Scalable Interprocedural Analysis

A Pragmatic Approach

Mark Marron[1] Ondřej Lhoták[2] Anindya Banerjee[1]

[1]IMDEA-Software, [2]University of Waterloo

{mark.marron, anindya.banerjee}@imdea.org, olhotak@uwaterloo.ca

Abstract

When analyzing a program via an abstract interpretation framework we would like to analyze the program in a context-sensitive interprocedural manner. Analyzing the program in a manner that considers interprocedural flow can lead to much more accurate results than local or context-insensitive analyses. However, the computational cost (both time and memory consumption) associated with context-sensitive interprocedural analysis techniques makes them infeasible for all but very small programs or simple domains.

This paper presents several novel, domain independent, heuristics for reducing the cost of analyzing a program in a context-sensitive manner while having a small impact on the precision of the results. The heuristics are motivated by observations about fundamental properties of software design and the semantics of iterative dataflow analysis. We validate the effectiveness of the heuristics via experimental evaluation which shows both good scalability and high precision of the results.

1. Introduction

When analyzing a program in the framework of abstract interpretation [4, 25, 27] there is a fundamental tension between precision and performance when deciding how to handle call-context sensitivity. While call-context insensitive approaches substantially improve the performance of the analysis, they often result in overly approximate analysis results. Conversely fully call-context sensitive analyses are computationally intractable for all but trivial programs (or simple properties such as points-to relations [13, 33]). Thus, there is interest in the development of heuristics that tradeoff small amounts of precision for large increases in the analysis performance [11, 17, 23]. In this paper we present a comprehensive set of heuristics for addressing the performance bottlenecks that arise when performing a context-sensitive analysis. We integrate them into a context-sensitive and flow-sensitive dataflow analysis algorithm that we use to drive the shape analysis of substantial Java programs (the largest is 15KLOC and 90 classes, involving dynamic/recursive call sequences, and substantial library usage).

Scalability Challenges. The standard approach for analyzing a program in a flow and call-context sensitive manner involves selecting a base domain (intervals [4], linear relations [12], shape [1, 30], regions [18], etc.) and then lifting this base domain to powersets. Thus at each program point the state of the program, $\theta = \{\sigma_1, \dots, \sigma_k\}$, is a set of *models* where each *model*, σ_j, is an element from the base domain, referred to as the *abstract analysis states* (or simply *states*). When analyzing method calls/returns the abstract state is passed to and from the callee method and usually (for efficiency) the analysis memoizes the input state and resulting output state pairs for each method.

When analyzing a program in this manner there are three major issues that lead to large increases in analysis runtime and memory usage: (a) growth in the number of models that the analysis must process due to methods having multiple possible return values (b) the related problem of growth in the number of models that the analysis must process due to method calls that have multiple possible implementations (i.e., overloaded virtual methods in object oriented programs) and (c) the production of large numbers of memoized call-context states for each method (which is particularly important to the memory use of the analysis).

These issues lead to the analysis generating intractably large sets of models that must be manipulated and large memotables (which keep track of memoized call/return states for each method) which require large amounts of memory and large numbers of comparison operations. In this work we address these problems by identifying points in the program where the sets of abstract models can be aggressively merged into a smaller number of more abstract models and construct a set of heuristics to decide how aggressive to be in the merging. These merges produce smaller abstract states and memotables and have only a small impact on the precision of the final analysis results.

Method Call/Return Merge Heuristics. We address the first challenge (the combinatorial growth in the size of the model sets due to method calls) by using one of the fundamental concepts of object-oriented programming and modular software development in general: abstraction via method pre/post conditions. We assume in this paper that the program we are analyzing follows good object-oriented software development practices where the methods have well defined pre/post conditions. However, we *do not* require that these pre/post conditions be explicitly recorded in the code and our analysis *does not* depend on knowing what the conditions are.

Based on this assumption we can, with a negligible impact on the precision of the analysis, merge all of the callee accessible [6, 17, 29] parts of the abstract models at the given call site and then again at the return from the callee (this differs from simple context insensitivity as we only merge models from the same call site and call-context). The return model from the callee is then expanded back into each of the models at the caller site. Thus, when the analysis encounters a call the cardinality of the model set after the call is the same cardinality as the model set before the call (each model before the call produces a single model after the call).

Virtual Method Merge Heuristics. The second challenge is combinatorial growth in the number of models when dealing with virtual methods where there may be many possible method targets for a single method invocation (generating multiplicative growth even with the call/return merge described above). Our approach to this problem is similar to the case of multiple return values from a single method. We know that all of these methods must return program states that respect the post-condition given in the base class and further that each subclass implementation of the virtual method should not assume anything that is not a consequence of the base class pre-condition [14]. Thus, we can, with a negligible loss of precision, merge some of the results of these calls returning a reduced set of models from a virtual method invocation.

Hybrid Call Context Merge Heuristics. The final issue that we examine is how to reduce the number of entries that are in the method entry/exit state memotables. The challenge is to eliminate redundant entries created during the iterative processes inherent to dataflow analysis while ensuring that we do not cause the repeated analysis of a given method body with the same abstract states. To address these challenges we use a hybrid memotable matching scheme that first takes into account the call path and then, depending on the call path information the memotable lookup either uses the domain equality or the weaker concept of *subsumption* ($\preceq$, Sec. 2) to find a matching memo entry. This hybrid call-path call-context approach first checks if the call paths for the two call entry states are equivalent. If they are then they must represent partial approximations of the call/return semantics for the method on the same call path and can be safely merged (i.e. there is some recursive call or loop body that the analysis is iterating on producing multiple approximations of a fixpoint). If the call paths differ then we use the equality operator to determine if the two entries are equal. This approach ensures that partial approximate contexts are merged together (eliminating redundant contexts while not impacting the hit rate of the memo table) while contexts from distinct call states are maintained precisely.

Contributions. In addition to the technical contributions, introducing heuristics for call/return merging (Sec. 3) and hybrid call-context sensitivity (Sec. 4), this paper presents the integration of these concepts into an interprocedural analysis framework (Sec. 5). This framework, which is a general-purpose method for efficiently and precisely performing context-sensitive interprocedural dataflow analysis, is the overarching contribution of the paper. Our experimental results (Sec. 6) with this dataflow analysis framework indicate that the heuristics greatly increase the scalability of the analysis, enabling us to perform shape analysis on a 15KLoc program (which could not be analyzed without the heuristics) in only 115 seconds and 122MB of memory. Our experimental results also demonstrate that the heuristics have a negligible impact on the precision of the analysis; in fact the results on the benchmarks suite are near the precision limits of the base shape domain.

