Sharing Analysis of Arrays, Collections, and Recursive Structures

Mark Marron[1] Mario Méndez-Lojo[1]
Manuel Hermenegildo[1,2] Darko Stefanovic[1] Deepak Kapur[1]

[1] University of New Mexico (USA) [2] IMDEA-Software and Technical University of Madrid (Spain)

{marron,mario,herme,darko,kapur}@cs.unm.edu

Abstract

Precise modeling of the program heap is fundamental for understanding the behavior of a program, and is thus of significant interest for many optimization applications. One of the fundamental properties of the heap that can be used in a range of optimization techniques is the sharing relationships between the elements in an array or collection. If an analysis can determine that the memory locations pointed to by different entries of an array (or collection) are disjoint, then in many cases loops that traverse the array can be vectorized or transformed into a thread-parallel version. This paper introduces several novel sharing properties over the concrete heap and corresponding abstractions to represent them. In conjunction with an existing shape analysis technique, these abstractions allow us to precisely resolve the sharing relations in a wide range of heap structures (arrays, collections, recursive data structures, composite heap structures) in a computationally efficient manner. The effectiveness of the approach is evaluated on a set of challenge problems from the JOlden and SPECjvm98 suites. Sharing information obtained from the analysis is used to achieve substantial thread-level parallel speedups.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages (program analysis).

General Terms Languages, Performance.

Keywords Shape analysis, Shared structures, Parallelism.

1. Introduction

The transformation of *foreach* style traversals on arrays/collections of scalars via vectorization and scheduling optimizations has had a major impact in improving the performance of optimized programs, and thread–level parallelization of these loops is becoming increasingly important with the proliferation of multi–core processors. The ability to apply these transformations in modern object-oriented programs that make heavy use of pointer structures and where arrays/collections often contain references to other heap structures (instead of scalar values) is severely limited by the difficulty of reasoning precisely about the structure of the program heap and the sharing relations between the pointers stored in these collections.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'08, Atlanta, Georgia, USA
Copyright © 2008 ACM 978-1-60558-382-2/08/11…$5.00

```
1 Data[] AU = new Data[N];
2 for(int i = 0; i < N; ++i)
3  AU[i] = new Data(i);

4 Data[] AS = new Data[N];
5 for(int i = 0; i < N; ++i)
6  AS[i] = AU[f(i)];
```

```
7 for(int i = 0; i < N; ++i)
8   AU[i].val++;

9 for(int i = 0; i < N; ++i)
10  AS[i].val *= 2;
```

Figure 1. Array initialization (left) and *foreach* processing (right)

A significant amount of work has been devoted to the problem of *shape analysis*, which strives to provide precise information about the connectivity/reachability properties of the heap. A major focus of this work has been on accurately modeling the construction and update of recursive data structures such as lists or trees [2, 6, 11, 15, 16, 18, 19], while recent work explored how these recursive structures are connected to form *composite* heap structures [2, 15]. These results are important steps in the development of a general-purpose heap analysis technique and allow for the parallelization of recursive data structure traversals, but they do not adequately capture the sharing (or lack thereof) between the entries in a given array or collection.

Take the simple program segment shown in Figure 1, which manipulates arrays of `Data` objects, each containing a single integer field `val`. The first loop fills the array `AU` with a number of fresh `Data` objects, and thus there is no aliasing between the entries in the array. The second loop fills the array `AS` with objects selected from the first array via some indexing function `f(i)`, which we assume cannot be understood by the analysis (e.g., `f` is a complex non–linear transform or uses some form of randomization). Therefore, each entry in `AS` may potentially alias with another entry in the array.

Figure 2(a) shows one possible concrete heap after initialization of `AS` (depending on the exact behavior of `f`). We see that there is sharing between elements of the array `AS`, since `AU[0]` is referenced by both `AS[0]` and `AS[2]`, but there is no sharing between elements of the original array `AU`. If we look at the resulting heap abstraction using the classic *storage shape graph* [5] approach we get the model shown in Figure 2(b). In this figure we have associated a unique integer identifier with each node/edge in the model and we also show the type of objects abstracted by each node and the storage specifier for each edge location. We use the special storage specifier `?` to represent all the storage locations in an array. Notice that in Figure 2(b) there is no information to distinguish between edge 2, which abstracts the pointers stored in array `AU` which we know do not alias, and edge 4, which abstracts the pointers stored in array `AS` which may alias with each other. Thus the optimizer must always conservatively assume that there may be sharing between the references stored in an array and cannot parallelize/vectorize the update loops on lines 7-8 and 9-10 (even though in practice it is safe to transform the first loop).

