RC22240 (W0108-036) August 24, 2001
Computer Science

IBM Research Report

Points-to Analysis for Java with Applications for Loop Optimizations

Peng Wu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Paul Feautrier
A3 Project
INRIA Rocquencourt
78153 Le Chesnay, France

David Padua, Zehra Sura
Department of Computer Science
University of Illinois
Urbana, IL 61801

Points-to Analysis for Java with Applications to Loop Optimizations

Peng Wu[†] Paul Feautrier[‡] David Padua[§] Zehra Sura[§]

[†] IBM T.J. Watson Research Center
P.O.Box 218
Yorktown Heights, NY 10598

[‡] A3 Project
INRIA Rocquencourt
78153 Le Chesnay, France

[§] Department of Computer Science
University of Illinois
Urbana, IL 61801

pengwu@us.ibm.com

paul.feautrier@inria.fr

{padua,zsura}@cs.uiuc.edu

ABSTRACT

In this paper, we present a pointer analysis that aims to enable the following optimizations for Java: loop-based dependence analysis in the presence of pointers and the identification of exception-free regions.

The analysis computes a property called *element-wise points-to* (ewpt) *mapping*. An ewpt mapping summarizes precise points-to information for all *instances* of a pointer or all *elements* of an array of pointer type. This is done with the help of a single, compact representation. Such instance-wise and element-wise information is especially important for loop-based dependence analysis and for a language where multi-dimensional arrays are implemented as arrays of pointers. We describe an iterative algorithm to compute ewpt mappings. We also present techniques to kill objects from ewpt mappings for destructive updates.

The points-to algorithm was implemented and evaluated on a set of benchmark programs. We demonstrate that ewpt information can significantly improve the precision of dependence analysis. In many cases, the dependence analysis reports no false dependences due to array accesses. In addition, all redundant null-pointer checks on array element accesses can be removed.

1. INTRODUCTION

If Java is to be used for high performance computing, we must enable classical loop optimizations for it. Due to the rigid exception semantics of Java and its pervasive use of pointers, two problems need to be solved before loops can be optimized: finding accurate loop-based dependences in the presence of pointers and identifying large exception-free regions.

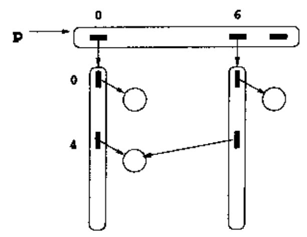

Figure 1: 2-dimensional Java array

This paper presents a pointer analysis that can be used to solve these two problems. The main difficulties that are faced by this pointer analysis are as follows:

- Java objects are heap allocated and are accessed through pointers only. The presence of many heap and pointer accesses demands an aggressive pointer analysis.
- A loop-based dependence analysis not only relates pointers from different program points, but also pointers from different loop iterations.

  Consider the following code fragment:

```
1    for (i = 0; i < m; i++) {
2      p = new Object();
3      p.x = ···;
4    }
```

  Is statement 3 output-dependent over loop i? This depends on whether p at different iterations of i may point to a common object. In this case, the pointer analysis needs to provide points-to information for each instance of p.

  Now, consider a similar loop with array accesses:

```
1    for (i = 0; i < n; i++)
2      for (j = 0; j < m; j++) {
3        p[i][j] = ...;
4      }
```

  Is statement 3 output-dependent over loop i? Since Java multi-dimensional arrays are trees of one-dimensional arrays (see Fig. 1), this would depend on whether p[i] for different values of i may be aliased. Then, the pointer analysis needs to provide points-to information for each element of p.

- Restricted by the exception semantics of Java, an instruction cannot be moved across a point where exceptions may occur. Therefore, it is important for high-level optimizers to identify regions that are free of exceptions. One particular problem is to eliminate redundant null-pointer checks for heap-residing pointers (e.g., p[i]).[1]

This paper presents a store-based points-to analysis to address the problems discussed above. The analysis treats all references uniformly as array areferences of some dimensions. It computes a property called *element-wise points-to mapping* that summarizes what heap objects are attached at each element of an array. This is compactly represented by a mapping from array elements to their points-to sets. An iterative fixed point algorithm is proposed to compute ewpt mappings. The algorithm is implemented and evaluated on a set of benchmark programs. We demonstrate that our pointer analysis can significantly improve the precision of dependence analysis. In most cases, the dependence analysis reports no false dependences due to array accesses. In addition, all redundant null-checks on array accesses can be removed.

Contributions. In summary, the paper makes the following contributions:

- It proposes a new points-to abstraction, *element-wise points-to mapping*, that summarizes points-to information for pointer instances and array element pointers in a single form.
- It introduces a scheme to name heap objects by allocation statement instances. It also provides the necessary techniques (i.e., aging and binding) to enable such a naming scheme.
- It proposes a store-based pointer analysis that is suitable for loop-based dependence tests and analyzing pointer arrays.
- It presents a technique to kill objects from ewpt mappings for destructive updates. This technique can help to identify redundant null-pointer checks on heap-residing pointers.

Organization. The rest of the paper is organized as follows. Section 2 gives an overview of the analysis. Section 3 describes our assumptions on the use of some language features and introduces basic notations. Section 4 presents the transfer functions and Section 5 gives the iterative algorithm. Section 6 describes how to kill objects from ewpt mappings for destructive updates. Section 7 discusses the applications of the analysis and experimental results. Section 8 outlines some related work and Section 9 concludes.

[1]Pointers stored on the heap are heap-residing pointers; those stored on the stack are stack-residing pointers.

2. OVERVIEW OF THE METHOD

The analysis is based on a points-to abstraction called the element-wise points-to mapping. An ewpt mapping summarizes the objects pointed to by individual elements of an array in a single form. To give an example of ewpt mappings, consider the loop in Fig. 2 and an arbitrary array element a[x]. The objects pointed to by a[x] at program point 4 in iteration i (denoted as $4(i)$) can be represented as

$$\text{PointsTo}(a[x]) = \{\texttt{null}, \langle N_t[x], x \leq i\rangle\},$$

where

- $N_t[x]$ denotes the object allocated by statement instance t at iteration x.
- $\langle N_t[x], x \leq i\rangle$ represents a step function that evaluates to $N_t[i]$ when $x \leq i$ and is undefined at other points.

