Interprocedural Compatibility Analysis for Static Object Preallocation

Ovidiu Gheorghioiu, Alexandru Sălcianu, and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{ovy, salcianu, rinard}@lcs.mit.edu

ABSTRACT

We present an interprocedural and compositional algorithm for finding pairs of *compatible* allocation sites, which have the property that no object allocated at one site is live at the same time as any object allocated at the other site. If an allocation site is compatible with itself, it is said to be *unitary*: at most one object allocated at that site is live at any given point in the execution of the program. We use the results of the analysis to statically preallocate memory space for the objects allocated at unitary sites, thus simplifying the computation of an upper bound on the amount of memory required to execute the program. We also use the analysis to enable objects allocated at several compatible allocation sites to share the same preallocated memory. Our experimental results show that, for our set of Java benchmark programs, 60% of the allocation sites are unitary and can be statically preallocated. Moreover, allowing compatible unitary allocation sites to share the same preallocated memory leads to a 95% reduction in the amount of memory preallocated for these sites.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*compiler, optimization, memory management (garbage collection)*

General Terms

Languages, Algorithms

Keywords

Static analysis, interprocedural analysis, memory preallocation

This research was supported by DARPA Contract F33615-00-C-1692, NSF Grant CCR-0086154, NSF Grant CCR-0073513, NSF Grant CCR-0209075, and the Singapore-MIT Alliance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-628-5/03/0001 ...$5.00.

1. INTRODUCTION

Modern object-oriented languages such as Java present a clean and simple memory model: conceptually, all objects are allocated in a garbage-collected heap. While this abstraction simplifies many aspects of the program development, it can complicate the calculation of an accurate upper bound on the amount of memory required to execute the program. Scenarios in which this upper bound is especially important include the development of programs for embedded systems with hard limits on the amount of available memory and the estimation of scoped memory sizes for real-time threads that allocate objects in sized scoped memories [9].

This paper presents a static program analysis designed to find pairs of *compatible* allocation sites; two sites are compatible if no object allocated at one site may be live at the same time as any object allocated at the other site. If an allocation site is compatible with itself (we call such allocation sites *unitary* allocation sites), then at any time during the execution of the program, there is at most one live object that was allocated at that site. It is therefore possible to statically preallocate a fixed amount of space for that allocation site, then use that space to hold all objects allocated at that site. Any further space usage analyses can then focus only on the non-unitary allocation sites.

Our analysis uses techniques inspired from register allocation [2, 6] to reduce the amount of memory required to hold objects allocated at unitary allocation sites. The basic approach is to build and color an *incompatibility graph*. The nodes in this graph are the unitary allocation sites. There is an undirected edge between two nodes if the nodes are not compatible. The analysis applies a coloring algorithm that assigns a minimal number of colors to the graph nodes subject to the constraint that incompatible nodes have different colors. This information enables the compiler to statically preallocate a fixed amount of memory for each color. At each unitary allocation site, the generated code bypasses the standard dynamic allocation mechanism and instead simply returns a pointer to the start of the statically preallocated memory for that allocation site's color. The object is stored in this memory for the duration of its lifetime in the computation. Our algorithm therefore enables objects allocated at compatible allocation sites to share the same memory.

Results from our implemented analysis show that, for our set of Java benchmark programs, our analysis is able to identify 60% of all allocation sites in the program as unitary allocation sites. Furthermore, our incompatibility graph col-

oring algorithm delivers a 95% reduction in the amount of memory required to store objects allocated at these unitary allocation sites. We attribute the high percentage of unitary allocation sites to specific object usage patterns characteristic of Java programs: many unitary allocation sites allocate exception, string buffer, or iterator objects.

We identify two potential benefits of our analysis. First, it can be used to simplify a computation of the amount of memory required to execute a given program. We have implemented a memory requirements analysis that, when possible, computes a symbolic mathematical expression for this amount of memory [16]. Our results from [16] show that preceding the memory requirements analysis with the analysis presented in this paper, then using the results to compute the memory requirements of unitary sites separately, can significantly improve both the precision and the efficiency of the subsequent memory requirements analysis. The second potential benefit is a reduction in the memory management overhead. By enabling the compiler to convert heap allocation to static allocation, our analysis can reduce the amount of time required to allocate and reclaim memory.

This paper makes the following contributions:

- **Object Liveness Analysis:** It presents a compositional and interprocedural object liveness analysis that conservatively estimates the set of objects that are live at each program point.

- **Compatibility Analysis:** It presents a compositional and interprocedural analysis that finds sets of compatible allocation sites. All objects allocated at sites in each such set can share the same statically preallocated memory. This analysis uses the results of the object liveness analysis.

- **Implementation:** We implemented our analyses in the MIT Flex [3] compiler and used them to analyze a set of Java benchmark programs. Our results show that our analyses are able to classify the majority of the allocation sites as unitary allocation sites, and that many such sites can share the same memory. We also implemented and evaluated a compiler optimization that transforms each unitary allocation site to use preallocated memory space instead of invoking the standard memory allocator.

The rest of this paper is organized as follows. Section 2 presents the analysis algorithm. Section 3 describes the implementation and presents our experimental results. We discuss related work in Section 4 and conclude in Section 5.

2. ANALYSIS PRESENTATION

Given a program P, the goal of the analysis is to detect pairs of compatible allocation sites from P, i.e., sites that have the property that no object allocated at one site is live at the same time as any object allocated at the other site. Equivalently, the analysis identifies all pairs of *incompatible* allocation sites, i.e., pairs of sites such that an object allocated at the first site and an object allocated at the second site may both be live at the same time in some possible execution of P. An object is live if any of its fields or methods is used in the future. It is easy to prove the following fact:

FACT 1. *Two allocation sites are incompatible if an object allocated at one site is live at 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. There are two complications. First, we have an abstraction problem: the analysis must use a finite abstraction to reason about the potentially unbounded number of objects that the program may create. Second, some parts of the program may read heap references created by other parts of the program. Using a full-fledged, flow-sensitive pointer analysis would substantially increase the time and space requirements of our analysis; a flow-insensitive pointer analysis [18, 5] would not provide sufficient precision since liveness is essentially a flow-sensitive property. We address these complications as follows:

- We use the *object allocation site* model [13]: all objects allocated by a given statement are modelled by an *inside node*[1] associated with that statement's program label.