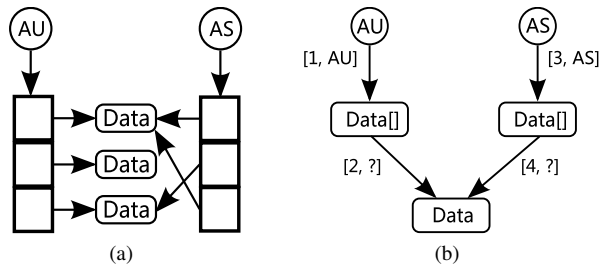


Figure 2. A possible concrete state (a) and its abstraction (b).

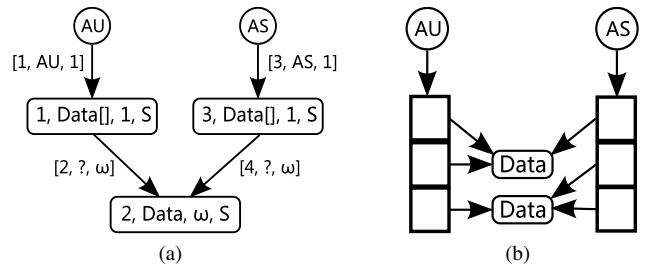


Figure 3. Basic abstraction (a) and a possible concretization (b).

The major contributions in this paper appear in Section 3, where we define a set of useful sharing properties in the concrete heap and where we demonstrate how these properties can be efficiently encoded in an abstract domain. In Section 4 we evaluate the extended analysis and the effectiveness of the sharing information by looking at a detailed case study from the JOlden [14] suite, the well known **Barnes-Hutt** benchmark (**bh**), which has not been successfully analyzed by any other shape analysis technique. We show how the analysis results obtained for this benchmark can be used to achieve a 23% performance improvement in single-threaded execution, a 37% reduction in heap use, and a factor of 3 parallel speedup on a quad–core machine. In Section 5 we evaluate the computational cost of the analysis and the impact of the sharing information in performing thread–level parallelization on our set of benchmarks.

2. Base Heap Model

To analyze a program we first transform the Java 1.4 source into a core sequential imperative language called MIL (Mid-level Intermediate Language). The MIL language is statically typed, has method invocations, conditional constructs (`if`, `switch`), exception handling (`try-throw-catch`) and the standard looping statements (`for`, `do`, `while`). The state modification and expressions cover the standard range of program operations: load, store, and assignment along with logical, arithmetic, and comparison operators. This mid-level language allows us to support the majority of the Java 1.4 language while substantially simplifying the analysis. During this transformation step we also load in our specialized standard library implementations, so we can analyze programs that use classes from `java.util`, `java.lang` or `java.io`.

The semantics of the language is defined in the usual way, using 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. We model the concrete heap as a labeled, directed multi-graph (N,E) where each node $n \in N$ is an object in the store or a variable in the environment, and each labeled directed edge $e \in E$ represents a reference (a pointer between objects or a variable reference). Each edge is given a label that is either an identifier from the program or an integer $i \in \mathbb{N}$ (the integers label the pointers stored in the arrays/collections). For an edge $(a,b) \in E$ labeled with p, the notation $a \xrightarrow{p} b$ indicates that the object/variable a points to b via the field name or identifier p.

A *region* of the heap $\mathfrak{R}$ is a subset of the objects, 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 $O \subseteq N$ be a subset of objects, and let $P_i = \{p \mid \exists o_1, o_2 \in O, o_1 \xrightarrow{p} o_2\} \wedge$ and $P_c = \{p \mid \exists o \in O, x \notin O, x \xrightarrow{p} o \vee o \xrightarrow{p} x\}$ be respectively the set of internal and cross-region references for O. Then a region is the tuple (O, P_i, P_c).

2.1 Basic Properties

Our base abstract heap domain [15] is a directed graph in which 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 properties:

Types. Each non–variable node in the graph represents a region of the heap (which may contain objects of many types). To track the types of these objects we use a set of type names as part of the label of each node. This set contains the type of any object that may be in the region of the heap that is abstracted by the given node.

Linearity. To model the number of objects abstracted by a given node (or pointers by an edge) we use a *linearity* property which has two possible values: 1, which indicates that the node (edge) concretizes to either 0 or 1 objects (pointers), and the value ω, which indicates that the node (edge) concretizes to any number of objects (pointers) in the range $[0,\infty)$.

