Unified Analysis of Array and Object References in Strongly Typed Languages

Stephen Fink[1], Kathleen Knobe[2], and Vivek Sarkar[1]

[1] IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA
[2] Compaq Cambridge Research Laboratory
One Cambridge Center, Cambridge, MA 02139, USA

Abstract. We present a simple, unified approach for the analysis and optimization of object field and array element accesses in strongly typed languages, that works in the presence of object references/pointers. This approach builds on Array SSA form [14], a uniform representation for capturing control and data flow properties at the level of array elements. The techniques presented here extend previous analyses at the array element level [15] to handle both array element and object field accesses uniformly.
In the first part of this paper, we show how SSA-based program analyses developed for scalars and arrays can be extended to operate on object references in a strongly typed language like Java. The extension uses Array SSA form as its foundation by modeling object references as indices into hypothetical *heap arrays*. In the second part of this paper, we present two new sparse analysis algorithms using the heap array representation; one identifies redundant loads, and the other identifies dead stores. Using strong typing to help disambiguation, these algorithms are more efficient than equivalent analyses for weakly typed languages. Using the results of these algorithms, we can perform scalar replacement transformations to change operations on object fields and array elements into operations on scalar variables.
We present preliminary experimental results using the Jalapeño optimizing compiler infrastructure. These results illustrate the benefits obtained by performing redundant load and dead store elimination on Java programs. Our results show that the (dynamic) number of memory operations arising from array-element and object-field accesses can be reduced by up to 28%, resulting in execution time speedups of up to 1.1×.

Keywords: static single assignment (SSA) form, Array SSA form, load elimination, store elimination, scalar replacement, Java object references.

1 Introduction

Classical compiler analyses and optimizations have focused primarily on properties of scalar variables. While these analyses have been used successfully in practice for several years, it has long been recognized that more ambitious analyses must also consider non-scalar variables such as objects and arrays.

Past work on analysis and optimization of array and object references can be classified into three categories — 1) analysis of array references in scientific programs written in languages with named arrays such as Fortran, 2) analysis of pointer references in weakly typed languages such as C, and 3) analysis of object references in strongly typed languages such as Modula-3 and Java. This research focuses on the third category.

Analysis and optimization algorithms in the first category were driven by characteristics of the programming language (Fortran) used to write scientific programs. These algorithms typically use dependence analysis [22] to disambiguate array references, and limit their attention to loops with restricted control flow. The algorithms in this category did not consider the possibility of pointer-induced aliasing of arrays, and hence do not apply to programming languages (such as C and Java) where arrays might themselves be aliased.

Analysis and optimization algorithms in the second category face the daunting challenge of dealing with pointer-induced aliases in a weakly typed language. A large body of work has considered pointer analysis techniques (*e.g.,* [17,6,13,16,7,12]) that include powerful methods to track pointer references both intra- and inter-procedurally. However, many of these techniques have limited effectiveness for large "real-world" programs because the underlying language semantics force highly conservative default assumptions. In addition, these algorithms are known to be complex and time-consuming in practice.

The research problem addressed by our work falls in the third category *viz.,* efficient and effective analysis and optimization of array and object references in strongly typed object-oriented languages such as Java. Recent work on type-based alias analysis [11] has demonstrated the value of using type information in such languages in analysis and optimization.

In this paper, we present a new unified approach for analysis and optimization of object field and array element accesses in strongly typed languages, that works in the presence of object references/pointers. We introduce a new abstraction called *heap arrays*, which serves as a uniform representation for array and object references. Our approach is flow-sensitive, and therefore more general than past techniques for type-based alias analysis. In addition, our approach leverages past techniques for sparse analyses, namely Array SSA form (SSA) [14,15] and scalar global value numbering [1], to obtain analysis and optimization algorithms that are more efficient than the algorithms used for weakly type languages such as C.

To illustrate this approach, we present two new analysis algorithms for strongly typed languages: one identifies redundant loads, and one identifies dead stores. We have implemented our new algorithms in the Jalapeño optimizing compiler [3], and we present empirical results from our implementation. Our results show that the (dynamic) number of memory operations arising from array-element and object-field accesses can be reduced by up to 28%, resulting in execution time speedups of up to 1.1×.

Our interest in efficient analysis arises from our goal of optimizing large Java programs. For this context, we need optimization algorithms that are efficient enough to apply at runtime in a dynamic optimizing compiler [3]. However, we believe that the approach developed in this paper will also apply to other applications that require efficient analysis, such as program understanding tools that need to scale to large programs.

The rest of the paper is organized as follows. Section 2 outlines the foundations of our approach: Section 2.1 introduces heap arrays, and Section 2.2 summarizes an extension of global value numbering to efficiently precompute *definitely-same* and *definitely-different* information for heap array indices. Section 3 describes our algorithms to identify redundant loads and dead stores. Section 4 contains our preliminary experimental results. Section 5 discusses related work, and Section 6 contains our conclusions.

2 Analysis Framework

In this Section, we describe a unified representation called *extended Array SSA* form, which can be used to perform sparse dataflow analysis of values through scalars, array elements, and object references. First, we introduce a formalism called *heap arrays* which allows us to analyze object references with the same representation used for named arrays [14]. Then, we show how to use the extended Array SSA representation and global value numbering to disambiguate pointers with the same framework used to analyze array indices.

2.1 Heap Arrays

In this section, we describe our approach to analyzing accesses to object fields and array elements as accesses to elements of hypothetical *heap arrays*. The partitioning of memory locations into heap arrays is analogous to the partitioning of memory locations using type-based alias analysis [11]. The main difference is that our approach also performs a flow-sensitive analysis of element-level accesses to the heap arrays.

We model accesses to object fields as follows. For each field x, we introduce a hypothetical one-dimensional heap array, $\mathcal{H}^x$. Heap array $\mathcal{H}^x$ consolidates all instances of field x present in the heap. Heap arrays are indexed by object references. Thus, a GETFIELD[1] of $p.x$ is modeled as a read of element $\mathcal{H}^x[p]$, and a PUTFIELD of $q.x$ is modeled as a write of element $\mathcal{H}^x[q]$. The use of distinct heap arrays for distinct fields leverages the fact that accesses to distinct fields must be directed to distinct memory locations in a strongly typed language. Note that field x is considered to be the same field for objects of types C_1 and C_2, if x is declared in class C_1 and class C_2 extends class C_1 *i.e.,* if C_2 is a subclass of C_1.

