Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models

Mark Marron[1], Manuel Hermenegildo[1,2], Deepak Kapur[1], and Darko Stefanovic[1]

[1]University of New Mexico, {marron, kapur, darko}@cs.unm.edu
[2]Technical University of Madrid and IMDEA-Software, herme@fi.upm.es

Abstract. The performance of heap analysis techniques has a significant impact on their utility in an optimizing compiler. Most shape analysis techniques perform interprocedural dataflow analysis in a context-sensitive manner, which can result in analyzing each procedure body many times (causing significant increases in runtime even if the analysis results are memoized). To improve the effectiveness of memoization (and thus speed up the analysis) *project*/*extend* operations are used to remove portions of the heap model that cannot be affected by the called procedure (effectively reducing the number of different contexts that a procedure needs to be analyzed with). This paper introduces *project*/*extend* operations that are capable of accurately modeling properties that are important when analyzing non-trivial programs (sharing, nullity information, destructive recursive functions, and composite data structures). The techniques we introduce are able to handle these features while significantly improving the effectiveness of memoizing analysis results (and thus improving analysis performance). Using a range of well known benchmarks (many of which have not been successfully analyzed using other existing shape analysis methods) we demonstrate that our approach results in significant improvements in both accuracy and efficiency over a baseline analysis.

1 Introduction

Recent work on shape analysis techniques [25, 28, 1, 14, 15, 9, 8] has resulted in a number of techniques that are capable of accurately representing the properties (connectivity, interference, and shape) that are needed for a range of optimization and parallelization applications. However, the computational cost of performing these analyses has limited their applicability. A significant component of the analysis runtime is due to the need to perform a context-sensitive interprocedural analysis, where each procedure body may be analyzed multiple times (once for each different calling context).

The practice of using a memo-table to avoid recomputing analysis results and the use of a *project* operation to remove portions of the heap that cannot affect or be affected by the called procedure are standard techniques for minimizing the number of times each function needs to be analyzed during interprocedural dataflow analysis [2, 17, 16, 19]. The two major goals of the *project* operation are improving the effectiveness of memoizing analysis results by removing portions of the heap that could cause spurious inequalities between calling contexts and preventing the loss of precision that occurs when recursive procedures use a summary representation for multiple out-of-scope references (e.g. local reference variables with the same name but that exist in different call frames).

The *project* operation for heap models and the utility of locality axioms have been analyzed in a number of papers [22, 21, 7, 12, 4]. These techniques use variations on the notion of a *frame rule* as presented in [11, 20] and identify a number of features of the *project* operation that are of particular importance for interprocedural analysis using heap domains. A major distinction is made between the projection operation in *cutpoint*-free cases, where there are no pointers that cross from a section of the heap that is *unreachable* from the procedure arguments into a section of the heap that is *reachable* from the procedure arguments, and cases where such pointers may exist.

This paper presents a method for using cutpoints to support interprocedural heap analysis. We then use the technique to quickly analyze (10's of seconds) programs that are larger (by a factor of 2-4) and more varied (in terms of data structures and algorithms) than any other analysis technique to date. Our first contribution is the reformulation of the project/extend operations in [21] so that they can be used in a graph based (as opposed to an access path based) heap model which allows us to use a very compact and efficient representation of heap connectivity. Our second contribution is the extension of the original approach to handle two classes of programatic events that are critical to analyzing real world programs, analyzing programs that involve non-trivial sharing and composite data structures [1, 15] and propagating nullity test information from callee to caller scope. Finally we use the results of the heap analysis to drive the parallelization of a range of benchmarks (several of which have not been successfully analyzed/parallelized using shape information) achieving an average parallel speedup of 1.69 on a dual-core machine.

2 Example Code

To develop intuition about the mechanism and purpose of *project*/*extend* operations we look at a simple function (Figure 1) that illustrates the basic functioning of the *project*/*extend* operations and the propagation of nullity information from the callee to the caller scope. Our lists are made of objects of *type* `LNode`, each `LNode` object has two fields, a `nx` field which refers to the next element in the list and a field `f` which stores a boolean.

Accurately analyzing the initialization method (LInit) requires the analysis to propagate information inferred about cutpoints in the callee scope back into the caller scope. If the analysis is unable to use the `l == null` test in the callee scope to infer that `l.nx` is `null` in the caller scope then the analysis will not be able to infer that after the method returns the argument list is either `null` or must have the `true` value in all the `f` fields.

3 Heap Model

We model the concrete heap as a labeled, directed multi-graph (V,E) where each vertex $v \in V$ is an object in the store or a variable in the environment, and each labeled directed

```
LNode LInit(LNode l)
  if(l == null)
    return;

  tin = l.nx;
  LInit(tin);
  l.f = true;
```

Fig. 1. Recursive List Initialize

edge $e \in E$ represents a pointer between objects or a reference from a variable to an object. Each edge is given a label that is an identifier from the program, an edge $(a,b) \in E$ labeled with p, we use the notation $a \xrightarrow{p} b$ to indicate that a points to the object b via the field name (or identifier) p.

