machines for high-performance multi-processor systems using cache-coherent non-uniform memory architectures*. PhD thesis, Technische Universität Wien, 2002.

[29] VITEK, J., HORSPOOL, R. N., AND KRALL, A. Efficient type inclusion tests. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1997), ACM, pp. 142–157.