Recall that arrays in an OO language like Java are also allocated in the heap — the program uses both an object reference and an integer subscript to access an array element. Therefore, we model such arrays as *two-dimensional*

[1] We use GETFIELD and PUTFIELD to denote general field access operators that may appear in three-address statements, not necessarily in Java bytecode.

heap arrays, with one dimension indexed by the object reference, and the second dimension indexed by the integer subscript. To avoid confusion, we refer to the array declared in the program as a "program array", and its representation in our model as its corresponding heap array.

The notation $\mathcal{H}^{\mathcal{T}[\,]\mathcal{R}}$ denotes a heap array, where $\mathcal{R}$ is the rank (dimensionality) of the underlying program array, and $\mathcal{T}$ is the element type. We introduce a distinct heap array for each distinct array type in the source language. Java contains seven possible array element types — bytes, chars, integers, longs, floats, doubles and objects. We denote the heap arrays for one-dimensional program arrays of these element types by $\mathcal{H}^{b[\,]}$, $\mathcal{H}^{c[\,]}$, $\mathcal{H}^{i[\,]}$, $\mathcal{H}^{l[\,]}$, $\mathcal{H}^{f[\,]}$, $\mathcal{H}^{d[\,]}$, and $\mathcal{H}^{O[\,]}$ respectively[2]. Thus, a read/write of a one-dimensional integer program array element `a[i]` corresponds to a read/write of heap array element $\mathcal{H}^{i[\,]}[a, i]$. In general, heap arrays for $\mathcal{R}$-dimensional arrays of these types are denoted by $\mathcal{H}^{b[\,]\mathcal{R}}$, $\mathcal{H}^{i[\,]\mathcal{R}}$, $\mathcal{H}^{l[\,]\mathcal{R}}$, $\mathcal{H}^{f[\,]\mathcal{R}}$, $\mathcal{H}^{d[\,]\mathcal{R}}$, and $\mathcal{H}^{O[\,]\mathcal{R}}$.

Note that we have only one heap array, $\mathcal{H}^{O[\,]\mathcal{R}}$, that represents all $\mathcal{R}$-dimensional arrays of objects. We could refine this approach by examining all the object array types used in the method being compiled, and replacing $\mathcal{H}^{O[\,]\mathcal{R}}$ by a set of heap arrays, one for each LCA (Least Common Ancestor) in the class hierarchy of the object array types.

Having modeled object and array references as accesses to named arrays, we can rename heap arrays and scalar variables to build an extended version of Array SSA form [14]. First, we rename heap arrays so that each renamed heap array has a unique static definition. This includes renaming of the dummy definition inserted at the start block to capture the unknown initial value of the heap array.

We insert three kinds of ϕ functions to obtain an *extended* Array SSA form that we use for data flow analyses[3]. Figure 1 illustrates the three types of ϕ function.

1. A *control* ϕ (denoted simply as ϕ) corresponds to the standard ϕ function from scalar SSA form [10], which represents a control flow merge of a set of reaching definitions.
2. A *definition* ϕ ($d\phi$) is used to deal with "non-killing" definitions. In scalar SSA, form a definition of a scalar kills the value of that scalar previously in effect. An assignment to an array element, however, must incorporate previous values. $d\phi$ functions were introduced in our past work on Array SSA form [14,15].
3. A *use* ϕ ($u\phi$) function creates a new name whenever a statement reads a heap array element. $u\phi$ functions were not used in prior work, and represent the extension in "extended" Array SSA form.

   The main purpose of the $u\phi$ function is to link together load instructions for the same heap array in control flow order. Intuitively, the $u\phi$ function

[2] By default, $\mathcal{R} = 1$ in our notation *i.e.,* we assume that the array is one-dimensional if $\mathcal{R}$ is not specified.

[3] The extended Array SSA form can also be viewed as a sparse data flow evaluation graph [8] for a heap array.

creates a new SSA variable name, with which a sparse dataflow analysis can associate a lattice variable.

We present one dataflow algorithm that uses the $u\phi$ for redundant load identification and one algorithm (dead store elimination) that does not require a new name at each use. In this latter case the $u\phi$ function is omitted.

We will sometimes need to distinguish between references (definitions and uses) that correspond directly to references in source and references added by construction of our extended Array SSA form. We refer to the first as *source* references and the second as *augmenting* references. In figure 1.c the references to $x_1[j]$, $x_2[k]$ and $x_3[i]$ are source references. The other references in that code fragment are all augmenting references.

Original program:

```
{$x$ in
effect here. }
$x[j]$ := ...
```

Insertion of $d\phi$:

```
{$x_0$ in
effect here. }
 $x_1[j]$ := ...
 $x_2$ := $d\phi$($x_1$,$x_0$)
```

(a)

Original program:

```
{$x$ in
effect here. }
if ... then
    $x[j]$ := ...
endif
```

Insertion of ϕ:

```
{$x_0$ in
effect here. }
if ... then
     $x_1[j]$ := ...
     $x_2$ := $d\phi$ ...
endif
 $x_3$ := $\phi$($x_2$, $x_0$)
```

(b)

Original program:

```
{$x$ in
effect here. }
$x[j]$ := ...
... := $x[k]$
... := $x[i]$
```

Insertion of $u\phi$:

```
{$x_0$ in
effect here. }
 $x_1[j]$ := ...
 $x_2$ := $d\phi$($x_1$,$x_0$)
... := $x_2[k]$
 $x_3$ := $u\phi$($x_2$)
... := $x_3[i]$
 $x_4$ := $u\phi$($x_3$)
```

(c)

Fig. 1. Three examples of ϕ nodes in extended Array SSA form.