2. Domain Requirements

The interprocedural analysis framework presented in this paper is designed to work with any abstract domain that satisfies the standard requirements for *abstract interpretation* [4, 25]. Thus we assume that we have a base domain $\mathscr{D}$ which defines a set of *abstract models* $\{\sigma_1, \ldots, \sigma_k\}$ which form a lattice with the standard concretization (γ), join ($\sqcup$), and order ($\sqsubseteq$) operations. To ensure good performance and accuracy we place the following additional requirements on the domain:

- We assume that the domain provides project/extend operations [6, 17, 24, 29] as needed by the domain.
- We require an upper approximation operation $\tilde{\sqcup}$ with the property, $\forall \sigma_i, \sigma_j\ \gamma(\sigma_i) \subseteq \gamma(\sigma_i \sqcup \sigma_j) \subseteq \gamma(\sigma_i \tilde{\sqcup} \sigma_j)$.
- Using the upper approximation operator we define a subsumption relation $\preceq$: $\mathscr{D} \times \mathscr{D} \mapsto$ `bool` as $\sigma_i \preceq \sigma_j$ iff $\sigma_j = \sigma_i \tilde{\sqcup} \sigma_j$. We note that this definition implies: if $\sigma_i \preceq \sigma_j$ then $\gamma(\sigma_i) \subseteq \gamma(\sigma_i \tilde{\sqcup} \sigma_j) = \gamma(\sigma_j)$. Thus the subsumption relation is order preserving with respect to the concrete domain and we can safely use it to determine when a fixpoint of the interprocedural analysis has been reached.

We assume the local dataflow analysis of each method is done in a flow-sensitive manner using a *partially disjunctive* [15] set. Thus, each abstract *state* θ is a set of abstract *models* $\{\sigma_1, \ldots, \sigma_j\}$. Sec. 6.1 contains a specific domain that satisfies these properties.

3. Call/Return Merge

The first topic we examine is how to deal with the combinatorial growth in the size of the abstract states that can occur when analyzing method calls. In an object oriented programming language there are two sources of this combinatorial growth. One source is the potential that passing in and returning sets of abstract models ends up lifting callee-local control flow information into the caller scope, that is if there are k paths through the callee method and we pass in a set of m models then the result of analyzing the method can contain $k * m$ models. The other source of exponential growth in the size of the abstract state is that a dynamic method invocation may have multiple possible targets (since we cannot always resolve a precise invocation target), resulting in the creation of new models for each possible target.

3.1 Entry/Exit Merge

We first look at how the growth in the number of models when analyzing a single callee method can be controlled by merging all of the models at call entry and exit (a heuristic used with minimal explanation/justification in [17]). These merges ensure that the number of models before the call is proportional to the number after the call and based on the pre/post conditions of the methods the precision loss due to this merge is negligible.

1. If the callee depends on some property P holding as part of the precondition of the method then all models at each call site *must* satisfy P.
2. If a property P is not enforced by the preconditions then the callee *must* test for this property internally if needed.
3. If a property is part of the post condition then every state at the exit of the method *must* satisfy P.
4. If a property is not enforced by the post conditions then the caller *must* test for this property after the call if needed.

The first two conditions ensure that for every property that holds on the program state *accessible* to the callee (e.g., for heap properties this is the section of the heap accessible from the call arguments [6, 17, 29]), that either it is part of the precondition and holds for every abstract model or that the callee makes no assumption on the property. For the first case, assuming the $\tilde{\sqcup}$ operator is reasonably precise, the result of merging the models should have a small impact on the accuracy of the model.[1] For example if the callee assumes that a given argument (say variable `x`) is always *non-null* then all calls to this method must only occur with states where `x` is *non-null*. Thus all abstract models should entail that `x` is *non-null* and the join of all these models should result in a model that also entails that `x` is *non-null*. In the second case, assuming that tests for the property P in the callee can restore (or at least closely approximate) the information lost by the join, then there is again a negligible loss in precision. Consider a singly linked list class with a `printList` method which is called from a location where the argument may be *null* or *non-null*. If the precondition of the `printList` method does not place conditions on the nullity of the argument, the method must contain a test of the form `if(l == null) {...}`. At this point we can split (assuming the analysis models nullity information) the abstract

[1] Techniques for reducing the possible information loss in the merge (in particular relational information) on call/return are examined in [16].

model $\sigma_?$ where `l` may be `null` or `non-null` into two new models σ_{null} (where `l` *must* be *null*) and $\sigma_{\text{non-null}}$ where (where `l` *must* be *non-null*), thus recovering much of the precision of the original set of abstract models.

From the perspective of the caller the third and fourth properties (along with call scope visibility) ensure that for every property P of the program state after the call, either the property P is part of the post-condition and holds for all models at method exit, it is not enforced by the post-condition, or it involves parts of the model that were not accessible to the callee and the value of the property is assumed to be unchanged. In the first case, as the property holds for all models, the join at method return loses no information (as in the case of properties joined at the call entry). In the second case the caller makes no assumptions about the property, thus assuming that later tests for the property can restore (or at least closely approximate) the information lost by the join, there is a negligible loss in precision (as in the case of recovering nullity information above). Finally in the third case if the property does not involve part of the program state accessible to the callee then the joins will not affect it (since the use of the *project*/*extend* operations [6, 17, 24, 29] ensure these parts of the abstract model are not involved in the joins).

While the above approach works well in general, to obtain the best performance we handle small *leaf* methods such as *getters*, *setters*, and other simple methods as special cases. The computational cost of directly analyzing these small methods by passing in and returning full model sets is less than the cost of performing the required project/join/analyze/extend operations.

3.2 Dynamic Merge

The second issue that contributes to combinatorial growth when analyzing method invocations in object-oriented programs is that there may be many possible targets for the invocation of each call. A simple example of this is an arithmetic expression evaluation program. In this type of program there is often an abstract base class (`Exp`) with a number of pure virtual methods (`eval`, ...) and many concrete subclasses (`Add`, `Mult`, ...) that inherit from the abstract base class and override the required methods. Even in our relatively simple expression evaluator there are 6 subclasses of `Exp` which all override a number of methods and there can easily be much larger numbers. Thus, even if we employ the call/return joins for the individual methods (ensuring that the analysis of each method body produces a single model) the analysis of a call to one of these overridden methods will, in this case, result in a factor of 6 growth in the number of models.

To address this problem we use a similar approach as was used for the reduction in the number of models produced when analyzing a single method body. One of the fundamental concepts of object-oriented design is that when calling a virtual method it should be transparent to the caller which of the child class implementations is actually invoked. That is, the pre/post conditions of the method implementations in the subclasses must respect the pre/post conditions of the overridden method in the base class.

Based on this observation we could decide to, just as for the case of multiple models at call/return, merge the analysis results for each possible invocation target into a single model that we return as the final abstract model for the call. However, our experience indicates that this is slightly too strong an approach. While the pre/post of each subclass implementation must be consistent with the pre/post conditions of the base class definition, the way in which the semantics of the pre/post condition is satisfied can vary substantially depending on how each particular subclass is implemented. These differences are particularly strong when some of the subclasses have non-recursive implementations of the virtual method while other classes have implementations that are recursive. Our `Exp` evaluator example illustrates this point: several subclasses (`Add`, `Mult`, ...) of the `Exp` class implement the virtual `eval` method by recursively evaluating other sub-expressions, while other subclasses (`Variable` and `Const`, which may be shared between several other `Exp` objects) have simple non-recursive implementations of the `eval` method.