A *region* of memory $\mathfrak{R}$ is a subset of the objects in memory, with all the pointers that connect these objects and all the cross-region pointers that start or end at an object in this region. Formally, let $C \subseteq V$ be a subset of objects, and let $P_i = \{p \mid \exists a,b \in C, a \xrightarrow{p} b\}$ and $P_c = \{p \mid \exists a \in C, x \notin C, a \xrightarrow{p} x \vee x \xrightarrow{p} a\}$ be respectively the set of internal and cross-region pointers for C. Then a region is the tuple (C, P_i, P_c). For a region $\mathfrak{R} = (C, P_i, P_c)$ and objects $a,b \in C$, we say a and b are *connected* in $\mathfrak{R}$ if they are in the same weakly-connected component of the graph (C, P_i). Objects a and b are *disjoint* in $\mathfrak{R}$ if they are in different weakly-connected components of the graph.

3.1 Abstract Heap Model

The underlying abstract heap domain is a graph where each node represents a region of the heap or a variable and each edge represents a set of pointers or a variable target. The nodes and edges are augmented with additional instrumentation predicates. The abstract domain evaluates the predicates using a *3-valued* semantics: predicates are either definitely true, definitely false, or unknown [25]. Our analysis tracks the following set of instrumentation predicates. Our choice of predicates is influenced by common predicates tracked in previous papers on shape analysis [5, 24, 28, 20].

Types. For each type t in the program, there is an instrumentation predicate (also written t) that is true at a concrete heap node if any concrete object represented by the node may have type t.

Linearity. Each abstract node has a *linearity* that represents whether it represents at most one concrete node (linearity 1) or any set of 0 or more concrete nodes (written #).

Abstract Layout. To track the connectivity and shape of the region a node abstracts, the analysis uses *abstract layout* predicates *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle*. The *Singleton* predicate states that there are no pointers between any of the objects represented by an abstract node. The *List* predicate is similar to the inductive *List* predicate in separation logic [20]. The other predicates correspond to the definitions for Trees, Dags, and Cycles in the literature, for the formal definitions see [14].

Interference. The heap model uses two properties to track the potential that two references can reach the same memory location in the region that a node represents.

The first property is for references that are represented by different edges in the heap model. Given the concretization function γ and two edges e_1, e_2 that are incoming edges to the node n, the predicate that defines *inConnected* in the abstract domain is: e_1, e_2 are *inConnected* with respect to n if it is possible that $\exists r_1 \in \gamma(e_1) \wedge \exists r_2 \in \gamma(e_2) \wedge \exists a, b \in \gamma(n)$ s.t. $(r_1$ refers to $a) \wedge (r_2$ refers to $b) \wedge$ $(a$, b connected). For improved precision we also track *may* and *must* aliasing (e_1, e_2 are *inConnected* and $a = b$) between the references the edges abstract (*must* aliasing is only meaningful if the edge represents a single references, see [15] for an approach that generalizes *must-aliasing* to sets of references).

The second property is for the case where the references are represented by the same edge. To model this the *interfere* property is introduced. An edge e represents interfering references if there may exist references $r_1, r_2 \in \gamma(e)$ such that the objects that r_1, r_2 refer to are connected/aliased. A three-element lattice, $np < ip < ap$, np for edges with all non-interfering references and ip for potentially interfering references and ap for potentially aliasing references, is used to represent the interference property.

The Heap Graph Each node in the graph either represents a region of the heap or a variable. The variable nodes are labeled with the variable that they represent. Nodes representing the concrete heap regions contain a record that tracks the types of the concrete objects that the node represents (*types*), the number of objects (either 1 or #) that may be in the region (*count*), and the abstract layout of a node (*layout*). Each node also tracks the connectivity relation between pairs of incoming edges. A binary relation *connR* is used to track the *inConnected* relation. Although the connectivity relation is a property of the nodes, for readability in the figures we associate the information with the edges. Thus, each node is represented as a record of the form `[types layout count]`.

As in the case of the nodes, each edge contains a record that tracks additional information about the edge. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. The number of references that the edge may represent is tracked with the *maxCut* property. The *interfere* property tracks the possibility that the edge represents references that interfere. Finally, we have a field *connto* which is a list of all the other edges/variables that the edge may be connected to according to the *connR* relation (we add a (`!`) for the edges in the list that represent references which *may* alias and a ($\sim$) if the edges represent single references that *must* alias). To simplify the figures if the *connto* field is empty we omit it entirely from the record in the figure. Since the variable edges always represent single references and the offset label is implicitly the name of the variable the record simply contains the *connR* information or is omitted entirely if the *connR* relation is empty. To simplify the discussion of the examples each edge also has a unique label. The pointer edges in the figures are represented as records $\{$`label offset maxCut interfere connto`$\}$.

