Statically Determining Memory Consumption of Real-Time Java Threads

by

Ovidiu Gheorghioiu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 28, 2002

Certified by . . . .
Martin Rinard
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Statically Determining Memory Consumption of Real-Time Java Threads

by

Ovidiu Gheorghioiu

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2002, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In real-time and embedded systems, it is often necessary to place conservative upper bounds on the memory required by a program or subprogram. This can be difficult and error-prone process. In this thesis, I have designed and implemented two (related) compile-time analyses to addresses this problem. The first analysis computes a symbolic upper bound on the maximum number of allocations of each object, showing undetermined facts about the program as symbols. The second analysis determines objects in the program that may be allocated statically, without changing the semantics of the program. The symbolic expression is then simplified by removing factors for statically allocated objects. The overall result is a simplified procedure for computing conservative upper bounds on memory. Results on a number of benchmarks are provided.

Thesis Supervisor: Martin Rinard
Title: Associate Professor

Acknowledgments

First and foremost, I have to thank my supervisor, Martin Rinard, for providing me with the opportunity to work on this challenging project. Martin's unique combination of energy and patience, more than anything else, propelled this project to completion. I also owe big thanks to Alexandru Sălcianu, whose competent advice and prompt feedback were immensely useful.

More thanks go to all of the FLEX group members, who were there to answer my questions, and to my friends and family for supporting me through this challenging period. A special "thank you" goes to my mother, for her constant interest and support for my work in a field as remote for her own calling as possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When writing Java programs for embedded or real-time systems, there is often the need to accurately determine a conservative upper bound on the memory consumption of a program or program part before it executes. Embedded systems usually have a small, fixed amount of memory, and we want to know whether that amount will suffice for running a certain program. In many such systems, the correctness of such upper bounds is especially important, since errors may result in real-world consequences, such as a critical control system failing.

In the case of real-time systems, a second problem arises due to the Java's garbage collection, as the operation of the garbage controller conflicts with the strict constraints on the time required for critical operations. The Real-Time specification for Java [8] allows one to avoid garbage collection by having dynamic allocations in real-time threads done in preallocated, fixed-size "memory regions" . This means that in order to use that feature the programmer must determine how much memory each real-time thread will require in the worst case.

Currently, the only way to find an upper bound on the memory the program uses is by hand, through visually examining the source code. This can be difficult and error-prone, due to a number of reasons:

- **Implicit object allocations:** Many allocations, and calls leading to allocations, are implicit in Java, i.e., they are not readily visible in the source code.

Objects such as exceptions, strings and `StringBuffer`s are often allocated implicitly.

- **Many allocation sites:** Java relies heavily on objects and dynamic allocations, so the sheer number of allocation sites can be overwhelming.
- **Global reasoning:** To compute memory consumption, one needs to know how many times control reaches each method or allocation site during the execution of the *whole* program. This may violate abstraction boundaries between different parts of the program.

This paper presents two, related, static analyses which address these issues.

1.1 Symbolic Upper Bounds

The goal of the first analysis is to determine a symbolic upper bound on the memory dynamically allocated within a given program or program part. Finding numerical upper bounds is infeasible, for two reasons: 1) the amount of memory allocated may depend on run-time information that we do not know at compile time, such as values read from external input, and 2) we can prove that determining whether a conservative, finite upper bound exists is undecidable in the general case, as implied by the Halting Problem, even when all the inputs are known in advance.

Therefore, the analysis returns an expression representing the upper bound. The expression contains, as symbols, those aspects of the program which the analysis could not determine. These might include:

- bounds on inputs,
- bounds on the number of times complex loops are executed,
- bounds on the number of times recursions are executed, or
- bounds on other variables in the program (such as the result of a complex calculation).

With this analysis, calculating memory consumption becomes much easier and safer, as the programmer need only replace the unknowns with numerical values to obtain the desired bound. In some cases, when the analysis is able to determine every relevant aspect of the computation, we even provide a numerical value directly.

1.2 Static Object Allocations

The goal of the second analysis is to eliminate some of the dynamic allocations in the program by making them static, while preserving correctness. We accomplish this by conservatively constructing a symmetric "incompatibility" relation between allocation sites; essentially, two sites are incompatible if two objects allocated at those sites may be live at the same point in the execution of the program. In particular, when an allocation site is compatible (i.e., not incompatible) with itself, objects at that site can be safely allocated statically, in the same memory space, since no two such objects can be alive at the same time.

In practice, many object allocations in Java are "short-lived", that is, they are only used within a limited region of the program (for example, exceptions, iterators, `StringBuffer`s and sometimes collections). Our analysis is able to correctly make most of these allocations static. As a result, the symbolic upper bound computed in the first analysis is greatly simplified – unlike dynamic allocations, static allocations consume a constant amount of memory, regardless of how many times control flow reaches the allocation site. This is true even in the absence of garbage collection.

One additional goal for this analysis is to reduce the memory required by static allocations, since that memory will not be freed. Fortunately, our "incompatibility relation" approach readily enables that. If two different allocation sites that have been made static are mutually compatible, then objects allocated at the two sites can share the same statically allocated memory space, without altering program semantics. Optimizing the total memory for static allocations is, in fact, similar to the graph coloring problem; therefore, we use one of the known heuristics for minimal graph coloring to obtain a nearly-optimal static allocation scheme.

1.3 Implementation and Experience

I have implemented these analyses using the FLEX compiler infrastructure [1]. This has necessitated a thorough (and welcome) familiarization with the concepts and code base of FLEX.

The results of this implementation on a number of benchmarks have been encouraging. With the analysis in place, the burden the programmer faces is substantially reduced, and the task of determining accurate bounds on the memory footprint becomes largely automated. We can think of the compound of these two analyses as a single tool for computing such bounds. A typical usage pattern for this tool might consist of the following steps:

1. The programmer needs a conservative approximation of the memory his/her program will require.

2. The programmer runs our tool on the program, obtaining the desired approximation in symbolic form.

3. The programmer visually examines the program and reasons about numerical bounds on the unknowns in the symbolic expression (note that we indicate filenames and line numbers for each of those symbols). As mentioned, he or she might have to:

   - Determine lower/upper bounds on values read from external input.
   - Determine upper bounds on the number of times loops are executed.
   - Determine upper bounds on the number of times recursions are executed.
   - Determine lower/upper bounds on other values in the program that affect memory allocations.

4. Finally, the programmer replaces the unknowns with numerical values in the symbolic expression, obtaining the desired numerical bound on the memory footprint of the program.

This approach is clearly better than the unassisted examination of the source code, for several reasons. By reducing the problem to a number of well-defined unknowns in the program (we the relevant file and line number for each symbol), we give the programmer a goal-directed way of achieving the desired result. We lessen the need for global reasoning, thus making better use of abstractions; the only cases such reasoning is still needed is when placing bounds on global variables and on mutual recursions between two or more methods (however, note that mutual recursions are unlikely to cross abstraction boundaries). Finally, our static allocation analysis makes many program unknowns irrelevant (i.e., they no longer have an effect on memory footprint).

The two analyses have been integrated into the FLEX infrastructure for future use and enhancement by the group.

1.4 Document Structure

The rest of this document is organized as follows. Chapter 2 presents aspects of the FLEX compiler infrastructure relevant to the analysis. Chapter 4 presents the incompatibility analysis. Chapter 3 presents the symbolic analysis. Chapter 5 presents statistics of our analysis' performance on a number of benchmarks. Chapter 6 shows possible improvements to the analysis. Chapter 7 presents related work, and chapter 8 concludes.