The $d\phi$ and $u\phi$ functions in extended Array SSA form do not lead to excessive compile-time overhead because we introduce at most one $d\phi$ function for each heap array def and at most one $u\phi$ function for each heap array use. Instructions that operate on scalar variables do not introduce any heap array operations[4]. So, the worst-case size of the extended Array SSA form is proportional to the

[4] Note that local variables (stack elements) cannot be subject to pointer-induced aliasing in a strongly typed language such as Java.

size of the scalar SSA form that would be obtained if each heap array access is modeled as a def. Past empirical results have shown the size of scalar SSA form to be linearly proportional to the size of the input program [10], and the same should hold for extended Array SSA form.

2.2 Definitely-Same and Definitely-Different Analyses for Heap Array Indices

In this section, we show how the heap arrays of extended Array SSA form reduce questions of pointer analysis to questions regarding array indices. In particular, we show how global value numbering and allocation site information can be used to efficiently compute *definitely-same* ($\mathcal{DS}$) and *definitely-different* ($\mathcal{DD}$) information for heap array indices. For simplicity, the $\mathcal{DS}$ and $\mathcal{DD}$ analyses described in this section are limited in scope to scalar references.

As an example, consider the following Java source code fragment annotated with heap array accesses:

```
r = p ;
q = new Type1 ;
p.y = ... ;     // $\mathcal{H}^y[p] := \dots$
q.y = ... ;     // $\mathcal{H}^y[q] := \dots$
... = r.y ;     // $\dots := \mathcal{H}^y[r]$
```

Our analysis goal is to identify the redundant load of `r.y`, enabling the compiler to replace it with a use of scalar temporary that captures the value stored into `p.y`. We need to establish two facts to perform this transformation: 1) object references p and r are identical (definitely same) in all program executions, and 2) object references q and r are distinct (definitely different) in all program executions.

For a program in SSA form, we say that $\mathcal{DS}(a, b) = true$ if and only if variables a and b are known to have exactly the same value at all points that are dominated by the definition of a and dominated by the definition of b. Analogous to $\mathcal{DS}$, $\mathcal{DD}$ denotes a "definitely-different" binary relation *i.e.,* $\mathcal{DD}(a, b) = true$ if and only if a and b are known to have distinct (non-equal) values at all points that are dominated by the definition of a and dominated by the definition of b.

The problem of determining if two symbolic index values are the same is equivalent to the classical problem of *global value numbering* [1,18,21]. We use the notation $\mathcal{V}(i)$ to denote the value number of SSA variable i. Therefore, if $\mathcal{V}(i) = \mathcal{V}(j)$, then $\mathcal{DS}(i, j) = true$. For the code fragment above, the statement, `p = r`, ensures that p and r are given the same value number (*i.e.,* $\mathcal{V}(p) = \mathcal{V}(r)$), so that $\mathcal{DS}(p, r) = true$.

In general, the problem of computing $\mathcal{DD}$ is more complex than value numbering. Note that $\mathcal{DD}$, unlike $\mathcal{DS}$, is not an equivalence relation because $\mathcal{DD}$ is not transitive. $\mathcal{DD}(a, b) = true$ and $\mathcal{DD}(b, c) = true$, does not imply that $\mathcal{DD}(a, c) = true$.

For object references, we use allocation-site information to compute the $\mathcal{DD}$ relation. In particular, we rely on two observations:

1. Object references that contain the results of distinct allocation-sites must be different.
2. An object reference containing the result of an allocation-site must differ from any object reference that occurs at a program point that dominates the allocation site. (As a special case, this implies that the result of an allocation site must be distinct from all object references that are method parameters.)

For example, in the above code fragement, the presence of the allocation site in `q = new Type1` ensures that $\mathcal{DD}(p, q) = true$.

For array references, we currently rely on classical dependence analysis to compute the $\mathcal{DD}$ relationship within shared loops. Global $\mathcal{DD}$ is the subject of future work.

Although the computations of $\mathcal{DD}$ for object references and array indices differ, the algorithms presented here use both types of $\mathcal{DD}$ relation in the same way, resulting in a unified analysis for arrays and objects. This unified approach applies, for example, to analysis of Java arrays, which are themselves accessed by reference. In this case we need to determine 1) if two arrays references are definitely not aliased and 2) if the array indices referenced are definitely not the same.

In the remainder of the paper, we assume that the index of a heap array is, in general, a vector whose size matches the rank of the heap array *e.g.,* an index into a one-dimensional heap array $\mathcal{H}^x$ will be a vector of size 1 (*i.e.,* a scalar), and an index into a two-dimensional heap array $\mathcal{H}^{b[\,]}$ will be a vector of size 2. (For Java programs, heap arrays will have rank ≤ 2 since all program arrays are one-dimensional.) Given a vector index $\boldsymbol{k} = (k_1, \dots)$, we will use the notation $\mathcal{V}(\boldsymbol{k})$ to represent a vector of value numbers, $(\mathcal{V}(k_1), \dots)$. Thus, $\mathcal{DS}(\boldsymbol{j}, \boldsymbol{k})$ is *true* if and only if vectors $\boldsymbol{j}$ and $\boldsymbol{k}$ have the same size, and their corresponding elements are definitely-same *i.e.,* $\mathcal{DS}(j_i, k_i) = true$ for all i. Analogously, $\mathcal{DD}(\boldsymbol{j}, \boldsymbol{k})$ is *true* if and only if vectors $\boldsymbol{j}$ and $\boldsymbol{k}$ have the same size, and at least one pair of elements is definitely-different *i.e.,* $\mathcal{DD}(j_i, k_i) = true$ for some i.

3 Scalar Replacement Algorithms

In this Section, we introduce two new analyses based on extended Array SSA form. These two analyses form the backbone of *scalar replacement* transformations, which replace accesses to memory by uses of scalar temporaries. First, we present an analysis to identify fully redundant loads. Then, we present an analysis to identify dead stores.