The abstract heap domain is restricted via a normal form [14, 15]. The normal form ensures that the heap graph remains finite, and that equality comparisons are efficient. The local data flow analysis is performed using a *Hoare (Partially Disjunctive) Power Domain* [13, 26] over these graphs. Interprocedural analysis is performed in a context-sensitive manner and the procedure analysis results are memoized. At each call/return site the portion of the heap graphs passed to the call are joined into a single graph. The design of the join operation is such that, in general, information lost in the join can be recovered when needed later in the program. The decision to perform joins at call sites (programs tend to have uniform expectations of the portion of the heap passed to and returned from calls) and to perform the join only on the portion of the heap passed to the called method results in very little loss of precision while ensuring the abstract model remains compact.

Abstract Call Stack. Our concrete model for the *call stack* is a function $S_m : (LV \times \mathbb{N}) \mapsto O$, where LV is the set of local variable names and $\mathbb{N}$ represents the depth in the call sequence (main is at depth 1) and O is the set of all live objects. Thus, the pair $(\mathtt{v}, 4)$ refers to the value of the variable `v` in the scope of the 4th call frame.

To represent the concrete call stack we introduce *stack variables* which represent the values of local variables on the stack (for a variation on this approach see [22]). In our extension each *stack variable* summarizes all the possible targets (in a given graph) for a given variable name on the stack. Given a variable name `v` and a heap graph G we define a variable name `v'` for use in the abstract domain (we will select a better naming scheme in Section 4) where: `v'` is the abstraction of all the variables in the call stack, $\exists i \in \mathbb{N},$ node $n \in G,$ object o_n s.t. $o_n \in \gamma(n) \wedge S_m(\mathtt{v}, i) = o_n$.

By associating the set of stack locations that are abstracted with the set of targets in a given abstract heap graph, we can naturally partition the *stack variables* along with the heap graphs. Since each *stack variable* is associated with only the values on the stack that point into a region of the heap represented by the given heap graph, it is straightforward to partition and join them when partitioning the heap graphs.

Thus, during the local analysis the heap graph represents the portion of the program heap that is visible from the local variables and is augmented with some number of *stack variables* and *cutpoint variables* which relate variable values and the heap in the caller scope to the portions of the heap reachable from callee scope local variables.

For efficiency and in order to ensure analysis termination the naming scheme we choose will result in situations where multiple cutpoint (or stack) edges are given the same name. This may result in some amount of information loss (particularly with respect to reachability and aliasing). To minimize the loss that occurs we introduce an instrumentation domain for the stack/cutpoint variable edges, *nameColl* = {*pdj*, *pua*, *pa*}. Where *pdj* indicates a cutpoint/stack name representing (a single edge) or edges where the edges do not represent any pairwise *connected* references, *pua* indicates a name representing multiple edges where there are no pairwise *aliases*, while *pa* is the indicates the name represents edges that they may have pairwise *aliasing*. Thus, the cutpoint variable edges are represented with records {`maxCut interfere connto nameColl`} (stack variables are not used in this example).

4 Stack Variables, Cutpoint Labels

When performing the project operation in heaps with cutpoints we need to name the *stack variables* as well as the *cutpoint* edges. We use a simple technique for the stack variables: given a variable name `v` defined in the caller function `fcaller` we use the name `$fcaller*v` to represent this variable in the callee scope. This naming scheme can create false dependencies on the local scope names unless the variable information is normalized during the comparisons of entries in the memo-table.

Naming edges that cross the cutpoints is more complex since we need to balance the accuracy of the analysis with the potential of introducing spurious differences resulting from isomorphic (or nearly so) cutpoint edges being given different names. For the renaming of the cutpoint edges we assume that special names for the arguments to the function have been introduced. The first pointer parameter is referred to by the special variable name `p1` and the i^{th} pointer argument is referred to by the variable `pi`.

Figure 2(c) shows a recursive call to `LInit` where the special argument name `p1` has been added to represent the value of the first argument to the function. In this figure the edge *e1* is a cutpoint edge since it starts in the portion of the heap that is unreachable from the argument variables and ends in a portion of the heap that is reachable from the argument variables (this differs slightly from the definition for cutpoints in [21] but allows us to handle edges uniformly).