Treating x as a parameter, $\text{PointsTo}(a[x])$ is in fact an ewpt mapping of a.

Table 1 illustrates how to compute ewpt mappings using the same example. In Table 1, objects names are separated by ";". We give a detailed explanation of the algorithm below.

Point 1, a = new Complex[n]. a points to the object allocated by s (i.e., N_s). Elements of N_s are initialized to null, hence, a[x] points to null.

Point 2, for (...), first iteration.

Point 3, b = new Complex(). b points to the object allocated by statement instance $\mathtt{t}(i)$ (i.e., $N_t[i]$).

Point 4, a[i] = b. The statement makes a[i] point to $N_t[i]$, while other array elements still point to null. Hence,

$$\text{PointsTo}(a[x]) = \texttt{null}; \langle N_t[i], x = i\rangle$$

Then, the above points-to set is converted to an equivalent form

$$\text{PointsTo}(a[x]) = \texttt{null}; \langle N_t[x], x = i\rangle.$$

This transformation is necessary to preserve the precision of the mapping during aging (see Point 2, second iteration). This is because at Point 2 the second iteration, $\langle N_t[i], x = i\rangle$ and $\langle N_t[x], x = i\rangle$ will be aged to $\langle N_t[i^-], x = i^-\rangle$ and $\langle N_t[x], x = i^-\rangle$, respectively. It is obvious that the latter name (with minimum occurrences of i) preserves better precision.

Point 2, for (...), second iteration. Since i is advanced, occurrences of i in the mappings are replaced by i^-, which denotes the value of any iteration before i. This transformation is called *aging*.

Point 3, b = new Complex(). b points to $N_t[i]$. Note that, object $N_t[i]$ allocated at the previous iteration became $N_t[i^-]$ after aging.

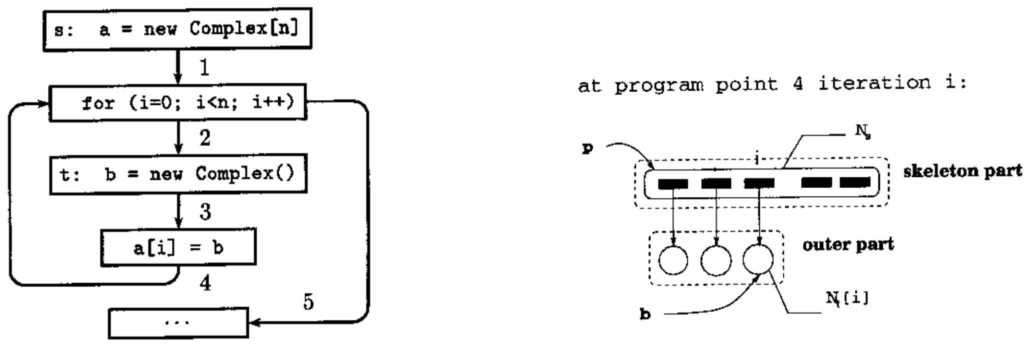

Figure 2: Constructing 1-dimensional array of `Complex` Objects

Point 4, **`a[i] = b`.** The statement makes `a[i]` point to $N_t[i]$, and leaves other elements of the array unchanged. Hence,

$$\begin{aligned}&\text{PointsTo}(a[x])\\ &\quad = \texttt{null}; \langle N_3[x], x = i^- \rangle; \langle N_3[x], x = i \rangle\\ &\quad = \texttt{null}; \langle N_3[x], x \in \{i, i^-\}\rangle.\end{aligned}$$

Point 2, 3, **and** 4, **third iteration.** The fixed point is reached.

Point 5**.** Mappings at point 4 are merged with those at point 1. Since `i` is not live after the loop, occurrences of `i` are substituted by its last value $(n-1)$. This transformation is called *binding*.

This example reveals some interesting features of the analysis.

- Heap objects are named by allocation statement instances. This naming scheme automatically encodes instance-wise information into the mapping. Names such as $\langle N_t[x], x \leq i\rangle$ are generated to represent the fact that an object is attached at a particular region of an array.
- Loop counters may occur in the ewpt mappings and are handled by aging and binding. Aging may introduce approximation but is needed for the convergence of the algorithm. It is the widening operator of the iterative algorithm.
- The analysis performs no kills (i.e., removing objects from ewpt mappings) for heap assignments (a.k.a. destructive updates). As a result, `null` is always present in PointsTo($a[x]$). A technique to enable kills for destructive updates will be presented in Section 6.

3. BASIC DEFINITION

This section describes our assumptions on the use of some language features and introduces basic notations.

3.1 Input Program Format

We assume that the source program conforms to the following constraints:

Stack Assignments

```
1.  p = null
2.  p = new Cls ()
3.  p = new Cls[m][]···[]
4.  p = q
5.  p = q.a
6.  p = q[e]
```

Heap Assignments

```
7.  p.a = q
8.  p[e] = q
```

Figure 3: Reference assignments

- Control statements are loops and conditionals. We assume that `while` loops have a loop counter, even if we have to create one before the analysis. Loop counters are initialized to 0 and have a step of 1.

  These conventions allow the use of notation, `s`(i), for statement instances, where s is a statement label and i is the iteration vector of s.
- A source program contains no `try` and `catch` constructs. Similarly, multithreaded programs or synchronization statements are not considered.
- Without loss of generality, only reference assignments of the form given in Fig. 3 are considered. In fact, all reference accesses supported by Java byte-code comply with these forms. More complicated accesses can be converted into these forms by introducing temporaries.
- Objects of type `Cls` are created by either invoking `new Cls[`m`][]`$\cdots$`[]` for array objects, or `new Cls()` for non-array objects. Object creation of the form, `new Cls[`m_1`]`$\cdots$`[`m_k`]`, is replaced by a loop nest of depth $(k-1)$ that creates the implied arrays of arrays.
- The current algorithm does not consider method invocation. Method calls are inlined before being processed by the analysis.