- The analysis tracks only the objects pointed to by local variables. Nodes whose address may be stored into the heap are said to *escape into the heap*. The analysis conservatively assumes that such a node is not unitary (to ensure this, it sets the node to be incompatible with itself). Notice that, in a usual Java program, there are many objects that are typically manipulated only through local variables: exceptions, iterators, string buffers, etc.[2]

Under these assumptions, a node that does not escape into the heap is live at a given program point if and only if a variable that is live at that program point refers to that node. Variable liveness is a well-studied dataflow analysis [2, 6] and we do not present it here. As a quick reminder, a variable v is live at a program point if and only if there is a path through the control flow graph that starts at that program point, does not contain any definition of v and ends at an instruction that uses v.

The analysis has to process the 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. Reanalyzing each method for each call instruction (which corresponds conceptually to inlining that method) would be inefficient. Instead, we use *parameter nodes* to obtain a single context-sensitive analysis result for each method. The parameter 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*: in the absence of recursion, it analyzes each method exactly once to extract a single analysis result.[3] At each call site, it instantiates the result for the calling context of that particular call site.

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

[2] It is possible to increase the precision of this analyis by tracking one or more levels of heap references (similar to [8]).

[3] The analysis may analyze recursive methods multiple times before it reaches a fixed point.

n^I_{lb}	$\in$	*INode*	inside nodes
n^P_k	$\in$	*PNode*	parameter nodes
n	$\in$	$Node = INode \cup PNode$	general nodes

Figure 1: Node Abstraction

Figure 1 presents a summary of our node abstraction. We use the following notation: *INode* denotes the set of all inside nodes, *PNode* denotes the set of parameter nodes, and *Node* denotes the set of all nodes. When analyzing a method M, the analysis scope is the method M and all the methods that it transitively invokes. The inside nodes model the objects allocated in this scope. n^I_{lb} denotes the inside node associated with the allocation site from label *lb* (the superscript I stands for "inside"; it is not a free variable). n^I_{lb} represents all objects allocated at label *lb* in the currently analyzed scope. The parameter nodes model the objects that M receives as arguments. The parameter node n^P_i models the object that the currently analyzed method receives as its ith argument of object type.[4]

The analysis has two steps, each one an analysis in itself. The first analysis computes the objects live at each allocation site or call instruction.[5] The second analysis uses the liveness information to compute the incompatibility pairs.

We formulate our analyses as systems of set inclusion constraints and use a bottom-up, iterative fixed-point algorithm to compute the least (under set inclusion) solution of the constraints. For a given program, the number of nodes is bounded by the number of object allocation sites and the number of parameters. Hence, as our constraints are monotonic, all fixed point computations are guaranteed to terminate.

The rest of this section is organized as follows. Section 2.1 describes the execution of the analysis on a small example. Section 2.2 presents the program representation that the analysis operates on. Section 2.3 describes the object liveness analysis. In Section 2.4, we describe how to use the object liveness information to compute the incompatibility pairs. Section 2.5 discusses how to apply our techniques to multithreaded programs.

2.1 Example

Consider the Java code from Figure 2. The program creates a linked list that contains the integers from 0 to 9, removes from the list all elements that satisfy a specific condition (the even numbers in our case), then prints a string representation of the remaining list. The program contains six lines that allocate objects. The two `Iterator`s from lines 3a and 3b are allocated in library code, at the same allocation site. The other four lines allocate objects directly by executing `new` instructions. For the sake of simplicity, we ignore the other objects allocated in the library. In our example, we have five inside nodes. Node n^I_1 represents the linked list allocated at line 1, node n^I_2 represents the `Integer`s allocated at line 2, etc. The iterators from lines 3a and 3b are both represented by the same node n^I_3 (they are allocated at the same site). Figure 3 presents the incompatibility graph for this example.

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

[5]The object liveness analysis is able to find the live nodes at any program point; however, for efficiency reasons, we produce an analysis result only for the relevant statements.

```
     static void main(String args[]) {
        List l = createList(10);
        filterList(l);
        System.out.println(listToString(l));
     }

     static List createList(int size) {
 1:     List list = new LinkedList();
        for(int i = 0; i < size; i++) {
 2:       Integer v = new Integer(i);
          list.add(v);
        }
        return list;
     }

     static void filterList(List l) {
3a:     for(Iterator it = l.iterator(); it.hasNext();) {
           Integer v = (Integer) it.next();
           if(v.intValue() % 2 == 0)
               it.remove();
        }
     }

     static String listToString(List l) {
 4:     StringBuffer buffer = new StringBuffer();
3b:     for(Iterator it = l.iterator(); it.hasNext();) {
           Integer v = (Integer) it.next();
           buffer.append(v).append(" ");
        }
 5:     return new String(buffer);
     }
```

Figure 2: Example Code

The analysis processes the methods in a bottom-up fashion, starting from the leaves of the call graph. The library method `LinkedList.add` (not shown in Figure 2) causes its parameter node n^P_2 (n^P_1 is the `this` parameter) to escape into the heap (its address is stored in a list cell). `createList` calls `add` with n^I_2 as argument; therefore, the analysis instantiates n^P_2 with n^I_2 and detects that n^I_2 escapes. In `filterList`, the parameter node n^P_1 (the list) escapes into the heap because `list.iterator()` stores a reference to the underlying list in the iterator that it creates.