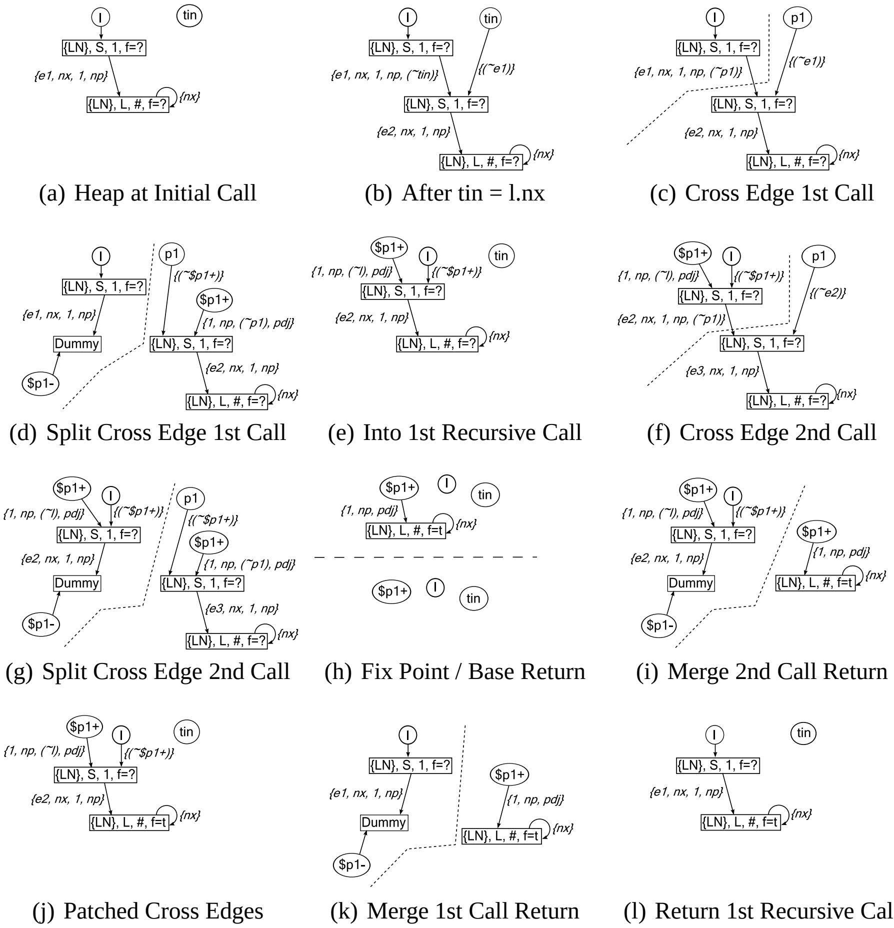
For each cutpoint edge we generate a pair of names: one is used in the unreachable section of the heap graph and one in the reachable section, which allows an abstract heap model to represent both incoming and outgoing cutpoint edges that are isomorphic and exist in the same abstract heap component without loss of precision.

If we are adding a cutpoint for the method call `fcaller` and the edge *e*, which is a cutpoint, starting at *n* and ending at n', and has edge label `fe`. We can find the shortest path (`f1` ... `fk`) from any of the `pi` variables to n' (using lexographic comparison on the path names to break ties). Using the `pi` argument variable and the path (`f1` ... `fk`) we derive the cutpoint `basename` = `fcaller*pi*f1*`...`*fk*fe` We compute a pair of static names (*unreachN*, *reachN*) where *unreachN* = `$basename-` and *reachN* = `$basename+`. In Figure 2(d) the cutpoint name `$p1+` (for brevity we simply label the cutpoint with the `pi` variable) is used to represent the endpoint of the cutpoint edge in the reachable component of the heap and `$p1-` to track a dummy node associated with the cutpoint edge in the unreachable component of the heap.

5 Example

The example program, Figure 1, recursively initializes the `f` fields in a linked list to the value `true`. Figure 2(a) shows the abstract heap model at the entry of the first call to the procedure (for simplicity we ignore any caller scope variables).

In Figure 2(a), variable `l` refers to a node that represents `LNode` objects (*types* = {`LNode`}, abbreviated to `LN`), that represents a region with no internal connections (*Layout* = *S*), which contains a single object (*count* = 1), and where all the incoming edges represent disjoint pointers (the `connto` lists on the edges are omitted). In this figure we also have that the elements in the list have unknown truth values in the `f` fields (`f=?`). There is a single edge out of this node representing pointers stored in the `nx` field of the object represented by the node. This edge represents a single pointer (*maxCut* = 1) and all the pointers are non-interfering (*interfere* = *np*). Finally, this edge refers to a node that also represents `LNode` objects but may represent many of these objects (*count* = #) and, since the *Layout* value is *List*, we know that the objects may be connected in a list-like shape. Since there is a single incoming edge and it represents a single pointer, we can safely assume that this edge refers to the head of the list structure.



(a) Heap at Initial Call

(b) After tin = l.nx

(c) Cross Edge 1st Call

(d) Split Cross Edge 1st Call

(e) Into 1st Recursive Call

(f) Cross Edge 2nd Call

(g) Split Cross Edge 2nd Call

(h) Fix Point / Base Return

(i) Merge 2nd Call Return

(j) Patched Cross Edges

(k) Merge 1st Call Return

(l) Return 1st Recursive Call

Fig. 2. Recursive Calls

Figure 2(b) shows the abstract heap model just after executing the statement `tin = l.nx`. Since we know that *e1* refers to the head element of the list from Figure 2(a) we replaced the single *List*-shaped node with a node representing the unique head element and a node representing the tail of the list. Since the head element is unique we set the *count* of this new node to 1. Additionally, the only possible layout for a node of *count* 1 is *Singleton*. Finally, if a node represents a single object then all the outgoing field edges can each represent a single pointer. Thus, we set the outgoing edge to have a *maxCut* =

1. Also note that after the load the analysis has determined that `tin` and `e1` must alias (indicated by the ∼`e1` and ∼`tin` entries in the connectivity lists).

