Identification of Heap–Carried Data Dependence Via Explicit Store Heap Models

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

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

Abstract. Dependence information between program values is extensively used in many program optimization techniques. The ability to identify statements, calls and loop iterations that do not depend on each other enables many transformations which increase the instruction and thread-level parallelism in a program. When program variables contain complex data structures including arrays, records, and recursive data structures, the ability to precisely model data dependence based on heap structure remains a challenging problem.
This paper presents a technique for precisely tracking heap based data dependence in non-trivial Java programs via static analysis. Using an abstract interpretation framework, the approach extends a shape analysis technique based on an existing graph model of heaps, by integrating read/write history information and intelligent memoization. The method has been implemented and its effectiveness and utility are demonstrated by computing detailed dependence information for two benchmarks (Em3d and BH from the JOlden suite) and using this information to parallelize the benchmarks.

1 Introduction

The concept of data dependence between program statements is a fundamental tool for the reordering of program statements and the determination of invariant values in basic blocks, loops, or methods. Knowledge of data dependence allows the introduction of instruction–level parallelism and thread–level parallelism (both in loops and method invocations). In past work effective techniques for computing data dependence between scalar variables have been developed. However, the extension of this work to tracking memory–carried data dependence has been much less successful, in large part due to the lack of suitable heap analysis techniques to support them.

Previous work focused broadly on two approaches for identifying possible heap–carried data dependence, shape or points-to analysis as a proxy for data dependence [4, 7, 2, 14] wherein the identification of various acyclic structures and/or access path information is used to infer which expressions cannot access the same portion of the heap, and the explicit tracking of read/written locations [8, 3, 9] which model the set of locations that may be read/written at each program point. This work introduced several fundamental concepts involved in modeling heap carried data dependence. However experimental work with these approaches was limited to small numbers of micro-benchmarks or used coarse points-to style analysis.

This work is supported in part by NSF grant 0540600.

This paper builds on the basic concepts developed in earlier work and makes several contributions which are critical to analyzing non-trivial programs. The first is a novel method for tracking read/write locations during the analysis. The approach presented in this paper only tracks a two program locations per object field (one read location and one write location) instead of a set of all possible read locations and a set of all possible write locations per field. This is sufficient to identify the most recent program point where each memory location may be used/modified while avoiding the additional space usage and computational cost of tracking a set of program locations per object field. The next contribution is a method to efficiently track read/write information through method boundaries, in particular how to ensure that the addition of *use–mod* information does not have a serious impact on the memoization of method body analysis results, which is critical to applying the technique to realistic programs.

Our analysis technique uses an explicit store model for the heap objects which allows us to easily track the identity of objects between program statements. This differs from some recent work on shape analysis, which uses logical models with implicit store representations [15, 5] that cannot be efficiently extended to track the properties of arbitrary heap locations. It also differs from approaches based on separation logic which restrict the program to regular recursive structures and limited sharing of objects on the heap in order to ensure termination [6, 1]. These features preclude the use of these approaches on many realistic application programs including the em3d and bh benchmarks, which we analyze as detailed case studies here.

2 Running Examples

We use examples in this paper to illustrate the various aspects of the analysis technique. The first is a small fragment created solely to illustrate the basics of the analysis. The second is a routine taken from em3d, one of the JOlden [13, 10] benchmarks.

```
m1  void main() {
m2     Pair p = new Pair(new Data(5), new Data(10));
m3     if(*)
m4       p.first.val = 0;
m5     swap(p);
m6     assert(p.first.val != 0);
m7  }

s1  void swap(Pair p) {
s2     Data temp = p.first;
s3     p.first = p.second;
s4     p.second = temp;
s5  }
```

Fig. 1. Conditional Modify and Swap

```
c1  static void computeNewValue(ENode n) {
c2    for(int i = 0; i < n.fromCount; i++)
c3      n.value -= n.coeffs[i] * n.fromN[i].value;
c4  }
```

Fig. 2. `Compute` (From `em3d`)

The first example 1 creates 2 `Data` objects, each of which has a single integer field `val`, and puts them in a `Pair` object. If the conditional holds the `first` element of the pair is modified and then the swap method is called to interchange the `first` and `second` elements of the pair. This example is simple but relevant since in order to determine that the asserted property always holds the analysis needs to be able to track how pointer stores affect reachability relations in the heap, to identify where each heap location may be written, and do so across method invocations.

The second program fragment is a method taken from the `em3d` benchmark. This program builds a bipartite heap structure. Each call to `computeNewValue` takes a `ENode` object from one side of the bipartite graph and updates the `value` field of this node based on the value fields of `ENode` objects on the opposite side of the bipartite graph. This example demonstrates the importance of precisely resolving the heap structure so the dependence analysis can determine that the set of heap location where the `value` field is written is distinct from the locations that are read.