Chapter 2

Background

The FLEX compiler infrastructure for Java [1] aims at providing comprehensive support for a range of compiler analyses and optimizations. It is almost completely implemented in Java, with native C code where required. The infrastructure has, among others, the following capabilities relevant to this project:

- Reading and linking Java compiled classes (as bytecode)
- Transforming Java bytecode into Intermediate Representations (IRs) amenable to efficient program analysis.
- Computing reasonably accurate *call graphs* for specified Java methods. It should be noted that Java's heavy use of dynamic dispatch poses significant challenges to the generation of accurate call graphs. By call graph accuracy we mean the ability to eliminate as many variants for dynamic dispatch as possible, while preserving program semantics.
- A number of already-implemented analyses (such as liveness, use-def chains and pointer analysis).
- A number of general data structures and algorithms, such as extensions to the Java Collection API (in particular, our implementation relies heavily on *multimaps*), computation of strongly connected components, etc.

Of the IRs that FLEX provides, we use exclusively the quadruple-based SSI representation (QuadSSI). The instructions, in this representation, are in quadruple form (as presented, for example, in [3]), having the general form $a \leftarrow b \oplus c$. b and c are source operands, either constants or variables, a is the destination operand, always a variable; finally, $\oplus$ is some binary operator. Quadruples also contain information about control-flow edges.

In a quadruple-based representation, variables do not directly correspond to program variables (although there is some overlap); instead, they represent storage for intermediate values of the computation, quite similar to registers.

The SSI (Single Static Information) form [4] is an extension of the more widely-used SSA (Single Static Assignment) form [5]. SSA simplifies program analysis by transferring some of the control flow information to variables; in particular, every use of a variable in the SSA form has a single reaching definition. To accomplish this, SSA form assigns unique names to unique static values of a variable, and introduces "variable-merging" ϕ-functions at program join points when necessary.

SSI adds further control-flow information to variables, by generating different names for the same variable on different control-flow paths. Thus, it "provides us with a one-to-one mapping between names and information about the variables at each program point" [4]. To preserve program semantics, SSI form requires the insertion of "variable-splitting" σ-functions at branches, in addition to the SSA ϕ-functions.

To further familiarize the reader with our Intermediate Representation, we show, in Figure 2-1 a list of quadruples relevant to our analysis.

Name	Format	Informal semantics
Copy	$v_1 = v_2$	copy a local variable into another
New	$v = \texttt{new}\ C$	create one object of class C
New Array	$v = \texttt{anew}\ C[v_1][v_2]\ldots[v_k]$	create a k-dimensional array of base class C and sizes $v_1, \ldots, v_k$
Store	$v_1.f = v_2$	assign a value to a field
Get	$v_1 = v_2.f$	read a value from a field
Return	$\texttt{return}\ v$	normal return from a method
Throw	$\texttt{throw}\ v$	exceptional return from a method
Call	$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$	method invocation
Phi	$v = \phi(v_1, \ldots, v_k)$	SSA ϕ nodes in join points
Typeswitch	$\langle v_1, v_2 \rangle = \texttt{typeswitch}\ v : C$	"instanceof" tests
Switch	$\texttt{switch}\ v : c_1, ..., c_{k-1}$ $(v_{11}, \ldots, v_{1k}) = \sigma(v_1)$ $\vdots$ $(v_{n1}, \ldots, v_{nk}) = \sigma(v_n)$	branch on the integer value of v (with σ-functions)
Conditional jump	$\texttt{cjmp}\ v$ $(v_{11}, v_{12}) = \sigma(v_1)$ $\vdots$ $(v_{n1}, v_{n2}) = \sigma(v_n)$	branch on the boolean value of v (with σ-functions)
Operation	$v_1 = v_2\ \mathit{binop}\ v_3$, or $v_1 = \mathit{unop}\ v_2$	arithmetic/logic operation

Figure 2-1: Instructions relevant for the analysis.

Chapter 3

Computing Symbolic Upper Bounds on Memory Consumption

In this chapter, we show a method of computing symbolic upper bounds on the memory required by a program or program part. Note that we currently do not take stack frames into account.

3.1 Overview

As mentioned in the introduction, obtaining an conservative, finite upper bound on the memory consumption of a program is, in general, undecidable, even when the inputs are known (and they are generally not). A short proof of this statement follows.

Consider a Java program P_1. Let P_2 be another Java program obtained from P_1 by adding an trivial `Object` allocation after each instruction. Clearly, P_2 run on any input x allocates a finite amount of memory if and only if P_1, run on the same input x, terminates after a finite amount of time. Hence, if we had a way to decide whether a Java program allocates a finite amount of memory on a given input, that would give us a way to decide the halting problem on any Java program (the choice of P_1 was arbitrary). The latter problem is known to be undecidable, which means the former is undecidable as well.

Therefore, we have not attempted to provide a numeric upper bound for the memory footprint of the analyzed program. Instead, we provide a symbolic upper bound, in terms of those characteristics of the program that we could not determine statically, such as inputs or potentially unbounded loops.

3.2 Basic algorithm

The main idea of the algorithm is rather simple: given the entry method of the program or thread that needs to be analyzed, we start with a generic symbolic expression that corresponds to an upper bound on the memory footprint of that method. If M_m is the entry method, the starting expression is $alloc\_record(M_m)$. *alloc_record* is a symbolic *primitive* which is detailed in one of the following sections. We progressively expand the starting expression according to a set of rules, until there is nothing more we can determine, and output the result. We use an expression cache (a table mapping expressions to their expanded equivalents) to avoid expanding sub-expressions more than once, and also to detect recursive expressions.

Symbolic expressions are arbitrary combinations of *constants*, *primitives*, and *operators*. Each of these, along with the rules we use for expanding them, is presented the following sections.

Figure 3-1 shows a pseudocode representation of the algorithm we use. The algorithm operates as follows. First, it initializes the expression cache to the empty table. Then,

3.3 Constants

The constants we allow in our symbolic expressions are values of the 32-bit Java `int` type and its shorter variants, `short` and `byte`. We do not, as of yet, consider the `long` type and the floating point types. In practice, values of these types seldom intervene in memory allocations.

A constant value is not expanded any further. However, operators involving con-

```
procedure symbolic_bound)(M : Method) : Expression
{
    cache := <empty;
    E := expression(alloc_record(M, <none>));
    E_exp := expand(E);
    return E_exp;
}

procedure expand(E : Expression) : Expression
{
    E_cached := cache_lookup(cache, E);
    if E_cached != <none> return E_cached;
    // prevent looping with recursive expansions
    cache_add(cache, E->recursion_value(E));
    // apply expansion rules
    E_exp := expand_one_level(E);
    // recursively expand each subexpression
    for each E_sub subexpression of E_exp
    {
        E_sub_exp := expand(E_sub_exp);
        subexpression_replace(E_exp, E_sub->E_sub_exp);
    }
    // avoid recursive expressions, leave them undetermined
    if E subexpression of E_exp return E
    // add final form
    cache_add(cache, E->E_exp);
    // return final form
    return E_exp;
}
```

Figure 3-1: Algorithm for symbolic expansions

stant value can undergo simplification, as shown in the next section.

3.4 Operators

Expressions can contain seven mathematical operators, namely $+$, $-$, $*$, $/$, `max`, `min` and `abs`, which retain their usual sense, and three pseudo-operators, `maxval`, `minval` and `subst`

The `maxval` and `minval` pseudo-operators are needed because of our focus on determining an upper bound, instead of an exact representation, of the program's memory consumption. For example, suppose we have determined that a method which takes two parameters, v_1 and v_2, will allocate $v_1 - v_2$ objects of type `Object`. Then, the upper bound of that method's memory consumption is linear in the maximum value of v_1 minus the minimum value of v_2. The `maxval` and `minval` operators express the notion of the maximum and minimum value of a subexpression throughout the execution of the analyzed subprogram.