Figure 2(c) shows the state of the abstract heap at the entry of the *project* procedure. The special name *p1* has been added to represent the value of the first pointer argument to the function and we have added a dotted line to indicate the reachable and unreachable portions of the heap. Note that the edge *e1* is a cutpoint edge according to our definition.

The result of the project operation is shown in Figure 2(d). The *e1* edge, which was a cutpoint edge for the call, has been remapped to a dummy node and the static cutpoint names `$p1-` and `$p1+` (for brevity we omit the procedure name and edge labels from the static names) have been introduced at the dummy node and at the target of this edge in the reachable section. Since this cutpoint edge only represents the single cutpoint edge generated in this call frame `nameColl = pdj`. Also note that the analysis has determined that the formal parameter `p1` must alias the cutpoint edge `$p1+`.

Figure 2(e) shows the resulting abstract heap that is passed into the callee scope for analysis. Since all the local variables in the caller scope either did not refer to nodes in the callee reachable section or are dead after the call return we do not have to give them stack names and can remove them entirely from the heap model. Figure 2(f) shows the abstract heap at the entry to the project function for the second recursive call. Again we have a cutpoint edge *e2*. Note that the reachable cutpoint label, `$p1+` introduced in the previous call is now in the unreachable portion of the heap, thus (`$p1+`) does not conflict with the unreachable name added in this call (`$p1-`). The result of the project operation is shown in Figure 2(g).

Figure 2(h) shows the eventual fixpoint approximation (above the dotted line) of the analysis of this function and also the base case return value (below the dotted line). Notice in the base case return value we were able to determine that the test `l == null` implies that `l` must be null and since we preserved must alias information through the cutpoint introduction we can infer that `l` must alias `$p1+`, which implies the cutpoint edge (`$p1+`) must also be null. Thus, the analysis can infer that on return the cutpoint edge is either `null` or is non-null and refers to some list in which all the `f` fields have been set to `true` (`f=t` in the figure).

In Figure 2(i) we show how the fixpoint approximation for the reachable section of the heap is recombined with the unreachable section of the heap using the *extend* operation. After the recombination we get the abstract heap model shown in Figure 2(j). In Figure 2(i) we have unioned the graphs and are ready to patch up the cutpoint cross edge information. The static name `$p1+` in the reachable portion of the heap has been used to compute the associated unreachable name (`$p1-`). Then the algorithm identifies the edge associated with the dummy node referred to by `$p1-` (*e2*) and remapped this edge to end at the target of `$p1+` (`tin` has been nullified since it is dead).

Figure 2(k) shows the *extend* operation at the return from the first recursive call which is similar to the situation in the second recursive call. The resulting abstract heap is shown in Figure 2(l) which can be joined with the result of the base case test and then completes the analysis of the method. As desired, the analysis has determined that the recursive list initialize procedure preserves the list shape of the argument list and that all of the `f` fields in the list have been set to `true` (`f=t` in the figures).

6 Project and Extend Algorithms

Project. We assume that before the *projectHeap* function is invoked all of the special argument variable names have been added to the heap model. This allows *projectHeap* (Algorithm 1 below) to easily compute the section of the heap model that is reachable in the callee procedure and then compute the set of nodes that comprise the unreachable portion of the heap model.

Algorithm 1: projectHeap

```
input  : $h$: the heap model to be partitioned
output: $h_r$, $h_u$: the reachable and unreachable partitions, $snu$, $ncs$: the static names used
        and newly created
reachNodes ← set of nodes reachable from args;
unreachNodes ← set of nodes unreachable from args;
crossEdges ← set of edges that start in unreachNodes and end in reachNodes;
snu ← ∅;
ncs ← ∅;
foreach edge $e$ in crossEdges do
    (sn, isnew) ← procCrossEdge($h$, $e$, reachNodes);
    snu.add(sn);
    if isnew then ncs.add(sn);

$h_u$ ← subgraph of $h$ on the nodes unreachNodes ∪ {dummy nodes from procCrossEdge};
$h_r$ ← subgraph of $h$ on the nodes reachNodes;
return ($h_r$, $h_u$, snu, ncs);
```

For each edge that crosses from the unreachable section into the reachable section we add a pair of static names to represent the edge (Algorithm 2). Since the heap model stores a number of domain properties in each edge, we create a dummy node and remap the edge to end at this node. Then, the *unreachN* static name is set to refer to this dummy node. In the reachable portion of the heap graph we simply set the *reachN* static name to refer to the target of the cross edge.