3 Abstract Heap Domain

The underlying abstract heap domain that we extend is a graph in which each node represents a region of the heap (a set of objects or data structures) 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.

Types. Since each node in the graph represents a region of the heap (which may contain objects of many types) we use a set of type names for each node in the heap graph which 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 2 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 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 states that each object has at most one pointer to another object in the region. The other predicates correspond to the standard definitions for Trees, Dags, and Cycles in the literature.

Interference. The heap model uses two properties to track the potential that multiple pointers or variables can reach the same memory location in the region that a particular

node represents. In this work the examples only require one of these properties (*interference*) so we omit the discussion of the other property (*connectivity*) and refer the interested reader to [12] for a more detailed description.

Each edge abstracts a set of pointers in the concrete program. The *interfere* property has three possible values, to track that some of the pointers may alias (*ap*), that none of the pointers alias but they may point into the same data structure (thus can interfere, *ip*), or that each of the pointers refers to a unique and disjoint data structure in the node that the edge ends at (they are disjoint and non-interfering, *np*).

Heap Representation. We represent abstract heaps pictorially as labeled, directed multigraphs. The variable nodes are labeled with the variable that they represent. The nodes representing the regions are represented as a record `[type, linearity, layout]` that tracks the instrumentation predicates.

The edges (which represent sets of pointers) in the figures are represented as records `[offset, linearity, interfere]`. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. These labels may be any of the field identifiers that are used in the program or the special label, *?*, which is the label given to the summary field representing all the elements in a collection object `Vector`, `List`, or an array.

To simplify the figures we omit entries in the labels when they are the default domain value. The default values for the nodes are *layout* = *(S)ingleton* and *linearity* = 1. The default edge values are *linearity* = 1 and *interfere* = *np*. The variable edges always represent single references and the label is always implicitly the variable name.

3.1 Heap Structure Examples

Pair Example. Figure 3 shows the heap model (without any read/write information) that is computed as the result of executing the pair constructor in the first example program. The variable `p` points to a single object of type `Pair` (the *linearity* is 1 and the shape in *Singleton*, as described above this default information is omitted from the figure). The node representing the `Pair` object has 2 outgoing edges representing the two pointers stored in the `first` and `second` fields. The analysis determines that these edges each represent a single pointer (and since any edge representing a single pointer cannot have any interference the *interfere* property is *np*). Again the default properties of linearity 1 and non-interference are omitted from the figure. Finally, the model shows that the `first` and `second` pointers each refer to a single `Data` object.

Em3d Example. The state of the heap at the entry to the `computeNewValue` method in the program `em3d` is shown in Figure 4 (again without any read/write information). The `em3d` program computes electro-magnetic field values in a 3–dimensional space by constructing a list of `ENode` objects, each representing an electric field value and a second list of `ENode` objects, which represent a magnetic field values. To compute how the electric/magnetic field value for a given `ENode` object is updated at each step the `computeNewValue` method uses an array of `ENode` objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result.

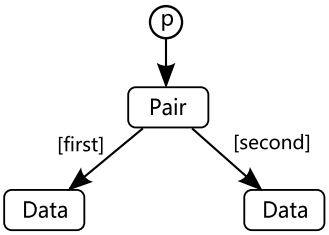


Fig. 3. `Pair` Allocation, Structure Only

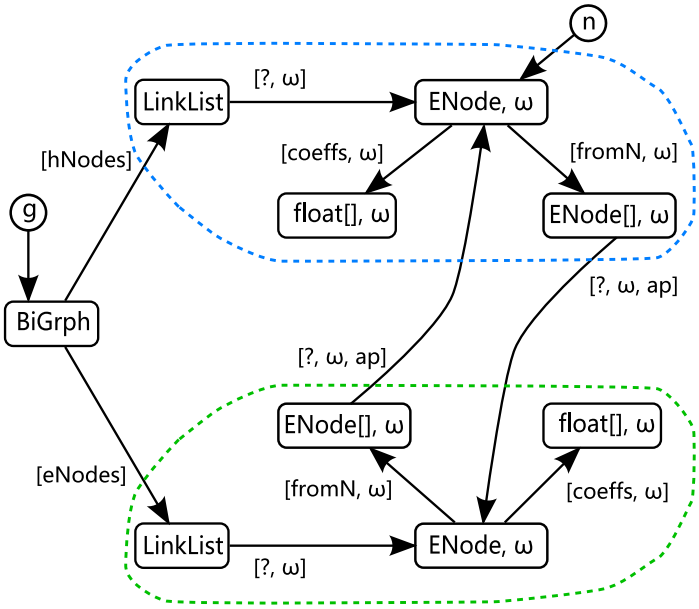


Fig. 4. `computeNewValue`, Structure Only

Figure 4 shows the heap structure computed for the `computeNewValue` method. We have placed dashed lines around the structures that represent the magnetic field (in blue if color is available) and the electric field (in green). Variable `g` points to a single object of type `BiGrph`, which is the data structure that encapsulates all the objects of interest. The `BiGrph` object has 2 fields, the `hNodes` field pointing to a linked list of `ENode` objects that make up the magnetic field and, the `eNodes` field pointing to a linked list of `ENode` objects that make up the electric field.