In the `listToString` method, n^I_4 is live "over the call" to `list.iterator()` that allocates n^I_3: it is pointed to by the local variable `buffer`, which is live both before and after the call. Therefore, n^I_4 is incompatible with n^I_3. Because n^I_4 is live at line 5, n^I_4 is also incompatible with n^I_5. n^I_3 is not live

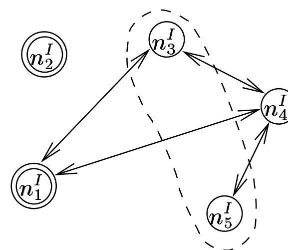


Figure 3: Incompatibility graph for the code from Figure 2. Circles represent inside nodes; a double circle indicates that the node escapes into the heap. n^I_3 and n^I_5 are compatible unitary nodes.

Name	Format	Informal semantics
COPY	$v_1 = v_2$	copy one local variable into another
NEW	$v = \texttt{new}\ C$	create one object of class C
STORE	$v_1.f = v_2$	create a heap reference
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

Figure 4: Instructions relevant for the analysis.

at line 5, so n_3^I and n_5^I are still compatible. The parameter node n_1^P (the list) is live at lines 4 and 3b (but not at 5). Therefore, n_1^P is incompatible with n_4^I and n_3^I.

The analysis of `main` detects that `l` points to n_1^I (because `createList` returns n_1^I). As the parameter of `filterList` escapes into the heap, the analysis detects that n_1^I escapes. When processing the call to `listToString`, the analysis instantiates n_1^P with n_1^I and discovers the incompatibility pairs $\langle n_1^I, n_3^I \rangle$ and $\langle n_1^I, n_4^I \rangle$. The analysis has already determined that n_1^I escapes into the heap and is not an unitary node; we generate the last two incompatibility pairs for purely expository purposes.

The graph coloring algorithm colors n_3^I and n_5^I with the same color. This means that the two iterators and the String allocated by the program have the property that no two of them are live at the same time. Hence, the compiler can statically allocate all of these objects into the same memory space.

2.2 Program Representation

We work in the context of a *static* compiler that compiles the entire code of the application before the application is deployed and executes. Our compiler provides full reflective access to classes and emulates the dynamic loading of classes precompiled into the executable. It does not support the dynamic loading of classes unknown to the compiler at compile time. This approach is acceptable for our class of target applications, real time software for embedded devices, for which memory consumption analysis is particularly important.

The analyzed program consists of a set of methods $m_1, m_2, \ldots \in \mathit{Method}$, with a distinguished main method. Each method m is represented by its control flow graph CFG_m. The vertices of CFG_m are the labels of the instructions composing m's body, while the edges represent the flow of control inside m. Each method has local variables $v_1, v_2, \ldots v_l \in \mathit{Var}$, and parameters $p_1, \ldots, p_k \in \mathit{Var}$, where Var is the set of local variables and method parameters.

Figure 4 contains the instructions that are relevant for the analysis. We assume that the analyzed program has already been converted into the Single Static Information (SSI) form [4], an extension of the Static Single Assignment (SSA) form [15] (we explain the differences later in this section).

Our intermediate representation models the creation and the propagation of exceptions explicitly. Each instruction that might generate an exception is preceded by a test. If an exceptional situation is detected (e.g., a null pointer dereferencing), our intermediate representation follows the Java convention of allocating and initializing an exception object, (e.g., a `NullPointerException`), then propagating the exception to the appropriate catch block or throwing the exception out of the method if no such block exists. Notice that due to the semantics of the Java programming language, each instruction that can throw an exception is also a potential object allocation site. Moreover, the exception objects are *first class* objects: once an exception is caught, references to it can be stored into the heap or passed as arguments of invoked methods. In practice, we apply an optimization so that each method contains a single allocation site for each automatically inserted exception (for example, NullPointerException and ArrayIndexOutOfBoundsException) that the method may generate but not catch. When the method detects such an exception, it jumps to that allocation site, which allocates the exception object and then executes an exceptional return out of the method.

To allow the inter-procedural propagation of exceptions, a CALL instruction from label lb has two successors: $succ_N(lb)$ for the normal termination of the method and $succ_E(lb)$ for the case when an exception is thrown out of the invoked method.

In both cases — locally generated exceptions or exceptions thrown from an invoked method — the control is passed to the appropriate catch block, if any. This block is determined by a succession of "instanceof" tests. If no applicable block exists, the exception is propagated into the caller of the current method by a THROW instruction "`throw` v". Unlike a `throw` instruction from the Java language, a THROW instruction from our intermediate representation always terminates the execution of the current method.

Note: we do not check for exceptions that are subclasses of `java.lang.Error`.[6] This is not a significant restriction: as we work in the context of a static compiler, where we know the entire code and class hierarchy, most of these errors cannot be raised by a program that compiled successfully in our system, e.g. `VirtualMachineError`, `NoSuchFieldError` etc. If the program raises any one of the rest of the errors, e.g., `OutOfMemoryError`, it aborts. In most of the cases, this is the intended behavior. In particular, none of our benchmarks catches this kind of exception.

We next present the informal semantics of the instructions from Figure 4. A COPY instruction "$v_1 = v_2$" copies the

[6]In the Java language, these exceptions correspond to severe errors in the virtual machine that the program is not expected to handle.

value of local variable v_1 into local variable v_2. A PHI instruction "$v = \phi(v_1, \ldots, v_k)$" is an SSA ϕ node that appears in the join points of the control flow graph; it ensures that each use of a local variable has exactly one reaching definition. If the control arrived in the PHI instruction on the ith incoming edge, v_i is copied into v. A NEW instruction "$v = \texttt{new}\ C$" allocates a new object of class C and stores a reference to it in the local variable v.