The remaining operator, `subst`, is used to perform variable replacement when encountering method calls. It has the general form `subst`$(E, v_1 \to E_1, ..., v_n \to E_n$, denoting that the variables v_1 through v_n should be replaced in the "body" expression E with the corresponding expressions E_1 through E_n.

The seven mathematical operators are not expanded any further; however, they undergo one extra *simplification* step (for example, by replacing $0 * E$ with 0, or by combining constants). The `maxval` and `minval` pseudo-operators, are expanded in certain cases, according to the rules in Figure 3-2.

Note that the expansions for maximum and minimum of $*$ and $/$ operators are expressed in terms of the absolute values of their arguments. This is because we have no way of knowing the sign of the arguments in advance, if they are symbolic.

Finally, the expansion of the `subst` operator is obtained by replacing every occurrence of the variables in the "body" expression, after the latter has been expanded.

Expression	Expansion
$\texttt{maxval}(E_1 + E_2)$	$\texttt{maxval}(E_1) + \texttt{maxval}(E_2)$
$\texttt{minval}(E_1 + E_2)$	$\texttt{minval}(E_1) + \texttt{minval}(E_2)$
$\texttt{maxval}(E_1 - E_2)$	$\texttt{maxval}(E_1) - \texttt{minval}(E_2)$
$\texttt{minval}(E_1 - E_2)$	$\texttt{minval}(E_1) - \texttt{maxval}(E_2)$
$\texttt{maxval}(E_1 * E_2)$	$\texttt{abs}(\texttt{maxval}(E_1)) * \texttt{abs}(\texttt{maxval}(E_2))$
$\texttt{minval}(E_1 * E_2)$	$-\texttt{abs}(\texttt{maxval}(E_1)) * \texttt{abs}(\texttt{maxval}(E_2))$
$\texttt{maxval}(E_1/E_2)$	$\texttt{abs}(\texttt{maxval}(E_1))/\texttt{abs}(\texttt{minval}(E_2))$
$\texttt{minval}(E_1/E_2)$	$-\texttt{abs}(\texttt{maxval}(E_1))/\texttt{abs}(\texttt{minval}(E_2))$
$\texttt{maxval}(\texttt{max}(E_1, ..., E_n))$	$\texttt{max}(\texttt{maxval}(E_1), ..., \texttt{maxval}(E_n))$
$\texttt{minval}(\texttt{max}(E_1, ..., E_n))$	$\texttt{max}(\texttt{minval}(E_1), ..., \texttt{minval}(E_n))$
$\texttt{maxval}(\texttt{min}(E_1, ..., E_n))$	$\texttt{min}(\texttt{maxval}(E_1), ..., \texttt{maxval}(E_n))$
$\texttt{minval}(\texttt{min}(E_1, ..., E_n))$	$\texttt{min}(\texttt{minval}(E_1), ..., \texttt{minval}(E_n))$

Figure 3-2: Expansions for the `maxval` and `minval` operators

3.5 Primitives

Primitives are symbolic placeholders for aspects of the program that must be analyzed. Whenever possible, the analysis expands primitives to more accurate symbolic constructs. Those primitives that cannot be further refined will appear as such in the final symbolic expression corresponding to the memory footprint of the program.

We use six types of primitives: *allocation*, *array_allocation*, variable primitives, *looptimes*, *rectimes* and *alloc_record*.

3.5.1 Allocations

The simplest primitive is *allocation*. It stands for a single, non-array object allocation of a given class (for example, *allocation*(`String`) represents an allocation of a `String` object).

The *allocation* primitive is not further expanded. Two *allocation* primitives referring to the same class C are considered equal, and may undergo simplification on that basis. For example, the expression $E_1 * allocation(C) + E_2 * allocation(C)$, becomes, after simplification, $(E_1 + E_2) * allocation(C)$.

3.5.2 Array allocations

array_allocation primitives are similar to *allocation* primitives, with one important difference: they also contain information about the dimensions of the allocated array. As such, the general form of this primitive is $array\_allocation(C, E_1, ..., E_n)$. C is the runtime type of the array being allocated, which in FLEX encapsulates information about the number of dimensions (for example, `int[][][]`). E_1 through E_n are symbolic expressions for each of the n dimensions of the array. See section 3.5.6 for an explanation of how we create *array_allocation* primitives.

Just like *allocation* primitives, we do not expand *array_allocation* primitives. However, we expand their expression arguments, if possible. In this respect, *array_allocation* primitives are similar to regular operators, like $+$ or `max`, and in fact use the same implementation as these.

We consider two *array_allocation* primitives equal, for purposes of simplification, if and only if they refer to the same type and their expression parameters are equivalent.

3.5.3 Variable primitives

A variable primitive stands for the integer value of an intermediary variable a certain point in the program. Because we use the SSI form, a variable encapsulates information about the program point. Hence, a variable primitive simply points to the corresponding variable.

We expand a variable v by looking at the single definition (denoted by def_v of the corresponding FLEX `Temp`, as follows:

- If def_v is a copy instruction, $v = v'$, then the expansion of v is v'.
- If def_v is a binary operator, $v = v_1 \ binop \ v_2$, where $binop \in \{+, -, *, /\}$, then the expansion of v is $v_1 \ op \ v_2$.
- If def_v is a `const` instruction, $v =$ `const` C, then the expansion of v is the integer constant C.

- If v is defined in a σ function $(v_1, ..., v, ..., v_k) = \sigma(v_\sigma)$, contained in def_v, we have several sub-cases, since we are aiming to use the single-information property of SSI for a more accurate expansion:
  - If def_v is a `switch` instruction, and its index variable is v_σ, then we expand v to the integer constant corresponding to the branch that defines v.
  - If def_v is a `cjmp` instruction, and its condition is a comparison between v_σ and some other variable v_{comp}, then we expand v to either $\max(v_\sigma, v_{comp})$ or $\min(v_\sigma, v_{comp})$, on the type of the comparison. and the `cjmp` branch which corresponds to v. For example, if the condition is $v_\sigma < v_{comp}$ and v is defined on the positive branch of def_v, we expand v to $\texttt{min}(v_\sigma, v_{comp})$.
  - Otherwise, we expand v to v_σ.

A special case occurs when v is the result of a ϕ function, $v = \phi(v_1, ..., v_k)$. Note that the above expansions are all exact (we know that v is necessarily equal to the expansion); however, the exact value of a ϕ function is not easily expressed symbolically. Instead, we want to take either the minimum or the maximum of the ϕ function's arguments, depending on what kind of bound (lower or upper, respectively) we are currently trying to place on v. Since that information is expressed by a `minval` or `maxval` operator that is above v in the symbolic expression, we do not expand v by itself, instead defining two more expansions for `maxval` and `minval`:

$$\texttt{maxval}(v), \text{ where } v = \phi(v_1, ..., v_k) \longrightarrow \texttt{max}(\texttt{maxval}(v_1), ..., \texttt{maxval}(v_k))$$

$$\texttt{minval}(v), \text{ where } v = \phi(v_1, ..., v_k) \longrightarrow \texttt{min}(\texttt{maxval}(v_1), ..., \texttt{maxval}(v_k))$$

Note that the ϕ expansions presented above may lead to recursive expansions of `maxval` or `minval` operators. Without going into details, a loop with a non-trivial induction variable, $v = f(v)$, but with a condition that does not depend on v, will cause an expansion like $\texttt{maxval}(v_1) \longrightarrow \texttt{max}(\texttt{maxval}(v_0), \texttt{maxval}(v_1))$, where v_0 corresponds, in SSI form, to the initial value of v, and v_1 corresponds to the value of v within the loop. To avoid unbounded growth of symbolic expressions, we do not allow recursive expansions, that is, we do not allow an expansion $E \longrightarrow E'$ if E' contains E as a subexpression. In the above example, the expression $\texttt{minval}(v_1)$ will not be expanded and will be reported unknown.