Looking at the structures in the magnetic field we see the edge labeled *[?, ω]* which represents all the pointers stored in the linked list. Since the linearity is ω we know the edge may represent multiple pointers but each of these pointers must point to a unique `ENode` object (the default interference value of non-interfering *np* is omitted). The figure also shows that the magnetic field is represented by many `ENode` objects (the node labeled *[ENode, ω]*) each of which has a pointer to a unique array of `floats` (the edge labeled `coeffs`) and an array of `ENode` objects (the edge labeled `fromN`) which are used as the set of nodes from the opposite field. The edge that represents the

pointers stored in this array is labeled *[?, ω, ap]*, which indicates that it represents all the pointers stored in the array and, that the pointers it represents may alias (*ap*).

4 Data Dependence Extensions

To track the read/write histories of objects on the heap we extend the model presented in Section 3 with information to track the identity of the objects represented by a given node, and for each field in the object we track the *most recent* program location (statement or control flow structure) where a read/write of that field *may* have occurred.

In order to ensure that the initial shape analysis when augmented with the read/write domain remains efficient it is critical to minimize the amount of additional information that is added to the heap model. The key observation is that for most optimization applications the shape analysis only needs to provide precise information about the *most recent* program location at which each field *may* have been read or written. Thus, the analysis does not need to track every possible program location where a field may have been read/written, and this significantly reduces the computational requirements.

4.1 Intermediate Representation

Before we introduce the domain extensions we need to specify how program locations are represented. To simplify the analysis the Java programs are transformed into a structured mid-level intermediate language (called MIL). The partial grammar below provides a sample of the language constructs in the intermediate representation.

$$
\begin{array}{lll}
\mathit{atom} & ::= & \mathit{var} \mid \mathit{literal} \\
\mathit{expr} & ::= & \mathit{atom} \mid \mathit{atom} + \mathit{atom} \mid \text{new } \mathit{type}(\mathit{atom}, \ldots, \mathit{atom}) \mid \mathit{var}.f \\
 & & \mid \mathit{var}.m(\mathit{atom}, \ldots, \mathit{atom}) \mid \mathit{var} \text{ instanceof } \mathit{type} \mid \ldots \\
\mathit{stmt} & ::= & \mathit{var}{=}\mathit{expr} \mid \mathit{var}.f{=}\mathit{atom} \mid \text{break} \mid \ldots \\
\mathit{contol} & ::= & \text{if}(\mathit{atom})\ \mathit{block}\ \text{else}\ \mathit{block} \mid \text{while}(\mathit{atom})\ \mathit{block} \mid \ldots \\
\mathit{block} & ::= & (\mathit{stmt} \mid \mathit{control})*
\end{array}
$$

The language has method invocations, conditional constructs (`if`, `switch`), exception handling (`try-throw-catch`) and looping statements (`for`, `do`, `while`). The state modification and expressions cover the standard range of program operations (load, store and assign along with logical, arithmetic and comparison operators). We associate with each statement and each control flow structure a program location ℓ.

4.2 Extended Domain

Read–Write Locations. Each node may represent a number of objects of different types ($\tau_1 \ldots \tau_m$) and each type may have many fields ($f^1_{\tau_i} \ldots f^n_{\tau_i}$). For each of these fields we keep two program locations (ℓ), the last time the field *may* have been read (ℓ_r) and the last time the field *may* have been written (ℓ_w).

Node Identity. In order to efficiently analyze method invocations we memoize the input and return abstract states and reuse them as possible. In order to prevent spurious inequalities between the read/write program locations (that refer to the locations in the

caller scope) in the memoized models we replace them with a generic *modified outside* value. To allow us to match the identities of the objects in the input state with their position in the output state we add a unique identity tag (a value in $\mathbb{N}$) to each node that is passed into a method call.

In our extended domain each node in the heap is now represented as a tuple `[type, linearity, layout, scalar-fields, identity]`. The entries `type`, `layout` and `count` are as described in Section 3. The `scalar-fields` entry is a list of `field-readloc-writeloc` entries, one for each scalar field, where `readloc` and `writeloc` are either a program location ℓ or the special entry 0 (*modified outside*). The `identity` entry is a *set* of identity tags or is omitted entirely if the node does not have a identity tag associated with it (or for clarity if it is not relevant to the example).

To track the read write information for the pointer fields we extend each edge label to `[offset, linearity, interfere, readloc-writeloc]` where `readloc` and `writeloc` are defined the same as for the scalar fields in the nodes. Again, for clarity, we omit `readloc-writeloc` information if it is irrelevant to the example.