When adding the *reachN* static name to the reachable section of the heap graph the name may or may not already be present in the heap graph. If the name is not present then we add it to the static name map and for later use we note that this is the call where the name is introduced. Otherwise a name collision has occurred and we must mark the edges representing the possible cutpoints appropriately (for simplicity we mark all the edges). If there may be aliasing we note that the cutpoints from different frames may have aliasing targets (*pa*) and similarly if the new cutpoint edge may be connected with an existing cutpoint edge we mark them as being pairwise connected (*pua*). The functions *makeEdgeForUnreachCutpoint* and *makeEdgeForReachCutpoint* are used to produce edges to represent the cutpoint (based on the static name and the cutpoint edge properties) in the unreachable and reachable portions of the heap.

Once all of the cutpoint edges have been replaced by the required static names, the heap can be transformed into the unreachable version (where all the nodes in the reachable section and all the variables/static names that only refer to reachable nodes have been removed) and the reachable version (where the nodes in the unreachable section and the associated names have been removed).

Algorithm 2: procCrossEdge

```
input  : $h$: the heap, $e$: the cross edge, $reachNodes$: set of reachable nodes
output: $rsn$: the name used, $isnew$: true if $rsn$ a new name
$n_e \leftarrow$ the node $e$ ends at;
$n_i \leftarrow$ new dummy node;
$(ursn, rsn) \leftarrow$ genStaticNamePairForEdge($h$, $e$);
$e_u \leftarrow$ makeEdgeForUnreachCutpoint(e, $ursn$);
set endpoint of $e_u$ to $n_i$;
add $e_u$ as an edge for $ursn$;
$e_r \leftarrow$ makeEdgeForReachCutpoint(e, $rsn$);
set endpoint of $e_r$ to $n_e$;
remap the endpoint of $e$ to $n_i$;
if the name $rsn$ exists and has edges pointing to a node in $reachNodes$ then
    $rsnes \leftarrow \{e' | e'$ is an edge for the cutpoint var $rsn\}$;
    add $e_r$ as an edge for $rsn$;
    if $e_r$ is inConnected with an edge in rsnes then set edges in $rsnes$ and $e_r$ to $pua$;
    if $e_r$ may alias with an edge in rsnes then set edges in $rsnes$ and $e_r$ to $pa$;
    return (rsn, false);
else
    add the name $rsn$ to $h$;
    add $e_r$ as an edge for $rsn$;
    return (rsn, true);
```

Extend. After the call return we need to rejoin the unreachable portion of the heap that we extracted before the procedure call entry with the result we obtained from analyzing the callee procedure. This is done by looking at each of the static names that was used to represent a cutpoint edge and reconnecting as required. Then, each of the newly introduced cutpoint names can be removed from the heap model. The pseudo-code to do this is shown in Algorithm 3.

This algorithm merges all edges with the same reachable cutpoint name so that there is at most one target edge for a given cutpoint name in the reachable heap h_r (this simplifies the algorithm and is in our experience is quite accurate). The algorithm then pairs up the two cutpoint names and remaps the edge we saved in the unreachable section to the target node in the reachable section subject to a number of tests to propagate sharing information (the nullity information is propagated due to the fact that the dummy node and all incoming edges are always removed but the foreach loop on the targets of *ursn* does not execute since the target set is empty). The e_r.*nameColl* = *pua* test is true if this edge represents sets of pointers that do not have pairwise aliases. Thus, we mark the newly remapped edge and e_r as pairwise unaliased. Similarly, the e_r.*nameColl* = *pdj* test is true if this edge represents cutpoint/stack edges that are pairwise disjoint. Thus, we mark the newly remapped edge and e_r as pairwise disjoint.

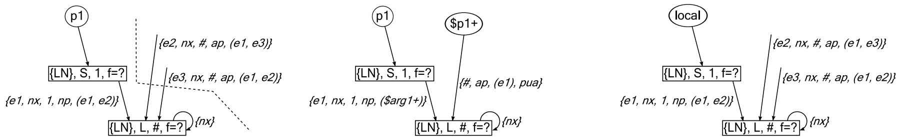
```
Algorithm 3: extendHeap
input  : $h_r$, $h_u$: the reachable and unreachable partitions, snu, ncs: the static names used
         and newly created
output: $h$: the joined heap model
$h \leftarrow$ new heap();
$h$.heapGraph $\leftarrow$ mergeGraphs($h_r$.heapGraph, $h_u$.heapGraph);
foreach static name sn in snu do
    ursn $\leftarrow$ reachNameToUnreachName(sn);
    $n_r \leftarrow$ the target of sn in $h_r$.nameMap;
    foreach node $n_u$ that is a target of ursn in $h_u$.nameMap do
        $e_r \leftarrow$ the single incoming edge to $n_u$;
        remap $e_r$ to end at the target of $n_r$;
        $e_r$.interfere = $e_r$.interfere $\sqcup$ $n_r$.interfere;
        if $e_r$.nameColl = pua then set $e_r$ and $n_r$ as unaliased;
        if $e_r$.nameColl = pdj then set $e_r$ and $n_r$ as disjoint;

    $h_u$.removeNodeAllEdges(target of ursn);
    $h_u$.unmapStaticName(ursn);
    if sn in ncs then $h_r$.unmapStaticName(sn);

$h$.nameMap $\leftarrow$ mergeNameMaps($h_r$.nameMap, $h_u$.nameMap);
return $h$
```

The major components of this algorithm are the separation of the *mergeGraphs* action from the *mergeNameMaps* action and the elimination of the static cutpoint edge names that were introduced for this call.