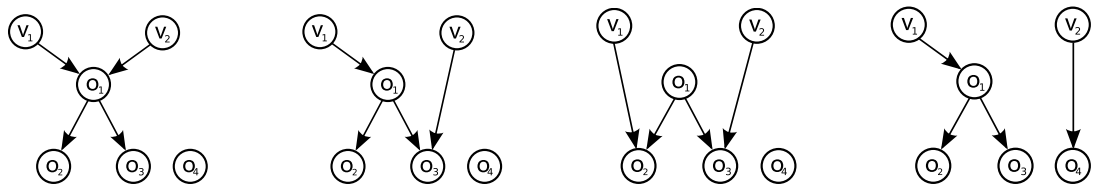
Abstract Layout. To approximate the shape of the region a node abstracts, the analysis uses the *abstract layout* properties $\{$*(S)ingleton*, *(L)ist*, *(T)ree*, *(M)ultiPath*, *(C)ycle*$\}$. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node. The *(L)ist* property states each object has at most one pointer to another object in the region. The other properties correspond to the standard definitions for trees, DAGs, and cycles.

Pictorially, we represent abstract heaps as labeled, directed multi-graphs. The variable nodes are labeled with the name of the variable they represent. Nodes abstracting concrete regions are denoted as `[id, type, linearity, layout]`; the first field (`id`) contains a unique identifier, while the rest correspond to the predicates described above.

The abstract edges, which approximate sets of pointers, are represented in the figures as records `[id, offset, linearity]`. The *offset* component indicates the label of the references that are abstracted by the edge: a field identifier, a variable name when the edge connects a variable and a node, or the special label *?* denoting the summary field for all the elements in an array or a collection object such as `List` or `Set`.

EXAMPLE 1: Figure 3(a) shows how our basic abstract domain represents the heap state after initializing the two arrays of Figure 1. We can see that the analysis is able to represent a number of properties that are not present in the classic storage shape graph in Figure 2(b). The figure represents two variables (`AS` and `AU`) pointing to two different arrays (nodes 1 and 3) through edges 1 and 3, each with linearity 1 (since a variable can only refer to a single location). Each of the abstract arrays represents objects of type `Data[]` and the *linearity* of value 1 indicates that each node abstracts at most one array. The *(S)ingleton layout* means there are no pointers between the objects in the region abstracted by the node (which in this case we already knew based on the type information).

Each array may store multiple pointers, which are abstracted by edges 2 and 4. The edge offset *?* indicates that the pointers may be



(a) Aliasing (b) Related (c) Related (d) Unrelated

Figure 4. Concrete Reference Relations

stored in any array index, while the *linearity* value of ω indicates that there may be many pointers stored in the arrays. Since there are multiple pointers in the arrays pointing into the same region of `Data` objects (node 2), sharing may occur between the two arrays. In particular, note that edges 2 and 4 have no information to distinguish the different sharing properties between the edge which abstracts the pointers stored in array `AU` (which do not alias) and the edge which abstracts the pointers stored in array `AS` (which may alias with each other). Based on this limited information, a possible concrete heap layout after the initialization of `AU` and `AS` is the one shown in Figure 3(b), even though this particular configuration can never occur in practice. A consequence of this imprecision is that elements of `AU` are conservatively assumed to potentially alias with each other and thus we cannot determine from the information in the model that the loop on lines 7-8 can be safely parallelized. □

3. Extensions for Sharing

In the base heap domain, each edge represents a set of pointers or a variable reference. These concrete references may point to several objects in the region of the heap abstracted by the node that an abstract edge (or edges) refer to. Since each region may contain many objects which may or may not be in the same data structure, a possible question we might ask is: do any of those references (pointers/variables) point to the same object or into the same data structure? The query can refer to references that are abstracted via different edges (e.g., *may* one of the pointers stored in array `AU` alias with one of the pointers stored in array `AS`), or about references that are abstracted by the same edge (e.g., *may* any of the pointers stored in array `AU` alias). To answer questions of this type, we first define three predicates that describe how concrete references are connected/share in the heap. Then we introduce the abstract predicates connectivity and interference, which model the concrete connected/share information.

Given a concrete region of the heap $\mathfrak{R} = (O, P_i, P_c)$, and incoming references $r, r' \in P_c$ pointing to objects $o, o' \in O$ respectively, we define the following *relation* predicates on the references:

- $\mathsf{alias}(r, r', \mathfrak{R})$ is true iff $o = o'$ in $\mathfrak{R}$.
- $\mathsf{related}(r, r', \mathfrak{R})$ is true iff $o \neq o'$ and o, o' are in the same *weakly–connected* component of the subgraph (O, P_i).
- $\mathsf{unrelated}(r, r', \mathfrak{R})$ is true iff o, o' are in different *weakly–connected* components of (O, P_i). Stated in a different way, $\mathsf{unrelated}(r, r', \mathfrak{R}) \Leftrightarrow (\neg\mathsf{alias}(r, r', \mathfrak{R})) \wedge (\neg\mathsf{related}(r, r', \mathfrak{R})))$.

