Data Flow Analysis for Software Prefetching Linked Data Structures in Java

Brendon Cahoon and Kathryn S. McKinley
Department of Computer Science
University of Massachusetts
Amherst, MA 01002
{cahoon,mckinley}@cs.umass.edu

Abstract

In this paper, we describe an effective compile-time analysis for software prefetching in Java. Previous work in software data prefetching for pointer-based codes uses simple compiler algorithms and does not investigate prefetching for object-oriented language features that make compile-time analysis difficult. We develop a new data flow analysis to detect regular accesses to linked data structures in Java programs. We use intra and interprocedural analysis to identify profitable prefetching opportunities for greedy and jump-pointer prefetching, and we implement these techniques in a compiler for Java. Our results show that both prefetching techniques improve four of our ten programs. The largest performance improvement is 48% with jump-pointers, but consistent improvements are difficult to obtain.

1. Introduction

Software controlled data prefetching improves memory performance by hiding memory latency. Its goal is to bring data into the cache before the demand access to that data. Existing research shows the benefits of software prefetching techniques for array-based, scientific programs [6, 21, 4, 19]. Given an array, the size of each element, and a regular access pattern, a compiler can compute the address of any element in the array and prefetch it.

Prefetching in pointer-based codes is difficult because separate dynamically allocated objects are disjoint, and the access patterns are thus less regular and predictable. Given an object `o`, we know the address of objects that `o` references, and we cannot prefetch other objects without following pointer chains. Recent pointer prefetching work considers C programs only [16, 18, 24, 15, 28]. Object-oriented Java programs pose additional analysis challenges because they mostly allocate data dynamically, contain frequent method invocations, and often implement loops with recursion.

The memory penalty can be high for object-oriented programs that frequently traverse linked data structures. Figure 1 illustrates the percentage of time spent servicing memory requests in an object-oriented Java implementation of the Olden benchmark suite [7] using RSIM [22] with a current, aggressive processor model. Memory stalls account for 15% to 95% of the execution time in these programs; thus, compiled Java programs suffer from the same latency problems as imperative languages.

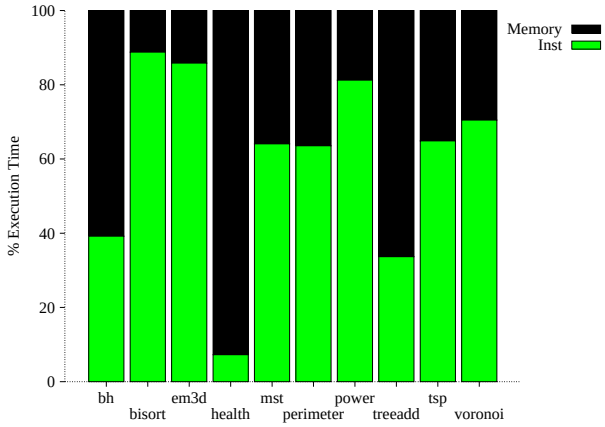


Figure 1. Memory penalty in compiled Java programs

This paper includes a new data flow analysis for discovering accesses to linked data structures. We implement our analysis in Vortex, an optimizing compiler for object-oriented programs [9]. We use our linked structure analysis to identify opportunities for the compiler to insert prefetches of objects. Our first prefetching implementation, greedy prefetching, prefetches directly connected object(s) during each iteration of a loop or recursive function call. We also present an automatic, compile-time algorithm, and implementation of jump-pointer prefetching. Jump-pointer prefetching may hide additional latency by using an extra pointer to prefetch objects further than a single link away. Our algorithm uses our new analysis to identify objects to which the compiler adds a jump-pointer field when either a program creates or traverses a linked data structure. The compiler generates prefetches using the jump-pointer field when the program traverses a linked structure.

Our data flow analysis contains several interesting features. We improve the precision of our analysis by tracking data flow facts across stores and loads to object fields and

arrays. In our Java programs, analyzing object fields, when combined with inlining or interprocedural analysis, is necessary to discover linked structure accesses that programmers hide using encapsulation. Our analysis also identifies indirect objects that are referenced by linked structures as candidates for prefetching.

Both prefetching techniques improve four of the ten programs in our benchmark suite. The largest run-time improvement is 48% which occurs using jump-pointers. Across all benchmarks, greedy and jump-pointer prefetching improve performance by a geometric mean of 4% and 10%, respectively. Even with prefetching, memory latency is still a problem. Future work should combine other techniques with prefetching to hide or eliminate latency.