Iter.	PrgmPts	PointsTo(a)	PointsTo(b)	PointsTo($a[x]$)
	1	N_s	`null`	`null`
1	2	N_s	`null`	`null`
	3	N_s	$N_t[i]$	`null`
	4	N_s	$N_t[i]$	`null`; $\langle N_t[i], x = i\rangle \Rightarrow$ `null`; $\langle N_t[x], x = i\rangle$
2	2	N_s	$N_t[i^-]$	`null`; $\langle N_t[x], x = i^-\rangle$
	3	N_s	$N_t[i]$	`null`; $\langle N_t[x], x = i^-\rangle$
	4	N_s	$N_t[i]$	`null`; $\langle N_t[x], x \in \{i^-, i\}\rangle$
3	2	N_s	$N_t[i^-]$	`null`; $\langle N_t[x], x = i^-\rangle$
	3	N_s	$N_t[i]$	`null`; $\langle N_t[x], x = i^-\rangle$
	4	N_s	$N_t[i]$	`null`; $\langle N_t[x], x \in \{i^-, i\}\rangle$
	5	N_s	`null`; $N_t[n-1]$	`null`; $\langle N_t[x], 0 \leq x \leq n-1\rangle$

Table 1: The step-by-step output of the analysis

3.2 Location, Object and Reference

A *location* is a place in memory where a primitive data type is stored. An *object* is an aggregation of locations and is named by a `new` statement instance. More precisely,

- $N_s[i]$ denotes the object created by statement instance $\mathtt{s}(i)$;
- $N_s[i].a$ denotes the location occupied by field a of object $N_s[i]$.

A *reference* is a strongly typed pointer to an object. It never points inside an object. The *dimension* of a reference is the number of "[ ]" in its type declaration. The dimension of a non-array reference is 0.

Consider a reference p of n dimensions. $\textsc{Reach}(p)$ denotes all objects that can be reached from p. Objects in $\textsc{Reach}(p)$ are further divided into two parts:

Skeleton part contains all objects pointed to by p with less than n subscripts, i.e., `p[`x_1`]`$\cdots$`[`x_k`]` where $k < n$ (see Fig. 2).

Outer part contains all objects that can be *reached* from p with n subscripts, i.e., `p[`x_1`]`$\cdots$`[`x_n`]`.

Intuitively, the skeleton part of p captures the backbone of the array referenced by p, whereas the outer part captures the leaf objects. Objects in the skeleton part are necessarily array objects.

3.3 Element-wise Points-to Mapping

Given a reference p of n dimensions, the following *element-wise points-to mappings* are defined for p:

- We define mapping p_k for $0 \leq k < n$. The domain of p_k is a set of k-tuples that contains all subscripts of p of length k. The value of $p_k(x)$, where x is a k-tuple, is the set of all objects that $p[x]$ may point to. Due to Java's strong typing, these objects must have the same type as $p[x]$, and they necessarily belong to the skeleton part of p.
- If the last dimension of p is of reference type, we define an additional mapping p_n. For any n-tuple, x, the value of $p_n(x)$ is the set of all objects that may be *reached* through $p[x]$. These objects can be of any type and dimension (since they may not be directly pointed to by $p[x]$), and they necessarily belong to the outer part of p.

We call p_k the ewpt mapping of p at level k. Consider the sets computed in Table 1. There, PointsTo(a) is a_0, and PointsTo($a[x]$) is a_1.

4. TRANSFER FUNCTIONS

The points-to algorithm views every reference assignment as a transfer function of ewpt mappings. For ease of understanding, this section presents the transfer functions without specifying the representation of ewpt mappings and the implementation of the operations used. A concrete implementation of the transfer functions will be presented in Section 5.

4.1 Stack Assignments

Let s be the current statement and $i_1, \ldots, i_d$ be the counters of the loops surrounding s. Assume that p has ewpt mappings from level 0 to level n. The transfer function of each of the stack assignments is as follows:

- `p = null`: Since accesses attempted via a null pointer generate an exception, $p[x_1]\cdots[x_k]$ refers to nothing, i.e., it maps to the empty set. Hence,

$$\begin{aligned} p_0() &= \{\mathtt{null}\} \\ p_k(x_1, \ldots, x_k) &= \varnothing, 1 \leq k \leq n. \end{aligned} \quad (1)$$

- `p = new Cls()`: `p` is necessarily 0-dimensional. It has only one ewpt mapping, p_0. Hence,

$$p_0() = \{N_s[i_1, \ldots, i_d]\}. \quad (2)$$

- `p = new Cls[m]`$_1$`[ ]`$_2\cdots$`[ ]`$_n$: Elements of a newly allocated array are initialized to `null`. Hence,

$$\begin{aligned} p_0() &= \{N_s[i_1, \ldots, i_d]\} \\ p_1(x_1) &= \{\mathtt{null}\} \\ p_k(x_1, \ldots, x_k) &= \varnothing,\ 2 \leq k \leq n. \end{aligned} \quad (3)$$

- p = q: p[x_1]$\cdots$[x_k] points to the same objects as q[x_1]$\cdots$[x_k] does. Hence,

$$p_k(x_1,\ldots,x_k) = q_k(x_1,\ldots,x_k),\ 0 \le k \le n. \quad (4)$$

- p = q.a: p[x_1]$\cdots$[x_k] points to the same objects as q.a[x_1]$\cdots$[x_k] does. The legality of q.a implies that q is of 0 dimension. Then, $q_0()$ contains all objects that can be reached by q, including those pointed to by q.a. Hence,

$$p_k(x_1,\ldots,x_k) = q_0(),\ 0 \le k \le n. \quad (5)$$

  This transfer function is conservative. Since objects in $p_k(x_1,\ldots,x_k)$ must have the same type as p and must be of $(n-k)$ dimensions, a possible refinement is to filter out objects in q_0 that do not match the type of p_k.

- p = q[e]: The type rules ensure that q has one more dimension than p. Hence,

$$p_k(x_1,\ldots,x_k) = q_{k+1}(e,x_1,\ldots,x_k),\ 0 \le k \le n. \quad (6)$$

4.2 Heap Assignments

Since a heap location may be accessed through different references, one heap assignment may change the ewpt mappings of several references at the same time. For example, consider the assignment in Fig. 4. Since the location of p[e] can be reached from both p and r, after p[e] = q, both p_1 and r_2 will change.