The *mergeGraphs* function computes the union of the graph structures that represent the abstract heap objects, while the *mergeNameMaps* function computes the union of the name maps (which are maps from the stack/variable/cutpoint names to the nodes in the graph structure that represent them). This separation allows the algorithm to nullify the names created for this call which prevents the propagation of unneeded cutpoint edge targets to the caller scope. The function *unmapStaticName* is used to eliminate a given static name from the abstract heap model name map.

Example Name Collision. The introduction of the *nameColl* domain minimizes the precision loss that occurs when a cutpoint or stack variable name collision occurs. Figure 3 shows an example of such a situation. In this figure we show part of a heap where the edges *e2* and *e3* are both cutpoint edges and they do not represent any pairwise aliasing pointers (no `!` in the *connTo* lists) although they each represent sets of pointers that may alias, *interfere* = *ap*.

In this example our naming scheme will result in *e2* and *e3* being represented with the same cutpoint name. However, our method will mark this cutpoint edge as *nameColl* = *pua* (Figure 3(b)). This means that on return the *extend* algorithm will set the edges that are mapped to this cutpoint as being pairwise unaliased (Figure 3(c)) as desired. Thus, even though there was a name collision for the cutpoints we avoided (in this case completely) the loss of sharing information about the heap.


p1
{e2, nx, #, ap, (e1, e3)}
{LN}, S, 1, f=?
{e3, nx, #, ap, (e1, e2)}
{e1, nx, 1, np, (e1, e2)}
{LN}, L, #, f=?
{nx}
p1
$p1+
{LN}, S, 1, f=?
{#, ap, (e1), pua}
{e1, nx, 1, np, ($arg1+)}
{LN}, L, #, f=?
{nx}
local
{e2, nx, #, ap, (e1, e3)}
{LN}, S, 1, f=?
{e3, nx, #, ap, (e1, e2)}
{e1, nx, 1, np, (e1, e2)}
{LN}, L, #, f=?
{nx}

(a) Colliding Names (b) To Same Cutpoint (c) PUA on Return

Fig. 3. Name Collision

7 Experimental Results

The proposed approach has been implemented and the effectiveness and efficiency of the analysis have been evaluated on the source code for programs from a variation of the Jolden [3, 18] suite and several programs from SPEC JVM98 [27] (raytrace, modified to be single threaded, db and compress). The analysis algorithm is written in C++ and was compiled using MSVC 8.0. The parallelization benchmarks were run using the Sun 1.6 JVM. All runs are from our 2.8 GHz PentiumD machine with 1 GB of RAM.

We ran the analysis with the project/extend operations enabled (the *Project* column) and disabled (the *No-Project* column) and recorded the analysis time, the average number of times a method needed to be analyzed, and used the resulting shape information to parallelize the programs, shown in Figure 4. The results indicate that the project/extend operations have a significant impact on the performance of the analysis, reducing the number of contexts that each function needs to be analyzed in (on average reducing the number of contexts by a factor of 4.3) which results in a substantial decrease in analysis times (by a factor of 18.4). As expected this reduction becomes more pronounced as the size and complexity of the benchmarks increases, in the case of raytrace the analysis time without the project/extend operation is impractically large (772.6 seconds) but when we use the project/extend operations the analysis time is reduced to 35.11 seconds.

We used the shape information from the analysis to drive the parallelization of the benchmarks by using multiple threads in loops and calls, resulting in the speedup columns in Figure 4. Given the shape information produced by the analysis it is straight forward to compute what parts of the heap are read and written by a loop body or method call and thus which loops and calls can be executed in parallel (in raytrace we treated the memoization of intersect computations as spurious dependencies). Once the analysis identified locations that could be parallelized we inserted calls to a simple thread pool (since our current work is focused on the analysis this is done by hand but can be fully automated [6, 23, 10]). In 8 of 9 benchmarks that are suitable for shape driven parallelization (compress, db and mst do not have any data structure operations that are amenable to shape driven parallelization) we achieve a promising speedup, averaging a factor of 1.69 over the benchmarks.