We organize the rest of the paper as follows. Section 2 presents an overview of our contributions to prefetching linked data structures. We describe our analysis for discovering accesses to linked structures in Section 3. In Section 4 and Section 5, we present our implementations of greedy and jump-pointer prefetching. We evaluate our prefetching implementations on a set of object-oriented Java programs in Section 6. Section 7 summarizes the related work.

2. Overview

To effectively insert prefetches, our compiler must identify regular accesses to linked data structures. We define a new data flow analysis with intra and interprocedural components to discover linked structure traversals in Java programs. We implement an existing compile-time technique for prefetching linked structures, called greedy prefetching, which is able to prefetch directly connected objects in linked data structures. We design and implement a compile-time, jump-pointer prefetching algorithm which potentially improves latency tolerance by prefetching further ahead in the linked structure, and, thus, increasing the amount of work between the prefetch and the demand request of an object.

We implement our data flow analysis and software prefetching schemes in Vortex, an optimizing compiler for object-oriented programs [9]. Vortex is a whole program optimizer that performs high-level optimizations targeted to object-oriented languages such as intraprocedural class analysis and class hierarchy analysis. Vortex implements call graph construction algorithms and uses the call graph to drive a general interprocedural data flow analyzer. Vortex performs other high-level optimizations, such as inlining, and low-level optimizations, including common subexpression elimination, copy propagation, and dead code elimination, to help produce efficient code.

3. Discovering accesses to linked structures

We identify regular traversals of a linked data structure by a *recurrent* update to a pointer variable. A recurrent update is a field assignment of the form `o = o.next` that appears within a loop or recursive call, as shown in the examples below. Each execution of the assignment updates the pointer variable with a new object of the same type, either directly or by using a temporary.

```
while (o != null) {
  o.compute();
  o = o.next;
}
```

```
while (o != null) {
  o.compute();
  t = o.next;
  ...;
  o = t;
}
```

Our analysis, which we call *recurrent analysis* (RA), contains an intraprocedural component and an interprocedural component. The intraprocedural algorithm finds recurrent pointer variables that occur in loops, and the interprocedural algorithm finds recurrent pointer variables that occur due to recursive function calls. In this paper, we use the term pointer variable to describe a variable that is defined as a Java reference type.

In this section, we describe our basic intraprocedural algorithm next. We follow with extensions to handle fields and arrays that contain recurrent pointer variables, and for indirect recurrent pointer variables. Then, we briefly describe our interprocedural analysis.

3.1. Basic intraprocedural analysis

Intraprocedural recurrent analysis discovers the field assignments that are recurrent due to loops. Our analysis is similar to reaching definitions analysis combined with computing definition-use chains for field references [1]. We discover linked structure traversals using a unified, forward data flow analysis.

We define the following sets in our data flow analysis. Let PV be the set of pointer variables in a method, F be the set of object fields, S be the set of statements in the method, and RS be the recurrent status. The basic analysis information is a set of tuples:

$$\mathrm{R} \subseteq \mathcal{P}(\mathrm{PV} \times \mathrm{F} \times \mathrm{S} \times \mathrm{RS})$$

We define a function RA that maps program statements to the analysis information, RA:$s \rightarrow$ R, where $s \in$ S.

The tuple contains the field name (F) to improve precision by reducing the number of recurrent pointer variables that the analysis discovers. For example in Figure 2 (b), we improve the precision of our analysis and effectiveness of prefetching by recording that the `next` field causes the recurrence and that the `prev` field does not.

We use the statement number (S) to properly handle the case when we have two field assignments that occur outside a loop or recursive call. For example, if the sequence

```
class SList {
  int   data;
  SList next;
  int sum() {
    prefetch(next);
    if (next != null)
      return data + next.sum();
    return data;
  }
}
```

(a) Linked list

```
class DList {
  int   data;
  DList next, prev;
  int sum() {
    prefetch(next);
    if (next != null)
      return data + next.sum();
    return data;
  }
}
```

(b) Doubly linked list

```
class Tree {
  int   data;
  Tree  left, right;
  int sum() {
    prefetch(left);
    prefetch(right);
    int s = data;
    if (left != null)
      s += left.sum();
    if (right != null)
      s += right.sum();
    return s;
} }
```

(c) Binary tree

Figure 2. Examples of prefetching linked data structures

`o=o.next; o=o.next` is not in a looping construct, we do not want to mark `o` as a recurrent pointer variable.

The recurrent status (RS) indicates when a program uses a pointer variable to traverse a linked data structure. Let $rs \in \text{RS} = \{$ `nr`, `pr`, `r` $\}$. We order the elements of RS such that `nr` $\prec$ `pr` $\prec$ `r`. We define the element values as follows:

Not recurrent (`nr`). The initial value which indicates a pointer variable is not involved in a traversal.