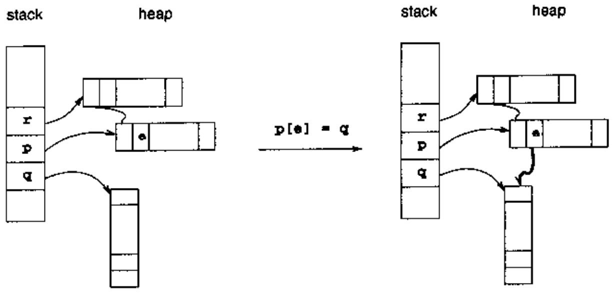

Figure 4: Heap assignments

Statement of the form p.a = q. Consider any reference r that can reach the location of p.a. Suppose that r is of n dimensions. Since p cannot possibly point to an array, if p.a can be reached from r, it must be in the outer part of r (i.e., r_n). It may seem at first that the reachability condition is $r_n(x_1,\ldots,x_n) \cap p_0() \ne \emptyset$. However, if the intersection is null, p.a cannot be reached by r because null.a is not a valid location. Hence, the correct reachability condition is

$$r_n(x_1,\ldots,x_n) \cap p_0() \not\subseteq \{\mathtt{null}\}.$$

The assignment may also remove objects from r_n; however, there is not enough information to find them. Conservatively, we add REACH(q) to all mappings that can reach the location of p.a.

All in all, let r'_n be r_n after the assignment. The transfer function is

$$r'_n(x_1,\ldots,x_n) = r_n(x_1,\ldots,x_n) \cup \{\langle \textsc{Reach}(q), \Sigma\rangle\} \quad (7)$$

where Σ is the system of constraints obtained by imposing the condition

$$r_n(x_1,\ldots,x_n) \cap p_0() \not\subseteq \{\mathtt{null}\}.$$

For example, if Σ is computed by imposing the condition $N_s[x_1,x_2] \cap N_s[n,n] \not\subseteq \{\mathtt{null}\}$, after simplification, we have $\Sigma \equiv \{x_1 = n, x_2 = n\}$.

Statement of the form p[e] = q. Consider any reference r that may reach the location of p[e]. Suppose that r is of n dimensions, p is of m dimensions, then q must be of $(m-1)$ dimensions.

- If the location of p[e] belongs to the skeleton part of r, p must point to one of the subarrays of r, call it r[u_1]$\cdots$[u_h].[2] Then, the following condition must be satisfied:

$$r_h(u_1,\ldots,u_h) \cap p_0() \not\subseteq \{\mathtt{null}\}.$$

  In this case, the effect of p[e] = q is to replace subarrays of r[u_1]$\cdots$[u_h][e] with those of q. That is to replace r[u_1]$\cdots$[u_h][e][x_{h+2}]$\cdots$[x_{h+1+k}] by q[x_{h+2}]$\cdots$[x_{h+1+k}]. All in all, for all k such that $1+h+k \le m$,

$$r'_{1+h+k}(x_1,\ldots,x_{h+1+k}) = r_{1+h+k}(x_1,\ldots,x_{1+h+k}) \cup \{\langle q_k(x_{h+2},\ldots,x_{h+1+k}), \Sigma\rangle\} \quad (8)$$

  where Σ is the system of contraints got by imposing the condition

$$r_h(x_1,\ldots,x_h) \cap p_0() \not\subseteq \{\mathtt{null}\} \wedge x_{h+1} = e.$$

- If the location of p[e] belongs to the outer part of r, REACH(q) is added to all $r_n(x_1,\ldots,x_n)$ that can reach the location of p[e], which is the same as the case of p.a = q. Hence, the transfer function is (7).

4.3 Handling Loop Counters

There are two transformations to be applied on loop counters in ewpt mappings. The first one is aging, which is applied at every program point that follows a back-edge of a loop. It reflects the fact that i has been advanced. Consider a loop counter i. Aging would replace i in the mappings by $i-1$. However, this may generate infinite number of terms, such as i, $i-1$, $i-1-1$, etc., during the iterative process. To limit the number of forms that aging may generate, we introduce a symbol i^- that represents the value of the loop counter in any iteration before i. Then, during aging, occurrences of i in the mappings are replaced by i^-. This may introduce approximation in the algorithm. A better scheme is to

[2]Since the dimension of this subarray (i.e., $n-h$) must be the same as the dimension of p (i.e., m), we can further determine h: $h = n - m$ provided that $n \ge m$.

k-limit the number of forms that a loop variable can be aged into. Readers can refer to [17] for details.

The second transformation is binding, which is applied at any program point that follows a loop exit edge. This is to ensure that ewpt mappings at different program points contain only variables that are in scope at those points. Consider a loop variable i. If the last iteration of i is n, during binding, occurrences of i in the mappings are replaced by n, and occurrences of i^- are replaced by a range $[0, n-1]$. In the case of unknown loop bounds, i and i^- are replaced by $*$ that represents any positive integer value.

5. THE ITERATIVE ALGORITHM

The iterative framework requires transfer functions and a meet operation. The meet operation is set-union. The transfer functions are described in Section 4. This section gives a concrete implementation of these transfer functions by defining the representation of ewpt mappings and providing the operations in the transfer functions based on this representation.

5.1 Symbolic Name

The algorithm operates on tables called ewpt tables. Each entry of an ewpt table contains a set of heap names in the following format:

$$\langle N_s[a_1, \ldots, a_d], x_1 = a_{d+1}, \cdots, x_k = a_{d+k}, \Sigma_A, \Sigma_B \rangle$$

- s is a statement and d is the nesting level of s;
- $x_1, \ldots, x_k$ are the parameters of any ewpt mapping at level k where k is called the *rank* of the heap name;
- $A = \{a_1, \ldots, a_{d+k}\}$ is a set of bound variables;
- B is another set of bound variables that can be renamed at will;
- Σ_A is a system of constraints over A where *at most* one constraint per $a \in A$. Let e be a subscript expression in the program and i be the counter of a loop surrounding s. Constraints in Σ_A take one of the following forms:
  - $a = e[i \leftarrow b]$ where $b \in B$.
  - $a = e[i \leftarrow u]$ where u is the last value of i.
  - $e[i \leftarrow 0] \leq a \leq e[i \leftarrow u-1]$ with the same conventions.