A CALL instruction "$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$" calls the method named mn of the object pointed to by v_1, with the arguments $v_1, \ldots, v_k$.[7] If the execution of the invoked method terminates with a RETURN instruction "`return` v", the address of the returned object is stored into v_N and the control flow goes to $succ_N(lb)$, where lb is the label of the call instruction. Otherwise, i.e., if an exception was thrown out of the invoked method, the address of the exception object is stored into v_E and the control flow goes to $succ_E(lb)$.

A TYPESWITCH instruction "$\langle v_1, v_2 \rangle = \texttt{typeswitch}\ v : C$" corresponds to a Java "instanceof" test. It checks whether the class of the object pointed to by v is a subclass of C. v is split into two variables: v_1 is v's restriction on the true branch, while v_2 is v's restriction on the false branch. Therefore, the object pointed to by v_1 is an instance of C, while the object pointed to by v_2 is not. A TYPESWITCH instruction is a simple example of an SSI "sigma" node, "$\langle v_1, v_2 \rangle = \sigma(v)$", that the SSI form introduces to preserve the flow sensitive information acquired in the test instructions. SSI thus allows the elegant construction of predicated dataflow analyses. Apart from this "variable splitting", SSI is similar to the SSA form. In particular, the SSI conversion seems to require linear time in practice [4].

Finally, a STORE instruction "$v_1.f = v_2$" sets the field f of the object referenced by v_1 to point to the object referenced by v_2. The other instructions are irrelevant for our analysis. In particular, as we do not track heap references, the analysis cannot gain any additional information by analyzing the instructions that read references from memory. However, we do analyze the STORE instructions because we need to identify the objects that escape into the heap.

We assume that we have a precomputed call graph: for each label lb that corresponds to a CALL instruction, $callees(lb)$ is the set of methods that that call instruction may invoke. The analysis works with any conservative approximation of the runtime call graph. Our implementation uses a simplified version of the Cartesian Product Algorithm [1].

2.3 Object Liveness Analysis

Consider a method M, a label/program point lb inside M, and let $live(lb)$ denote the set of inside and parameter nodes that are live at lb. We conservatively consider that a node is live at lb iff it is pointed to by one of the variables that are live at that point:

$$live(lb) = \bigcup_{v \text{ live in } lb} P(v)$$

where $P(v)$ is the set of nodes to which v may point. To interpret the results, we need to compute the set E_G of inside nodes that escape into the heap during the execution of the program. To be able to process the calls to M, we also 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, the analysis computes the following mathematical objects:

$$\begin{array}{rcl} P & : & Var \to \mathcal{P}(Node) \\ E_G & \subseteq & INode \\ R_N, R_E & : & Method \to \mathcal{P}(Node) \\ E & : & Method \to \mathcal{P}(PNode) \end{array}$$

We formulate the analysis as a set inclusion constraint problem. Figure 5 presents the constraints generated for a method $M \in Method$ with 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 $P(v_1) = P(v_2)$.[8] 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 and THROW add more nodes to $R_N(M)$ and $R_E(M)$, respectively. A STORE instruction "$v_1.f = v_2$", causes all the nodes pointed to by v_2 to escape into the heap. Accordingly, the nodes from $P(v_2)$ are distributed between E_G (the inside nodes) and $E(M)$ (the parameter nodes).

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 5, $SubTypes(C)$ denotes the set of all subtypes (i.e., Java subclasses) of C (including C). We can precisely determine the type $type(n_{lb'}^I)$ 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.[9]

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 $m \in callees(lb)$, we include the nodes from $R_N(m)$ into $P(v_N)$. Note that $R_N(m)$ is a parameterized result. We therefore instantiate $R_N(m)$ before use by replacing each parameter node n_i^P with the nodes that the corresponding argument v_i points to, i.e., 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 instantiates the set $E(m)$ and the uses the nodes from the resulting set $E(m)\langle P(v_1), \ldots, P(v_k) \rangle$ to update E_G and $E(M)$.

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

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

[7] For the sake of simplicity, in the presentation of the analysis we consider only instance methods (in Java terms, non-static methods), i.e., with v_1 as the `this` argument. The implementation handles both instance methods and static methods.

[8] 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 "$\supseteq$".

[9] 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	$P(p_i) = \{n_i^P\},\ \forall 1 \le i \le k$, where $p_1, \ldots, p_k$ are M's parameters.
COPY: $v_1 = v_2$	$P(v_1) = P(v_2)$
NEW: $v = \texttt{new}\ C$	$P(v) = \{n_{lb}^I\}$
STORE: $v_1.f = v_2$	$E(M) \supseteq P(v_2) \cap PNode, \quad E_G \supseteq P(v_2) \cap INode$
RETURN: `return` v	$R_N(M) \supseteq P(v)$
THROW: `throw` v	$R_E(M) \supseteq P(v)$
CALL: $\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$	$P(v_N) = \bigcup_{m \in callees(lb)} R_N(m)\langle P(v_1), \ldots, P(v_k)\rangle$ $P(v_E) = \bigcup_{m \in callees(lb)} R_E(m)\langle P(v_1), \ldots, P(v_k)\rangle$ `let` $A = \bigcup_{m \in callees(lb)} E(m)\langle P(v_1), \ldots, P(v_k)\rangle$ `in` $E(M) \supseteq A \cap PNode, \quad E_G \supseteq A \cap INode$
PHI: $v = \phi(v_1, \ldots, v_k)$	$P(v) = \bigcup_{i=1}^k P(v_i)$
TYPESWITCH: $\langle v_1, v_2\rangle = \texttt{typeswitch}\ v : C$	$P(v_1) = \{n_{lb'}^I \in P(v) \mid type(n_{lb'}^I) \in SubTypes(C)\} \ \cup\ \{n^P \in P(v)\}$ $P(v_2) = \{n_{lb'}^I \in P(v) \mid type(n_{lb'}^I) \notin SubTypes(C)\} \ \cup\ \{n^P \in P(v)\}$ $SubTypes(C)$ denotes the set of subclasses of class C.