Figure 5 shows the model that is computed as the result of executing the pair constructor in the first example program. The pair is marked as having read and written the two pointer fields at initialization (the `m2-m2` entries on the `first` and `second` edges) and the identity tag is omitted (since this object was allocated in the current scope). The two `Data` objects which had their *val* fields initialized at program location *m2* have the entry `m2-m2` in their *scalar-fields* read/write entry.

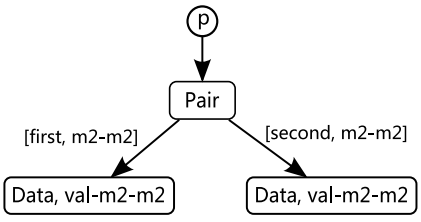


Fig. 5. `Pair` with *use-mod*

4.3 Local Data Dependence

Now that we have extended the model with the required instrumentation properties we can define a set of dataflow operations to model the effects of program operations on the read/write information. The changes for load and store operations are simple, only requiring an update of the last read/write value for the target object to the current program location, thus we omit a detailed description of these operations.

Data Flow Domain. If $\mathscr{G}$ is the set of all possible heap graphs and $\mathscr{B} = \wp(var) \times \wp(var)$ (a simple domain to track which variables *must* be *true*, the first element of the pair, and which *must* be false, the second element) then our abstract domain is $\mathscr{D} = \wp(\mathscr{G} \times \mathscr{B})$.

Given an element in the abstract domain, $\sigma \in \mathscr{D}$, we assume the abstract semantics are defined for expressions and statements. Thus, given an expression e, the abstract semantics of this expression on the abstract state σ are given by $\mathscr{S}[\![e]\!]\sigma$ and similarly for statement s, the abstract semantics are given by $\mathscr{S}[\![s]\!]\sigma$. Using the $\mathscr{B}$ component of the domain and a boolean condition b, we can filter an abstract state $\sigma = \{\theta_1, \ldots, \theta_k\}$ into two new abstract states $\sigma_{true} = \mathscr{S}[\![b]\!]_{true}(\sigma) = \{\ \theta_i \mid \text{b } \textit{may} \text{ be true in } \theta_i\}$ and $\sigma_{false} = \mathscr{S}[\![b]\!]_{false}(\sigma) = \{\ \theta_i \mid \text{b } \textit{may} \text{ be false in } \theta_i\}$.

Abstract Conditional Semantics. Using the above definitions we can write the standard definition for the `if` statement, $\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma = \mathscr{S}[\![block_t]\!](\sigma_{true}) \cup \mathscr{S}[\![block_f]\!](\sigma_{false})$. However, using this definition of the semantics can result in exponential growth in the number of states that the analysis must deal with (since for most cases at the union of the abstract states that result from analyzing *true* and *false* branches will have many models that are identical except for a few *readloc–writeloc* entries).

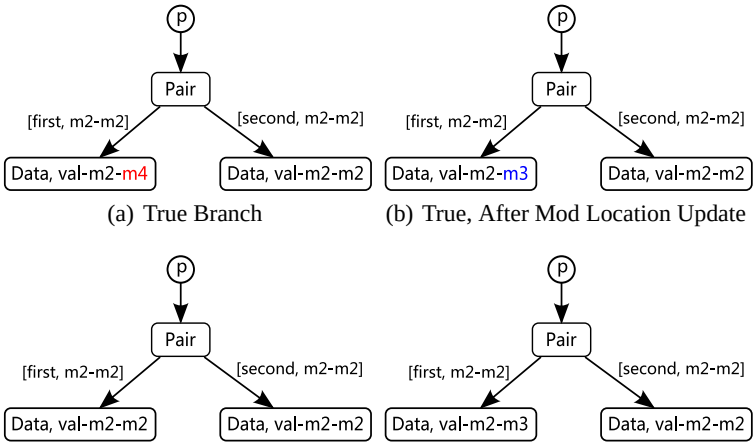
To avoid this we replace all the *readloc–writeloc* entries that refer to program locations in the *true* or *false* branches of the conditional with the program location of the conditional before the union operation. Thus any differences that are solely due to *readloc–writeloc* entries are removed and exponential growth is avoided. Given $\sigma = \{\theta_1, \ldots, \theta_k\}$ and a *block* which contains statements/control structures at program locations $pl = \{v_1, \ldots, v_i\}$, we define the operator $\clubsuit(\sigma, block, \mu) = \left\{ \theta_i|_{pl}^{\mu} \;\middle|\; \theta_i \in \sigma \right\}$, which performs the required replacements in the heap graph models. With this definition the improved semantics for the conditional operation (at program location κ) are:

$$\begin{aligned}&\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma = \\ &\quad \clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, \kappa) \cup \clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, \kappa)\end{aligned}$$

Disjunctive Domain. To speed program analysis we employ a *partially disjunctive domain* [11] which we use to discard elements in the abstract states (θ_i) that contain redundant read/write information. This is done by defining an order on the program locations based on their control–flow order. In general this order is not total (e.g. statement locations in the *true* and *false* branches of an `if` statement). However, our replacement of locations inside nested control–flow structures with the program location of the structure that contains them ensures that we can always compare the program locations that appear in the *readloc–writeloc* entries.