3.3 Call Analysis With Merge

Our approach to merging the call/return models for the invocation of a single call to a dynamic method is to first partition the abstract models and call targets based on the structure of the call target (simple leaf methods, builtin methods, non-recursive methods, recursive methods), then process each of the possible targets in each group, and finally to merge all the call results within each group. This gives us the *AnalyzeProc* method shown in Alg. 1.

Algorithm 1: AnalyzeProc

```
input : program $p$, caller $m_{from}$, call sig. $c_{sig}$, state $\theta$
$(m_{to}^{bi}, m_{to}^{sl}, m_{to}^{dag}, m_{to}^{rec}) \leftarrow p$.getImpls($c_{sig}$, $\theta$);
$\theta_f \leftarrow []$;
for $m_{\text{to}} \in m_{\text{to}}^{\text{bi}}$ do
    $\theta_t \leftarrow \theta$.copy();
    $p$.analyzeBuiltin($m_{to}$, $\theta_t$);
    $\theta_f$.addAll($\theta_t$);
for $m_{\text{to}} \in m_{\text{to}}^{\text{sl}}$ do
    $\theta_t \leftarrow \theta$.copy();
    $p$.analyzeDirect($m_{to}$, $\theta_t$);
    $\theta_f$.addAll($\theta_t$);
if ! $m_{\text{to}}^{\text{dag}}$.empty() then
    analyzeCallGroup($\theta$.copy(), $\theta_f$, $c_{sig}$, $m_{from}$, $m_{to}^{dag}$);
if ! $m_{\text{to}}^{\text{rec}}$.empty() then
    analyzeCallGroup($\theta$.copy(), $\theta_f$, $c_{sig}$, $m_{from}$, $m_{to}^{rec}$);
$\theta \leftarrow \theta_f$;
```

The *getImpls* algorithm takes a call signature c_{sig} and an abstract state θ. It then looks up every possible method target of that call signature given the possible types of the receiver object (`this`). These possible targets are partitioned into four categories m_{to}^{bi} the methods with special builtin semantics, m_{to}^{sl} the methods with simple bodies that which should be analyzed directly, m_{to}^{dag} the methods which are not recursive, and m_{to}^{rec} the methods which are recursive. These groups of methods are then analyzed as needed with the argument abstract state and the results are accumulated to produce the result abstract state (θ_f). The method *analyzeBuiltin* handles the application of builtin domain semantics for specific methods (i.e., `java.io` or `java.util`), and the method *analyzeDirect* preforms the direct analysis of the bodies of simple leaf methods.

Algorithm 2: analyzeCallGroup

```
input : in $\theta$, out $\theta_f$, sig. $c_{sig}$, caller $m_{from}$, targets $m_{trgt}$
$(\theta_r, \theta_u) \leftarrow$ projectAll($c_{sig}$, $\theta$.copy());
$(\theta_m, \theta_c) \leftarrow ([\tilde{\bigsqcup}(\theta_r)], [])$;
for $m_{\text{to}} \in m_{\text{trgt}}$ do
    $\theta_t \leftarrow \theta_m$.copy();
    $\theta_t$.assertReciverTypeIs($m_{to}.declInClassType()$);
    $p$.analyzeInvoke($m_{to}$, $m_{from}$, $\theta_t$);
    $\theta_c$.addAll($\theta_t$);
$\sigma_r \leftarrow \tilde{\bigsqcup}(\theta_c)$;
extendAllInto($\theta_u$, $\sigma_r$, $\theta_f$);
```

The *analyzeCallGroup* method (Alg. 2) takes a set of method implementations $m_{targets}$ for a given virtual call c_{sig}, and manages the analysis for each of the calls followed by merging the results. The first step is to project out the inaccessible state of each model followed by merging the accessible portions of each model. Then for each possible implementation we assert that the receiver object is of the type that corresponds to the implementing method class and analyze the implementation, accumulating the results of each method analysis. Finally, we merge the results from each possible target and then extend this summary result back to the inaccessible sections of each model extracted earlier. These extended models are then added to the final state set, θ_f, as part of the *extendAllInto* method. The method *analyzeInvoke* examines the call graph (computed by the frontend) and selects one of the call graph exploration strategies from Sec. 5. The two methods *projectAll* and *extendAllInto* are used to manage the extraction and recombination of the callee accessible and inaccessible program state (via domain specific *project*/*extend* operations as described in [6, 17, 29]).

Algorithm 3: projectAll

```
input  : called method sig. $c_{sig}$, abstract state $\theta$
output: callee accessible and inaccessible components
$(\theta_r, \theta_u) \leftarrow ([], [])$;
for $\sigma \in \theta$ do
    $(\sigma_r, \sigma_u) \leftarrow project(\sigma, c_{sig}.\text{args})$;
    $\theta_r$.add($\sigma_r$);
    $\theta_u$.add($\sigma_u$);
return $(\theta_r, \theta_u)$;
```

Algorithm 4: extendAllInto

```
input  : inaccessible component $\theta_u$, result model $\sigma_r$, out $\theta$
for $\sigma_u \in \theta_u$ do
    $\theta$.add($extend(\sigma_r.copy(), \sigma_u)$);
```

4. Partial Context Sensitivity

In this section we develop a context sensitivity heuristic that uses both call-path information and domain equivalence to control the creation and merging of call-contexts. Our goal in this combination is to produce a heuristic that is able to use domain equality information to ensure that semantically distinct call states to the method are not merged together (as can happen when using a k-limited call string or partial token approach [10, 23, 31]) but that the analysis does not produce redundant call states that only represent intermediate steps in the fixpoint computation (as happens when using pure domain equality as the call context [17]).

The problem of introducing infeasible call paths though merging call states (e.g., the classic k-limited call string approach) is well known. However, the problem of creating large numbers of semantically meaningless call-contexts that represent intermediate states in the fixpoint computation is more subtle. Consider:

```
void ppath(int k) {
    assert (foo(5) == 0);
    for(int i = 0; i < k; ++i)
        System.out.println(foo(i));
    assert (foo(5) == 0);
}

int foo(int x) {
    return (x == 5) ? 0 : x;
}
```

If we were analyzing this program with a simple numeric domain such as intervals [4] in a fully context sensitive manner the loop would produce a number of partial call contexts for the method `foo`. Depending on the exact widening operation we could get the models, x = [0], x = [0, 1] and x = [0, ∞), corresponding to the first, second and third analysis passes over the loop body. However, notice that the first two contexts are in fact just partial approximations of the final loop body fixpoint [0, ∞) and are never used again (the other calls create/use the call context x = [5]). Although these contexts are no longer useful they remain in the memotable, requiring additional storage space and are continually involved in comparisons when checking for memotable matches.