Benchmark Info			No-Project			Project		
Benchmark	Stmt	Method	Time	Avg Cont.	Speedup	Time	Avg Cont.	Speedup
bisort	260	13	0.86s	10.6	1.00	0.28s	1.9	1.72
em3d	333	13	0.12s	2.5	1.75	0.08s	1.8	1.75
mst	457	22	0.06s	3.2	NA	0.04s	3.0	NA
tsp	510	13	1.51s	22.4	1.84	0.17s	7.0	1.84
perimeter	621	36	54.57s	105.9	1.00	2.97s	50.2	1.00
health	643	16	3.24s	12.9	1.00	2.26s	4.2	1.76
voronoi	981	63	20.89s	61.4	1.00	2.67s	37.2	1.68
power	1352	29	5.71s	26.8	1.93	0.17s	1.3	1.93
bh	1616	51	8.64s	32.8	1.75	2.68s	7.3	1.75
compress	1102	41	0.29s	2.9	NA	0.18s	2.2	NA
db	1214	30	0.94s	3.7	NA	0.68s	2.8	NA
raytrace	3705	173	772.60s	293.1	1.00	35.11s	15.6	1.76
Overall	12794	523	869.43s	48.2	1.36	47.29s	11.2	1.69

Fig. 4. The Stmt and Method columns list the number of statements and methods for each benchmark. The columns for the No-Project and Project variations of the analysis list: the analysis time in seconds, the average number of times each method was analyzed and parallel speedup achieved on a 2 core 2.8 GHz PentiumD processor.

Our experimental results show that the information provided by the analysis can be effectively used (in conjunction with existing techniques) to drive the parallelization of programs. To the best of our knowledge this analysis is the only shape analysis that is able to provide the information required to perform shape driven parallelization for five of these benchmarks (em3d, health, voronoi, bh and raytrace). Given the speed with which the analysis is able to produce the information needed for the parallelization and the consistent parallel speedup that is obtained in the benchmarks (1.69 over all of the benchmarks and 1.77 if we exclude the benchmark mst), we find the results encouraging.

Of particular interest is the raytrace benchmark. This program is 2-4 times larger than any benchmarks used in the related work, builds and traverses several heap structures that have significant sharing between components. It also makes heavy use of virtual methods and recursion. This benchmark presents significant challenges in terms of the complexity and size of the program as well as in terms of the range of heap structures that need to be represented in order to accurately and efficiently analyze the program. Our analysis is able to manage all of these aspects and is able to produce a precise model of the heap (allowing us to obtain a speedup of 1.76 using heap based parallelization techniques). Further, the analysis is able to produce this result while maintaining a tractable analysis runtime.

8 Conclusion

We presented and benchmarked project/extend operations for a store-based heap model that is capable of precisely representing a range of shape, connectivity and sharing properties. The project and extend operations we introduced are designed to minimize the analysis time by reducing the number of unique calling contexts for each function and to minimize the imprecision introduced by the collisions that occur between stack/cutpoint names.

Our experimental results using the project/extend operations are very positive. The analysis was able to efficiently analyze benchmarks that build and manipulate a variety of data structures. Our benchmark set includes a number of kernels that were originally designed as challenge problems for automatic parallelization (the Jolden suite) and several benchmarks from the SPEC JVM98 suite (including a single threaded version of raytrace). Our experimental results demonstrate that the project/extend operations are effective in minimizing the number of contexts that need to be analyzed (on average a factor of 4.3 reduction), improving analysis accuracy (seen as improved parallelization results, in 4 out of 12 benchmarks) and substantially reducing the analysis runtime (by a factor of nearly 20). Our heap analysis was also able to provide sufficient information to successfully parallelize the majority of benchmarks we examined, including several that cannot be successfully analyzed/parallelized using other proposed shape analysis methods.

Acknowledgments

This work is supported under subcontract R7A824-79200004 from the Los Alamos Computer Science Institute and Rice University and by the National Science Foundation (grant 0540600). Manuel Hermenegildo is also supported by the Prince of Asturias Chair at UNM, and projects MEC-MERIT, CAM-PROMESAS, and EU-MOBIUS.

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. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *J. Log. Program.*, 10:91–124, 1991.
3. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.
4. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.
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. R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *CC*, 1998.
7. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

8. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
9. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
10. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1), 1990.
11. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
12. B. Jeannet, A. Loginov, T. W. Reps, and S. Sagiv. A relational approach to interprocedural shape analysis. In *SAS*, 2004.
13. R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.
14. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
15. M. Marron, R. Majumdar, D. Stefanovic, and D. Kapur. Dominance: Modeling heap structures with sharing. Tech. report, CS Dept., Univ. of New Mexico, Aug 2007.
16. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
17. K. Muthukumar and M. V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *J. Log. Program.*, 1992.
18. Modified Jolden Benchmarks, August 2007. `http://www.cs.unm.edu/~marron`.
19. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
20. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
21. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.
22. N. Rinetzky and S. Sagiv. Interprocedural shape analysis for recursive programs. In *CC*, 2001.
23. R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, 1999.
24. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
25. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
26. M. B. Smyth. Power domains and predicate transformers: A topological view. In *ICALP*, 1983.
27. Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`.
28. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.