Analyze Conditional Example. Figure 6(a) is the abstract heap that approximates the state of the program after the *true* branch ($\mathscr{S}[\![block_t]\!](\sigma_{true})$), where the first element of the pair had the `val` field written. In the node that represents the `Data` object that was written we updated the *writeloc* entry to program location *m4* (where the write occurred, marked in red if color is available). Figure 6(b) shows the result of $\clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, m3)$, where we replaced the *readloc–writeloc* locations that appear in the *true* branch with program location of the `if` statement (program location *m3*, shown in blue).

Figure 6(c) shows the abstract heap from the *false* branch where no write occurred ($\mathscr{S}[\![block_f]\!](\sigma_{false})$). The most recent *mod* location is unchanged (program location *m2*,



(a) True Branch (b) True, After Mod Location Update

(c) False, After Mod Location Update (d) Discard Subsumed False Branch

Fig. 6. Updating Read/Write Locations At Control Flow Join

where the object was initialized) in $\clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, m3)$ since program location *m2* is not nested in the conditional.

Given our order relation on the *use-mod* sites we can simplify the models resulting from the *true* and *false* branches into a single model shown in Figure 6(d). Intuitively the *may use-mod* information from the *true* branch indicates that the memory location at `p.first.val` *may* have been written at location *m3* (the `if` statement) or at some previous point in the program, while the result of the *false* branch indicates that the memory location at `p.first.val` *may* have been written at location *m2*. Since the possibility that the object may be written at or before program location *m2* is implied by the statement that the object *may* be written at or before program location *m3* we can safely discard the model from the *false* branch.

Abstract Loop Semantics. The semantics of a looping statement `while` at program location κ can be expressed in terms of accumulating all possible exit states. To do this we define the state of the heap at the loop test for the i^{th} iteration of the loop as:

$$\sigma_i = \begin{cases} \sigma & \text{if } i = 0 \\ \mathscr{S}[\![block]\!](\mathscr{S}[\![b]\!]_{true}(\sigma_{i-1})) & \text{otherwise} \end{cases}$$

Then we can define the semantics of the loop analysis as the union of all the possible exits from the loop with the read/write program locations that occur within the loop body replaced by the program location of the loop (κ). Formally:

$$\mathscr{S}[\![while(b)\ block]\!]\sigma = \bigcup \left\{ \clubsuit(\mathscr{S}[\![b]\!]_{false}(\sigma_i), block, \kappa) \;\middle|\; i \in \mathbb{N} \right\}$$

4.4 Interprocedural Data Dependence

In order to efficiently handle large programs we memoize results of analyzing each method. At method call sites, if we were to naively compare the memoized heap models with the current call state the method specific *readloc–writeloc* entries we embed in

the model would create many spurious inequalities. As an example consider the `swap` function from our running example. The `swap` method could be called from multiple locations in a program and at each of these call sites the `Pair` object may have a different *readloc–writeloc* entries for the `first` and `second` fields. If comparision is done in a naive manner these differences will result in spurious mismatches with memoized analysis values, forcing the method to be re-analyzed for each call.

To avoid this problem we anonymize the *readloc–writeloc* locations before attempting to find a match in the memo table. However, when doing this anonymization we need to ensure that we can figure out which locations in the result heap *may* have been read/written in the call and which *must* not have been read/written (and thus have the same *readloc–writeloc* entry as before the call).

Call Example. The anonymization and remapping operations are conceptually simple but without some intuition into how they function the definitions are difficult to follow. Thus, we first examine how the `swap` call is handled in the pair example. Figure 7 shows the steps that are taken to analyze the call at program location *m5* assuming that the memo table contains Subfigures 7(a) and 7(b) as a memoized result.