3.5.4 Loop execution times

The *looptimes* primitive serves as a symbolic placeholder for the maximum number of times a loop is executed, once control flow reaches that loop (note that the upper bound is implicit in the definition, as we never need lower bounds on loop execution times). By loops we mean *natural loops*, as defined in [3]. FLEX allows us to break a method into a hierarchy of natural loops, such that two loops are either disjoint or one of them is entirely contained in the other, with the method itself as the top-level loop. The *looptimes* primitive will always be expanded to 1 for the top-level loop. See section 3.5.6 for an explanation of how we create *looptimes* primitives.

We are currently able to determine integer `for` loops and loops similar to `for`, that is, loops that satisfy the two conditions below:

- The condition for continuing the loop is a comparison between an induction variable of integer type (the loop index) and a fixed final value variable whose value is not modified throughout the loop).
- During every loop iteration, the loop index is incremented exactly once, by a fixed increment.

For such a loop L, with the index variable v, an initial value v_0 (in SSI, v_0 is a variable corresponding to v before the loop), a final value v_1 and an increment of v_i, we expand *looptimes*(L) to an symbolic expression in v_0, v_1 and v_i, as shown in Figure 3-3. Note that we use $\texttt{max}(0, E)$, where E is a possible expansion, to ensure that the expansion of *looptimes* is positive (a loop cannot execute a negative number of times).

Loop condition	Expansion
$v < v_1$	$\texttt{max}(0, \texttt{maxval}(v_1) - \texttt{minval}(v_0))/\texttt{minval}(v_i)$
$v <= v_1$	$\texttt{max}(0, \texttt{maxval}(v_1) - \texttt{minval}(v_0) + 1/\texttt{minval}(v_i)$
$v > v_1$	$\texttt{max}(0, \texttt{maxval}(v_0) - \texttt{minval}(v_1))/\texttt{minval}(v_i)$
$v >= v_1$	$\texttt{max}(0, \texttt{maxval}(v_0) - \texttt{minval}(v_1))/\texttt{minval}(v_i)$
$v! = v_1$	$\texttt{max}(0, (\texttt{maxval}(v_1) - \texttt{minval}(v_0))/\texttt{minval}(v_i)$

Figure 3-3: *looptimes* expansions for `for`-loops

3.5.5 Recursion

We handle recursions through the addition of a special primitive, *rectimes*.

As the first step in our analysis, we use the call graph corresponding to the entry method to find groups of mutually recursive methods. These correspond to the strongly connected components of the call graph. Let SCC denote the set of strongly connected components; each component $\mathcal{C} \in SCC$ is in turn a set of methods that belong to that component.

The *rectimes* primitive takes as its single argument a strongly connected component $\mathcal{C}$, and stands for the maximum number of times any method $M \in \mathcal{C}$ might recursively enter itself once control flow enters that component.

We only expand $rectimes(\mathcal{C})$ in one case, namely when $\mathcal{C}$ consists of a single, non-recursive method. In this case $rectimes(\mathcal{C})$ expands to 1, since the method will only execute once. We do not currently have a way of determining any "proper" recursions; in case the analyzed program contains such a recursion, the corresponding *rectimes* primitive will be reported as undetermined.

3.5.6 Allocation Record

The *alloc_record* primitive stands for the maximum number of allocations that might occur when a method is called. Its general form is $alloc\_record(M, \mathcal{C})$, where M is a method and $\mathcal{C}$ is the strongly-connected component from which M was called. We need the $\mathcal{C}$ parameter to avoid multiple *rectimes* primitives for the same strongly connected component.

Instruction at label lb	$instralloc(lb)$
$v = \texttt{new}\ C$	$allocation(C)$
$v = \texttt{anew}\ C[v_1][v_2]\dots[v_k]$	$array\_allocation(C, \texttt{maxval}(v_1), \dots, \texttt{maxval}(v_k))$
$\langle v_N, v_E \rangle = v_1.mn(v_2, \dots, v_k)$	$\max_{m \in CG(lb)} \{\texttt{subst}(alloc\_record(m, \mathcal{C}_M), p_1 \to v_1, \dots, p_k \to v_k)\}$, where $p_1, \dots, p_k$ are the declared parameters of m.
all others	0

Figure 3-4: Computation of $instralloc(lb)$

This primitive is always expanded, by analyzing the method it refers to (since we cache expansions, each method is analyzed at most once). To compute the expansion, we scan the method for instructions that may cause allocations, directly or indirectly, and for each of those we add to the expansion a symbolic expression describing the allocations that take place at that site, multiplied by a *looptimes* expression for each loop that contains that instruction.

More formally, let $instralloc(lb)$, with $lb \in Label(M)$, denote a symbolic expression describing an upper bound on the allocations caused by the instruction at label lb. Also let $\mathcal{C}_M \in SCC$ denote the strongly connected component of M. We compute *instralloc* according to the table in Figure 3-4. Then we expand $alloc\_record(M, \mathcal{C})$ as follows:

$$alloc\_record(M, \mathcal{C}) \longrightarrow \sum_{lb \in Label(m)} \left(instralloc(lb) \cdot \prod_{L \in Loops(M), L \ni lb} looptimes(L) \right)$$

Furthermore, if M's strongly connected component $\mathcal{C}_M$ differs from the $\mathcal{C}$ argument of $alloc\_record(M, \mathcal{C}_M)$, the expression above is multiplied by an additional factor of $looptimes(\mathcal{C})$, to account for multiple executions of M due to recursion.

3.6 Summary

The symbolic upper bound analysis we have presented in this chapter greatly reduces the work required from the programmer to compute worst-case memory consumption. It takes as input a program or program part, specified by its entry method, and outputs a symbolic expression specifying a conservative approximation of the memory footprint of that portion of code. The expression contains, as symbols, unknown or hard-to-compute facts about the behavior of the program that affect its memory consumption. By replacing those symbols with appropriate numerical values, the programmer can obtain a safe, numerical upper bound.

Chapter 4

Computing Static Allocations

This chapter presents a static analysis that finds object allocation sites that can be transformed to static allocations, without changing program semantics[1]. Furthermore, the analysis groups preallocated sites into a small number of classes, such that all of the objects in a class can use the same preallocated memory, again while preserving program semantics. In our benchmarks, the analysis is able to identify, on average, over 60% of the allocation sites as static. Grouping those sites into compatible classes resulted in a reduction of over 80% in the amount of memory needed for static preallocations.

Note that this analysis only considers single object allocations. We do not attempt to statically preallocate arrays, as this would require exact knowledge of the size of the array.

4.1 Overview

Given a subprogram P, the analysis identifies all pairs of *incompatible* allocation sites, i.e., pairs of sites such that one object allocated at the first site and one object allocated at the second one may be live at the same time in some possible execution of P. An object is live if any of its fields/methods is used in the future. It is easy to

[1]The text in this chapter borrows heavily from an article submission by myself, Alexandru Sălcianu and Martin Rinard to the Static Analysis Symposium 2002. The tile of the article is "Interprocedural Incompatibility Analysis for Static Object Preallocation".

see that two allocation sites are incompatible if one object from one site is live in the program point that corresponds to the other site.

To identify the objects that are live at a program point, the analysis needs to track the use of objects throughout the program. This is a difficult whole-program analysis. First, we have an interesting abstraction problem: the program may create an unbounded number of objects, that need to be abstracted by a bounded number of analysis objects. Next, some parts of the program might read heap references created by other parts. Tracking objects through the heap requires an expensive pointer analysis. Here are the simplifications we do, in order to obtain a more approachable problem:

- We use the *object allocation site* model: all objects allocated by a given statement are modeled by an *inside node*[2] attached to that statement / program label. *INode* is the set of all inside nodes; n^I_{lb} is the inside node for the allocation site from label lb.