  The notation $e[x \leftarrow y]$ stands for substituting occurrences of x in e by y.
- Σ_B is a system of constraints over B. For each $b \in B$, the constraint is either of the form $b \in i$ or $b \in i^-$. To bound the size of B, redundant variables in B are removed:
  - When both $b \in i$ and $c \in i$ belong to Σ_B, b in Σ_A is replaced by c, and b is removed from B.
  - When b does not occur in Σ_A, it is removed from B.

The above representation is called the *standard format*. Heap names in the standard format are called *symbolic names*.

5.2 Transfer Functions

There is a one-to-one correspondence between any ewpt mapping p_k and ewpt[p,k]. That is symbolic names in ewpt[p,k] represents the value of $p_k(x_1, \ldots, x_k)$. For instance, the following table entry

$$\texttt{ewpt[p,1]} = \{\texttt{null}; \langle N_s[a_1], x_1 = a_1, a_1 \leq n \rangle\}$$

represents the mapping

$$p_1(x) = \{\texttt{null}; \langle N_s[x], x \leq n \rangle\}.$$

Derived directly from (1) - (8), Table 2 gives the transfer functions based on the ewpt table representation where s denotes the current statement and $i_1, \ldots, i_d$ denote the counters of the loops that surround s. Table 2 also uses several operations applied to sets of symbolic names. These operations are defined below. All but the first one are defined on symbolic names, but can be extended to sets in the usual way.

Let A_k be a set of symbolic names and ν_k be a symbolic name. Both are of rank k.

$\nu_k(e)$ binds a subscript expression, e, to the first parameter of ν_k, x_1. The resulting symbolic name is of rank $(k-1)$. Given $\nu_k = \langle N_s[a_1, \ldots, a_d], \Sigma_k \rangle$, this operation eliminates x_1 from ν_k by adding the constraint $x_1 = e$ to Σ_k. The computation is performed in three steps:

1. If the system is unfeasible, it returns $\varnothing$.
2. Else, from the new constraint $x_1 = e$, we deduce $a_{d+1} = e[i \leftarrow b]$. This constraint is added to Σ_A iff a_{d+1} was not constrained in ν_k. If it was constrained, then there are two constraints for a_{d+1}, which is forbidden in the standard format. Therefore, we chose to ignore one of them. This is in fact a widening operation. Since the new constraint always defines a singleton set while the old one may define a range, we chose to replace the old constraint with the new one.
3. Finally, the parameters of ν_k are shifted: $[x_2 \leftarrow x_1, \cdots, x_k \leftarrow x_{k-1}]$. This substitution will generate a symbolic name of rank $(k-1)$.

To give an example, suppose

$$\nu_1 = \langle N_s[a, b], x_1 = a, x_2 = b, a \leq i, b \leq j \},$$

then,

$$\begin{aligned} \nu_1(i+1) &= \varnothing \\ \nu_1(i-1) &= \langle N_s[a, b], x_1 = b, a = i-1, b \leq j \rangle. \end{aligned}$$

This operation is also used to compute the points-to set of an array element from ewpt mappings, for instance, to compute read-sets/write-sets of a reference access for dependence analysis.

`p = null`	$\mathtt{ewpt}[p,0] \leftarrow \{\mathtt{null}\}$ $1 \le k \le n : \mathtt{ewpt}[p,k] \leftarrow \emptyset$
`p = new Cls()`	$\mathtt{ewpt}[p,0] \leftarrow \langle N_s[a_1,\ldots,a_d], a_1 = b_1,\ldots,a_d = b_d, b_1 \in i_1,\ldots,b_d \in i_d\rangle$

`p = new Cls[m]`
`[]...[]`
	$\mathtt{ewpt}[p,0] \leftarrow \langle N_s[a_1,\ldots,a_d], a_1 = b_1,\ldots,a_d = b_d, b_1 \in i_1,\ldots,b_d \in i_d\rangle$
	$\mathtt{ewpt}[p,1] \leftarrow \{\mathtt{null}\}$
	$2 \le k \le n : \mathtt{ewpt}[p,k] \leftarrow \emptyset$

`p = q`	$0 \le k \le n : \mathtt{ewpt}[p,k] \leftarrow \mathtt{ewpt}[q,k]$
`p = q.a`	$0 \le k \le n : \mathtt{ewpt}[p,k] \leftarrow \mathtt{ewpt}[q,0]$
`p = q[e]`	$0 \le k \le n : \mathtt{ewpt}[p,k] \leftarrow \mathtt{ewpt}[q,k+1](e)$
`p.a = q`	$\forall r, n = \mathit{rank}(r)$: $\mathtt{ewpt}[r,n] \leftarrow \mathtt{ewpt}[r,n] \sqcup \langle \textsc{Reach}(q), \mathtt{ewpt}[r,n] \cap \mathtt{ewpt}[p,0] \not\subseteq \{\mathtt{null}\}\rangle$
`p[e] = q`	$\forall r, n = \mathit{rank}(r), m = \mathit{rank}(p)$, if $h = n - m \ge 0$: $\mathtt{ewpt}[r,h+1+k] \leftarrow \mathtt{ewpt}[r,h+1+k] \sqcup \langle \mathtt{ewpt}[q,k](x_{h+2},\ldots,x_{h+1+k}),$ $\quad \mathtt{ewpt}[r,h] \cap \mathtt{ewpt}[p,0] \not\subseteq \{\mathtt{null}\}, x_h = e\rangle$ $\mathtt{ewpt}[r,n] \leftarrow \mathtt{ewpt}[r,n] \sqcup \langle \textsc{Reach}(q), \mathtt{ewpt}[r,n] \cap \mathtt{ewpt}[p,0] \not\subseteq \{\mathtt{null}\}\rangle$

Table 2: Transfer functions of ewpt tables

$\nu_k(x_{m+1},\ldots,x_{m+1+k})$ substitutes x_i by x_{i+m} in ν_k for any $1 \le i \le k$. The resulting symbolic name is of rank $(k+h+1)$. For instance, given

$$v_1 = \langle N_s[a], x_1 = a, a = i\rangle,$$

then,

$$v_1(x_2) = \langle N_s[a], x_2 = a, a = i\rangle.$$

$A_k \sqcup \nu_k$ adds ν_k to set A_k. The resulting set is simplified by removing common names. Furthermore, if a symbolic name subsumes another, the latter is removed from the resulting set. For instance, given

$$\begin{aligned} \nu_1 &= \langle N_s[c], x_1 = c, c < i\rangle \\ A_1 &= \{\langle N_s[a], x_1 = a, a \le i\rangle\}, \end{aligned}$$

then,

$$A_1 \sqcup \nu_1 = \{\langle N_s[d], x_1 = d, d \le i\rangle\}.$$

In this example, the resulting set represents the same mapping as A_1 (after intermediate variables are renamed).