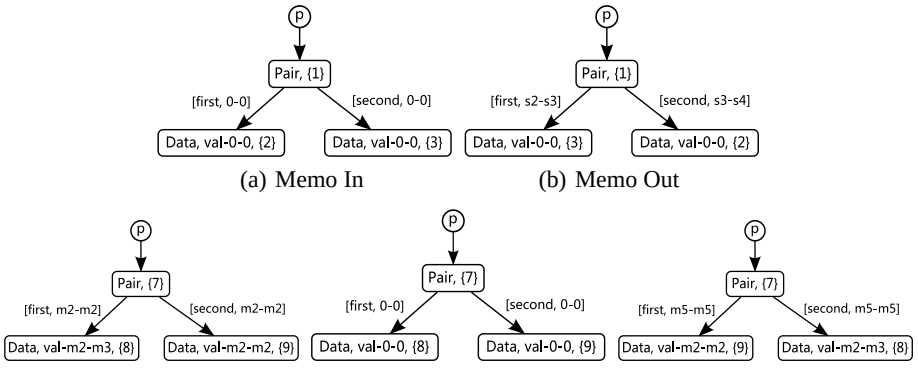
Figures 7(a) and 7(b) show that during the analysis of the `swap` method the analysis has determined the `first` and `second` fields have been read and written (the *readloc* and *writeloc* entries refer to program locations within the `swap` method, *s2*, *s3* and *s4*) but that the `val` fields are neither read nor written. The *readloc* and *writeloc* entries are the *modified–outside* value 0. Further, based on the identity tag sets we know that the object which was stored in the `first` field at the method entry (Figure 7(a)) and was given the identity tag 2 is stored in the `second` field at the method exit (Figure 7(b)). A similar situation holds for the object stored in the `second` field at the method entry, which was assigned the identity tag 3.

Figure 7(c) shows the state of the heap model at the call site (location *m5*) after we have added fresh tags (7, 8, and 9) to uniquely identify the nodes. After anonymizing the locations of the *readloc–writeloc* entries to the *modified–outside* value (0) we have the model shown in Figure 7(d), which is isomorphic (up to identity tags) to the model in our memo table, Figure 7(a).

During the anonymization we construct a map from the identity tags we added and the field identifiers to the *readloc–writeloc* entries in the caller scope that we are anonymizing. This gives us the map $ModM = \{(7,\texttt{first}) \rightarrow (m2,m2), (7,\texttt{second}) \rightarrow (m2,m2), (8,\texttt{val}) \rightarrow (m2,m3), (9,\texttt{val}) \rightarrow (m2,m2)\}$. Using the isomorphism from $\sigma_{in} \mapsto \sigma_{call}$ we have a map $\Pi = \{1 \rightarrow 7, 2 \rightarrow 8, 3 \rightarrow 9\}$.

Using these maps we transfer the read/write information from the call input to the memoized output, replacing any *readloc–writeloc* entries that refer to program locations in the callee body (`swap`) with the program location of the call site (program location *m5*) and replacing any occurrences of the *modified outside* value with the appropriate entry from *modM*. In Figure 7(b) the node with identify tag 2 has the *modified outside* value for the *readloc*/*writeloc* of the `val` field (`val-0-0`). To place the correct *readloc–writeloc* values into this node we look up the node that it maps to in the caller scope (via the Π map), which gives us the identity tag 8. Then we look up the caller scope *readloc–writeloc* information in the *modM* map, which gives us the read/write information for the field, `m2-m3`.

This remapping gives us the result in Figure 7(e), which shows that the object stored in the `second` field of the `Pair` object may have been written at program location *m3* but that the object stored in the `first` field has not been modified since initialization at program location *m2*. Thus, we can determine that the read from `p.first.val` is non-zero and the assertion will always succeed.



(a) Memo In (b) Memo Out

(c) Call In (d) Anonymized (e) Call Out

Fig. 7. Mapping Through Memoization

Dataflow Operations. For a method invocation at call site ℓ_{call} we give each node in the call state σ_{call} a unique tag $\kappa \in \mathbb{N}$, set the read/write location to the *modified outside* value and build a map $ModM : \mathbb{N} \times field \mapsto (\ell_r, \ell_w)$.

We then compare the anonymized version of σ_{call} with the entries in the memo table ignoring the read/write information. If a match $(\sigma_{in}, \sigma_{out})$ is found then there is a graph isomorphism $\Phi : \sigma_{in} \mapsto \sigma_{call}$. This isomorphism and the fact that the set of location tags in σ_{in} and σ_{out} are the same implicitly defines a map, $\Pi : \{\, \kappa \mid \kappa \textit{ a location tag } \in \sigma_{out}\} \mapsto \{\, \kappa' \mid \kappa' \textit{ a location tag } \in \sigma_{call}\}$. Using this map we can then compute the result of the call by replacing any *readloc–writeloc* values (ℓ_x) for the fields in each node *n* with:

$$(\ell'_x) = \begin{cases} \ell_{call}, \text{ if } \ell'_x \text{ is a location in the callee method} \\ \max(\{\, n'.\ell_x \mid \kappa \in n.\text{identity} \wedge n' \in \sigma_{call} \wedge \Pi(\kappa) \in n'.\text{identity}\}), \text{ otherwise} \end{cases}$$

5 Experimental Results

In this section we examine how the data dependence information can be used to perform thread level parallelization on variations of two of the more complex JOlden benchmarks, `em3d` and `bh` [13, 10]. To asses the performance of our approach we examine the analysis runtime on the JOlden suite, several of the SPECjvm98 benchmarks [16], and a logic formula manipulation program we developed as test case.