- The analysis tracks only the objects pointed to by local variables. Nodes whose addresses may be stored into the heap are said to *escape into the heap*. The analysis conservatively assumes that such a node is live for the rest of the program.

- To simplify the object liveness analysis, we conservatively assume that any object returned from a method is used after the method returns.

With these assumptions, a node is live at a given program point if it is pointed to by a variable that is live at that program point. Variable liveness is a well-studied data-flow analysis [3, 5] and we do not present it here. As a quick reminder, a variable v is live at a program point iff there is a path through the control flow graph that starts at that program point, does not contain any definition of v and ends in an instruction that uses v. It should also be noted that the SSI form seems to require linear time in practice [4].

[2]We use the adjective "*inside*" to make the distinction from the "*parameter*" nodes that we introduce later in the chapter.

Additionally, a node that escapes into the heap is live at a program point lb if the above condition is true, or if lb is accessible on a control flow path from any of the program points where the node escapes.

The analysis has to process call instructions accurately. For example, it needs to know the nodes returned from a call and the nodes that escape into the heap during the execution of an invoked method. Re-analyzing each method for each call instruction (which corresponds, conceptually, to inlining that method) would be inefficient. Instead, we parameterize the results that the analysis computes for every method by introducing *parameter nodes.* A method with k parameters of object type[3] has k parameter nodes: $n_1^P, \ldots, n_k^P$; these nodes are placeholders for the nodes passed as actual arguments. When the analysis processes a call instruction, it replaces the parameter nodes with the nodes sent as arguments. Hence, the analysis is *compositional*: a method is abstracted by some analysis results that are computed once and used multiple times. $PNode$ is the set of parameter node and $Node = INode \cup PNode$ is the set of all nodes. When analyzing a method M, the analysis scope is the method and all the methods it transitively calls. The inside nodes models the objects allocated in this scope, while the parameter nodes models the objects that M receives as arguments.

The analysis has two steps, each one an analysis in itself. First, it computes the objects live at each relevant label. Next, it uses the liveness information to compute the incompatibility pairs. The labels that are relevant for the second step (the object liveness analysis) are those labels that appear in the constraints for the second step, i.e., those corresponding to new, call and get instructions[4].

Each of these analyses is formulated as a group of set constraints, and examines the methods in a bottom-up way, starting with the leaves of the call graph. For each method M, we examine the relevant instructions, generate a set of constraints and then we find their least fixed point by using an iterative fixed point algorithm. The analysis results for M use parameter nodes to abstract over the calling context.

[3]I.e., not primitive types such as int, char etc.

[4]The object liveness analysis is able to find the live nodes in any label; however, for efficiency reasons, we want to study only the relevant ones.

Normally, each method is analyzed once. However, some of the constraints for M may use the results of the analysis for the methods called from M. Therefore, each set of mutually recursive methods requires a fixed point algorithm. For a given program, the number of nodes is bounded by the number of object allocation sites and the number of parameters. Hence, as all our constraints are monotonic, all fixed point computations are guaranteed to terminate.

Once we have computed the incompatibility relation, we mark as static every whose corresponding node to a node is self-compatible, i.e., not incompatible with itself. Based on the incompatibility relation between different nodes, we group nodes into compatible classes, such that all allocation sites corresponding to nodes in the same class use the same preallocated memory.

4.2 Object Liveness Analysis

Consider a method M a label/program point *lb* inside it, and let *live*(*lb*) be the set of inside and parameter nodes that are live at that point. We conservatively consider that a node is live at *lb* if it is pointed to by one of the variables live at that point, or if it escaped at a point *lb*' such that control flow path from *lb*' to *lb* exists.

$$live(lb) = \bigcup_{v \text{ live in } lb} P(v) \quad \cup \quad \bigcup_{\substack{lb' \in Label(M) \\ lb' \rightsquigarrow lb}} E(lb')$$

where $P(v)$ is the set of nodes that may be pointed to by v, and $E(lb')$ is the set of nodes that may escape at the instruction at *lb*'; finally, $lb' \rightsquigarrow lb$ is the reachability relation, true iff there is a path from lb' to lb in the control flow graph.

To be able to process the calls to M, we need to compute the set of nodes that can be normally returned from M, $R_N(M)$, the set of exceptions thrown from M, $R_E(M)$, and the set of parameter nodes that may escape into the heap during the execution of M, $E(M)$. More formally, here are the functions we want to compute:

$$\begin{aligned} P &: V \rightarrow \mathcal{P}(\mathit{Node}) \\ R_N, R_E &: \mathit{Method} \rightarrow \mathcal{P}(\mathit{Node}) \\ E &: \mathit{Method} \rightarrow \mathcal{P}(\mathit{PNode}) \end{aligned}$$

Suppose we analyze method $M \in \mathit{Method}$, which has k parameters $p_1, p_2, \ldots, p_k$. At the beginning of the method, p_i points to the parameter node n_i^P. A COPY instruction "$v_1 = v_2$" sets v_1 to point to all nodes that v_2 points to; accordingly, the analysis generates the constraint $P(v_1) = P(v_2)$ [5]. The case of a PHI instruction is similar. A NEW instruction from label lb, "$v = \texttt{new}\ C$", makes v point to the inside node n_{lb}^I attached to that allocation site. The constraints generated for RETURN add more nodes to $R_N(M)$; similarly, the constraints generated for THROW add nodes to $R_E(M)$. A SET instruction "$v_1.f = v_2$", causes all the nodes pointed to by v_2 to escape into the heap. Accordingly, we add the nodes from $P(v_2)$ to $E(M)$.

A TYPESWITCH instruction "$\langle v_1, v_2 \rangle = \texttt{typeswitch}\ v : C$" works as a type filter: v_1 points to those nodes from $P(v)$ that may represent objects of a type that is a subtype of C, while v_2 points to those nodes from $P(v)$ that may represent objects of a type that is not a subtype of C. In Figure 4.2, $ST(C)$ denotes the set of all subtypes (i.e., Java subclasses) of C (including C). We can precisely determine the type $\mathit{type}(n_{lb}^I)$ of an inside node n_{lb}^I by examining the NEW instruction from label lb. Therefore, we can precisely distribute the inside nodes between $P(v_1)$ and $P(v_2)$. As we do not know the exact types of the objects represented by the parameter nodes, we conservatively put these nodes in both sets[6].

A CALL instruction "$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$" sets v_N to point to the nodes that may be returned from the invoked method(s). For each possible callee $M' \in CG(lb)$, we have to include the nodes from $R_N(M')$ into $P(v_N)$. As $R_N(M')$ is

[5]As we use the SSI form, this is the only definition of v_1; therefore, we do not lose any precision by using "=" instead of "$\supseteq$".

[6]A better solution would be to consider the declared type C_p of the corresponding parameter and check that C_p and C have at least one common subtype.