Figure 2 illustrates three different cases of scalar replacement for object fields. All three cases can be identified by the algorithms presented in this paper. For the original program in figure 2(a), introducing a scalar temporary `T1` for the store (def) of `p.x` can enable the load (use) of `p.x` to be eliminated *i.e.,* to

Original program:

```
  p   := new Type1
  q   := new Type1
   . . .
 p.x  := ...
 q.x  := ...
...   := p.x
```

After redundant load elimination:

```
  p   := new Type1
  q   := new Type1
   . . .
  T1  := ...
 p.x  := T1
 q.x  := ...
...   := T1
```

(a)

Original program:

```
  p   := new Type1
  q   := new Type1
   . . .
...   := p.x
 q.x  := ...
...   := p.x
```

After redundant load elimination:

```
  p   := new Type1
  q   := new Type1
   . . .
  T2  := p.x
...   := T2
 q.x  := ...
...   := T2
```

(b)

Original program:

```
  p   := new Type1
  q   := new Type1
  r   := p
   . . .
 p.x  := ...
 q.x  := ...
 r.x  := ...
```

After dead store elimination:

```
  p   := new Type1
  q   := new Type1
  r   := p
   . . .
 q.x  := ...
 r.x  := ...
```

(c)

Fig. 2. Object examples of scalar replacement

Original program:

```
x[p]     := ...
x[p+1]   := ...
   ...   := x[p]
```

(a)

Original program:

```
   ...   := x[p]
x[p+1]   := ...
   ...   := x[p]
```

(b)

Original program:

```
x[p]     := ...
x[p+1]   := ...
x[p]     := ...
```

(c)

Fig. 3. Array examples of scalar replacement

be replaced by a use of T1. Figure 2(b) contains an example in which a scalar temporary (T2) is introduced for the first load of p.x, thus enabling the second load of p.x to be eliminated *i.e.,* replaced by T2. Finally, figure 2(c) contains an example in which the first store of p.x can be eliminated because it is known to be dead (redundant); no scalar temporary needs to be introduced in this case.

Figure 3 shows array-based examples. To highlight the uniform approach for both arrays and objects, these examples are totally analagous to the object based examples in figure 2.

Past algorithms for scalar replacement (*e.g.,* [4,2]) have been based on data dependence analysis or on exhaustive (dense) data flow analysis (*e.g.,* [5]). In this section, we show how extended Array SSA form, augmented with the definitely-same and definitely-different analysis information described in section 2.2, can be used to obtain a simple sparse scalar replacement algorithm. In addition, the use of SSA form enables our algorithm to find opportunities for scalar replacement that are not discovered by past algorithms that focus exclusively on innermost loops.

The rest of this section is organized as follows. Section 3.1 describes our analysis to identify redundant loads with respect to previous defs and previous uses, and Section 3.2 outlines our algorithm for dead store elimination.

3.1 Redundant Load Elimination

Input: Intermediate code for method being optimized, augmented with the $\mathcal{DS}$ and $\mathcal{DD}$ relations defined in Section 2.2.

Output: Transformed intermediate code after performing scalar replacement.

Algorithm:

1. **Build extended Array SSA form for each heap array.**
   Build Array SSA form, inserting control ϕ, $d\phi$ and $u\phi$ functions as outlined in Section 2.1, and renaming of all heap array definitions and uses.
   As part of this step, we annotate each call instruction with dummy defs and uses of each heap array for which a def or a use can reach the call instruction. If interprocedural analysis is possible, the call instruction's heap array defs and uses can be derived from a simple flow-insensitive summary of the called method.
2. **Perform index propagation.**
   (a) Walk through the extended Array SSA intermediate representation, and for each ϕ, $d\phi$, or $u\phi$ statement, create a dataflow equation with the appropriate operator as listed in Figures 5, 6 or 7.
   (b) Solve the system of dataflow equations by iterating to a fixed point.
   After index propagation, the lattice value of each heap array, A_i, is $\mathcal{L}(A_i) = \{ \mathcal{V}(\boldsymbol{k}) \mid \text{location } A[\boldsymbol{k}] \text{ is "available" at def } A_i \text{ (and all uses of } A_i) \}$.
3. **Scalar replacement analysis.**
   (a) Compute $UseRepSet = \{ \text{ use } A_j[\boldsymbol{x}] \mid \exists\ \mathcal{V}(\boldsymbol{x}) \in \mathcal{L}(A_j) \}$ *i.e.*, use $A_j[\boldsymbol{x}]$ is placed in *UseRepSet* if and only if location $A[\boldsymbol{x}]$ is available at the def of A_j and hence at the use of $A_j[\boldsymbol{x}]$. (Note that A_j uniquely identifies a use, since all uses are renamed in extended Array SSA form.)
   (b) Compute $DefRepSet = \{ \text{ def } A_i[\boldsymbol{k}] \mid \exists \text{ use } A_j[\boldsymbol{x}] \in UseRepSet \text{ with } \mathcal{V}(\boldsymbol{x}) = \mathcal{V}(\boldsymbol{k}) \}$ *i.e.*, def $A_i[\boldsymbol{k}]$ is placed in *DefRepSet* if and only if a use $A_j[\boldsymbol{x}]$ was placed in *UseRepSet* with $\mathcal{V}(\boldsymbol{x}) = \mathcal{V}(\boldsymbol{k})$.
4. **Scalar replacement transformation.**
   Apply scalar replacement actions selected in step 3 above to the *original* program and obtain the transformed program.

Fig. 4. Overview of Redundant Load Elimination algorithm.