EXAMPLE 2: Figure 4 contains several examples of the *relation* predicates. Each figure shows a region of the concrete heap, were $O = \{o_1, o_2, o_3, o_4\}$ and $P_i = \{o_1 \to o_2, o_1 \to o_3\}$. In Figure 4(a), variables v_1 and v_2 point to the same object ($P_c = \{v_1 \to o_1, v_2 \to o_1\}$), so they alias. In Figure 4(b) v_1 and v_2 point to different objects ($P_c = \{v_1 \to o_1, v_2 \to o_3\}$), but there is a path from o_1 to o_3, thus they are in the same weakly–connected component of the region and are related. Figure 4(c) shows a sample concrete heap where the two variables refer to the same structure ($P_c = \{v_1 \to o_2, v_2 \to o_3\}$). Since they belong to the same weakly–connected component they are related according to the above definition, even though there is no path between them. Finally, Figure 4(d) shows an example of a region where v_1 and v_2 are unrelated because they refer to disjoint data structures ($P_c = \{v_1 \to o_1, v_2 \to o_4\}$). □

3.1 Abstract Sharing

To model the concrete properties just defined, we introduce two instrumentation predicates, one to track relations between references which are abstracted by different edges in the model (connectivity), and one to track the relations between references which are abstracted by the same edge in the model (interference).

Connectivity. Given the relation predicates in the concrete regions we can define a series of connectivity properties, $\mathsf{connectivity} = \{\mathsf{share}, \mathsf{connected}, \mathsf{disjoint}\}$, on the edges in the abstract domain. The concrete region $\mathfrak{R}$ and the sets of references R, R' are a valid concretization of the edges e and e' if:

- if $\mathsf{disjoint}(e, e')$ is true, then $\nexists r \in R, r' \in R'$ s.t. $\mathsf{alias}(r, r', \mathfrak{R})$ or $\mathsf{related}(r, r', \mathfrak{R})$.
- if $\mathsf{connected}(e, e')$ is true, then $\nexists r \in R, r' \in R'$ s.t. $\mathsf{alias}(r, r', \mathfrak{R})$.
- if $\mathsf{share}(e, e')$ is true, then any of the *relation* predicates holds for the references $r \in R, r' \in R'$.

To represent the connectivity property in the figures we extend the label of every edge e with a list of the identifiers of the other edges in the graph that it has a share or connected relation with. If an edge identifier e' in this list is prefixed with a "!" then $\mathsf{share}(e, e')$ holds, while if there is no prefix then they are related by the connected predicate; if an edge identifier e' does not appear in the list, we will assume that $\mathsf{disjoint}(e, e')$ is true.

Interference. The interference property is closely related to the concept of connectivity. While the latter tracks *relation* predicates between references that are abstracted by different graph edges, the interference property tracks *relation* predicates between references that are abstracted by the same graph edge. Given the definitions for the *relation* predicates in the concrete regions we can define a series of interference predicates, $\mathsf{interference} = \{\mathsf{aliasing}(\mathsf{ap}), \mathsf{interfering}(\mathsf{ip}), \mathsf{non\text{–}interfering}(\mathsf{np})\}$, on the edges in the abstract domain. The concrete region $\mathfrak{R}$ and the set of references R are a valid concretization of edge e if:

- if $\mathsf{non\text{–}interfering}(e)$ is true, then $\nexists r, r' \in R, r \neq r'$ s.t. $\mathsf{alias}(r, r', \mathfrak{R})$ or $\mathsf{related}(r, r', \mathfrak{R})$.
- if $\mathsf{interfering}(e)$ is true, then $\nexists r, r' \in R, r \neq r'$ s.t. $\mathsf{alias}(r, r', \mathfrak{R})$.
- if $\mathsf{share}(e)$ is true, then any of the *related* predicates holds for the references $r, r' \in R$.

To represent the interference property in the figures each edge label is extended with one of the predicates $\{\mathsf{ap}, \mathsf{ip}, \mathsf{np}\}$.

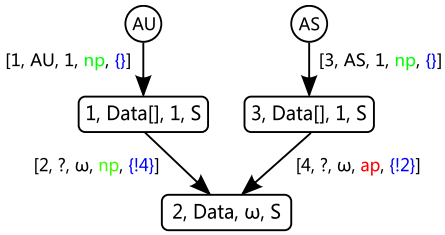


Figure 5. Graph Model With Sharing

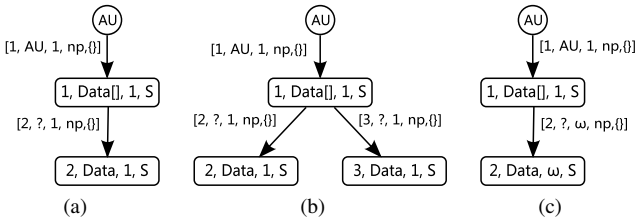


Figure 6. Abstract initialization of AU.