Instruction at label *lb* in method M	Generated constraints
method entry	$P(p_i) = \{n_i^P\},\ \forall 1 \leq i \leq k,$ where $p_1, \ldots p_k$ are M's parameters.
$v_1 = v_2$	$P(v_1) = P(v_2)$
$v = \phi(v_1, \ldots, v_k)$	$P(v) = \bigcup_{i=1}^{k} P(v_i)$
$v_1.f = v_2$	$E(lb) = P(v_2)$
$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$	
	$P(v_N) = \bigcup_{M \in CG(lb)} R_N(M)\text{<}P(v_1), \ldots, P(v_k)\text{>}$ $P(v_E) = \bigcup_{M \in CG(lb)} R_E(M)\text{<}P(v_1), \ldots, P(v_k)\text{>}$ $E(lb) = \bigcup_{M' \in CG(lb)} E(M')\text{<}P(v_1), \ldots, P(v_k)\text{>}$
$v =$ `new` C	$P(v) = \{n_{lb}^I\}$
$\langle v_1, v_2 \rangle =$ `typeswitch` $v : C$	
	$P(v_1) = \{n^I \in P(v) \mid type(n^I) \in ST(C)\} \cup \{n^P \in P(v)\}$ $P(v_2) = \{n^I \in P(v) \mid type(n^I) \notin ST(C)\} \cup \{n^P \in P(v)\}$
`return` v	$R_N(M) \supseteq P(v)$
`throw` v	$R_E(M) \supseteq P(v)$
$E(M) = \bigcup_{lb \in Label(M)} E(lb)$	

Figure 4-1: Constraints for the object liveness analysis.

a parameterized result, before using it, we have to instantiate $R_N(M')$ by replacing each n_i^P with the nodes from $P(v_i)$. The case of v_E is analogous. The execution of the invoked method *M'* may also cause some of the nodes passed as arguments to escape into the heap. Accordingly, the analysis generates a constraint that adds the set $E(M)'{<}P(v_1), \ldots, P(v_k){>}$ to $E(M)$.

Below is a formal definition of the previously mentioned instantiation operation: if $S \subseteq \mathit{Node}$ is a set that contains some of the parameter nodes $n_1^P, \ldots, n_k^P$ (not necessarily all), and $S_1, \ldots, S_k \subseteq \mathit{Node}$, then

$$S{<}S_1, \ldots S_k{>} = \{n^I \in S\} \;\cup\; \bigcup_{n_i^P \in S} S_i$$

4.3 Computing the Incompatibility Pairs

Once we have computed the object liveness information, the analysis detects the set of incompatible allocation sites $I_G \subseteq \mathit{INode} \times \mathit{INode}$ [7]. As previously noted, the incompatibility pairs are required for making allocation sites static and for the construction of the compatibility classes.

Figure 4-2 presents the constraints used to compute I_G. An allocation site from label *lb* is incompatible with all the allocation sites whose corresponding nodes are live at that program point.

However, as some of the nodes from *live*(*lb*) may be parameter nodes, we cannot generate all incompatibility pairs directly. Instead, for each method M, the incompatibility pairs involving one parameter node are collected into a set $I(M)$ that we instantiate at each call to M, similar to the way we instantiate $R_N(M)$, $R_E(M)$ and $E(M)$:

$$I(M){<}S_1, \ldots, S_k{>} = \bigcup_{\langle n_i^P, n\rangle \in I(M)} S_i \times \{n\}$$

(S_i is the set of nodes that the ith argument sent to M might point to). Notice that some S_i may contain a parameter node from M's caller. However, at some point into the call graph, each incompatibility pair will involve only inside node and will

[7]Remember that there is a bijection between the inside nodes and the allocation sites.

Instruction at label lb in method M	Generated constraints
$v = \texttt{new}\ C$	$live(lb) \times \{n^I_{lb}\} \subseteq J(M)$
$\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$ $\swarrow \quad \searrow$ $succ_N(lb) \qquad succ_E(lb)$	$\forall M \in CG(lb),$ $I(M)\text{<}P(v_1), \ldots, P(v_k)\text{>} \subseteq J(M)$ $(live(lb) \cap live(succ_N(lb))) \times A_N(M) \subseteq J(M)$ $(live(lb) \cap live(succ_E(lb))) \times A_E(M) \subseteq J(M)$

$$\begin{aligned}
&\forall M \in Method,\\
&\quad J(M) \cap (INode \times INode) \subseteq I_G\\
&\quad J(M) \setminus (INode \times INode) \subseteq I(M)
\end{aligned}$$

Figure 4-2: Constraints for computing the set of incompatibility pairs.

be passed to I_G.

To simplify the equations from Figure 4-2, for each method M, we compute the whole set of incompatibility pairs $J(M)$. After $J(M)$ is computed, the pairs that contain only inside nodes are put in the global set of incompatibilities I_G; the pair that contains a parameter node are put in $I(M)$. For efficiency reasons, our implementation would does this separation "on the fly", as soon as an incompatibility pair is generated, without the need for $J(M)$.

In the case of a CALL instruction, we have two kinds of incompatibility pairs. We've already mentioned the first one: the pairs obtained by instantiating $I(M), \forall M \in CG(lb)$. In addition, each node that is live "over the call" (i.e., before and after the call) is incompatible with all the nodes corresponding to the allocation sites from the invoked methods. To increase the precision, we treat the normal and the exceptional exit from the called method separately. Let $A_N(M), A_E(M) \subseteq INode$ be the sets of inside nodes that may be allocated during an invocation of the method M that returns normally, respectively with an exception. We describe later how to compute these sets; for the moment we suppose the analysis computes them right before starting to generate the incompatibility pairs. Let $succ_N(lb)$ be the successor corresponding to the normal return from the call instruction from label lb. The nodes from $live(lb) \cap live(succ_N(lb))$ are incompatible with all nodes from $A_N(M)$. A similar

Instruction at label lb in method M	Condition	Generated constraints
$v = \texttt{new}\ C$	$lb \rightsquigarrow \texttt{return}$	$n^I_{lb} \in A_N(M)$
	$lb \rightsquigarrow \texttt{throw}$	$n^I_{lb} \in A_E(M)$
$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$		
	$succ_N(lb) \rightsquigarrow \texttt{return}$	$A_N(M) \subseteq A_N(M), \forall M \in CG(lb)$
	$succ_N(lb) \rightsquigarrow \texttt{throw}$	$A_N(M) \subseteq A_E(M), \forall M \in CG(lb)$
	$succ_E(lb) \rightsquigarrow \texttt{return}$	$A_E(M) \subseteq A_N(M), \forall M \in CG(lb)$
	$succ_E(lb) \rightsquigarrow \texttt{throw}$	$A_E(M) \subseteq A_E(M), \forall M \in CG(lb)$

Figure 4-3: Constraints for computing A_N, A_E.

relation holds for $A_E(M)$.

Computation of $A_N(M)$, $A_E(M)$

Given a label lb from the code of some method M, we define the predicate "$lb \rightsquigarrow \texttt{return}$" to be true iff there is a path in CFG$_M$ from lb to a RETURN instruction (i.e., the instruction from label lb may be executed in an invocation of M that returns normally). Analogously, we define "$lb \rightsquigarrow \texttt{throw}$" to be true iff there is a path from lb to a THROW instruction. Computing these predicates is a trivial graph reachability problem. For a method M, $A_N(M)$ contains each inside node n^I_{lb} that corresponds to a NEW instruction at label lb such that $lb \rightsquigarrow \texttt{return}$. In addition, for a CALL instruction from label lb in M's, if $succ_N(lb) \rightsquigarrow \texttt{return}$, then we add all nodes from $A_N(M)$ into $A_N(M)$, for each possible callee M. Analogously, if $succ_E(lb) \rightsquigarrow \texttt{return}$, $A_E(M) \subseteq A_N(M)$. The computation of $A_E(M)$ is similar. Figure 4-3 formally presents the constraints for computing the sets $A_N(M)$, $A_E(M)$.

4.4 Computing Static Allocations and Static Allocation Classes

Let S denote the set of self-compatible nodes, as per the incompatibility relation computed above:

$$S = \bigcup_{\substack{n^I \in INode \\ \langle n^I, n^I \rangle \notin I_G}} \{n^I\}$$

Allocation sites corresponding to nodes in S can safely be made static, since we can prove no two objects allocated at one of these sites are live at the same time.