If we were to use a strict call path comparison to determine equality of call contexts this would eliminate the generation of spurious call contexts in the loop body but would result in the redundant analysis of the calls to `foo` in the assert statements (although these calls have identical call values they occur on different lines and thus have different call contexts wrt. to call path information). Other approaches, such as using a weaker form of comparison than full equality (such as $\sqsubseteq$ or partial context tokens [10, 23]) or using some sort of heuristics to remove call contexts that do not appear to be of use from the set of memoized analysis contexts, reduce these issues but are not, independently, robust solutions to the problem. The use of weak forms of equivalence of call states (partial tokens [10, 23] etc.) clearly leads to the propagation of infeasible flow information (since they by definition use imprecise matching) and heuristics that flush parts of the memotable (such as staleness) suffer from the standard problems of overly aggressively flushing useful states or letting the memotable fill with useless values.

The approach we take is to use a combination of call string information and model equivalence to determine how many contexts are needed for each method. This is done by handling contexts from different call paths (based on call string information) via full domain equality and handling contexts that arise from the same call path in a completely insensitive manner (merging them). This approach allows us to use equality to determine the creation of contexts in cases where merging abstract state information from different call sites would propagate infeasible flow information, while the call string information prevents the creation of multiple intermediate contexts that result from repeatedly analyzing the method with approximations of the final call/return state. Thus, the use of call strings limits the number of contexts created to one per call path (with the usual caveats about strongly-connected components in the call graph), while the use of abstract model equality prevents the creation of large numbers of contexts that have different call strings but the same abstract model value at call/return.

4.1 Call String Representation and Context Testing

We assume that we have a fixed call graph in the analysis. For non-recursive calls we track the line number of each method invocation in the call stack. For recursive calls (strongly-connected components in the call graph) we ignore any calls within the strongly-connected component (inserting $\star$ in the call path string) and when we encounter a call that exits the strongly-connected component we add that call site to the call string (i.e., we ignore path information inside the strongly-connected components of the call graph).

DEFINITION 1 (Call Path (η)). *A call path state for a given program p is a sequence of the form $\langle \ell_1, \ldots, \ell_k \rangle$ where the ℓ_i are from the set $\{\ell_i \in \mathbb{N} | \exists$ a method call at line i$\} \cup \{\star\}$.*

Given this representation for the call paths it is straightforward to handle the updates of the current call paths at each call/return via two methods *pushCall(callLine, targetMethod, callGraph)* and *popCall(callLine, targetMethod, callGraph)*.

During the analysis of a program in a classic context sensitive analysis, when we encounter a call to a method m with the call context described by the model (σ) we look at the call contexts that we have previously encountered when analyzing the method body

$\{\sigma_1^{in}, \dots, \sigma_k^{in}\}$, and if we find a call context that is the same as σ we simply return the already computed result (i.e., if $\sigma_j^{in} = \sigma$ then we can use the memoized analysis result σ_j^{out}). We want to revise this formulation to take into account the call path that lead to this method invocation as well as the relations between the memoized analysis values for the method and the current abstract model.

DEFINITION 2 (Memoized Abstract Analysis Models). *For each method m in the program we maintain a list of memoized (and for recursive calls, potentially partial) abstract models* σ_i^{in} *and* σ_i^{out}. *Where when m is analyzed with the abstract model* σ_i^{in} *the resulting abstract model is* σ_i^{out}. *We augment these input/output pairs with call path information on the set of call paths that correspond to the given input/output pair. Thus, the memoized abstract analysis table is of the form* $m_{\text{table}} = \{\lambda_1, \dots, \lambda_k\}$ *where* $\lambda_i =$ *(paths* $= \{\eta_1, \dots, \eta_j\}, in = \sigma_i^{\text{in}}, out = \sigma_i^{\text{out}}$*).*

Alg. 5 shows how the path information is used to determine if a given context is memoized and can be reused directly, if it is part of a more general call state, or is a fresh call state.

The algorithm first determines which category the current call model falls into. If there is a memotable entry that is associated with the same call path as the current call (η) then we check if the current call model (σ) is subsumed by the current input model (λ.in) and if so we can just return the memoized output state (λ.out). Otherwise we need to merge the current call model and the current approximate input state to create a more general model (and we will need to recompute the memoized result at some later point in time as well). Before we generalize the memoized input we make sure that this particular memo entry is associated with only the path η via the *splitEntryAsNeeded* method (if we didn't do this we could, as other call paths may be associated with this memo entry, spuriously generalize the state for other models from unrelated call paths).

The other two cases correspond closely to what is done in standard context sensitive analysis. We first check to see if there exists some memotable entry with the same input model (λ.in) as the current call model (σ) regardless of the call path. If there is we simply return the memoized output model (λ.out) and add the call path information to the set of call paths that this state is related to (for later use in later invocations). If we cannot find such a memotable entry we create a new entry with the special bottom value ($\perp$) and note that the caller needs to decide how to compute the output model.

Algorithm 5: memoCheck

```
input  : input model $\sigma$, input call path $\eta$, memo table $m_{table}$
output: ($\texttt{flag}$, $\lambda$) where $\texttt{flag} \in \{\texttt{Memo}, \texttt{Compute}\}$
if $m_{\text{table}}$ contains $\lambda$ s.t. $\eta \in \lambda.paths$ then
    if $\sigma \preceq \lambda.in$ then
        return ($\texttt{Memo}$, $\lambda$);
    else
        $m_{table}$.splitEntryAsNeeded($\lambda$);
        $\lambda$.in $\leftarrow$ $\lambda$.in $\tilde{\sqcup}$ $\sigma$;
        return ($\texttt{Compute}$, $\lambda$);
else if $m_{\text{table}}$ contains $\lambda$ s.t. $\sigma = \lambda.in$ then
    $\lambda$.paths.add($\eta$);
    return ($\texttt{Memo}$, $\lambda$);
else
    $\lambda \leftarrow$ ($\{\eta\}$, $\sigma$.copy(), $\perp$);
    $m_{table}$.add($\lambda$);
    return ($\texttt{Compute}$, $\lambda$);
```

5. Analysis Algorithm

Now that we have techniques for handling calls/returns with multiple models, reducing the multiplicative effects of virtual method invocation, and have a context heuristic for minimizing the size of method memotables, we can combine these techniques into a complete interprocedural dataflow analysis method. This method is then invoked from the local analysis phase by calling *analyzeProc* (Alg. 1) whenever a call is encountered.

The order in which methods are analyzed has a substantial impact on the efficiency of the program analysis [11, 22, 24, 26, 28] and thus we use a hybrid approach for call graph exploration that varies the order based on structural information about the call graph and the method bodies (similar to what is done in [11, 22, 28]). In particular we distinguish between calls that are in a recursive cycle (a strongly connected component in the call graph) and calls that are non-recursive. In the non-recursive case the analysis also distinguishes between small leaf (or *getter*/setter type methods) and more complex methods, Alg. 1.

Analyze Recursive Call Inside SCC. The *analyzeRecSame* method (Alg. 6) is used to analyze calls that are within the same recursive component of the call graph. For these calls we want the analysis to explore the call graph in a breadth-first manner as opposed to the depth-first exploration used for non-recursive calls. This avoids problems with depth first path exploration in densely connected recursive call structures which can result in a number of long call paths through the component being explored. Thus Alg. 6 simply checks for the current approximate abstract semantics of the given abstract state (creating a new entry if needed) and returns this approximate result. If the new call state is not subsumed by the existing input approximation then we note that this result value is approximate and needs to be reanalyzed.