Possibly recurrent (`pr`). The first time we process a field reference use, it is potentially recurrent.

Recurrent (`r`). This value indicates a pointer variable is involved in a linked structure traversal.

The first time we analyze a loop, an object occurring on the left hand side of a pointer field assignment becomes `pr` (*e.g.*, `t = o.next`). On the second iteration of the analysis, the object on the left hand side becomes `r` if the base object of the field reference (*i.e.*, `o`) is `pr`. If the base object is `nr`, then `t`'s value remains the same.

The data flow equations for recurrent analysis (RA) are:

$$\begin{aligned} \text{RA}_{in}(s) &= \bigsqcup_{p \in pred(s)} \text{RA}_{out}(p) \\ \text{RA}_{out}(s) &= (\text{RA}_{in}(s) \setminus \text{KILL}_{RA}(s, \text{RA}_{in}(s))) \\ &\quad \sqcup \text{GEN}_{RA}(s, \text{RA}_{in}(s)) \end{aligned}$$

Given tuples, $t_1=(p_1,f_1,s_1,rs_1)$ and $t_2=(p_2,f_2,s_2,rs_2)$, we define the join operation, $t_1 \sqcup t_2$, as follows. If $(p_1=p_2 \wedge f_1=f_2 \wedge s_1=s_2)$ then $t_1 \sqcup t_2 = (p,f,s,rs_1 \sqcup rs_2)$. Otherwise, $t_1 \sqcup t_2 = \{t_1, t_2\}$. Given our ordering of the elements $rs \in$ RS, $rs \sqcup$ `nr`$=rs$, `pr`$\sqcup$`pr`$=$`pr`, and $rs \sqcup$ `r`$=$`r`.

We define the GEN_{RA} and KILL_{RA} functions as follows:

$$\text{GEN}_{RA}, \text{KILL}_{RA} : \text{S} \times \text{R} \rightarrow \text{R}$$

At the initial statement, init(S), we initialize the function RA_{in} to $\{(pv, \emptyset, \emptyset,$`nr`$) | pv \in \text{PV}\}$

The statements which affect the analysis include field loads and assignments. We describe the details of our GEN and KILL functions for each interesting program statement below. In the following function definitions, f'∈F, s'∈S, and rs'∈RS.

`o=p.`f_i A field assignment at statement i may create a recurrent update when it occurs in a loop. Informally, the expression causes a recurrent update when the value assigned to `o` is propagated to `p`, the base object on the right hand side. The canonical example is `o = o.next` in a loop with no other assignments to `o`. The KILL_{RA} and GEN_{RA} functions for a field assignment are:

$$\text{KILL}_{RA}(\text{o=p.f}_i, \text{R}) = \{(\text{o,f,i,pr}), (\text{o,}\emptyset,\emptyset,\text{nr})\}$$

$$\text{GEN}_{RA}(\text{o=p.f}_i, \text{R}) = \begin{cases} \{(\text{o,f,i,pr})\} & : \text{if (p,}\emptyset,\emptyset,\text{nr)} \in \text{R} \\ \{(\text{o,f,i,r})\} & : \text{if (p,f,i,pr)} \in \text{R} \end{cases}$$

The first time we process a field expression, we create a tuple containing `o` with the `pr` recurrent status. If the analysis does not reach a fixed point due to a loop, the field assignment expression is processed again. If there exists a tuple containing `p` with the recurrent status `pr`, then there is no intervening assignment to `p`. In this case, we create a tuple containing `o` with the `r` recurrent status.

`o=p` A pointer variable assignment expression creates an alias between `p` and `o`. For each tuple containing a pointer variable `p`, we create a new tuple containing `o` with the same field, statement, and recurrent status as `p`. We kill the old information associated with `o`. The KILL_{RA} and GEN_{RA} functions for an assignment are:

$$\text{KILL}_{RA}(\text{o=p}, \text{R}) = \{(\text{o,f',s',rs')}\}$$

$$\text{GEN}_{RA}(\text{o=p}, \text{R}) = \{(\text{o,f',s',rs')} \mid (\text{p,f',s',rs'}) \in \text{R}\}$$