To make multiple static allocations use the same preallocated memory in an optimal way, we need to compute a set *Classes* of "compatible" classes that satisfies the conditions below, and has minimal cardinality:

$$\bigcup_{\mathcal{C} \in Classes} \mathcal{C} = S$$
$$\forall \mathcal{C}_1, \mathcal{C}_2 \in Classes,\ \mathcal{C}_1 \cup \mathcal{C}_2 = \emptyset$$
$$\forall \mathcal{C} \in Classes, \forall n_1^I, n_2^I \in \mathcal{C},\ \langle n_1^I, n_2^I \rangle \notin I_G$$

Consider a graph $G(V, E)$ such that its vertex set V is the set S of self-compatible nodes, and its set of edges E is given by the incompatibility relation between these nodes: $\langle n_1^I, n_2^I \rangle \in E$ iff $\langle n_1^I, n_2^I \rangle \in I_G$. We can see that computing "compatible classes", as per the requirements above, is the same as determining an optimal[8] graph coloring of G; the compatible classes are simply the sets of inside notes marked with the same color.

The graph coloring problem is provably NP-complete; however, a number of fast heuristics are known to give nearly optimal results in practice. Our implementation uses the *DSATUR* heuristic [9], and achieved in our benchmarks more than 80% reduction in the memory needed for static allocations.

4.5 Simplifying Symbolic Upper Bounds on Memory Consumption

The analysis we have presented in this chapter safely reduces many dynamic allocations in a program or program part to static allocations. This enables us to simplify

[8]I.e., using the fewest number of colors.

the symbolic upper bound we compute using the symbolic analysis in the previous chapter, by reducing those terms corresponding to dynamic allocations made static to 1.

However, we cannot perform this simplification on the final symbolic expression, since we have no way of distinguishing between terms corresponding to different allocation sites. Rather, we simplify the symbolic expression as we build it: when the symbolic analysis encounters an allocation site that has been marked static, it is not added to the symbolic expression, but to a separate expression describing only static allocations. This expression, denoted by E_{static} is computed as follows:

$$E_{static} = \sum_{n^I \in S} allocation(type(n^I))$$

With the optimization technique show in the previous section, the symbolic expression denoting static allocations becomes:

$$E_{static} = \sum_{\mathcal{C} \in Classes} \max_{n^I \in \mathcal{C}} allocation(type(n^I))$$

Finally, once the symbolic analysis is complete, we add E_{static}, as computed above, to our symbolic expression.

In practice, allocation sites that can be made static using the method presented here are surprisingly many (over 60% of all the allocation sites). By keeping the contribution from those sites to the symbolic upper bound minimal, we are able to greatly simplify that upper bound.

Chapter 5

Experimental Results

In this chapter we present results of our analyses on several benchmark programs. Table 5.1 presents a description of the benchmarks. Most of the them are from the SPECjvm98 benchmark suite; in addition, we analyze JavaCUP and JLex. For reference, Table 5.2 indicates the size of these programs in methods, Java bytecodes and elements of the SSI Intermediate Representation.

Table 5.3 shows our results in terms of the allocation sites we analyze and the precision of our analysis on these sites. We present two sets of results, the first with static allocations disabled and the second with static allocations enabled. The numbers are to be interpreted as follows:

- S: the number of allocation sites that only execute once
- L_e: the number of allocation sites in exact loops (i.e., loops whose bounds we have determined numerically)
- L_s: the number of allocation sites in symbolic loops (i.e., loops whose bounds we have determined symbolically; the symbolic expression corresponding to such a site contains at least one variable primitive)
- L_w: the number of allocation sites in unbounded loops (i.e., loops whose bounds we could not determine; the symbolic expression for such a site contains at least one *looptimes* primitive)

- R: the number of allocation sites in unbounded recursions (i.e. recursions whose number of executions we could not determine; the symbolic expression for such a site contains at least one *rectimes* primitive.

Note that the last four of the above categories may overlap (for example, we can have allocations contained in several nested loops and several nested recursions). Also, FLEX contains an optimization that reduces the number of exception allocation sites by allocating and throwing all exceptions of the same type in a method from a single location in the method. If an exception check fails, the generated code branches to the single throw site for the corresponding exception. Thus, the number of exception allocation sites is (in some cases considerably) lower than the number of exception check sites.

Table 5.4 shows our results in terms of the unknown symbols of different types that we could not determine. It is meant to illustrate the amount of effort required from the programmer to provide our analysis with the bounds it could not determine. Again, the table contains two sets of numbers, with the static allocation analysis disabled and enabled, respectively. The columns should be interpreted as follows:

- N_R: the number of recursion bounds the analysis could not determine
- N_L: the number of loop bounds the analysis could not determine
- N_v: the number of bounds on variables the analysis could not determine.

Finally, Table 5.5 presents the results of our static allocation analysis independently of the symbolic analysis; we show the total number of allocation sites, the number of sites that we have identified as static and the corresponding percentage.

These results entail several important conclusions:

- Substantial effort would be required from an unassisted programmer to determine a memory upper bound, as many of the allocation sites are enclosed in non-trivial loops or recursions (Table 5.3 shows that). The programmer would have to keep track of all enclosing loops and recursions to determine the number of times objects are allocated at a particular site.

Application	Description
compress	File compression tool (SPECjvm98)
jess	Expert system shell (SPECjvm98)
raytrace	Single thread raytracer
db	Database application (SPECjvm98)
mpegaudio	Audio file decompression tool (SPECjvm98)
jack	Java parser generator (SPECjvm98)
JLex	Java lexer generator
JavaCUP	Java parser generator

Table 5.1: Analyzed Applications

- Our symbolic analysis of upper bounds can focus the programmer's effort on the specific parts of the program that he or she needs to analyze by hand (see Table 5.4). We believe this information will substantially reduce the amount of programmer effort required to determine the worst-case memory usage. But as the numbers in Table 5.4 indicate, for some of our applications, there is still a significant amount of analysis left to the programmer.

- By making some allocation sites static, based on object liveness, we considerably simplify the computation of upper bounds. Table 5.5 shows that a majority of allocation sites can be made static using our method.

We view these results as providing a positive indication that our analysis may provide substantial assistance to a programmer attempting to determine an upper bound on the amount of memory that a program requires to execute successfully. One interesting aspect of our experimental results is that, because of our inability to find Java programs written for devices with limited memory, our benchmarks may not be representative of programs with strict memory usage constraints. In particular, we speculate that our analysis may provide much more precise information for these kinds of programs.

Application	Analyzed methods	Bytecode instrs	SSI IR size (instr.)
compress	274	7643	13311
jess	727	25725	46935
raytrace	94	2101	3502
db	353	12153	21041
mpegaudio	471	17341	33865
jack	573	22957	43100
JLex	445	21995	36221
JavaCUP	584	19674	38865

Table 5.2: Size of the Analyzed Applications

Application	Without static allocs					With static allocs				
	S	L_e	L_s	L_w	R	S	L_e	L_s	L_w	R
compress	9	0	107	33	13	83	0	33	1	0
jess	37	0	111	581	850	526	0	20	119	355
raytrace	18	0	3	0	0	20	0	1	0	0
db	80	0	0	79	13	143	0	0	10	0
mpegaudio	7	58	421	318	313	300	3	128	55	52
jack	15	0	104	181	265	329	0	4	119	58
JLex	233	0	49	169	79	333	0	11	39	16
JavaCUP	256	5	13	460	76	495	0	2	182	22

Table 5.3: Compound Analysis Results by Allocation Sites

Application	Without static allocs			With static allocs		
	N_R	N_L	N_v	N_R	N_L	N_v
compress	2	8	5	0	1	5
jess	21	70	68	20	38	43
raytrace	0	0	5	0	0	4
db	2	11	3	0	6	3
mpegaudio	1	20	20	1	19	11
jack	16	51	39	14	32	32
JLex	2	50	34	2	30	24
JavaCUP	6	72	20	5	47	20

Table 5.4: Compound Analysis Results by Number of Unknowns

Application	Allocation sites	Static Allocation Sites	%
_201_compress	117	80	68%
_202_jess	902	526	58%
_205_raytrace	21	17	63%
_209_db	162	122	75%
_222_mpegaudio	428	304	71%
_228_jack	459	341	74%
jlex	388	302	78%
javacup	771	536	70%
Total	3248	2218	68%

Table 5.5: Static Allocation Analysis Results

Chapter 6

Future Improvements

A number of improvements can be made to both analyses to improve their performance. Additionally, there are several ways to improve the usability of our analyses.