EXAMPLE 3: Figure 5 shows the abstract heap of Figure 3(a) extended with the interference and connectivity information. Every edge is now of the form [id, offset, linearity, interference, connectivity]. The abstract references 1 and 3 represent at most one concrete pointer -either from variable AU to node 1, or from variable AS to node 3-, and by definition cannot interfere (thus edges 1 and 3 both have the np interference property), and are disjoint from any other edge in the model, as indicated by their empty connectivity lists. More interesting is the sharing information that the extended abstract domain captures about the relation between the arrays and the elements they contain. The node representing array AU (node 1) may contain many pointers to Data objects (abstracted by node 2), which are known not to interfere, and thus edge 2 is tagged as np. However, pointers abstracted by edge 4 might refer to the same object, since the edge is labeled ap. This information represents a fundamental addition to what is inferred by the base domain, since we now know that the pointers stored in AU do not alias and therefore we can safely parallelize the loop in lines 7 and 8 of Figure 1. Finally, the connectivity information indicates that the set of pointers abstracted by edge 2 might alias objects also referred to by pointers abstracted by edge 4. □

3.2 Sharing and Abstract Semantics

For brevity, we illustrate how the analysis simulates the effects of the program statements and control flow structures used in the motivating example and refer the reader to [1, 15] for detailed descriptions of the operations. The running example covers a range of basic operations including object allocation, storing a reference into memory (the version without strong updating), memory loads and -since we are analyzing loops- the use of the normalization (or widening) operation.

Figure 6 shows the state of the abstract heap at several key points during the analysis of the first loop (lines 2-3). During the first analysis pass through the loop body we need to model the allocation of the new Data object, which will be stored in the array AU. To accomplish this we create a new node to represent this object, node 2 in Figure 7(a). Since this node abstracts a single object it has *linearity* 1, and a *(S)ingleton layout*. To simulate the effects of the store of a reference to this object into array AU we create a new edge (edge 2), which connects nodes 1 and 2. Since this edge represents a single pointer it is given *linearity* 1. Because by definition a single reference cannot interfere with itself the edge is labeled *non–interfering* (np), and since there are no other edges incident to node 2 we know that edge 2 is disjoint from all other edges. These operations result in the model shown in Figure 7(a), which is the approximation returned by the analysis after a first pass over the loop body.

A second pass through the loop body results in an array of two distinct elements. The abstract state representing this is shown in Figure 6(b). The model explicitly represents the two pointers stored in the array with edges 2 and 3 respectively. If we continue adding edges in this fashion we get an infinite graph. For that reason, after the second analysis pass of the loop body we apply the normalization operator. Based on the determination that the sections of the heap graph that edges 2, 3 refer to represent similar heap structures (see [1,15]), the normalization operation merges nodes 2 and 3 into a single summary node and edges 2, 3 into a single summary edge shown in Figure 6(c). In this summarization, since the two edges end at nodes that do not have an edge between them (and thus represent disjoint sections of the heap), we know that the summary edge represents pointers that refer to disjoint sections of the heap and thus abstracts unrelated pointers. Therefore, the resulting summary edge is labeled np.

Figure 6(c) shows the heap model that abstracts the program state at the exit of the loop. According to this model, there is a variable AU which points to an array of Data objects. This array, represented by node 2, may have many pointers stored in it, all of which are represented by edge 2. However, since the edge is labeled with the interfere property np we know that all of these pointers must refer to unique Data objects (i.e., they do not alias).

We now describe how the construction of array AS, a shallow copy of AU, is modeled in our domain. Figure 7(a) shows the state of the abstract heap after the first analysis pass of the loop body, line 6. The new array node pointed to by AS has a single edge (4) representing the pointer that was stored into it on line 6. Based on the assignment statement we know that this edge represents a pointer to a Data object that is also referred to by a pointer abstracted by edge 2. To capture this information, the analyzer updates the connectivity properties to indicate that a pointer abstracted by edge 4 may alias with the pointer abstracted by edge 2 (in the concrete heap the share predicate may be satisfied). This information is indicated in the figure with the !4 tag in the label for edge 2, and the !2 tag in the label for edge 4. The analyzer is able to infer that the store into AS[0] does not affect pointers stored in AU, thus the np property is not altered and since edge 4 currently abstracts a single pointer, it also has the np property.

Figure 7(b) shows the model representing the abstract heap after the second analysis pass over the loop body. We have added edge 5 to represent the pointer that was created by the store operation. The analysis has determined that there is possible sharing between all the edges (2, 4, 5) that point to node 2. Although taken pairwise these edges abstract potentially aliasing pointers, each edge individually abstracts a set of pointers that all refer to disjoint sections of the heap.