We use two global worklists *wlDown* and *wlUp* which keep track of the methods in a strongly connected component of a call graph that need to be re-analyzed. The *wlDown* list is added to when we encounter a memo entry that needs to be re-analyzed because the input model changed while the *wlUp* list contains memo entries that need to be re-analyzed because the results of one methods that they depend on has changed. The separation of these two allows us a relatively fine grain of control over the order in which the methods in the call graph are processed.

In the case where this is the first time the method m_{to} has been called from within the SCC (λ.out $= \perp$) we compute the base case approximation of the output model by calling *analyzeAssumeNoRec* which analyzes the body of m_{to} but assumes that only control flow paths that are non-recursive can be executed. This is a simple but important optimization to reduce the number of times that the SCC needs to be traversed to reach a fixed point [22, 26] (we immediately compute a first step up the lattice preventing unneeded iterations of the fixpoint algorithm).

Algorithm 6: analyzeRecSame

```
input  : program $p$, callee $m_{to}$, abstract model $\sigma$
($mval$, $\lambda$) $\leftarrow$ $m_{to}$.memoCheck($\sigma$);
if mval = $\texttt{Memo}$ then
    return $\lambda$.out.copy();
else
    if $\lambda.out = \perp$ then
        body $\leftarrow$ $m_{to}$.getBody();
        $\sigma_c \leftarrow \lambda$.in.copy();
        $\lambda$.out $\leftarrow$ $p$.analyzeAssumeNoRec(body, $\sigma_c$);
    wlDown.add(($m_{to}$, $\lambda$));
    return $\lambda$.out.copy();
```

Analyze Call Into SCC. The *analyzeRecNew* (Alg. 7) algorithm is used for the first call into a recursive component of the call graph. It sets up the worklists and then runs the analysis of the methods until the fixed point is reached (thus safely approximating the result of the call). The *analyzeRecNew* method (Alg. 7) initializes the worklists by analyzing the method body (the *analyzeBody* method, which triggers the creation of the needed new memo entries in the *analyzeRecSame* method). While the *wlUp* and *wlDown* worklists are non-empty we select a memo entry that may need to be reprocessed and perform the analysis (Alg. 8). Memo states that need to be recomputed are added to the worklists as needed in *analyzeRecPending* (Alg. 8) and *analyzeRecSame* (Alg. 6). The entries are processed in a breadth first fashion (methods are taken from the front of the *wlDown* worklist and added at the back, while they are added to the back and removed from the back of the *wlUp* worklist). We also favor the *down* calls to the *up* calls as we want to reach a back edge in the SCC (giving a large step up the lattice) before we begin propagating partial information back through the SCC.

Algorithm 7: analyzeRecNew

```
input : program $p$, callee $m_{to}$, abstract model $\sigma$
$(mval, \lambda) \leftarrow m_{to}$.memoCheck($\sigma$);
if mval = Memo then
    return $\lambda$.out.copy();
else
    $(wlUp, wlDown) \leftarrow ([], [])$;
    $(change, pend) \leftarrow (\text{true}, \text{true})$;
    body $\leftarrow m_{to}$.getBody();
    $\lambda$.out $\leftarrow p$.analyzeBody(body, $\lambda$.in.copy());
    while $change \vee pend$ do
        while (! wlUp.empty()) $\vee$ (! wlDown.empty()) do
            if (! wlDown.empty()) then
                ($m_{pend}$, $\lambda'$) $\leftarrow$ wlDown.popFront();
                $p$.analyzeRecPending($m_{pend}$, $\lambda'$);
            else
                ($m_{pend}$, $\lambda'$) $\leftarrow$ wlUp.popBack();
                $p$.analyzeRecPending($m_{pend}$, $\lambda'$);
        $\sigma_{old} \leftarrow \lambda$.out;
        $\lambda$.out $\leftarrow$ analyzeBody(body, $\lambda$.in.copy());
        change = ($\sigma_{old} \neq \lambda$.out);
        pend = (! wlUp.empty()) $\vee$ (! wlDown.empty());
```

Algorithm 8: analyzeRecPending

```
input : program $p$, method $m_{to}$, memo entry $\lambda$
$\sigma \leftarrow p$.analyzeBody($m_{to}$.getBody(), $\lambda$.in.copy());
if $\sigma \neq \lambda.out$ then
    $\lambda$.out $\leftarrow \sigma$;
    foreach ($m_{\text{from}}$, $\lambda'$) that may depend on ($m_{\text{to}}$, $\lambda$) do
        wlUp.add(($m_{from}$, $\lambda'$));
```

Analyze DAG Call. The *analyzeDAGCall* method is used to analyze the larger non-recursive methods. The *analyzeDAGCall* simply passes the state into the local analysis method and returns the result. Analyzing the callee before returning to the analysis of the caller results in the exploration of the call graph in a depth first order for non-recursive calls, which is an optimal order for these calls. Thus, this method is a straight forward check if the given path and model are memoized and either computing the result model (or returning a memoized value) as needed.

6. Experimental Evaluation

In order to evaluate the effectiveness of the interprocedural analysis presented in this work we examine both the precision of the results produced and the computational cost when analyzing various programs.[2] Since the analysis algorithm is a general purpose framework we need to select a specific domain with which to instantiate the algorithm. Since shape analysis is known to be a challenging analysis domain and we have experience with the MTSA domain (Sec. 6.1) we use this domain for the evaluation.

We use a number of benchmarks from a version of the Jolden suite [8], two programs from SPECjvm98 [32], and two programs (exp and interpreter) which were developed as challenge problems for interprocedural shape analysis. The JOlden suite contains pointer-intensive kernels (taken from high performance computing applications with minor changes to reflect modern object oriented programming idioms). The benchmarks raytrace and db are taken from SPECjvm98 (with minor modifications to remove test harness code and threading). The other benchmarks are a simple arithmetic expression evaluator (exp) designed to stress the ability of the analysis to deal with virtual method invocations with multiple targets, and a simple interpreter for the computational core of Java which stresses both the interprocedural analysis and shape domain. Although we use the hypothesis that methods have well defined pre/-post conditions to motivate the development of the heuristics in this paper, we emphasize that the analysis *does not* require them to be explicit and the benchmark code is not annotated in any way.

The exp program is an excellent challenge problem for this area of research as it contains all of the fundamental difficulties of interprocedural analysis (virtual methods, recursion, and non-trivial calls out of recursive calls), a variety of heap analysis challenges (heap structures with and without sharing, copy and destructive traversals of the structures), and is small enough to understand (both what is happening in the benchmark and how the analysis proceeds through the program). The interpreter program stresses many aspects of both the shape and interprocedural analysis. It is a large program with many virtual classes and overridden methods. The heap structures are varied, from a large well defined tree structure in the AST, symbol and local variable tables, a call stack of pending call frames, and a very poorly defined cyclic structure in the internal model of the heap built by the interpreter (thus the heap analysis must be both precise and able to deal with ambiguity efficiently). It also has substantial amounts of sharing (variables, method signatures, and objects on the interpreter's internal representation of the heap are shared in multiple structures). The interprocedural analysis must also deal with a mixture of recursive/non-recursive calls, methods that are called from multiple locations in the program, and methods that are called in loops/recursive calls that produce multiple approximate fixpoint states.