Figure 5: Constraints for the object liveness analysis. For each method M, we compute $R_N(M)$, $R_E(M)$, $E(M)$ and $P(v)$ for each variable v live in at a relevant label. We also compute the set E_G of inside nodes that escape into the heap.

2.4 Computing the Incompatibility Pairs

Once the computation of the object liveness information completes, the analysis computes the (global) set of pairs of incompatible allocation sites $Inc_G \subseteq INode \times INode$.[10] The analysis uses this set of incompatible allocation sites to detect the unitary allocation sites and to construct the compatibility classes.

Figure 6 presents the constraints used to compute Inc_G. An allocation site from label lb is incompatible with all the allocation sites whose corresponding nodes are live at lb.

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 analysis collects the incompatibility pairs involving one parameter node into a set of *parametric* incompatibilities $ParInc(M)$. It instantiates this set at each call to M, similar to the way it instantiates $R_N(M)$, $R_E(M)$ and $E(M)$:

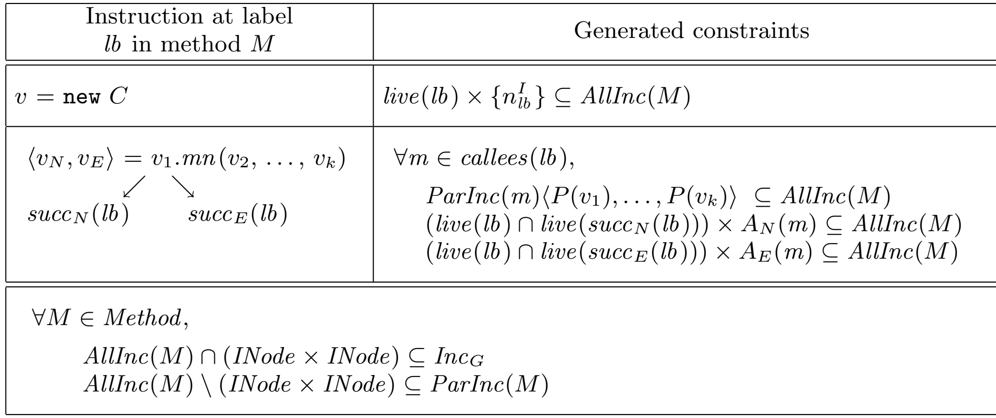
$$ParInc(M)\langle S_1, \ldots, S_k\rangle \ = \bigcup_{\langle n_i^P, n\rangle \in ParInc(M)} S_i \times \{n\}$$

[10]Recall that there is a bijection between the inside nodes and the allocation sites.

(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 in the call graph, each incompatibility pair will involve only inside nodes and will be passed to Inc_G.

To simplify the equations from Figure 6, for each method M, we compute the entire set of incompatibility pairs $AllInc(M)$. After $AllInc(M)$ is computed, the pairs that contain only inside nodes are put in the global set of incompatibilities Inc_G; the pair that contains a parameter node are put in $ParInc(M)$. Our implementation of this algorithm performs this separation "on the fly", as soon as an incompatibility pair is generated, without the need for $AllInc(M)$.

In the case of a CALL instruction, we have two kinds of incompatibility pairs. We have already mentioned the first kind: the pairs obtained by instantiating $ParInc(m), \forall m \in callees(lb)$. 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



Instruction at label lb in method M	Generated constraints
$v = \texttt{new}\ C$	$live(lb) \times \{n^I_{lb}\} \subseteq AllInc(M)$
$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$ → $succ_N(lb)$, $succ_E(lb)$	$\forall m \in callees(lb)$, $ParInc(m)\langle P(v_1), \ldots, P(v_k)\rangle \subseteq AllInc(M)$ $(live(lb) \cap live(succ_N(lb))) \times A_N(m) \subseteq AllInc(M)$ $(live(lb) \cap live(succ_E(lb))) \times A_E(m) \subseteq AllInc(M)$

$\forall M \in Method$,
$AllInc(M) \cap (INode \times INode) \subseteq Inc_G$
$AllInc(M) \setminus (INode \times INode) \subseteq ParInc(M)$

Figure 6: Constraints for computing the set of incompatibility pairs.

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

Figure 7: Constraints for computing A_N, A_E. For each relevant instruction, if the condition from the second column is satisfied, the corresponding constraint from the third column is generated.

the exceptional exit from an invoked method separately. Let $A_N(m) \subseteq INode$ be the set of inside nodes that represent the objects that may be allocated during a method execution that returns normally. Similarly, let $A_E(m) \subseteq INode$ be the set of inside nodes that represent the objects that may be allocated during an invocation of m that returns with an exception. We describe later how to compute these sets; for the moment we suppose the analysis computes them just before it starts 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 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 an easy 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 code, 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 7 formally presents the constraints for computing the sets $A_N(M)$ and $A_E(M)$.

2.5 Multithreaded Applications

So far, we have presented the analysis in the context of a single-threaded application. For a multithreaded application, the analysis needs to examine all methods that are transitively called from the main method and from the `run()` methods of the threads that may be started. In addition, all nodes that correspond to started threads need to be marked as escaped nodes. The rest of the analysis is unchanged.

In Java, each thread is represented by a thread object allocated in the heap. For an object to escape one thread to be accessed by another, it must be reachable from either the thread object or a static class variable (global variables are called static class variables in Java). In both cases, the analysis determines that the corresponding allocation site is not unitary. Therefore, all objects allocated at unitary allocation sites are local to the thread that created them and do not escape to other threads. Although we know that no two objects allocated by the same thread at the same unitary site are live at any given moment, we can have multiple live objects allocated at this site by different threads. Hence, for each group of compatible unitary sites, we need to allocate one memory slot *per thread*, instead of one per program.