The normalization of this model replaces edges 4 and 5 in Figure 7(b) with the single summary edge 4 in Figure 7(b). Based on the information that edges 4 and 5 satisfy the share predicate -which means that there may be a pointer abstracted by edge 4 and a pointer abstracted by edge 5 that alias-, the analysis must assume that the resulting summary edge 4 in Figure 7(c) may represent aliasing pointers and thus is given the ap property.

The model shown in Figure 7(c) is identified as a safe abstraction of the program at the loop exit. The model shows that variables AU and AS refer to distinct arrays, each of which contains some number of pointers. Additionally, the model is now able to distinguish between the aliasing properties of the pointers stored in array

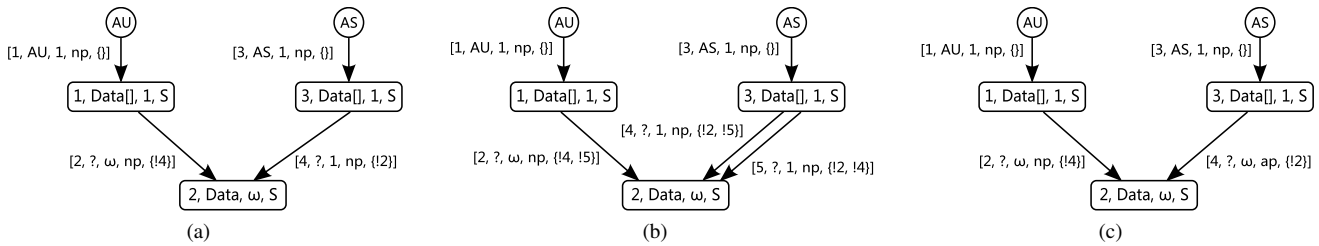


Figure 7. Abstract copy of AU into AS.

AU, all of which are known to be un–aliased with any other pointers in the array (edge 2 has the np label), and the pointers stored in the array AS, which may alias with the other pointers in the array (edge 4 has the ap label). This allows an optimizing compiler to determine that the loop in lines 7-8 may be safely parallelized. Although not needed for parallelization, the analysis is able to determine that the two arrays may contain references to the same object (or objects), as indicated by the !2 and !4 tags in the connectivity lists.

4. Extended Case Study: Barnes-Hutt

In this section we look in some detail at the analysis results for the **bh** program, from the JOlden suite [3]. We examine the types of structural and sharing information that the analysis can extract and how this information can be used to optimize this program.

The **bh** program performs a *fast–multipole* algorithm on the gravitational interaction between a set of bodies (the Body objects) and uses a space decomposition tree 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 vectors for accessing the bodies, bodyTab and bodyTabRev. The majority of the computation in **bh** is done by iterating over each Body object and walking the space decomposition tree (the root field) to determine a new acceleration value for the Body object, which is then stored in the newAcc field:

```
Iterator b = root.bodyTabRev.iterator();
while(b.hasNext())
  ((Body) b.next()).hackGravity(rsize, root);
```

Figure 8 shows the state of the heap model after the loop body. For clarity, we omit those attributes with default values: in the case of nodes, *layout* = *(S)ingleton* and *linearity* = 1, and in the case of edges, *linearity* = 1 and interference = np.

Our analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be a cyclic structure, as indicated by the *(C)ycle* layout value of node 17, which represents the Cell objects. However, the analysis is able to determine that the Cell objects and the Body objects represent distinct regions in the program. 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 and bodyTabRev) and the space decomposition tree. In particular, it has assumed that the pointer from the Cell objects in the space decomposition tree may have aliasing pointers (the ap entry in edge 18, marked in red if color is available) while there are no aliasing pointers in the vectors bodyTab and bodyTabRev. This is indicated by the omitted default interference attribute (np) of edges 3 and 11 abstracting these pointers. The analysis is also able to determine that the two vectors and the space decomposition tree may all point to some of the same Body objects (the (!18, !21) in edge 3, (!3, !18) in edge 21, and (!3, !21) in edge 18, respectively; these connectivity relations are marked in blue if color is available).

The sharing information about the pointers stored in each vector, combined with the observation that the space decomposition tree is only read in the loop body and that the only part of the heap which is modified is never read (the newAcc field) is sufficient to ensure that there is no heap–carried dependence in this loop. Thus, we can safely thread–parallelize the loop body, achieving a factor of 3.09 speedup on our quad–core test machine.