`o=`*`expr`* Any other assignment to a pointer variable kills the analysis information for `o`. Our analysis sets the

recurrent status of any tuple containing `o` to *not recurrent* (`nr`). The kill$_{RA}$ and GEN$_{RA}$ functions for all other assignments are:

$$\text{KILL}_{RA}(\text{o=}expr, \text{R}) = \{(\text{o,f',s',rs'})\}$$
$$\text{GEN}_{RA}(\text{o=}expr, \text{R}) = \{(\text{o},\emptyset,\emptyset,\texttt{nr})|(\text{o,f',s',rs'}) \in \text{R}\}$$

3.2. Storing an object into an array or field

In the previous section, we assume that the left hand side expression is a simple object reference. We improve the analysis by also tracking the recurrent information of objects assigned to object fields and array elements. For example, in the following code sequence, the analysis in Section 3.1 does not indicate that object `o` is *recurrent* because the analysis does not propagate the recurrent information to `temp.f`. This sequence occurs in Java programs that use the `Enumeration` class to traverse linked lists, when inlining is enabled.

```
while (temp.f != null) {
  o = temp.f;
  o.compute();
  t = o.next;
  temp.f = t;
}
```

To improve the analysis, we extend the data flow tuple to include field references and arrays. Let PR be the set of pointer variables, field references, and arrays. We define:

$$\text{R'} \subseteq \mathcal{P}(\text{PR} \times \text{F} \times \text{S} \times \text{RS})$$

We also define a new analysis function, RA':$s \rightarrow$ R'.

For object fields, we associate the analysis information with the field name and we ignore the base object. We prepend the field name with its class name to avoid ambiguity between fields from different classes. We can potentially improve the precision by tracking the base object which increases the analysis complexity cost. We treat arrays as monolithic objects in our analysis, *i.e.*, as an assignment or use of the whole array.

We define the GEN$_{RA'}$ and KILL$_{RA'}$ functions to include the same definitions as GEN$_{RA}$ and KILL$_{RA}$ with the following extensions:

`p.f=o, a[j]=o` Create data flow information for a field or array reference. The GEN$_{RA'}$ and KILL$_{RA'}$ functions are similar to a pointer variable assignment.

$$\text{KILL}_{RA'}(\text{p.f=o}, \text{R'}) = \{(\text{p.f,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{p.f=o}, \text{R'}) = \{(\text{p.f,f',s',rs'})|(\text{o,f',s',rs'}) \in \text{R'}\}$$
$$\text{KILL}_{RA'}(\text{a[j]=o}, \text{R'}) = \{(\text{a,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{a[j]=o}, \text{R'}) = \{(\text{a,f',s',rs'})|(\text{o,f',s',rs'}) \in \text{R'}\}$$

`o=p.f, o=a[j]` For any tuple containing `p.f` or `a`, we create a new tuple containing `o` which includes the field, statement, and recurrent status. The GEN$_{RA'}$ and KILL$_{RA'}$ functions are:

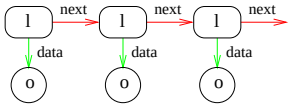
$$\text{KILL}_{RA'}(\text{o=p.f}, \text{R'}) = \{(\text{o,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{o=p.f}, \text{R'}) = \{(\text{o,f',s',rs'})|(\text{p.f,f',s',rs'}) \in \text{R'}\}$$
$$\text{KILL}_{RA'}(\text{o=a[j]}, \text{R'}) = \{(\text{o,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{o=a[j]}, \text{R'}) = \{(\text{o,f',s',rs'})|(\text{a,f',s',rs'}) \in \text{R'}\}$$

`p.f=`*expr*`, a[j]=`*expr* Any other assignment to a field or array kills the data flow information for `p.f` or `a`. The GEN$_{RA'}$ and KILL$_{RA'}$ functions are:

$$\text{KILL}_{RA'}(\text{p.f=}expr, \text{R'}) = \{(\text{p.f,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{p.f=}expr, \text{R'}) = \{(\text{p.f},\emptyset,\emptyset,\texttt{nr})|(\text{o,f',s',rs'}) \in \text{R'}\}$$
$$\text{KILL}_{RA'}(\text{a[j]=}expr, \text{R'}) = \{(\text{a,f',s',rs'})\}$$
$$\text{GEN}_{RA'}(\text{a[j]=}expr, \text{R'}) = \{(\text{a},\emptyset,\emptyset,\texttt{nr})|(\text{o,f',s',rs'}) \in \text{R'}\}$$

3.3. Indirect recurrent pointer variables

An indirect pointer variable is a unique object that is referenced by a recurrent pointer variable, but is not recurrent itself. An object is *unique* if it is referenced by at most one other object. In contrast, an object is *shared* if it is referenced by multiple objects. An example of an indirect recurrent pointer occurs in a traversal of a generic linked list, where the data elements are separate objects from the list nodes. Below, `l` is a recurrent pointer variable for a linked list traversal, and both `l` and `o` are unique. In this example, `o` is also an indirect recurrent pointer variable because it is unique, and it is referenced by a recurrent pointer variable.




```
while (l != null) {
    o = l.data;
    o.compute();
    l = l.next;
}
```

Both `l` and `o` are candidates for prefetching because each iteration of the loop accesses a new list node and a new data element. We do not want to prefetch shared objects because each iteration may access the same data element which results in wasteful prefetches.