6.1 MTSA Heap Domain

The MTSA [18, 19] heap domain is based on the *storage shape graph* approach [3]. In this approach the concrete state of the program is viewed as an environment, mapping variables into values, and a store, mapping addresses into values. We refer to the environment and the store together as the concrete heap, which is treated as a labeled, directed multi-graph (V, O, R) where each $v \in V$ is a variable, each $o \in O$ is an object on the heap and each $r \in R$ is a reference (either a variable reference or a pointer between objects).

We can partition the concrete heap into disjoint regions and define a number of structural concepts. A *region* of memory $\mathfrak{R} = (C, P, R_{in}, R_{out})$ consists of a subset $C \subseteq O$ of the objects in the heap, all the pointers $P = \{(a, b, p) \in R \mid a, b \in C \wedge p \in L\}$ that

[2] Benchmarks and analysis source code are available online at `www.software.imdea.org/~marron/software/software.html`.

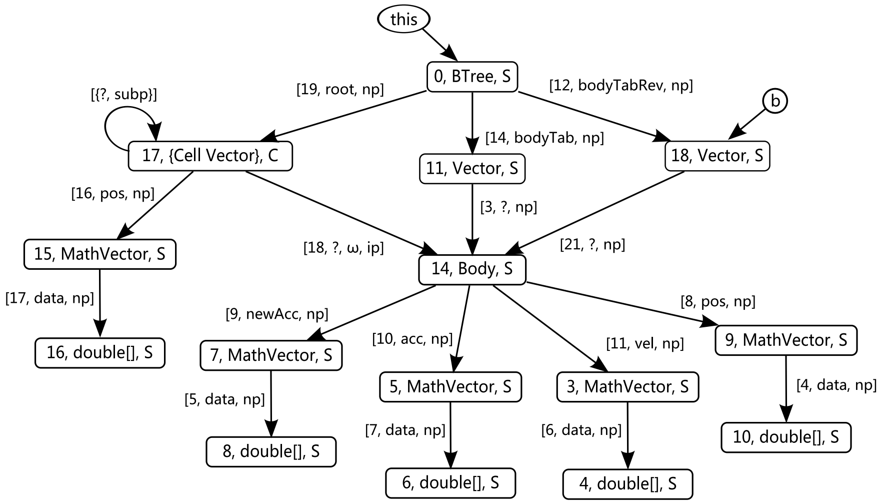

Figure 1: Analysis result for the program state at entry to the `stepSystem` method in `bh`.

connect these objects, the references that enter the region $R_{in} = \{ (a,b,r) \in R \mid a \in (V \cup O) \setminus C \wedge b \in C \wedge r \in L \}$ and references exiting the region $R_{out} = \{ (a,b,r) \in R \mid a \in C \wedge b \in O \setminus C \wedge r \in L \}$.

The abstract graph is a tuple of the form $(\hat{V}, \hat{N}, \hat{E}, \hat{U})$, where $\hat{V}$ is a set of abstract nodes representing the variables, $\hat{N}$ is a set of abstract nodes (each of which abstracts a region $\mathfrak{R}$ of the heap), and $\hat{E}$ are the graph edges (each of which abstracts a set of pointers). The set $\hat{U}$ is a set of labels that we attach to nodes and edges in the abstract graph to further restrict the set of concrete heaps that each shape graph abstracts. The labels allow us to more precisely model shape and sharing (among other properties) than is possible using only the connectivity information present in the graph structure. Based on the examination of client applications that use heap information (program optimization, error detection, and software engineering) we track the following properties in $\hat{U}$ which (in conjunction with the structural information in the graph) provide the information needed for many client applications.

Abstract Layout. To approximate the shape of the heap in the region that a node abstracts, the analysis uses *abstract layout* properties $\{$*(S)ingleton*, *(L)ist*, *(T)ree*, and *(C)ycle*$\}$. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node, given a node n where $\mathfrak{R} = (C, P, R_{in}, R_{out})$ then $P = \emptyset$. The other properties correspond to standard definitions for list, tree, and cyclic structures [1, 5].

Sharing. As each edge may represent a set of pointers we want to track how the targets of these pointers may be related (i.e., we want to distinguish an array where each index refers to a unique object from an array where the same object may be stored in multiple indicies). We accomplish this by using an interference property as described in [19]. Given an edge e that ends at node n, we say it is *non-interfering* (*np*) if it abstracts a set of references $\{r_1, \ldots, r_m\}$ s.t. all r_i, r_j refer to objects that are in different weakly-connected components of the region (C, P) represented by n.[3] We label an edge e as *interfering* (*ip*) if it abstracts a set of references $\{r_1, \ldots, r_m\}$ s.t. there may exist r_i, r_j that refer to objects in the same weakly-connected component of (C, P).

In the pictorial representations of the heap, nodes are represented as records of the form `id`, `types`, `layout`, where `types` is a set of all the object types the node may represent and *id* is a unique identifier we give to the node to simplify discussion. Similarly all (non-self) edges are labeled with records of the form `id`, `offset`, `interfere`, where `offset` is the field name the pointers are stored in (or *?* for pointers stored in collections) and *id* is again a unique identifier.

6.2 Analysis Case Study

We begin by examining in detail the results of analyzing a single program as a case study on the type of properties and the level of precision that the analysis can extract. The `bh` (Barnes-Hut) program preforms a gravitational interaction simulation on a set of bodies (the `Body` objects) using a *fast-multipole* technique with a space decomposition tree. The tree is composed of `Cell` objects each of which has a `Vector` containing references to other `Cell` objects or references to the `Body` objects. The program also keeps two `Vector` objects for accessing the bodies, `bodyTab` and `bodyTabRev`. The positions (`pos`), velocities (`vel`), and acceleration (`acc`) values of the bodies are represented with composite structures consisting of a `MathVector` object and a `double[]`. Using a common OOP idiom the `Cell` and `Body` objects both inherit from an abstract `Node` class.

The result of the shape analysis at entry to the `stepSystem` method in the `bh` program is shown in Fig. 1. The analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be cyclic (the `C` in node 17, representing the `Cell` objects). However, the analysis is able to determine that the `Cell` objects and the `Body` objects represent *disjoint regions* of the heap (nodes 14 and 17 have disjoint type sets). The analysis is also able to determine a number of useful sharing properties with respect to how the `Body` objects (node 14) are stored in the two vectors (`bodyTab`, node 11, and `bodyTabRev`, node 18) and the space decomposition tree. In particular it has assumed that the `Cell` objects in the space decomposition tree may have aliasing pointers (the *ip* entry in edge 18) while there are no aliasing pointers stored in `bodyTab` or `bodyTabRev` (edges, 3 and 21, abstracting these pointers have the *interfere* property *np*). The analysis has also determined that each `MathVector` object is stored in a single field of a single `Body` object (the *np interfere* label on the edges into the `MathVector` nodes) and the `MathVector` objects *own* their `double[]` arrays (the *np* label for the `data` edges).