Figure 4 outlines our algorithm for identifying uses (loads) of heap array elements that are redundant with respect to prior defs and uses of the same heap array. The algorithm's main analysis is *index propagation*, which identifies the set of indices that are *available* at a specific def/use A_i of heap array A.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\boldsymbol{i_1}), \dots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle(\boldsymbol{i'})\rangle$	$\top$	UPDATE$((\boldsymbol{i'}), \langle(\boldsymbol{i_1}), \dots\rangle)$	$\langle(\boldsymbol{i'})\rangle$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 5. Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := d\phi(A_1, A_0)$ is a definition ϕ operation

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\boldsymbol{i_1}), \dots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle(\boldsymbol{i'})\rangle$	$\top$	$\mathcal{L}(A_1) \cup \mathcal{L}(A_0)$	$\mathcal{L}(A_1)$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 6. Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{u\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := u\phi(A_1, A_0)$ is a use ϕ operation

$\mathcal{L}(A_2) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\boldsymbol{i_1}), \dots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \langle(\boldsymbol{i'_1}), \dots\rangle$	$\mathcal{L}(A_1)$	$\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 7. Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$, where $A_2 := \phi(A_1, A_0)$ is a control ϕ operation

(a) Extended Partial Array SSA form:	(b) After index propagation:	(c) Scalar replacement actions selected:	(d) After transforming original program:

`p := new Type1`
`q := new Type1`
`. . .`
$\mathcal{H}^x_1[p]$ `:= ...`
$\mathcal{H}^x_2$ `:=` $d\phi(\mathcal{H}^x_1, \mathcal{H}^x_0)$
$\mathcal{H}^x_3[q]$ `:= ...`
$\mathcal{H}^x_4$ `:=` $d\phi(\mathcal{H}^x_3, \mathcal{H}^x_2)$
`... :=` $\mathcal{H}^x_4[p]$
$\mathcal{H}^x_5$ `:=` $u\phi(\mathcal{H}^x_4)$
	$\mathcal{L}(\mathcal{H}^x_0) = \{\ \}$
	$\mathcal{L}(\mathcal{H}^x_1) = \{\mathcal{V}(p)\}$
	$\mathcal{L}(\mathcal{H}^x_2) = \{\mathcal{V}(p)\}$
	$\mathcal{L}(\mathcal{H}^x_3) = \{\mathcal{V}(q)\}$
	$\mathcal{L}(\mathcal{H}^x_4) = \{\mathcal{V}(p), \mathcal{V}(q)\}$
	$\mathcal{L}(\mathcal{H}^x_5) = \{\mathcal{V}(p), \mathcal{V}(q)\}$
		$\mathit{UseRepSet} = \{\mathcal{H}^x_4[p]\}$
		$\mathit{DefRepSet} = \{\mathcal{H}^x_1[p]\}$
			`p := new Type1`
			`q := new Type1`
			`. . .`
			$A\_temp_{\mathcal{V}(p)}$ `:= ...`
			`p.x :=` $A\_temp_{\mathcal{V}(p)}$
			`q.x := ...`
			`... :=` $A\_temp_{\mathcal{V}(p)}$

Fig. 8. Trace of load elimination algorithm from figure 4 for program in figure 2(a)

Index propagation is a dataflow problem, which computes a lattice value $\mathcal{L}(\mathcal{H})$ for each heap variable $\mathcal{H}$ in the Array SSA form. This lattice value $\mathcal{L}(\mathcal{H})$ is a set of value number vectors $\{\boldsymbol{i_1}, \dots\}$, such that a load of $\mathcal{H}[\boldsymbol{i}]$ is available (previously stored in a register) if $\mathcal{V}(\boldsymbol{i}) \in \mathcal{L}(\mathcal{H})$. Figures 5, 6 and 7 give the lattice computations which define the index propagation solution. The notation $\textsc{update}(\boldsymbol{i'}, \langle \boldsymbol{i_1}, \dots \rangle)$ used in the middle cell in figure 5 denotes a special update of the list $\mathcal{L}(A_0) = \langle \boldsymbol{i_1}, \dots \rangle$ with respect to index $\boldsymbol{i'}$. $\textsc{update}$ involves four steps:

1. Compute the list $T = \{ \ \boldsymbol{i_j} \mid \boldsymbol{i_j} \in \mathcal{L}(A_0) \text{ and } \mathcal{DD}(\boldsymbol{i'}, \boldsymbol{i_j}) = \textit{true} \ \}$. List T contains only those indices from $\mathcal{L}(A_0)$ that are *definitely different* from $\boldsymbol{i'}$.
2. Insert $\boldsymbol{i'}$ into T to obtain a new list, I.
3. If the size of list I exceeds the threshold size Z, then one of the indices in I is dropped from the output list so as to satisfy the size constraint. (Since the size of $\mathcal{L}(A_0)$ must have been $\leq Z$, it is sufficient to drop only one index to satisfy the size constraint.)
4. Return I as the value of $\textsc{update}(\boldsymbol{i'}, \langle \boldsymbol{i_1}, \dots \rangle)$.

After index propagation, the algorithm selects an array use (load), $A_j[\boldsymbol{x}]$, for scalar replacement if and only if index propagation determines that an index with value number $\mathcal{V}(\boldsymbol{x})$ is available at the def of A_j. If so, the use is included in *UseRepSet* , the set of uses selected for scalar replacement. Finally, an array def, $A_i[\boldsymbol{k}]$, is selected for scalar replacement if and only if some use $A_j[\boldsymbol{x}]$ was placed in *UseRepSet* such that $\mathcal{V}(\boldsymbol{x}) = \mathcal{V}(\boldsymbol{k})$. All such defs are included in *DefRepSet* , the set of defs selected for scalar replacement.

Figure 8 illustrates a trace of this load elimination algorithm for the example program in figure 2(a). Figure 8(a) shows the partial Array SSA form computed for this example program. The results of index propagation are shown in figure 8(b). These results depend on definitely-different analysis establishing that $\mathcal{V}(p) \neq \mathcal{V}(q)$ by using allocation site information as described in Section 2.2. Figure 8(c) shows the scalar replacement actions derived from the results of index propagation, and Figure 8(d) shows the transformed code after performing these scalar replacement actions. The load of `p.x` has thus been eliminated in the transformed code, and replaced by a use of the scalar temporary, $A\_temp_{\mathcal{V}(p)}$.

We now present a brief complexity analysis of the redundant load elimination algorithm in Figure 4. Note that index propagation can be performed separately for each heap array. Let k be the maximum number of defs and uses for a single heap array. Therefore, the number of $d\phi$ and $u\phi$ functions created for a single heap array will be $O(k)$. Based on past empirical measurements for scalar SSA form [10], we can expect that the number of control ϕ functions for a single heap array will also be $O(k)$ in practice (since there are $O(k)$ names created for a heap array). Recall that the maximum size of a lattice value list, as well as the maximum height of the lattice, is a compiler-defined constant, Z. Therefore, the worst case execution-time complexity for index propagation of a *single heap array* is $O(k \times Z^2)$.