We must first classify objects as *shared* or *unique* to compute the set of indirect recurrent pointer variables. We use an approximation because statically classifying dynamically allocated objects is not feasible. Our approximation classifies *class fields* as shared or unique. For example, in the code above, we classify the `next` and `data` fields as unique because the fields reference a single object.

Our interprocedural, context-sensitive analysis is similar to Aldrich et al.'s shared object analysis for eliminating unnecessary synchronization [2], and Dolby's analysis for finding inlinable objects [10]. We only briefly describe the data flow analysis here.

The analysis begins by assuming that all fields are unique. The analysis maintains a mapping between program variables and field names. When processing an assignment of a variable to a field, the analysis creates an association between the variable and the field name. The analysis removes other existing associations between the field name and any different variable. If the variable appears on the right hand side of multiple field store expressions, then the analysis associates the variable with each field name. When processing a field store, if there already exists an association between the variable and the field, then the field is *shared*.

The analysis also propagates the field information at assignments and field loads. At a function call, we assign the field information associated with each actual to each formal. After we process the function call, the caller updates the analysis information with changes made in the callee by assigning the field information associated with each formal to each actual. We compute aliases among the objects to determine if the object has previously been assigned to a field. All fields are identified as either *shared* or *unique* at the end of this analysis.

3.4. Interprocedural algorithm

We use an interprocedural algorithm to find linked structure traversals occurring across recursive method calls. The algorithm is a bidirectional, context-sensitive traversal of the call graph. A context-sensitive algorithm enables the analysis phase to determine the fields used in recurrent object updates across recursive function calls. A context-insensitive algorithm cannot track the recurrent fields from multiple call sites because the compiler analyzes each method only once. For example, in Figure 2 (c), a context-sensitive analysis determines that `this.left` and `this.right` are both recurrent fields in the recursive method, `sum`. A context-insensitive analysis only analyzes `sum` once, and will not determine that both `left` and `right` are recurrent fields, unless the analysis is able to determine the call sites are self-recursive.

A method definition has the form: `r = m(`p_0`,...,`p_n`)`, where p_i is a formal parameter, and `r` is the return value. At a call site, we create a new set of tuples, R_m, for the callee. We process each actual, a_0,...,a_n, as an assignment of the recurrent information from the actual to the formal, $p_i = a_i$. At a call site, we also add the recurrent field information to R_m for each of the actual's fields, a_i`.f`. After initializing R_m, we analyze the callee method with R_m using the intraprocedural analysis. Recursive calls cause the analysis to iterate until the formals reach a fixed point.

We process a function return as an assignment of `r` to the value on the left hand side of the method call by copying the analysis information from R_m to R_c. The analysis uses the appropriate GEN and KILL function, which depends on whether the left hand side expression is a simple object, field reference, or array reference. Upon return we must also update R_c with any tuple in R_m that contains a field reference as an element of PR.

A context-sensitive interprocedural algorithm can be quite expensive because each function may be analyzed multiple times. Our interprocedural analysis can analyze each function reached at each call site up to 3 times. Each call site may invoke multiple functions due to polymorphic function calls. We perform 0-CFA interprocedural class analysis to reduce the number of potential functions reachable at each call site [27].

4. Greedy prefetching

In this section, we describe our greedy prefetching algorithm which prefetches directly connected objects in recurrent accesses. Figure 2 shows simple class definitions for a singly linked list, a doubly linked list, and a binary tree. Each class contains a `sum` method which adds the elements in the data structure. In the list examples, we insert a prefetch instruction for the `next` object in the linked list. We cannot prefetch objects further ahead because we do not know the address of future objects.

Achieving the full benefits of prefetching requires that the computation time between the prefetch and use of the object be greater than or equal to the memory access time to completely hide the latency. If the computation time is less than the memory access time, the prefetch can partially hide the latency. In the linked list examples, we only partially hide the read latency of `next` if the cost of the addition and function call is less than the cost of a memory access. Similarly, we typically only partially hide the latency of the prefetch of `left` in the binary tree example. Since we also prefetch `right`, we may completely hide its latency.