5.1 Case Studies:

Em3d. The first application of the read/write dependence information we look at is performing thread–level parallelization of the em3d benchmark. In Figure 2 we show the code for updating the `value` field of a single `ENode` object. By applying our read/write analysis we obtain the model in Figure 8 at the end of the method body. We see that some object from the list of magnetic field nodes has had the `value` field both read and written in the loop, *readloc* = *c2* and *writeloc* = *c2* (marked in red if color is available), while there have been reads from the `coeffs` and `fromN` pointer fields, *readloc* = *c2* (marked in green), *writeloc* = 0. The pointers in the `fromN` array have also been read in order to access the `value` fields in the `ENode` objects in the opposite field, which have been read but not written (*readloc* = *c2*, *writeloc* = 0).

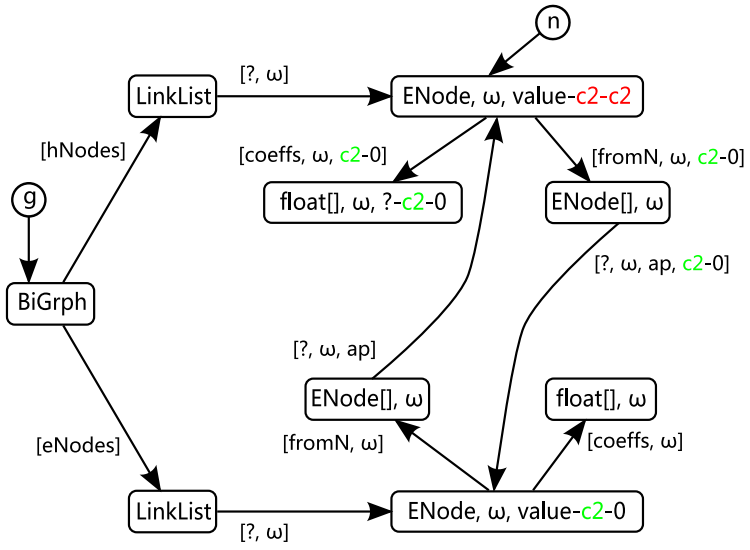


Fig. 8. Em3d With Read/Write Info

```
e1  for(int i = 0; i < this.hNodes.size(); ++i)
e2    computeNewValue((ENode) this.hNodes.get(i));
```

Fig. 9. Main Em3d Compute Loop

Using this information, the fact that each reference in the linked list (`LinkList`) of `ENode` objects refers to a unique object (the edge is *np*, the omitted default interference value) and the linear loop iteration, allows us to determine that each magnetic `ENode` object is written on a single iteration of the main update loop, program location *e2*, in Figure 9, which calls `computeNewValue`. Given this information it is valid to thread parallelize this loop (and to vectorize the loop in `computeNewValue`). Doing so results in a speedup of 3.21 on our quad-core test machine.

BH. Figure 10 shows the model that the analysis computes for the heap based read/write information in the `hackGravity` method of the *Barnes–Hutt* benchmark. For clarity we have simplified the heap structure in areas that are not relevant to this example.

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 a subtree or a reference to the `Body` objects. The program also keeps two vectors for accessing the bodies, `bodyTab` and `bodyTabRev`. Figure 10 shows the state of the heap model after the loop body (Figure 11) that contains the majority of the computation in `bh`. This loop takes each `Body` object and walks the space decomposition tree (the `root` field) to determine a new acceleration value for the `Body` object (stored in the `newAcc` field).