$\nu_k \cap \mu_0 \not\subseteq \{\mathtt{null}\}$ solves a system of constraints over $x_1,\ldots,x_k$: $\nu_k(x_1,\ldots,x_k) \cap \mu_0 \not\subseteq \{\mathtt{null}\}$. Consider

$$\begin{aligned} \nu_k &= \langle N_s[a_1,\ldots,a_d], \Sigma_k\rangle \\ \mu_0 &= \langle N_t[a_1,\ldots,a_d], T_0\rangle. \end{aligned}$$

The operation returns false when $s \ne t$, i.e., objects in ν_k and μ_0 are created by different statements. Otherwise, it returns $\Sigma' = \Sigma_k \cup T_0$. Again, Σ' needs to be simplified: unfeasible constraints are represented by false; and if some variable in A has more than one constraints, the most precise one is kept and the others are discarded.

Consider the following example,

$$\begin{aligned} \nu_2(x_1,x_2) &= \langle N_s[a_1,a_2], x_1 = a_1, \\ &\qquad x_2 = a_2, a_1 \le i, a_2 < j\rangle \\ \mu_0() &= \langle N_s[c_1,c_2], c_1 = i, c_1 = j\rangle \\ \mu_0'() &= \langle N_s[c_1,c_2], c_1 = i, c_2 = *\rangle. \end{aligned}$$

then,

$$\begin{aligned} \nu_2 \cap \mu_0 \not\subseteq \{\mathtt{null}\} &\Rightarrow \mathsf{false} \\ \nu_2 \cap \mu_0' \not\subseteq \{\mathtt{null}\} &\Rightarrow \{x_1 = a_1, x_2 < a_2, \\ &\qquad a_1 = i, a_2 < j\}. \end{aligned}$$

Reach(q) computes all objects that can be reached from reference variable q. It can be computed as the union of all ewpt entries of q, removing constraints over the parameters of those symbolic names.

Finally, the handling of loop counters is explained. Consider a loop counter i with an upper bound u. Aging is performed by replacing i by i^- in all symbolic names. Binding is performed as follows,

- For each $b \in i$ in B, b in Σ_A is replaced by u;
- Consider any b with a constraint $b \in i^-$ in B. If b occurs in any constraint $a = e$ in Σ_A, then $a = e$ is replaced by $e[b \leftarrow 0] \le a \le e[b \leftarrow u-1]$.

It can be checked that operations on symbolic names preserve the standard format. Hence, the number of symbolic names in a given program is finite. This is key for the convergence of the algorithm. Due to space constraint, the convergence proof can be found in [17].

5.3 Cost Analysis

A rough estimate of the complexity of the analysis can be the product of the following factors:

- the number of program points;
- the number of ewpt mappings at each point;
- the number of iterations to reach a fixed point;
- the number of symbolic names in each mapping.

The second factor can be easily computed. Let d_i be the dimension of the i-th reference in the program. The number of ewpt mappings is

$$N_e = \sum_i (d_i + 1).$$

To estimate the iteration count, let us consider aging first. Since each loop counter i in the ewpt mappings is aged to i^- at the end of a loop body, it may take up to 2 iterations to reach a fixed point plus one more iteration to test it. The iteration count also depends on how fast modifications can be propagated. While forward modifications are propagated instantly, it may need several iterations to do backward propagations. Consider the following example:

```
0   for(i=0; i<n; i++) {
1     p = q;
2     q = r;
3     r = new Cls();
4   }
```

3 iterations are needed until the effect of statement 3 is propagated back to p. An upper bound of the number of iterations in a m-nested loop would be m times the number of pointer assignments in the loop body.

We have already proved in [17] that the number of symbolic names is finite. Although likely to be an overestimate, this gives an upper bound on the number of symbolic names in an ewpt mapping. In practice, we can limit the size of symbolic names in an ewpt mapping to reduce the cost of the analysis.

One may wonder about the complexity of the operations on symbolic names. The complexity of these operations can be treated as a constant, although they may be quite expensive in practice. It is worth to mention, however, that symbolic names in ewpt mappings at level 0 are essentially points-to sets. Operations on them are inexpensive set union and intersection. They become expensive only when ewpt mappings at a level greater than 0 (for true array element accesses) are involved.

6. ADDITIONAL KILLING

This section presents a technique to perform kills for heap assignments. There are two reasons why the iterative algorithm performs no kill on heap assignments:

- To kill on heap assignment, one needs *must-alias* information, whereas ewpt mappings capture *may-points-to* information.
- To kill an object from a particular array element may involve restricting the constraint part of symbolic names. Then, aging that replaces a set by its super-set (i.e., replacing $i-1$ by i^-) is not safe any more.

The additional killing is performed after the iterative algorithm is terminated. As opposed to *widening*, this is the *narrowing* part of the points-to analysis. The technique is based on proving the solidness of objects. An object is *solid* if all its fields must not be **null**. Here, we focus on how to remove **null** from ewpt mappings. However, the technique can be easily extended to a general killing scheme. Consider the example in Fig. 2. At program point 5, the following ewpt mappings are computed,

$$\begin{aligned} a_0() &= N_s \\ a_1(x) &= \{\texttt{null}; \langle N_t[x], 0 \le x < n \rangle\}. \end{aligned}$$

Since a points to N_s, $a[x]$ and $N_s.[x]$ ought to point to the same objects. Hence, if object N_s is solid, $a[x]$, for any possible x, must not be **null**. This means that **null** can be removed from a_1. In general, if an ewpt mapping, p_k, contains only solid objects, **null** can be removed from p_{k+1}.

The solidness of an object can be proved in two steps:

Step One Pointer assignments in the program are converted to pseudo assignments. For every statement in form of $l = r$, the following pseudo assignment ($\leftarrow$) is constructed,

$$\text{Location}(l) \leftarrow \text{PointsTo}(r),$$

where Location(l) and PointsTo(r) are computed using the ewpt mappings from the previous iterative algorithm. A pseudo assignment is a *must-assignment* if its left-hand-side is a singleton non-null location.