The greedy prefetch algorithm consists of two parts; the phase described in Section 3 which finds objects to prefetch, followed by a phase which schedules the prefetch instructions. We describe the scheduling phase next.

4.1. Scheduling prefetch instructions

The scheduling phase computes which recurrent objects to prefetch, and when to insert the prefetch instructions. The algorithm is greedy because we do not perform any analysis to determine if an object is already in the cache, and we try to prefetch as much as possible.

For each *recurrent* object at each program point, we generate a prefetch for its recurrent fields when the object is not null. The scheduler computes the set of non-null objects using information from the program structure. The scheduler knows an object is not null under the following conditions: (1) immediately after an object allocation, (2) after a comparison to `null`, (3) the base object of a field access, (4)

the `this` object upon entering a method, and (5) the first parameter after a method call. For example, a loop that traverses a linked list typically compares the current head element to null at the start of the loop. In this case, the scheduler knows the head element is not null, and can generate a prefetch for the recurrent field in the list.

The scheduler uses alias analysis information to generate a single prefetch for groups of aliased recurrent objects, such as `t=o.next; o=t;`. In a loop, we mark both `o` and `t` as recurrent, but we only generate a prefetch for one of the objects. The following pseudo-code summarizes our intraprocedural scheduling process.

```
Let R = RA_out(exit(S)); exit(S) is the last statement
For each assignment, o = expr, at statement s:
  if o is not null
    for each tuple (o,f,s,r) ∈ R
      generate prefetch o.f
      remove (o,f,s,r) from R
      for each p that is an alias of o
        remove (p,f,s,r) from R
```

The compiler generates multiple prefetches if the size of the object is greater than the size of a cache line. A command line option specifies the cache line size.

The scheduling phase generates prefetches for all the individual elements of an array, if the array field is recurrent, and the size of the array is a small, compile-time constant [1]. Computing the size of a Java array is not trivial since Java programs allocate arrays dynamically at run time. Many programs allocate arrays of the same type using compile-time constants which makes it possible for the compiler to determine the size of an array. When performing interprocedural analysis, the compiler analyzes all the array allocation sites, and computes the set of constant size arrays of the same type and size.

Interprocedural Scheduling: We must take care not to generate redundant prefetches for recurrent objects passed as parameters by issuing a prefetch in the caller and the callee methods. We minimize redundant prefetches as follows. Our compiler performs a single, in-order pass over the call graph to schedule recurrent parameters as high as possible in the call graph. The scheduler does not issue a prefetch of a recurrent parameter when the scheduler generates a prefetch for the parameter in a calling method.

Another source of useless prefetches in non-recursive methods is due to method overriding. When a program contains several implementations of a method that has a recurrent parameter, but only one of the implementations is recursive, our analysis indicates that the parameter is recurrent in all the implementations. We eliminate this source of useless prefetches by not generating a prefetch for a field of a recurrent parameter if the callee does not reference the field.

[1] We do not prefetch regular array accesses, *e.g.*, `while(i<N) a[i++];`, although many Java programs use arrays.

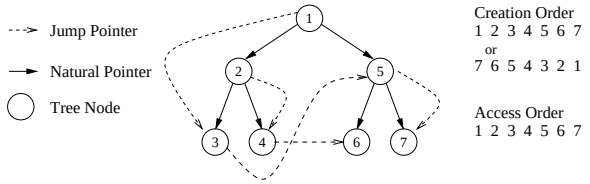


Figure 3. Jump-pointer prefetching: Binary tree traversal

5. Jump-pointer prefetching

In this section, we discuss our design and implementation of compile-time, jump-pointer prefetching. Figure 3 illustrates jump-pointer prefetching for a binary tree. Each tree node contains a jump-pointer to a tree node two links away. Thus, when the program accesses node 1, we issue a prefetch for node 3. The number of links depends upon the amount of latency we need to hide.

Jump-pointers are a flexible mechanism for linked data structures because we can prefetch arbitrary objects and not just directly connected objects. Jump-pointer prefetching is potentially able to tolerate any amount of latency by varying the distance between the two objects. Greedy prefetching restricts the amount of latency tolerance by prefetching direct links only, but does not require an additional field to store the jump-pointer. Jump-pointer prefetching may also reduce the number of prefetches, yet still remain effective. In Figure 3 for example, greedy prefetching adds two prefetches for each node reference, but jump-pointer prefetching adds one only. Furthermore, jump-pointer prefetching does not prefetch `null` objects at the leaf nodes.