6.1 Improvements to the Symbolic Upper Bound Analysis

The main direction of improvement for this analysis, is adding more cases of program constructs and computations that the analysis can determine. Below are several examples of such improvements:

- Handling *iterator loops*, a common construct in Java, which are clearly bounded by the size of the collection underlying the iterator. Following the symbolic analysis technique we use currently use, we would add a primitive that expresses collection size and use it to bound iterator loops. The harder part, of course, is to place a bound on the collection size itself. We would need pointer analysis to trace the collection through the program, and a way of reasoning about the maximum number of objects that might have been added to it.

- Handling some cases of induction (for induction variables and recursive methods), for example using the techniques highlighted in [11] and [12].

6.2 Static Allocation Analysis Improvements

For the second analysis, the main improvement would be to add a way of tracking object pointers through the heap, using some form of pointer analysis (in fact, a pointer analysis from Alexandru Sălcianu [14] is available in FLEX). This would greatly improve the precision of our liveness computation in the case of heap-referenced objects, which is currently very primitive (as a reminder, we consider a heap-referenced object as alive from the point it was defined until the end of the program).

At the very least, we could add a conservative approximation of the live-region of a heap-referenced object as the total live-region of all its possible references, including both local variables and heap references.. While we have not studied this enhancement in great detail, it may be hard to do it while keeping our analysis compositional. One special case, that of an object holding the sole reference to another objects, is important in practice, as it mirrors the "has a" relationship from object-oriented design. The most ubiquitous example is `String` objects that hold a pointer to a character array. This special case might well be the most approachable instance of pointer analysis for our object liveness computation.

It should also be noted that the precision of our analysis depends in no small part on the accuracy of the call graph we are using. Therefore, improvements in FLEX's call graph implementation will positively affect our static allocation analysis.

6.3 Usability Improvements

Currently, we do not have a code transformation to account for static allocations. Static allocations have to be explicitly supported by the generated code for our bounds to be correct. More specifically, the code generated every allocation site must use a well-specified portion of a statically allocated region of memory. This will be implemented in the future using FLEX's support for code transformation.

In addition, we want to add an automated way of incorporating manually computed numerical bounds into the symbolic expression. Ideally, this would work by prompting the user once for each unknown bound, then reducing the expression to a numerical values using the bounds entered.

Chapter 7

Related Work

There has been significant work in the field of worst-case execution timing analysis, especially in connection to embedded systems and real-time programming. Worst-case execution time (WCET) is a problem related to the one we are addressing, since we need the same type of reasoning on the number of time control flow passes to parts of the program. Some proposals involve computing parameterized WCET expressions that are instantiated at run-time to make scheduling decisions, without programming intervention [16]. Others require the program to insert constructs in the code to help the reasoning of the analysis, and use a similar, symbolic analysis approach [13]. However, such research typically focuses on detailed modeling of low-level details such as the operation of the real-time scheduling and cache hits/misses rather than the high-level computation of memory bounds that we propose.

As for our static allocation analysis, we present, to our knowledge, the first use of a pointer analysis to enable static object preallocation. Other researchers have used pointer and/or escape analyses to improve the memory management of Java programs [17, 7, 6], but these algorithms focused on allocating objects on the call stack. Researchers have also developed algorithms that are designed to correlate the lifetimes of objects with the lifetimes of invoked functions, then use this information to allocate objects in different regions [15]. The goal is to eliminate garbage collection overhead by statically deallocating all of the objects allocated in a given region when the corresponding function returns.

Of these analyses, the analysis of Bogda and Hoelzle [7] is closest in spirit to our analysis. Their analysis is more precise in that it can stack allocate objects reachable from a single level of heap references, while our analysis does not attempt to maintain precise points-to information for objects reachable from the heap. Our analysis is more precise in that it computes live ranges of objects and treats exceptions with more precision. In particular, we found that our predicated analysis of type switches (which takes the type of the referenced object into account) was necessary to give our analysis enough precision to statically preallocate exception objects.

Our combined liveness and incompatibility analysis and use of graph coloring to minimize the amount of memory required to store objects allocated at unitary allocation sites is similar in spirit to register allocation algorithms [5, Chapter 11]. However, register allocation algorithms are concerned only with the liveness of the local variables, which can be computed by a simple intra-procedural analysis. We found that obtaining useful liveness results for dynamically allocated objects is significantly more difficult. In particular, we found that we had to track the flow of objects across procedure boundaries to obtain results precise enough to enable useful amounts of static preallocation.

Chapter 8

Conclusions

Placing conservative bounds on memory consumption is important in many resource-limited environments, because of the strict constraints on memory (embedded systems) and execution time (real-time systems) that have to be met. It is a difficult and time consuming process for a human programmer to compute such bounds unassisted.

This thesis presented a symbolic analysis that significantly reduces the complexity of this task, asking the programmer to reason only about those aspects of the program that could not be determined automatically. A second analysis uses a different approach – making dynamic allocations static based on object liveness – to improve on the performance of the first, further reducing complexity.

Personally, I have gained a lot of useful experience on the course of this project. Having the opportunity to investigate some of the concepts and trends in modern program analysis was more than welcome. Also, it was new and challenging to me to tackle a problem infeasible in general, with the goal of reducing cases of that problem to a minimum of unknowns. Finally, the experience of building on a large and evolving code base such as FLEX is one I regard as very useful for the future.

Without doubt, there still exists significant room for improvement. We have mentioned two main directions: more sophisticated symbolic reasoning in the first analysis, and more precise computation of object liveness during the second. Still, our results show that our combination of two analyses is, in its current state, an effective tool for automating the computation of memory upper bounds.

Bibliography

[1] The MIT Flex Compiler Infrastructure for Java. `http://www.flex-compiler.lcs.mit.edu/Harpoon`.

[2] O. Agesen. The Cartesian Product Algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming.* Springer-Verlag LNCS, 1995.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[4] C. S. Ananian. The Static Single Information Form. S.M. thesis, Massachusetts Institute of Technology, September 1999. Available from `http://www.flex-compiler.lcs.mit.edu/Harpoon/thesis.pdf`.

[5] A. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[6] B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications,* Denver, Colorado, November 1998.

[7] J. Bogda and U. Hoelzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications,* Denver, Colorado, November 1999.

[8] G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, Reading, Massachusetts, June 2000. Available from `http://www.rtj.org/rtsj-V1.0.pdf`.

[9] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM* 22, 1979, pp. 251-256.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL)*, Austin, January 1989 pp. 26-35.

[11] M. R. Haghighat and C. D. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. In *ACM Transactions on Programming Languages and Systems*, 18-4, July 1996, pp. 477–518.

[12] T. Fahringer, B.Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proceedings of the 11th International Conference on Supercomputing*, 1997, pp. 261-268. Conference

[13] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. In *Journal of Real-Time Systems,* 2-1, September 1989, pp. 159-176.

[14] A. Sălcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Snowbird, Utah June 2001.

[15] M. Tofte and L. Birkedal. A Region Inference Algorithm. In *ACM Transactions on Programming Languages and Systems,* 20(4), July 1998.

[16] E. Vivancos, C. Healy, F. Mueller and D. Whalley. Parametric Timing Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Languages, compilers and tools for embedded systems,* Snow Bird, Utah, 2001.

[17] J. Whaley and M. Rinard. A Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications,* Denver, Colorado, November 1999.