For example, the following pseudo code is generated from the code in Fig. 2,

```
a ← $N_s$;
for (i = 0; i<n; i++) {
  b ← $N_t$[$i$];
  $N_s$.[$i$] ← $N_t$[$i$];
}
```

Step Two To prove that an object is solid, one considers all pseudo statements that assign to a field of the object. A must-assignment *generates* a non-null field if the right-hand-side (*rhs*) of the assignment contains no **null**. An assignment whose *rhs* contains **null** *kills* a non-null field. For array elements, the Gen and Kill sets are summarized as intervals over loops. Finally, if every field of an object does not point to **null**, the object is solid.

7. APPLICATIONS AND RESULTS

The transfer functions were implemented in Java and **javac** was augmented to drive the fixed point computation. We evaluate the analysis and demonstrate that ewpt mappings can be used to improve dependence test, loop parallelization, and exception optimization.

7.1 The Benchmarks

The experimental results are reported based on running the analysis over eight Java programs, as given in Table 3. All benchmark programs use either multi-dimensional arrays or one dimensional object arrays. Six of them are numerical codes that would benefit from classical loop optimizations. Program **listtbl** is an artificial example that constructs an array of linked lists in a loop, and is included in the benchmark because it shows an interesting pattern of reference assignments.

Since the points-to analysis is intra-procedural, we inlined method calls and commented out system calls that had no side-effects in the programs. Table 3 gives lengths of the programs before and after inlining.[3]

[3]**lufact** is smaller after inlining because of some dead code elimination done during inlining.

Program	Description	Lines (inlined)	Source
listtbl	constructing an array of linked lists	15	-
cmatmul	complex matrix multiplication	47	-
cholesky	cholesky factorization of a matrix	38	IBM
shallow	complex shallow-water simulation	197 (218)	IBM
sor	successive over-relaxation routine	40	Java Grande
lufact	LU factorization routine	287 (153)	Java Grande
moldyn	molecular dynamics simulation ·	234	Java Grande
euler	computational fluid dynamics	915 (2028)	Java Grande

Table 3: The benchmarks

7.2 Cost and Precision

The points-to analysis is applied to the seven benchmark programs. Analysis time is measured on an Ultra SPARC5 with a 270MHz processor, using java from SUN JDK1.2.2 with jit enabled. Table 4 summarizes the measurement for each program: "Prgm Pts" gives the number of program points where ewpt mappings were computed; "time" gives the actual analysis time; and "ewpt/javac" gives the percentage of the analysis time in a plain javac compilation.

Program	Prgm Pts	Analysis Time	
		time (ms)	ewpt/javac
listtbl	10	9	0.1%
cmatmul	25	162	2.1%
shallow	73	259	3.4%
cholesky	19	195	2.6%
sor	18	184	2.5%
lufact	40	168	2.9%
moldyn	53	77	1.0%
euler	299	2440	25%

Table 4: Analysis cost

Overall, the analysis time of the first seven programs is fairly small (0.1%- 3.4% of a plain javac compilation). The analysis time of euler (25% of javac compilation) is much higher because euler involves switching the four sub-arrays of a 2-dimensional array ug. As a result, the ewpt mappings of ug contains 18 symbolic names, whereas, in other benchmarks, most ewpt mappings contain about 2 symbolic names. This suggests that a reasonable k-limiting on the size of ewpt mappings can help reduce analysis cost on irregular assignment patterns.

To measure the precision, we checked the output ewpt mappings obtained. For array references, the ewpt mappings were fairly precise. In particular, the mappings of ug in euler capture correct points-to information due to the switching of elements. Furthermore, using the "killing" technique, the analysis is able to remove all of redundant null from the mappings of array elements.

7.3 Dependence Analysis

Three versions of dependence tests were implemented using different pointer information. All of them use the Omega library [13] to determine the dependences.

- type collects read- and write-sets as sets of types. Two accesses are reported dependent if their types are compatible and at least one of them is a write. Two array accesses a[x] and b[y] are dependent if a and b are of compatible types, and if x and y may denote the same offset. The latter condition is determined by the Omega library.

- flat assumes that there is no aliasing between array accesses (i.e., arrays are flat FORTRAN-like arrays). For non-array references, it uses a type-based analysis as in the previous test. Although the assumption made by this test is not safe, it can give a lower bound of the number of dependences in the program that involve arrays.

- ewpt computes read- and write-sets as sets of heap locations from ewpt mappings. Details can be found in [17]. Dependence testing on heap locations is the same as that on FORTRAN arrays.

Table 5 gives the statistics collected for the three implementations. Only dependences due to conflicts of heap locations were reported, and at most one dependence was reported between any pair of statements. Note that ewpt reports fewer dependences than flat. This is because although flat assumes perfect information about arrays, it is quite conservative about non-array objects; whereas ewpt has information of both types of objects.

It is worth to mention that using a conventional points-to analysis in place of type would not improve the dependence test significantly, due to their lack of ability to disambiguate different elements of an array.

Program	Dependences			Parallel Loops		
	type	flat	ewpt	type	ewpt	real
listtbl	5	3	2	0	1	1
cmatmul	8	3	3	3	7	7
cholesky	10	4	4	5	6	6
shallow	1092	152	152	6	17	17
sor	6	5	5	3	4	4
lufact	72	45	45	9	11	11
moldyn	2	0	0	17	19	19
euler	12559	2489	2489	36	55	55

Table 5: Dependences and parallel loops

Program	bound-check		null-check		safe loop	
	ewpt	type	ewpt	type	ewpt	real
listtbl	0	2	0	3	2	2
cmatmul	0	13	0	13	7	7
cholesky	0	19	0	19	8	8
shallow	0	278	0	278	20	20
sor	0	17	0	17	7	7
lufact	22	57	0	57	6	15
moldyn	2	2	0	8	22	24
euler	12	507	0	705	57	60

Table 6: Redundant checks and safe loops

Table 5 also shows the number of parallel loops detected by ewpt and type. The actual number of parallel loops is given in real. We assume that conflicts due to stack locations can be handled by techniques such as scalar privatization. Overall, ewpt is able to detect all actual parallel loops, whereas parallel loops detected by type are mostly initialization loops.