The compiler generates code such that each time the program starts a new thread, it preallocates memory space for all unitary allocation sites that may be executed by that thread. For each unitary allocation site, the compiler generates code that retrieves the current thread and uses the preallocated memory space for the unitary site in the current thread. When a thread terminates its execution, it deallocates its preallocated memory space. As only thread-local

objects used that space, this deallocation does not create dangling references. To bound the memory space occupied by the unitary allocation sites, we need to bound the number of threads that simultaneously execute in the program at any given time.

2.6 Optimization for Single-Thread Programs

In the previous sections, we consider a node that escapes into the heap to be incompatible with all other nodes, including itself. This is equivalent to considering the node to be live during the *entire* program. We can gain additional precision by considering that once a node escapes, it is live only for *the rest* of the program. This enhancement allows us to preallocate even objects that escape into the heap, if their allocation site executes at most once. This section presents the changes to our analysis that apply this idea.

We no longer use the global set E_G. Instead, for each label lb, $E(lb) \subseteq Node$ denotes the set of nodes that the instruction at label lb may store a reference to into the heap. This set is relevant only for labels that correspond to STOREs and CALLs; for a CALL, it represents the nodes that escape during the execution of the invoked method.

We extend the set of objects live at label lb (from method M) to include all objects that are escaped by instructions at labels lb' from M that can reach lb in CFG_M:

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

We change the constraints from Figure 5 as follows: for a STORE instruction "$v_1.f = v_2$", we generate only the constraint $E(lb) = P(v_2)$. For a CALL instruction "$\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$", we generate the same constraints as before for $P(v_N)$ and $P(v_E)$, and the additional constraint

$$E(lb) = \bigcup_{m \in callees(lb)} E(m)\langle P(v_1), \ldots, P(v_k)\rangle$$

The rules for STORE and CALL no longer generate any constraints for E_G (unused now) and $E(M)$. Instead, we define $E(M)$ as

$$E(M) = \bigcup_{lb \text{ in } M} E(lb)$$

Now, $E(M) \subseteq \mathcal{P}(Node)$ denotes the set of all nodes — not only parameter nodes as before, but also inside nodes — that escape into the heap during M's execution.

The rest of the analysis is unchanged. The new definition of $live(lb)$ ensures that if a node escapes into the heap at some program point, it is incompatible with all nodes that are live at any future program point. Notice that objects allocated at unitary sites are no longer guaranteed to be thread local, and we cannot apply the preallocation optimization described at the end of Section 2.5. Therefore, we use this version of the analysis only for single thread programs.

3. EXPERIMENTAL RESULTS

We have implemented our analysis, including the optimization from Section 2.6, in the MIT Flex compiler system [3]. We have also implemented the compiler transformation for memory preallocation: our compiler generates executables with the property that unitary sites use preallocated memory space instead of calling the memory allocation primitive. The memory for these sites is preallocated at the beginning of the program. Our implementation does not currently support multithreaded programs as described in Section 2.5.

We measure the effectiveness of our analysis by using it to find unitary allocation sites in a set of Java programs. We obtained our results on a Pentium 4 2.8Ghz system with 2GB of memory running RedHat Linux 7.3. We ran our compiler and analysis using Sun JDK 1.4.1 (hotspot, mixed mode); the compiler generates native executables that we ran on the same machine. Table 1 presents a description of the programs in our benchmark suite. We analyze programs from the SPECjvm98 benchmark suite[11] and from the Java version of the Olden benchmark suite [12, 11]. In addition, we analyze JLex, JavaCUP, and _205_raytrace.

Table 2 presents several statistics that indicate the size of each benchmark and the analysis time. The statistics refer to the user code plus all library methods called from the user code. As the data in Table 2 indicate, in general, the time required to perform our analysis is of the same order of magnitude as the time required to build the intermediate representation of the program. The only exceptions are _202_jess and _213_javac.

Table 3 presents the number of total allocation sites and unitary allocation sites in each program. These results show that our analysis is usually able to identify the majority of these sites as unitary sites: of the 14065 allocation sites in our benchmarks, our analysis is able to classify 8396 (60%) as unitary sites. For twelve of our twenty benchmarks, the analysis is able to recognize over 80% of the allocation sites as unitary.

Table 3 also presents results for the allocation sites that allocate exceptions (i.e., any subclass of java.lang.Throwable), non-exceptions (the rest of the objects), and java.lang.StringBuffers (a special case of non-exceptions). For each category, we present the total number of allocation sites of that kind and the proportion of these sites that are unitary. The majority of the unitary allocation sites in our benchmarks allocate exception or string buffer objects. Of the 9660 total exception allocation sites in our benchmarks, our analysis is able to recognize 6602 (68%) as unitary sites. For thirteen of our twenty benchmarks, the analysis is able to recognize over 90% of the exception allocation sites as unitary sites. Of the 1293 string buffer allocation sites, our analysis is able to recognize 1190 (92%) as unitary sites. For eight benchmarks, the analysis is able to recognize over 95% of the string buffer allocation sites as unitary sites.

Table 4 presents the size of the statically preallocated memory area that is used to store the objects created at unitary allocation sites. The second column of the table presents results for the case where each unitary allocation site has its own preallocated memory chunk. As described in the introduction of the paper, we can decrease the preallocated memory size significantly if we use a graph coloring algorithm to allow compatible unitary allocation sites to share the same preallocated memory area. The third column of Table 4 presents results for this case. Our compiler optimization always uses the graph coloring algorithm; we provide the second column for comparison purposes only.

[11] With the exception of _227_mtrt, which is multithreaded.