Of interest from a memory management and code scheduling standpoint is the behavior of the MathVector objects, which are used to represent k–dimensional vectors (using an array of k doubles where k is a small static final int known at compile time). Examining the heap through the entire program shows that each array is owned by a single MathVector at any time, the edges from the MathVector objects to the arrays are always non-interfering (np) (the omitted default value), and the array nodes never have multiple incoming edges from different MathVector nodes. This information allows us to safely inline the elements in the arrays directly into fields in the MathVector objects. This has the beneficial effect of increasing the data locality, removing many redundant loads and allowing aggressive loop scheduling, resulting in a 23% reduction in the runtime of the single threaded program, as well as reducing the size of the MathVector/Array composite structure object by a pointer (and the overhead of an array), resulting in a 37% reduction in memory usage.

5. Benchmarks

We have implemented a shape analyzer based on the shape predicates and sharing properties presented in this paper, and evaluated the effectiveness and efficiency of the analysis on programs from SPECjvm98 [17] and the entire non-trivial JOlden suite. The JOlden suite contains pointer-intensive kernels (derived from the Olden benchmarks [4]) that make use of recursive procedures, inheritance, and virtual methods. We modified the suite to use modern Java programming idioms and addressed major concerns in the

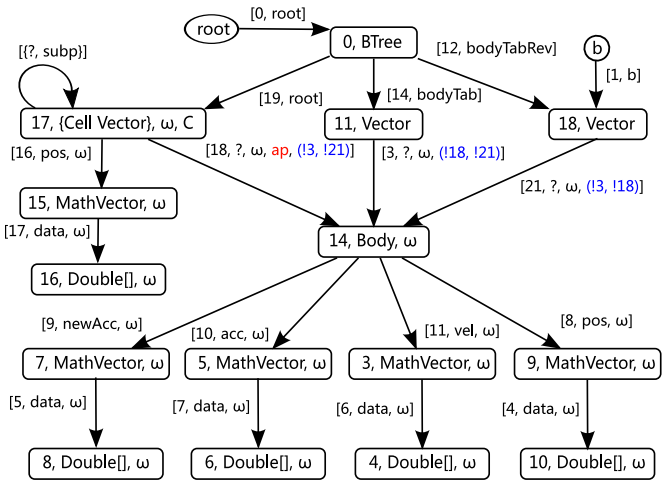


Figure 8. Barnes-Hutt

Benchmark	LOC	t	SU_e	SU_s	SU_b
bisort	560	0.26	2.38	2.38	2.38
mst	668	0.12	*NA*	*NA*	*NA*
tsp	910	0.15	3.16	3.16	3.16
em3d	1103	0.31	2.85	2.85	1.00
perimeter	1114	0.91	1.00	1.00	1.00
health	1269	1.25	3.21	3.21	1.00
voronoi	1324	1.80	2.43	2.43	2.43
power	1752	0.36	3.25	3.25	3.25
bh	2304	1.84	3.09	1.00	1.00
db	1985	1.42	*NA*	*NA*	*NA*
raytrace	5809	37.09	2.92	1.00	1.00

Figure 9. Analysis times (in seconds) and speedups obtained using the extended, simple, and base domains. LOC is of the MIL program including required library stubs.

literature [20]. The benchmarks **raytrace** (modified to be single–threaded) and **db** are taken from SPECjvm98.

The analyzer is written in C++ and compiled using MSVC 8.0. The analysis, as well as the parallelization benchmarks, were run on a 2.6 GHz Intel quad-core machine with 4 GB of RAM (although memory consumption never exceeded 160 MB). Both the analyzer and the benchmarks in use are publicly available [1].

The sharing information inferred by the analysis permits identifying loops and recursive calls that read from/write to disjoint sections of the heap. This allowed us to parallelize the benchmarks to use multiple threads in loops and calls [9, 12] to exploit the four cores of the test machine. The SU_e column in Figure 9 shows the resulting speedups by using the information of the extended domain described in Section 3. Two of the benchmarks (**mst** and **db**) do not use any algorithms that can be parallelized using the parallelization approaches in [9, 12]; we mark these with *NA*. In all but one of the remaining benchmarks we are able to achieve a significant performance improvement (up to 3.25× on **power**).

To isolate the impact that sharing has on these results, we ran the analysis with two alternative, simpler abstract domains: one in which sharing is tracked in a coarser manner [15], since there is no distinction between the alias and the interfere predicates, and one in which no sharing information is approximated. Columns SU_s and SU_b, respectively, show the speedups obtained by using the information inferred by these approaches. Runtimes for the SU_s and SU_b analysis are omitted as they were within 10% of the values for the analysis with sharing which are shown in the t column. The results show that the concept of sharing is critical to achieve results that can be used to effectively perform thread level parallelism transformations. Without sharing information (column SU_b) a large number of the benchmarks cannot be parallelized. The simple sharing properties of [15] are capable of providing the information needed to parallelize several of the other benchmarks, but are unable to provide the information needed to parallelize two of the more complex benchmarks (**bh** and **raytrace**).