To complete the complexity analysis, we define a size metric for each method, $S = \max(\#\text{ instrs in method}, k \times (\#\text{ call instrs in method}))$. The first term (#

instrs in method) usually dominates the max function in practice. Therefore, the worst-case complexity for index propagation for all heap arrays is

$$\sum_{\text{heap array } A} O(k_A \times Z^2) = O(S \times Z^2),$$

since $\sum_A k_A$ must be $O(S)$. Hence the execution time is a linear with a Z^2 factor. As mentioned earlier, the value of Z can be adjusted to trade off precision and overhead. For the greatest precision, we can set $Z = O(k)$, which yields a worst-case $O(S \times k^2)$ algorithm. In practice, k is usually small resulting in linear execution time. This is the setting used to obtain the experimental results reported in Section 4.

We conclude this section with a brief discussion of the impact of the Java Memory Model (JMM). It has been observed that redundant load elimination can be an illegal transformation for multithreaded programs written for a memory model, such as the JMM, that includes the memory coherence assumption [20]. (This observation does not apply to single-threaded programs.) However, it is possible that the Java memory model will be revised in the future, and that the new version will not require memory coherence [19]. However, if necessary, our algorithms can be modified to obey memory coherence by simply treating each $u\phi$ function as a $d\phi$ function *i.e.,* by treating each array use as an array def. Our implementation supports these semantics with a command-line option. As in interesting side note, we observed that changing the data-flow equations to support the strict memory model involved changing fewer than ten lines of code.

3.2 Dead Store Elimination

In this section, we show how our Array SSA framework can be used to identify redundant (dead) stores of array elements. Dead store elimination is related to load elimination, because scalar replacement can convert non-redundant stores into redundant stores. For example, consider the program in Figure 2(a). If it contained an additional store of `p.x` at the bottom, the first store of `p.x` will become redundant after scalar replacement. The program after scalar replacement will then be similar to the program shown in Figure 2(c) as an example of dead store elimination.

Our algorithm for dead store elimination is based on a backward propagation of *DEAD* sets. As in load elimination, the propagation is *sparse i.e.,* it goes through ϕ nodes in the Array SSA form rather than basic blocks in a control flow graph. However, $u\phi$ functions are not used in dead store elimination, since the ordering of uses is not relevant to identifying a dead store. Without $u\phi$ functions, it is possible for multiple uses to access the same heap array name. Hence, we use the notation $\langle A, s\rangle$ to refer to a specific use of heap array A in statement s.

Consider an augmenting def A_i, a source or augmenting use $\langle A_j, s\rangle$, and a source def A_k in Array SSA form. We define the following four sets:

$$DEAD_{def}(A_i) = \{\mathcal{V}(x) | \text{element } x \text{ of array } A \text{ is dead at augmenting def } A_i\}$$
$$DEAD_{use}(\langle A_j, s\rangle) = \{\ \mathcal{V}(x)\ |\ \text{element } x \text{ of array } A \text{ is dead at source use of } A_j \text{ in statement } s\}$$
$$KILL(A_k) = \{\ \mathcal{V}(x)\ |\ \text{element } x \text{ of array } A \text{ is killed by source def of } A_k\}$$
$$LIVE(A_i) = \{\ \mathcal{V}(x)\ |\ \exists \text{ a source use } A_i[x] \text{ of augmenting def } A_i\ \}$$

The *KILL* and *LIVE* sets are local sets; *i.e.,* they can be computed immediately without propagation of data flow information. If A_i "escapes" from the procedure (*i.e.,* definition A_i is exposed on procedure exit), then we must conservatively set $LIVE(A_i) = \mathcal{U}^A_{ind}$, the universal set of index value numbers for array A. Note that in Java, every instruction that can potentially throw an exception must be treated as a procedure exit, although this property can be relaxed with some interprocedural analysis.

1. **Propagation from the LHS to the RHS of a control ϕ:**
Consider an augmenting statement s of the form, $A_2 := \phi(A_1, A_0)$ involving a control ϕ.
In this case, the uses, $\langle A_1, s\rangle$ and $\langle A_0, s\rangle$, must both come from augmenting defs, and the propagation of $DEAD_{def}(A_2)$ to the RHS is a simple copy *i.e.,* $DEAD_{use}(\langle A_1, s\rangle) = DEAD_{def}(A_2)$ and $DEAD_{use}(\langle A_0, s\rangle) = DEAD_{def}(A_2)$.
2. **Propagation from the LHS to the RHS of a definition ϕ:**
Consider a $d\phi$ statement s of the form $A_2 := d\phi(A_1, A_0)$. In this case use $\langle A_1, s\rangle$ must come from a source definition, and use $\langle A_0, s\rangle$ must come from an augmenting definition. The propagation of $DEAD_{def}(A_2)$ and $KILL(A_1)$ to $DEAD_{use}(\langle A_0, s\rangle)$ is given by the equation, $DEAD_{use}(\langle A_0, s\rangle) = KILL(A_1) \cup DEAD_{def}(A_2)$.
3. **Propagation to the LHS of a ϕ statement from uses in other statements:**
Consider a definition or control ϕ statement of the form $A_i := \phi(\ldots)$. The value of $DEAD_{def}(A_i)$ is obtained by intersecting the $DEAD_{use}$ sets of all uses of A_i, and subtracting out all value numbers that are not definitely different from every element of $LIVE(A_i)$. This set is specified by the following equation:

$$DEAD_{def}(A_i) = \left(\bigcap_{s \text{ is a } \phi \text{ use of } A_i} DEAD_{use}(\langle A_i, s\rangle) \right) - \{v | \exists w \in LIVE(A_i) s.t. \neg DD(v, w)\}$$

Fig. 9. Data flow equations for $DEAD_{def}$ and $DEAD_{use}$ sets

The data flow equations used to compute the $DEAD_{def}$ and $DEAD_{use}$ sets are given in Figure 9. The goal of our analysis is to find the maximal

$DEAD_{def}$ and $DEAD_{use}$ sets that satisfy these equations. Hence our algorithm will initialize each $DEAD_{def}$ and $DEAD_{use}$ set to $= \mathcal{U}^{A}_{ind}$ (for renamed arrays derived from original array A), and then iterate on the equations till a fixpoint is obtained. After $DEAD$ sets have been computed, we can determine if a source definition is redundant quite simply as follows. Consider a source definition, $A_1[j] := \ldots$, followed by a definition ϕ statement, $A_2 := d\phi(A_1, A_0)$. Then, if $\mathcal{V}(j) \in DEAD(A_2)$, then def (store) A_1 is redundant and can be eliminated.