7.4 Exception Analysis

Table 6 gives the number of array bounds and null-pointer checks identified as redundant as well as the number of exception-free loops (safe loops). Some redundant array bounds checks are undetected because our implementation is not able to compare symbolic expressions. On the other hand, all null-pointer checks in the benchmarks have been identified as redundant due to the analysis' ability to remove null from ewpt mappings.

8. RELATED WORK

We compare our work with research in three areas: analyses of heap-directed pointers, dependence analyses in the presence of pointers, and Java exception analysis.

Pointer Analysis. There are two approaches to compute properties for heap-directed pointers. The first one is referred to as *store-based* because heap locations are named statically, using either the pointer type [5, 15, 2], or the allocation site [16]. Our points-to analysis is store-based, but it uses a more precise naming scheme. We name objects by their allocation statement *instances*. In many cases, naming heap objects using allocation sites is a good cost/precision trade-off. However, for loop-based dependence tests it is important to distinguish different objects created by the same allocation site. Furthermore, our points-to analysis is able to compute points-to information for individual array elements that most others cannot.

As opposed to the store-based approach, a store-less analysis directly computes alias properties without naming heap objects. These properties could be alias pair information [4, 6, 3] or shape information [10, 14, 7]. The shape of a pointer tells us whether a pointer refers to a list, a tree, etc. It is the "store-less" counterpart of the element-wise information that our analysis captures. Although the work by Deutsch [4] is store-less, our work shares one common feature with his. Both analyses try to summarize properties of unbounded objects in one symbolic form. His work uses symbolic access paths, whereas ours uses ewpt mappings (or symbolic names).(expensive) transfer functions

Dependence Testing with Pointers. Dependence tests that are based on store-less pointer analyses [9, 8, 12] represent read/write sets as sets of access paths. Access paths from different program points cannot be compared as they do not have any associated points-to information. Therefore, these dependence tests are applicable only to loops with no pointer assignments.

Like our points-to analysis, Ghiya and Hendren [7] also address pointer analysis in the context of dependence testing. Their analysis aims to enhance store-less schemes so that they can be used for dependence tests. Our scheme aims to improve the precision of store-based analyses for iteration-based dependence tests. They are able to handle pointers to flat arrays, but not array elements of pointer types since their points-to analysis is not element-wise. In addition, they cannot handle dependences in a loop with pointer assignments as their scheme is not instance-wise.

Chambers et. al. [2] address dependence analysis for Java in the presence of exceptions, multi-threading, and dynamic class loading. They use a type-based points-to analysis. They do not focus on getting precise information for loop iterations or array elements.

Overall, no previous work can decide that the following loop carries no dependence over statement 3.

```
1      for (i = 0; i<n; i++) {
2        p = new ...;
3        p.a = ...;
4      }
```

The fact that p points to a new object at each iteration cannot be abstracted by either shape or alias information. Our analysis can do this because of the instance-wise naming of objects.

Exception Analysis. Bodik, Gupta, and Sarkar [1] proposed a method to eliminate exception checks based on partial redundancy elimination. Lacking precise pointer information, their method only remove partially redundant checks for heap-residing references. Our points-to analysis, on the other hand, is able to directly remove redundant checks based on ewpt mappings.

Moreira, Gupta, and Midkiff [11] exploited exception-free regions for numerical Java programs. However, they did not address the techniques to identify such regions.

9. CONCLUSION AND FUTURE WORK

In this paper, we present a pointer analysis that computes points-to information for instances of references and elements of reference arrays based on an abstraction called the element-wise points-to mapping. This information can be used to enable a precise loop-based dependence test for Java. We also propose a technique to perform kills on ewpt mappings for heap assignments.

This technique can help identify redundant null-pointer checks for heap-residing pointers. We obtain promising results with a reasonable cost when using ewpt information on dependence analysis and exception analysis.

This work can be extended in many directions. The analysis is still intra-procedural. An open question is how to represent objects instance-wise across method calls, especially for recursive calls. We would also like to integrate the analysis in a Java environment with `try/catch` blocks, multi-threading and dynamic class loading. Another important application of the analysis that needs to be studied is heap optimizations. Since ewpt mappings capture precise links between references, objects and allocation sites, it is possible to use them for garbage collection, object layout optimization (e.g., flattening arrays), and object privatization.

10. REFERENCES

[1] Rastislav Bodik, Ragiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM Symp. on Programming Language Design and Implementation*, June 2000.

[2] C. Chambers, I. Pechtchanski, V. Sarkar, M. Serrano, and H. Srinivasan. Dependence analysis for Java. In *Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[3] Ben-Chung Cheng and Wen mei Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM Symp. on Programming Language Design and Implementation*, pages 57–69, June 2000.

[4] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Symp. on Programming Language Design and Implementation*, June 1994.

[5] A. Diwan, K.S. McKinley, and E.B. Moss. Type-based alias analysis. In *ACM Symp. on Programming Language Design and Implementation*, June 1998.

[6] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 1995.

[7] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *ACM Symp. on Principles of Programming Languages*, January 1998.

[8] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. In *IEEE Trans. on Parallel and Distributed Computing*, January 1990.

[9] Joseph Hummel, Laurie J. Hendren, and Alex Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *ACM Symp. on Programming Language Design and Implementation*, June 1994.

[10] Joseph Hummel, Laurie J. Hendren, and Alex Nicolau. A language for conveying the alising properties of dynamic, pointer-based data structures. In *Inthernational Parallel Processing Symposium*, pages 208–216, April 1994.

[11] Samuel P. Midkiff Jos E. Moreira and Manish Gupta. From flop to megaflops: Java for technical computing. Technical Report toplas, 2000.

[12] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *ACM Symp. on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[13] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape analysis problems in languages with destructive updating. In *ACM Symp. on Principles of Programming Languages*, January 1996.

[15] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symp. on Principles of Programming Languages*, January 1996.

[16] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM Symp. on Programming Language Design and Implementation*, June 1995.

[17] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Element-wise points-to analysis for loop-based dependence testing. Technical Report CSRD 1707, U. of Illinois at Urbana-Champaign, July 2001.