These results are quite encouraging as many of the programs that we have analyzed here are well known (have been available for a number of years in commonly used benchmark suites) and have not, to the best of our knowledge, been successfully analyzed prior to this work. The Olden benchmark suite was introduced in 1995 as a challenge problem to assess the effectiveness of parallelizing compilers and parallel architectures on programs that make extensive use of dynamically allocated data structures. Our survey of the literature indicates that the system used in this paper is the only heap analysis that can identify the connectivity and sharing properties needed to parallelize a number of the benchmarks (**em3d**, **voronoi**, **bh**). Similarly, the **raytrace** and **db** benchmarks included in SPECjvm98 have not been successfully analyzed by any other shape analysis technique, despite the widespread use of the suite.

6. Related work

A number of techniques have been developed to analyze the connectivity properties of the program heap. Early work on access path or graph based approaches with interference and reachability information [5, 7, 8, 13] can be used to successfully analyze simple sharing as in the array example in this paper, but are severely limited in their ability to deal with recursive data structures (particularly updates in these structures) and sharing properties in composite structures such as those found in **bh**.

More recent work has focused on the ability to precisely model the shapes of recursive data structures and destructive updates to them [2, 11, 19]. The focus has been mostly on sharing that occurs internally in recursive heap structures (recursive trees of list-type structures). This work assumes that more complex sharing, such as the sharing between the two different arrays of our motivating example, does not occur in the program. Thus, these techniques cannot be applied to the program in Figure 1, even to produce a conservative approximation, since they lack the expressivity to capture this type of sharing and always assume that the objects in arrays/lists are unshared.

A recent approach [10], which can analyze programs with sharing of objects in arrays/lists, extends the access path model in [7] with limited quantification over the index variables in access paths. This technique models sharing in shallow data structures in a more precise manner than our domain. However, as it is based on quantification over access paths and given the performance of the analysis in the preliminary results the ability to scale the approach to larger programs with significant amounts of sharing is uncertain.

In [15] we introduced the base domain of Section 2, augmented with simple support for sharing that is able to differentiate purely unshared sections of the heap sections which may have some sharing. This is sufficient for many simple types of heap structures, such as those built in **em3d** or **tsp**, but cannot precisely capture more complex sharing properties between the various heap structures built by programs like **bh** or **voronoi**.

7. Conclusion

This paper focuses on modeling the sharing of data elements in container constructs: arrays, library collections, and recursive data structures. The ability to precisely represent these sharing properties is critical to the precise modeling of the heap and to supporting a wide range of program optimization techniques.

In this paper we showed how sharing information can be characterized in the concrete heap and how this characterization can be used to extend an abstract heap model to precisely model these properties. We demonstrated the effectiveness of this approach using as a case study the Barnes-Hutt (**bh**) benchmark, which cannot be analyzed with other existing heap analysis techniques. With the sharing (and shape) information provided by our analysis, we are able to achieve a 23% performance improvement in single threaded execution, a 37% reduction in heap usage, and a factor of 3 parallel speedup on a quad–core machine. We also used this information to successfully thread–level parallelize a range of well–known benchmark programs. In addition to providing the accurate shape and sharing information required to parallelize these programs, the analysis presented in this paper is computationally efficient: each small benchmark is analyzed in less than 2s, while **raytrace** at 5809 LOC takes only 37s.

Based on these results, and the fact that the analysis (in conjunction with a simple compiler front end) is able to deal with nearly all the features in the Java 1.4 language, the analysis described in this paper is robust enough to be generally useful in the optimization of real–world small/medium size Java programs. We plan to continue work on scaling the analysis to handle larger programs with the same level of precision.

Acknowledgements. This work is supported in part by NNSF grant 0540600. Mario Méndez-Lojo and Manuel Hermenegildo are supported by the Prince of Asturias Chair in IST at UNM, by EU IST-FET project 15905 *MOBIUS*, by Spanish MCI project TIN-2005-09207 *MERIT*, and by Madrid Regional Government project *PROMESAS*.

References

[1] `http://cs.unm.edu/~marron`.

[2] 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.

[3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, September 2001.

[4] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. *J. Parallel and Distributed Computing*, 1996.

[5] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[6] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *PLDI*, pages 230–241, 1994.

[8] 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.

[9] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive darta structures. In *CC*, 1998.

[10] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.

[11] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

[12] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1), 1990.

[13] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *PLDI*, 1989.

[14] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[15] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.

[16] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

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

[18] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.

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

[20] C. Zilles. Benchmark health considered harmful. In *Computer Arch. News*, 2001.