Our compiler automates jump-pointer prefetching by inserting code to create and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program. It has three main parts, (1) identify the objects to prefetch using the recurrent analysis from Section 3, (2) schedule prefetches for the objects, and (3) generate the code to create the jump-pointers. Step (2) uses the scheduling algorithm from Section 4.1, but the prefetch code is different. Instead of generating a prefetch for each recurrent field, the compiler generates a prefetch for the *prefetch* field, *i.e.*, `prefetch` $o.prefetch$. Step (3) is the major difference between jump-pointer and greedy prefetching, and we describe it below.

5.1. Creating jump-pointers

The compiler creates jump-pointers when either an object is *created*, *e.g.*, using `new`, or when *traversing* a data structure. We use a compiler option to specify the choice.

```
class Tree
{
  int value;
  Tree left;
  Tree right;
  Tree prefetch;
}
```

```
Tree createTree(int l)
{
  if (l == 0) return null;
  else {
    Tree n = new Tree();
    jumpObj = jumpQueue[i];
    jumpObj.prefetch = n;
    jumpQueue[i++ % size] = n;
    Tree left = createTree(l-1);
    Tree right = createTree(l-1);
    n.left = left;
    n.right = right;
    return n;
} }
```

Figure 4. Inserting jump-pointers for a binary tree

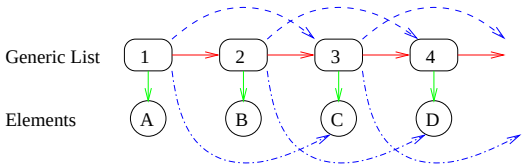


Figure 5. Indirect jump-pointer example

By default, our compiler builds jump-pointers at the object creation site. In our current implementation, the compiler adds only one jump-pointer field to a recurrent object. We do not add jump-pointers when linked structures are updated, unless the update occurs while traversing the linked structure. The effectiveness of jump-pointer prefetching depends on when and where the compiler creates the jump-pointers. We discuss each choice in detail below.

Figure 4 shows the code for building jump-pointers in a binary tree object at creation time. The circular queue, *jumpQueue*, maintains a list of the last *n* objects allocated. When an object allocation occurs, we create a jump-pointer from the object at the head of *jumpQueue* to the new object. Then, we insert the new object at the end of *jumpQueue*, and advance the circular queue index. Currently, our compiler creates jump-pointers from the *jumpQueue* object to the current object unless a command line option specifies the reverse direction. We also use a circular queue when the compiler updates the jump-pointers during a traversal.

Object Creation: Adding jump-pointers during object creation is beneficial for data structures with regular access patterns that do not frequently change. This choice minimizes the run-time cost because the jump-pointers are created once. Unfortunately, it is not always possible to create effective jump-pointers at the creation site. For example, in Figure 3, the creation phase must be preorder, beginning with either the left or right subtree. If the programs builds the tree bottom-up, then the jump-pointers will not be useful. Another problem occurs in programs that frequently update a linked structure containing jump-pointers because the original jump-pointers become invalid.

Traversal: Building jump-pointers during traversals is effective for programs that contain multiple instances of a linked structure that a program frequently traverses and may also update. Due to the overhead of maintaining the jump-pointer queue, this approach is less effective when programs do not change the linked structures, or when the traversal patterns change, *e.g.*, accessing a list in one direction followed by an access in the reverse direction. An advantage is that the code to create jump-pointers appears locally with the prefetches, which means the compiler does not need knowledge of the entire program.

5.2. Indirect jump-pointers

Section 3.3 discusses our analysis for discovering indirect recurrent pointers. An indirect recurrent pointer variable is a unique object that is referenced by a recurrent pointer variable. We prefetch indirect pointer variables by creating a second jump-pointer from the recurrent pointer variable to the indirect recurrent pointer variable. We illustrate indirect jump-pointers in Figure 5 which contains a generic linked list (the rectangles) with pointers to the list elements (the circles). If a program allocates the list objects in order, `A`, `B`, `C`, and `D`, then we add jump-pointers as illustrated (`1` to `C`, `2` to `D`, etc.). When the program traverses the linked list, we schedule prefetch instructions for the list and element jump-pointers. Greedy prefetching is unable to effectively prefetch these objects because there are no direct links between them.

6. Experimental results

In this section, we evaluate the effectiveness of greedy and jump-pointer prefetching. We implement our recurrent analysis and prefetching algorithms in the Vortex optimizing compiler [9]. We use Vortex to compile our Java programs, perform object-oriented and traditional optimizations, and generate Sparc assembly code.