As in the index propagation analysis in Section 3.1, the worst-case execution-time complexity for dead store elimination is $O(S \times k^2)$, where S is the size of the input method and k is an upper bound on the number of defs and uses for a single heap array. In practice, k is usually small resulting in linear execution time.

4 Experimental Results

We present an experimental evaluation of the scalar replacement algorithms using the Jalapeño optimizing compiler [3]. The performance results in this section were obtained on an IBM F50 Model 7025 system with four 166MHz PPC604e processors running AIX v4.3. The system has 1GB of main memory. Each processor has split 32KB first-level instruction and data caches and a 256KB second-level cache.

The Jalapeño system is continually under development; the results in this section use the Jalapeño system as of April 5, 2000. For these experiments, the Jalapeño optimizing compiler performed a basic set of standard optimizations including copy propagation, type propagation, null check elimination, constant folding, devirtualization, local common subexpression elimination, load/store elimination, dead code elimination, and linear scan register allocation. Previous work [3] has demonstrated that Jalapeño performance with these optimizations is roughly equivalent to that of the industry-leading IBM product JVM and JIT. The runs use Jalapeño‘s non-generational copying garbage collector with 300MB of heap (which is shared by the application and by all components of the Jalapeño JVM).

Our preliminary implementation has several limitations. Our current implementation does not eliminate the null-pointer and array-bounds checks for redundant loads. We do not use any interprocedural summary information, as the Jalapeño optimizing compiler assumes on “open-world” due to dynamic class loading. We do not perform any loop-invariant code motion or partial redundancy elimination to help eliminate redundant loads in loops. Most importantly, the Jalapeño optimizing compiler still lacks many classical scalar optimizations, which are especially important to eliminate the register copies and reduce register pressure introduced by scalar replacement. For these reasons, these experimental results should be considered a lower bound on the potential gains due to scalar replacement, and we expect the results to improve as Jalapeño matures.

Note that since the entire Jalapeño JVM is implemented in Java, the optimizing compiler compiles not only application code and library code, but also VM code. The results in this section thus reflect the performance of the entire body of Java code which runs an application, which includes VM code and libraries. Furthermore, the compiler inlines code across these boundaries.

For our experiments, we use the seven codes from the SPECjvm98 suite [9], and the Symantec suite of compiler microbenchmarks. For the SPEC codes, we use the medium-size (-s10) inputs. Note that this methodology does *not* follow the official SPEC run rules, and these results are not comparable with any official SPEC results. The focus of our measurements was on obtaining dynamic counts of memory operations. When we report timing information, we report the best wall-clock time from three runs.

Program	**getfield**	**putfield**	**getstatic**	**put-static**	**aload**	**astore**	**Total**
compress	171158111	33762291	4090184	377	39946890	19386949	268344802
jess	17477337	372777	109024	27079	7910971	60241	25957429
db	2952234	166079	88134	35360	2135244	428809	5805860
mpegaudio	81362707	13571793	18414632	3511	155893220	25893308	295139171
jack	9117580	2847032	226457	171130	1005661	860617	14228477
javac	5363477	1797152	188401	3421	449841	223629	8025921
mtrt	26474627	4788579	53134	1927	8237230	800812	40356309
symantec	28553709	15211818	41	0	303340062	123075060	470180690

Table 1. Dynamic counts of memory operations, without scalar replacement.

We instrumented the compiled code to count each of the six types of Java memory operations eligible for optimization by our scalar replacement algorithms: `getfield, putfield, getstatic, putstatic, aload` and `astore`. Table 1 shows the dynamic count of each operation during a sample run of each program.

Program	**getfield**	**putfield**	**getstatic**	**putstatic**	**aload**	**astore**	**Total**
compress	25.9%	0.0%	0.0%	0.0%	0.0%	0.0%	16.5%
jess	1.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%
db	21.8%	0.0%	0.0%	0.0%	0.0%	0.0%	11.1%
mpegaudio	57.1%	9.0%	0.0%	0.0%	20.3%	10.6%	27.8%
jack	15.2%	0.9%	0.1%	0.0%	0.0%	0.0%	9.9%
javac	3.2%	0.0%	0.0%	0.0%	0.1%	0.0%	2.2%
mtrt	1.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.7%
symantec	7.9%	3.8%	0.0%	0.0%	33.2%	0.4%	22.1%

Table 2. Percentage of (dynamic) memory operations eliminated.

Table 2 shows the percentage of each type of memory operation eliminated by our scalar replacement algorithms. The table shows that overall, the algorithms eliminate between 0.7% and 27.8% of the loads and stores. The table shows that redundant load elimination eliminates many more operations than dead store elimination. On two codes, (mpegaudio and symantec), elimination of loads from arrays play a significant role. On the others, the algorithm eliminates mostly getfields. Interestingly, the algorithms are mostly ineffective at eliminating references to static variables; however, table 1 shows that these references are relatively infrequent.

Program	**Time (no scalar replacement)**	**Time (scalar replacement)**	**Speedup**
compress	5.75	5.32	1.08
jess	1.80	1.80	1.00
db	1.98	1.97	1.00
mpegaudio	7.25	6.59	1.10
jack	8.13	8.12	1.00
javac	2.61	2.60	1.00
mtrt	3.07	3.05	1.00
symantec	16.22	15.46	1.05

Table 3. Speedup due to scalar replacement.

Table 3 shows the improvement in running time due to the scalar replacement algorithm. The results show that the scalar replacement transformations give speedups of at least 1.05× on each of the three codes where the optimizations were most effective. `Mpegaudio` shows the greatest improvement with a speedup of 1.1×.