Application	Description
SPECjvm98 benchmark set	
_200_check	Simple program; tests JVM features
_201_compress	File compression tool
_202_jess	Expert system shell
_209_db	Database application
_213_javac	JDK 1.0.2 Java compiler
_222_mpegaudio	Audio file decompression tool
_228_jack	Java parser generator
Java Olden benchmark set	
BH	Barnes-Hut N-body solver
BiSort	Bitonic Sort
Em3d	Models the propagation of electromagnetic waves through three dimensional objects
Health	Simulates a health-care system
MST	Computes the minimum spanning tree in a graph using Bentley's algorithm
Perimeter	Computes the perimeter of a region in a binary image represented by a quadtree
Power	Maximizes the economic efficiency of a community of power consumers
TSP	Solves the traveling salesman problem using a randomized algorithm
TreeAdd	Recursive depth-first traversal of a tree to sum the node values
Voronoi	Computes a Voronoi diagram for a random set of points
Miscellaneous	
_205_raytrace	Single thread raytracer (not an official part of SPECjvm98)
JLex	Java lexer generator
JavaCUP	Java parser generator

Table 1: Analyzed Applications

Application	Analyzed methods	Bytecode instrs	SSI IR size (instr.)	SSI conversion time (s)	Analysis time (s)
_200_check	208	7962	10353	1.1	4.1
_201_compress	314	8343	11869	1.2	7.4
_202_jess	1048	31061	44746	5.3	101.2
_209_db	394	12878	18162	2.7	12.3
_213_javac	1681	52941	71050	8.2	1126.2
_222_mpegaudio	511	18041	30884	5.2	15.9
_228_jack	618	23864	37253	11.6	55.6
BH	169	6476	8690	1.4	3.6
BiSort	123	5157	6615	1.2	2.9
Em3d	142	5519	7497	0.9	3.1
Health	141	5803	7561	0.9	3.2
MST	139	5228	6874	1.2	3.0
Perimeter	144	5401	6904	1.2	2.7
Power	135	6039	7928	1.0	3.2
TSP	127	5601	6904	0.9	3.1
TreeAdd	112	4814	6240	0.8	2.8
Voronoi	274	8072	10969	1.8	4.3
_205_raytrace	498	14116	20875	4.2	23.0
JLex	482	22306	31354	4.0	12.3
JavaCUP	769	27977	41308	5.8	32.0

Table 2: Analyzed Code Size and Analysis Time

Application	Allocation sites	Unitary sites		Exceptions		Non-exceptions		StringBuffers	
		count	%	total	unitary %	total	unitary %	total	unitary %
_200_check	407	326	80%	273	92%	134	57%	44	97%
_201_compress	489	155	32%	390	28%	99	44%	38	97%
_202_jess	1823	919	50%	1130	58%	693	38%	233	84%
_209_db	736	354	48%	565	48%	171	49%	65	98%
_213_javac	2827	1086	38%	1863	47%	964	23%	195	89%
_222_mpegaudio	825	390	47%	625	55%	200	24%	43	97%
_228_jack	910	479	53%	612	54%	298	50%	135	99%
BH	329	281	85%	243	98%	86	51%	18	94%
BiSort	234	198	85%	177	97%	57	47%	17	94%
Em3d	276	235	85%	206	98%	70	50%	20	95%
Health	276	227	82%	202	97%	74	42%	17	94%
MST	257	216	85%	194	97%	63	44%	16	93%
Perimeter	239	200	84%	180	97%	59	45%	16	93%
Power	262	213	81%	192	97%	70	39%	15	93%
TSP	235	199	85%	176	97%	59	49%	17	94%
TreeAdd	227	190	84%	170	96%	57	46%	15	93%
Voronoi	448	387	86%	349	98%	99	44%	28	96%
_205_raytrace	753	318	42%	525	44%	228	39%	43	95%
JLex	971	812	84%	645	99%	326	54%	72	86%
JavaCUP	1541	1211	79%	943	93%	598	56%	246	92%
Total	14065	8396	60%	9660	68%	4405	41%	1293	92%

Table 3: Unitary Site Analysis Results

Application	Preallocated memory size (bytes)		Size reduction %
	normal	sharing	
_200_check	5516	196	96%
_201_compress	2676	144	95%
_202_jess	17000	840	96%
_209_db	6028	252	96%
_213_javac	18316	332	98%
_222_mpegaudio	6452	104	98%
_228_jack	8344	224	97%
BH	4604	224	95%
BiSort	3252	96	98%
Em3d	3860	200	95%
Health	3716	96	97%
MST	3532	96	97%
Perimeter	3280	96	98%
Power	3540	196	94%
TSP	3292	104	97%
TreeAdd	3120	92	98%
Voronoi	6368	192	97%
_205_raytrace	5656	644	89%
JLex	13996	1676	88%
JavaCUP	20540	1180	94%
Total	143088	6984	95%

Table 4: Preallocated Memory Size

The graph coloring algorithm finds an approximation of the smallest number of colors such that no two incompatible allocation sites have the same color. For each color, we preallocate a memory area whose size is the maximum size of the classes allocated at allocation sites with that color. Our implementation uses the DSATUR graph coloring heuristic [10]. It is important to notice that the DSATUR heuristic minimizes the numbers of colors, not the final total size of the preallocated memory. However, this does not appear to have a significant negative effect on our results: as the numbers from Table 4 show, we are able to reduce the preallocated memory size by at least 88% in all cases; the average reduction is 95%.

Theoretically, the preallocation optimization may allocate more memory than the original program: preallocating a memory area for a set of compatible allocation sites reserves that area for the entire lifetime of the program, even when no object allocated at the attached set of compatible sites is reachable. An extreme case is represented by the memory areas that we preallocate for allocation sites that the program never executes. However, as the data from Table 4 indicate, in practice, the amount of preallocated memory for each analyzed application is quite small.