[3] Two nodes are in a weakly-connected component if there is a (possibly non-empty) path between them (treating all edges as undirected).

Benchmark	Region	Shape	Share
tsp	100%	100%	98%
em3d	100%	100%	99%
voronoi	97%	98%	98%
bh	100%	96%	96%
db	100%	100%	81%
raytrace	100%	95%	92%
exp	100%	100%	100%
interpreter	100%	97%	96%

Figure 2: Avg. accuracy of analysis results. Percentage of known *maximally precise* properties computed by the static analysis for region, shape and sharing properties of the abstract graph.

The results above are identical to the results obtained in previous work [18, 19] which used a dataflow analysis similar to the analysis described in [17]. Thus, in this example the dataflow analysis optimizations cause no precision losses that would prevent the application of the memory/TLP optimizations from [18, 19] and the analysis time is reduced by a factor of 2.5.

6.3 Quantitative Precision

The case study provides an example of a nontrivial program in which the analysis is able to extract a range of information on shape, sharing, and regions. Aside from the misidentification of the space decomposition tree as cyclic structure instead of a tree and that the references from the space decomposition tree to the Body objects may alias, the information precisely captures the set of feasible program states. Formally, we say an abstract graph g is *maximally precise* if there is no abstract heap graph g' in where $g' \sqsubseteq g$ and g' is also a *safe* approximation of the program state.

In this work we perform empirical evaluation of the precision of the dataflow analysis results based on this concept of *maximal precision*. We believe that this notion of precision is a better basis for examining the impact of the heuristics on analysis results than the use of a specific client application (which may hide many precision losses that *happen* to not matter for the particular client).

Recent work [20] has experimented with a general framework for translating concrete program execution states into abstract models for debugging purposes (and provides the needed machinery to accumulate these models at specific program points). Thus, for each method m we can construct an abstract graph g^c_{in}, constructed from the call states seen at the entry to m during a specific execution of the program. Since this graph must represent under approximation of the *maximally precise* heap graph (it is derived from a specific run of the program) then given our statically computed heap graph (g_{in}), if $g_{\text{in}} = g^c_{\text{in}}$ then g_{in} is *maximally precise*. In fact in we can further refine our comparison to be property specific (we can examine the maximal precision of individual properties such as region identification, sharing, or shape). In the bh example there are 15 nodes in the shape graph, all of which appear in the dynamically computed graph, thus the analysis is 100% accurate wrt. to region identification. In the case of the shape information the analysis correctly identified the shape of 14 of the 15 regions (i.e., 93% accurate) and the analysis also correctly identified the sharing properties of 15 of the 16 edges (i.e., 93% accurate).

Fig. 2 lists the accuracy of the analysis on each program averaged over all methods in the program. We rate the accuracy on region information, shape information, and sharing information (i.e., whether pointers abstracted by an edge *interfere*). We can see that overall the analysis results are very close to the limit of what can be computed via the abstract domain that we have selected to work with. Thus, for the benchmarks examined here, the proposed heuristics and dataflow analysis have a negligible impact on the precision of the results.

6.4 Analysis Performance

We next examine the time and memory requirements of analyzing a program using the method described in this paper. The cost of analyzing a program with a full powerset domain on call and return is generally infeasible (the No Opt column in Fig. 3) with the heap domain we are using (all but two of the benchmarks time out). The next variation of interprocedural analysis we examine (the Call/Ret column) uses an improved version of the technique described in [17] along with the call-site merge heuristic from Sec. 3.1. The analysis described in this paper (the Optimized column) uses the call-site merge heuristics described in Sec. 3.1, the dynamic call merge heuristics from Sec. 3.2, and the memotable matching operations described in Sec. 4. The analysis algorithm is written in C++ and compiled using MSVC 8.0. The analysis was run on a 2.6 GHz Intel quad-core machine with 2 GB of RAM.

For each benchmark we list the *normalized* LOC count as well as the number of classes (not including standard library classes from java.lang, java.io, and java.util). Normalized LOC is the number of lines in the program where expressions/calls are un-nested and the required stubs have been automatically added by the frontend for handling the builtin Java libraries (the stubs add a few hundred LOC on average, up to 1500 LOC for the interpreter program which uses many standard library classes).

Fig. 3 shows the aggregate performance of the analysis approaches. The timing measurements exclude the time required to startup and read/transform the source program into the internal IR. For the timeout we use a limit of 10 min. or 500MB of memory.

The *no opt* approach is unable to handle most of the benchmarks, so we will not discuss it further. When compared to the *call/ret* approach the *optimized* technique is a factor of 2-4 faster in all of the smaller benchmarks and enables the analysis to complete the two challenge programs where the *call/ret* analysis times out (exp and interpreter). Both the *call/ret* and *optimized* analysis require very little memory to analyze the simpler programs (less than 30MB). In fact in these simpler programs most of the memory that is used is taken up by the executable and internal representation of the program being analyzed. Thus, we do not attempt to further break down the exact impact of the modified analysis on memory use for these programs. However, as the size and complexity of the program increases we see that the *optimized* analysis improves memory usage as well. For the raytrace program the *optimized* interprocedural analysis uses slightly less memory (38MB vs 45MB) and in the interpreter program the analysis only uses 122MB (leading us to believe that the analysis will continue to scale up well). This is in contrast to the *call/opt* analysis which was using 400MB+ before it timed out on the interpreter program.

To understand the source of this difference in performance we examine the impact of the *optimized* analysis on the maximum number of models in the model set during the local flow analysis, the maximum number of memo entries for any method and the average number of memo entries per method (shown in Fig. 4). Looking at the results in the table we can see that the reduction in the maximum model set size is moderate, on average reducing the number by 1-2 models. However, as this number already is quite low in the *call/ret* column it is not clear that more reduction is possible without a large impact on precision. Conversely the impact on the number of memo entries per method is quite significant. This reduction occurs in the maximum number of entries (reduced to 13 from 87 for raytrace, with a factor of 2 or more for many others) and in the average number per method (near a factor of 4 in some cases and an average reduction of around 1 context per method for the rest of the benchmarks). This has a huge impact on

Benchmark Statistics			No Opt		Call/Ret Merge		Optimized	
Name	LOC	Classes	Time(s)	Mem(MB)	Time(s)	Mem(MB)	Time(s)	Mem(MB)
tsp	910	2	1.81	≤30	0.15	≤30	0.03	≤30
em3d	1103	5	1.12	≤30	0.31	≤30	0.09	≤30
voronoi	1324	6	-	-	1.80	≤30	0.50	≤30
bh	2304	7	-	-	1.84	≤30	0.72	≤30
db	1985	3	-	-	1.42	≤30	0.68	≤30
raytrace	5809	24	-	-	37.09	45	15.5	38
exp	3567	17	-	-	-	-	152.3	48
interpreter	15293	90	-	-	-	-	114.8	122

Figure 3: Benchmark statistics and aggregate performance of the *No Opt, Call/Ret,* and *Optimized* analysis.