We conclude this section with a brief discussion of the impact of scalar replacement on register allocation. It has been observed in the past that scalar replacement can increase register pressure [5]. For example, the scalar replacement transformations shown in Figure 2(a) and Figure 2(b) eliminate load instructions at the expense of introducing temporaries with long live ranges. In our initial experiments, this extra register pressure resulted in performance degradations for some cases. We addressed the problem by introducing heuristics for live range splitting into our register allocator, which solved the problem.

5 Related Work

Past work on analysis and optimization of array and object references can be classified into three categories — analysis of array references in scientific programs written in languages with named arrays such as Fortran, analysis of pointer references in weakly typed languages such as C, and analysis of object references in strongly typed languages such as Modula-3 and Java. Our research builds on past work in the third category.

The analysis framework based on heap arrays reported in this paper can be viewed as a flow-sensitive extension of *type-based alias analysis* as in [11]. Three different versions of type-based alias analysis were reported in [11] — *TypeDecl* (based on declared types of object references), *FieldTypeDecl* (based on type declarations of fields) and *SMTypeRefs* (based on an inspection of assignment statements in the entire program). All three versions are flow-insensitive. The disambiguation provided by heap arrays in our approach is comparable to the disambiguation provided by *FieldTypeDecl* analysis. However, the use of value numbering and Array SSA form in our approach results in flow-sensitive analyses of array and object references that are more general than the three versions of type-based alias analysis in [11]. For example, none of the three versions would disambiguate references `p.x` and `q.x` in the example discussed earlier in Figure 2(a).

In the remainder of this section, we briefly compare our approach with relevant past work in the first two categories of analysis and optimization of array and object references.

The first category is analysis and optimization of array references in scientific programs. The early algorithms for scalar replacement (*e.g.,* [4]) were based on data dependence analysis and limited their attention to loops with restricted control flow. More recent algorithms for scalar replacement (*e.g.,* [5,2]) use analyses based on PRE (partial redundancy elimination) as an extension to data dependence analysis. However, all these past algorithms focused on accesses to elements of named arrays, as in Fortran, and did not consider the possibility of pointer-induced aliasing of arrays. Hence, these algorithms are not applicable to programming languages (such as C and Java) where arrays might themselves be aliased.

The second category is analysis and optimization of pointer references in weakly typed languages such as C. Analysis of such programs is a major challenge because the underlying language semantics forces the default assumptions to be highly conservative. It is usually necessary to perform a complex points-to analysis before pointer references can be classified as *stack*-directed or *heap-directed* and any effective optimization can be performed [12]. To address this challenge, there has been a large body of research on flow-sensitive pointer-induced alias analysis in weakly typed languages *e.g.,* [17,6,13,16,7]. However, these algorithms are too complex for use in efficient analysis of strongly typed languages, compared to the algorithms presented in this paper. Specifically, our algorithms analyze object references in the same SSA framework that has been used in the past for efficient scalar analysis. The fact that our approach uses global value numbering in SSA form (rather than pointer tracking) to determine if two pointers are the same or different leads to significant improvements in time and space efficiency. The efficiency arises because SSA generates a single value partition whereas pointer tracking leads to a different connection graph at different program points.

6 Conclusions and Future Work

In this paper, we presented a unified framework to analyze object-field and array-element references for programs written in strongly-typed languages such as Java and Modula-3. Our solution incorporates a novel approach for modeling object references as heap arrays, and on the use of global value numbering and allocation site information to determine if two object references are known to be same or different. We presented new algorithms to identify fully redundant loads and dead stores, based on sparse propagation in an extended Array SSA form. Our preliminary experimental results show that the (dynamic) number of memory operations arising from array-element and object-field accesses can be reduced by up to 28%, resulting in execution time speedups of up to 1.1×.

In the near future, we plan to use our extended Array SSA compiler infrastructure to extend other classical scalar analyses to deal with memory accesses. An interesting direction for longer-term research is to extend SSA-based value numbering (and the accompanying $\mathcal{DS}$ and $\mathcal{DD}$ analyses) to include the effect of array/object memory operations by using the Array SSA analysis framework. This extension will enable more precise analysis of nested object references of the form `a.b.c`, or equivalently, indirect array references of the form `a[b[i]]`. Eventually, our goal is to combine conditional constant and type propagation, value numbering, PRE, and scalar replacement analyses with a single framework that can analyze memory operations as effectively as scalar operations.

Acknowledgments

We thank David Grove, Michael Hind, Harini Srinivasan, Mark Wegman and Bill Thies for their comments and suggestions. Thanks also to the entire Jalapeño team for their contribution to the infrastructure used for the experimental results reported in this paper.

References

1. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 1–11, January 1988. San Diego, CA. 156, 160
2. R. Bodik and R. Gupta. Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures. *Lecture Notes in Computer Science*, (1033):1–15, 1995. Proceedings of Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio, August 1995. 162, 171
3. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999. 156, 168

4. David Callahan, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, pages 53–65, June 1990. 162, 171
5. Steve Carr and Ken Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software—Practice and Experience*, (1):51–77, January 1994. 162, 170, 171
6. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, 25(6):296–310, June 1990. 156, 171
7. Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993. 156, 171
8. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991. 158
9. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998. 169
10. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. 158, 160, 165
11. Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, May 1998. 156, 157, 171
12. Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121–133, January 1998. 156, 171
13. Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992. 156, 171
14. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelization. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1998. 155, 156, 157, 158
15. Kathleen Knobe and Vivek Sarkar. Conditional constant propagation of scalar and array references using array SSA form. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 33–56. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*. 155, 156, 158
16. William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, May 1993. 156, 171
17. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 23(7):21–34, July 1988. 156, 171
18. Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997. 160

19. William Pugh. A new memory model for Java. Note sent to the JavaMemoryModel mailing list, http://www.cs.umd.edu/ pugh/java/memoryModel, October 22, 1999. 166
20. William Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, June 1999. 166
21. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 12–27, January 1988. San Diego, CA. 160
22. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing. 156