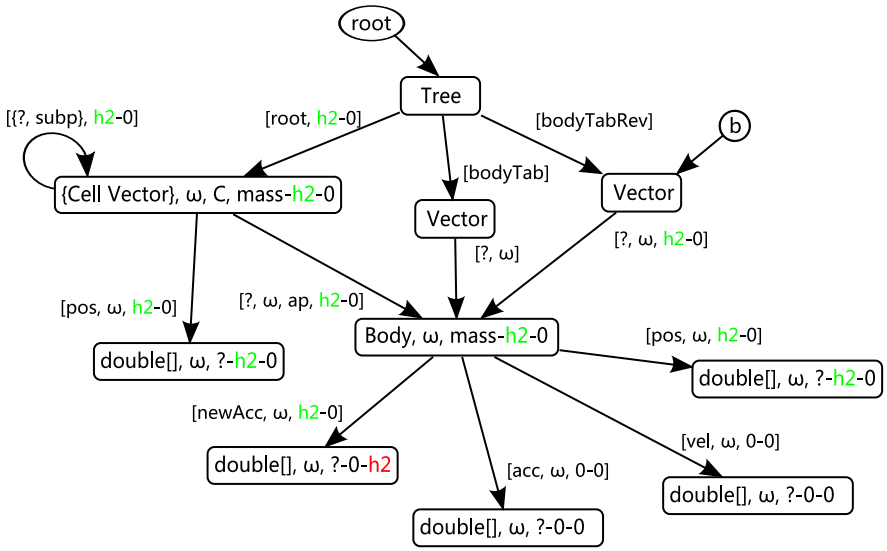


Fig. 10. BH With Read/Write Info

Our analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be a cyclic structure (shown by the `C` in the node representing the `Cell` objects). However, the analysis is able to determine that the `Cell` objects and the `Body` objects represent distinct regions in the program. This piece of information combined with the observation that the space decomposition tree is only read in the loop body (all the *readloc* entries set to *h2*, marked in green, and the *writeloc* entries set to 0), that the only part of the heap which is modified is never read (the `double[]` stored in the `newAcc` field, *writeloc* = *h2*, set to red), and that the collection being indexed over (the `Vector` referred to by the `bodyTabRev` field) does not have multiple references to the same object (the *?* edge is *np*, the omitted default interference value), is sufficient to ensure that there are no heap–carried dependence in this loop. Thus, we can safely thread–parallelize the loop body, achieving a factor of 2.98 speedup on our test machine.

```
h1  Iterator b = this.bodyTabRev.iterator();
h2  while(b.hasNext())
h3    ((Body) b.next()).hackGravity(rsize, root);
```

Fig. 11. Main Update, Gravity Computation

5.2 Performance

The analysis algorithm was written in C++ and compiled using gcc 4.2. 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 60 MB).

The original Java programs are transformed into MIL programs and the required stub code is added to enable the analysis of the standard Java libraries (which requires from 200-600 lines depending on which libraries the benchmark uses). These MIL programs are then processed by the analyzer. A demo version of the analyzer and benchmarks can be obtained at [13].

Benchmark	LOC	Classes	Methods	Analysis Time	Shape	RW Dep
bisort	560	36	348	0.26s	Y	Y
mst	668	52	485	0.12s	Y	Y
tsp	910	42	429	0.15s	Y	Y
em3d	1103	56	488	0.31s	Y	Y
perimeter	1114	44	381	0.91s	P	N
health	1269	59	534	1.25s	Y	Y
voronoi	1324	58	549	1.80s	Y	Y
power	1752	57	520	0.36s	Y	Y
bh	2304	61	576	1.84s	P	Y
db	1985	68	562	1.42s	Y	Y
logic	3960	72	620	48.26s	P	Y
raytrace	5809	63	506	37.09s	Y	Y

Fig. 12. LOC is the size of the program after transformation to MIL (including library stub code that must be analyzed), Classes/Methods are the number of classes/methods in the program (including Java Libraries that are used). Shape reports the heap connectivity is correctly identified and RW Dep reports if the RW information is useful (as in Section 5.1).

We report Y(es) in the *Shape* column if the analysis correctly identified all the relevant the shape information of the heap structures in the program. P(artial) means the analysis was able to determine the precise shape for some of the data structures but that some properties were missed.

We report similar information for the utility of the RW information. Y(es) means the read/write information would be sufficient to introduce substantial thread level parallelism (as in Section 5.1) and provides the information required to enable significant instruction level parallelism optimizations (e.g. code motion to improve scheduling, elimination of redundant loads/stores or the identification of loop invariant values). We Report (N)o for only one of the benchmarks, perimeter, where the read/write informa-

tion does not enable any thread level parallelism and only enables minor scheduling or load elimination opportunities.

Our experimental results show that the analysis is capable of efficiently computing very precise heap–carried dependence information over a range of benchmarks. In particular the ability to compute this information on the benchmarks bh, em3d, voronoi and raytrace is a significant advance in the state of the art for understanding the program heap. Computing precise shape and dependence information for these benchmarks requires the analysis to precisely model recursive data structures, Java collections, non-trivial sharing between components of the heap and, in order to compute the dependence information, to precisely track the part of the heap each read/write affects.

The analysis presented in this paper is not only capable of accurately modeling all of these features but is able to do so efficiently (analyzing the smaller benchmarks takes less than 2s per benchmark and raytrace at 5809 LOC takes only 37s). Based on these results we believe that the analysis reported in this paper is robust enough to be generally useful in the optimization of smaller Java programs and we plan to continue work on scaling the analysis to handle larger programs with the same level of precision.

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. B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. *ACM SIGPLAN Notices*, 2000.
3. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.
4. R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *CC*, 1998.
5. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
6. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
7. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1), 1990.
8. S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *PLDI*, 1989.
9. J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, 1994.
10. Jolden Suite. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.
11. R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.
12. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
13. Modified Jolden and Demo, May 2008. `http://www.cs.unm.edu/~marron`.
14. R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, 1999.
15. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
16. Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.