We compiled each benchmark with the memory preallocation optimization enabled. Each optimized executable finished normally and produced the same result as the unoptimized version. We executed the SPECjvm98 and the Olden applications with their default workload. We ran JLex and JavaCUP on the lexer and parser files from our compiler infrastructure. We instrumented the allocation sites to measure how many objects were allocated by the program and how many of these objects used the preallocated memory.

Application	Total objects	Preallocated objects	
		count	%
_200_check	725	238	33%
_201_compress	941	108	11%
_202_jess	7917932	3275	0%
_209_db	3203535	142	0%
_213_javac	5763881	335775	6%
_222_mpegaudio	1189	7	1%
_228_jack	6857090	409939	6%
BH	15115028	7257600	48%
BiSort	131128	15	0%
Em3d	16061	23	0%
Health	1196846	681872	57%
MST	2099256	1038	0%
Perimeter	452953	10	0%
Power	783439	12	0%
TSP	49193	32778	67%
TreeAdd	1048620	13	0%
Voronoi	1431967	16399	1%
_205_raytrace	6350085	4080258	64%
JLex	1419852	12926	1%
JavaCUP	100026	16517	17%

Table 5: Preallocated Objects

Table 5 presents the results of our measurements. For five of our benchmarks, at least one third of the objects resided in the preallocated memory. There is no correlation between the static number of unitary sites and the dynamic number of objects allocated at those sites. This is explained by the large difference in the number of times different allocation sites are executed. In general, application-specific details tend to be the only factor in determining these dynamic numbers. For example, in JLex, 95% of the objects are iterators allocated at the same (non-unitary) allocation site; _213_javac and JavaCUP use many StringBuffers that we can preallocate; both _205_raytrace and BH use many temporary objects to represent mathematical vectors, etc.

4. RELATED WORK

This paper presents, 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 [14, 20, 7], but these algorithms focus on allocating objects on the call stack. Researchers have also developed algorithms that correlate the lifetimes of objects with the lifetimes of invoked methods, then use this information to allocate objects in different regions [19]. The goal is to eliminate garbage collection overhead by atomically deallocating all of the objects allocated in a given region when the corresponding function returns. Other researchers [17] require the programmer to provide annotations (via a rich type systems) that specify the region that each object is allocated into.

Bogda and Hoelzle [8] use pointer analysis to eliminate unnecessary synchronizations in Java programs. In spite of the different goals, their pointer analysis has many technical similarities with our analysis. Both analyses avoid maintaining precise information about objects that are placed "too deep" into the heap. Bogda and Hoelzle's 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. On the other hand, 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 analysis has more aggressive aims than escape analysis. Escape analysis is typically used to infer that the lifetimes of all objects allocated at a specific allocation site are contained within the lifetime of either the method that allocates them or one of the methods that (transitively) invokes the allocating method. The compiler can transform such an allocation site to allocate the object from the method stack frame instead of the heap. Notice that the analysis does not provide any bound on the number of objects allocated at that allocation site: in the presence of recursion or loops, there may be an arbitrary number of live objects from a single allocation site (and an arbitrary number of these objects allocated on the call stack). In contrast, our analysis identify allocation sites that have the property that at most one object is live at any given time.

In addition, the stack allocation transformation may require the compiler to lift the corresponding object allocation site out of the method that originally contained it to one of the (transitive) callers of this original allocating method [20]. The object would then be passed by reference down the call stack, incurring runtime overhead.[12] The static preallocation optimization enabled by our analysis does not suffer from this drawback. The compiler transforms the original allocation site to simply acquire a pointer to the statically allocated memory; there is no need to move the allocation site into the callers of the original allocating method.

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 [6, Chapter 11]. However, register allocation algorithms are concerned only with the liveness of the local variables, which can be computed by a simple intraprocedural analysis. We found that obtaining useful liveness results for dynamically allocated objects is significantly more difficult. In particular, we found that we had to use a predicated analysis and track the flow of objects across procedure boundaries to identify significant amounts of unitary sites.

5. CONCLUSIONS

We have presented an analysis designed to simplify the computation of an accurate upper bound on the amount of memory required to execute a program. This analysis statically preallocates memory to store objects allocated at unitary allocation sites and enables objects allocated at compatible unitary allocation sites to share the same preallocated memory. Our experimental results show that, for our set of Java benchmark programs, 60% of the allocation sites are unitary and can be statically preallocated. Moreover, allowing compatible unitary allocation sites to share

[12]A semantically equivalent alternative is to perform method inlining. However, inlining introduces its own set of overheads.

the same preallocated memory leads to a 95% reduction in the amount of memory required for these sites. Based on this set of results, we believe our analysis can automatically and effectively eliminate the need to consider many object allocation sites when computing an accurate upper bound on the amount of memory required to execute the program. We have also used the analysis to optimize the memory managment.

6. ACKNOWLEDGEMENTS

We would like to thank Wes Beebee and Scott C. Ananian for their useful advice on implementing the preallocation optimization in the MIT Flex compiler system [3], and Viktor Kuncak for proofreading early drafts of the paper. We also want to thank the anonymous referees for their valuable comments.

7. REFERENCES

[1] Ole Agesen. The cartesian product algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, 1995.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., Reading, MA, second edition, 1986.

[3] C. Scott Ananian. MIT FLEX compiler infrastructure for Java. http://www.flex-compiler.lcs.mit.edu.

[4] C. Scott Ananian. Static single information form. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1999.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[6] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[7] 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, CO, November 1999.

[8] 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, CO, November 1999.

[9] Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000.

[10] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, (22):251–256, 1979.

[11] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[12] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1995.

[13] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 1990.

[14] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[16] Ovidiu Gheorghioiu. Statically determining memory consumption of real-time Java threads. MEng thesis, Massachusetts Institute of Technology, 2002.

[17] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation*, 2002.

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[19] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4), July 1998.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.