Benchmark	Call/Ret Merge			Optimized		
Name	Max Set	Max Cont.	Avg Cont.	Max Set	Max Cont.	Avg Cont.
tsp	4	9	2.1	4	4	1.2
em3d	7	6	2	4	2	1.2
voronoi	3	53	10	3	6	1.8
bh	8	13	3.5	6	8	1.6
db	12	9	3.1	12	3	1.9
raytrace	65	87	9.1	32	13	2.2
exp	-	-	-	18	11	2.1
interpreter	-	-	-	14	37	1.8

Figure 4: Comparison of *Call/Ret* and *Optimized* analysis on model set size and memo table entries. *Max Set* is the maximum number of models in an abstract state set during local dataflow analysis, *Max Cont.* is the maximum number of entries in the memotable of any method, and *Avg Cont.* is the average number of entries in each memotable.

the number of comparisons that need to be done to test if a given call model is memoized and in the number of times each method body is analyzed. As the program gets larger this reduction has a large impact in the amount of time and memory needed to analyze a program. We note that the impact of the techniques presented in this paper is small on the simpler benchmarks but become increasingly important as the complexity increases in db and raytrace, and is critical to successfully analyzing exp and interpreter.

These results emphasize the robustness of the scaling of the interprocedural analysis with respect to growth in model set and memo table sizes. In particular the average number of memo entries is nearly constant regardless of the size of the program (thus the memory required by the analysis scales in a fairly linear way with the number of methods). We have been unable to analyze larger programs due to the limited nature of the compiler frontend we used in our prototype implementation (which translates from a subset of the Java language to the intermediate form used by the analysis). However we believe that the analysis can continue to scale to larger programs and are currently integrating the analysis into a production level frontend. This integration will enable us to analyze and evaluate the analysis over a much larger set of programs and enable other researchers to utilize the analysis results.

7. Related Work

There has been a substantial amount of work on techniques for speeding up interprocedural dataflow analysis. These range from methods for efficiently encoding call contexts using BDD's [13, 33], to heuristics that alter the level of context sensitivity based on structural properties of the call or flow graph [11, 21, 28], to methods for generating partial call context tokens [9, 23, 34]. The work in this paper draws on the general concepts of context heuristics presented in [11, 15, 21, 24, 28] and work on project/extend operations [6, 17, 29], to efficiently and precisely manage the challenges presented by object oriented programs.

This work differs substantially from previous work in a number of respects. While there has been work on heuristics for introducing join points in programs and in support for flow structure aware join heuristics [15, 21] these have focused on frameworks for the joins and not specific heuristics. The related work on context sensitivity has used either partial context tokens [9, 23, 34] or limited context sensitivity based on call graph structure [11, 24]. Thus the approach taken in this paper of a combined call string and domain context information is distinct and complementary to the previous work (and in fact can be combined with previous techniques).

One of the key insights into this work is that there exist specific points in the program where the programmer makes generalizing assumptions about the state of the program and thus most of the subtle differences between abstract models in a given abstract state are not critical to understanding the later behavior of the program. This idea has been exploited implicitly in previous work on interprocedural analysis which often applies some set of heuristics for merging states either at call or local control flow joins [17, 21, 35, 24] but do not provide justification (beyond the experimental results) for why the specific set of join points selected is a good choice.

Recent work on separation logic-based shape analysis [1, 35] has also shown promise in scaling to large programs (up to a 12KLoc device driver). However, this work has focused on a class of analysis problems that present different challenges from what is encountered in this paper (and have different requirements on the level of precision needed from the analysis). In particular the programs they have analyzed are device drivers in C (thus do not have inheritance or virtual calls). Restrictions are also placed on sharing/aliasing and the types of data structures that are built. While these limitations are reasonable in the context of verifying low-level C programs such as device drivers (which use a small number of well-defined heap-allocated structures in limited ways and where the analysis results must be very accurate) they become overly restrictive when moving to general purpose programs.

This approach differs from recent work on abduction based approaches to shape analysis [2, 7] which infer call-context information from the callee method instead of relying on a whole program analysis to provide the call-contexts directly. While abductive analysis is very attractive from a scalability standpoint, as methods can be analyzed independently, performing the needed abductive inferences in the presence of complex heap structures is very difficult. In [2] Table 1 nearly half of the methods inspected could not be successfully analyzed even with the restricted heap model. However, the combination of an abductive analysis followed by a context sensitive interprocedural analysis presents a promising way to leverage the strengths of both approaches.

8. Conclusion

This work introduced and provided justification for a number of heuristics for minimizing the size of the abstract states and the memo tables that are created when performing flow-sensitive, context-sensitive interprocedural analysis. The development of novel heuristics for call/return sites and for dynamic methods with multiple implementations led to substantial reductions in the size of the abstract states that the analysis is required to process (by preventing combinatorial growth of the model based on the number of call paths/possible targets of a call). This work also developed a hybrid call path and domain context-sensitive equivalence relation which had a large impact on the number of entries in the memo tables while avoiding the propagation of infeasible flow information.

The impact of these contributions was demonstrated by integrating them into a complete flow-sensitive context-sensitive dataflow analysis and then using this algorithm, and an existing high precision shape analysis domain, to analyze a number of Java programs. The results of the analysis on our benchmarks provide empirical evidence that the heuristics drastically reduce the time required to analyze a program, constrain the memory required to perform the analysis, and have a small impact on precision. In the largest benchmark (a 15+ KLoc interpreter for the core of Java) the baseline shape analysis times out at 10 min. and uses 400MB+ of memory before being terminated. However, when we apply the heuristics described in the paper the program can be analyzed in 114 seconds and 122MB of memory. At the same time the precision loss is negligible and we are able to extract precise (in fact near optimal) region, shape, and sharing information about the program.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[2] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[3] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[5] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[6] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

[7] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis. In *SAS*, 2009.

[8] Jolden Suite. http://www-ali.cs.umass.edu/DaCapo/.

[9] N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *POPL*, 1979.

[10] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, 1982.

[11] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[12] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.

[13] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Method.*, 18(1), 2008.

[14] B. Liskov. Data abstraction and hierarchy. In *OOPSLA*, 1987.

[15] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.

[16] M. Marron, M. Hermenegildo, and O. Lhoták. Program analysis with write invariant properties. In *Submission*, 2009.

[17] M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.

[18] M. Marron, D. Kapur, and M. Hermenegildo. Identification of logically related heap regions. In *ISMM*, 2009.

[19] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[20] M. Marron, C. Sanchez, and Z. Su. Visualizing high-level heap abstractions for program understanding and debugging. In *Submission*, 2009.

[21] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.

[22] M. Méndez, J. Navas, and M. Hermenegildo. An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In *BYTECODE*, 2007.

[23] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 2002.

[24] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3), 1992.

[25] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[26] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *SAS*, 1996.

[27] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[28] T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*, 2007.

[29] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[30] S. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.

[31] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[32] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.

[33] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[34] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

[35] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.