We evaluate our prefetching algorithms using an object-oriented Java version of the Olden benchmark suite that our research group wrote[7]. Other researchers use the C version of the Olden suite to evaluate optimizations for pointer-based programs [8, 18, 24]. Table 1 lists the benchmarks we use in our experiments, along with characteristics about each program. We compile the programs using JDK 1.1.6. The lines of code (LOC) number excludes comments and blank lines. Since our prefetching techniques do not attempt to improve allocation and garbage collection performance, we disable garbage collection which often increases the execution time substantially. We plan to address the interaction between garbage collection and prefetching in future work.

We use RSIM to perform a detailed cycle by cycle simulation of our programs [22]. RSIM simulates an aggressive,

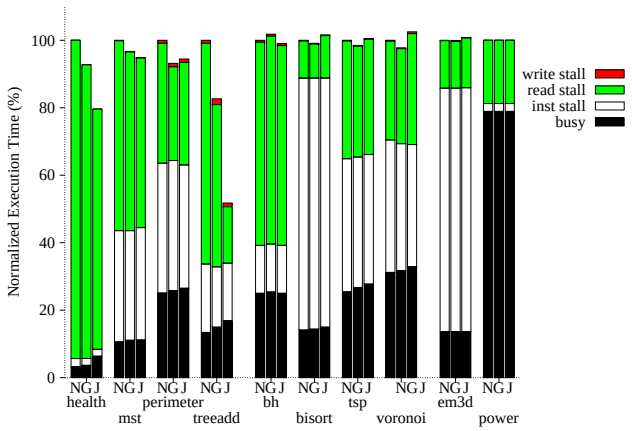


Figure 6. Prefetching performance

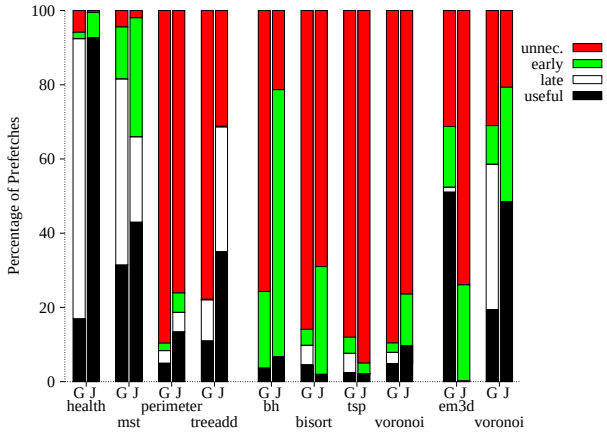


Figure 7. Prefetch effectiveness (L1)

out-of-order processor that issues up to 4 instructions per cycle, and has a 64 entry instruction window. The functional units include 2 ALU, 2 FP, 1 branch, and 2 address units. We use the default values for most of the parameters except for the memory hierarchy which we list below.

L1 Cache	32KB, direct WT, split, 32B line
L2 Cache	256KB, 4-way, WB, unified, 32B line
L1 Cache Ports	2
L1/L2/Mem hit time	1/12/60 cycles
Miss Handlers (MSHR)	8 L1, 8 L2
Memory Bandwidth	32B/cycle

We have also run experiments with different cache configurations. In general, changing the cache size and associativity parameters does not significantly affect the overall performance results. When we increase the cache size or set associatively, our prefetching results tend to improve slightly, until the cache grows large enough to eliminate most capacity misses.

In RSIM, a prefetch instruction causes one cache line to be brought into the L1 cache. If an object is larger than a cache line, our compiler issues multiple prefetch instructions.

Overall results: Figure 6 shows the results of greedy (G) and jump pointer (J) prefetching. We normalize the results to those without prefetching (N). For jump-pointer prefetching, we use a prefetch distance of 8 objects, except for `mst` which uses a distance of 2. In our programs, using a distance value greater than 8 does not improve performance significantly. The compiler could compute this distance based upon the number of instructions between accesses, but we have not implemented this cost model. Jump-pointer and greedy prefetching improve performance as much as 48% and 18%, respectively. Across all benchmarks, we see improvements of 10% for jump-pointer and 4% for greedy prefetching using the geometric mean.

Both prefetching techniques produce noticeable improvements on four of the ten programs, *i.e.*, `health`, `mst`, `perimeter`, `treeadd`. Either greedy or jump-pointer prefetching slightly improves performance in `bh`, `bisort`, `tsp`, and `voronoi`. We describe each program that shows an improvement below. Neither prefetching technique is able to improve `em3d` and `power` because the number of cache misses in very low in these programs so most of the prefetches hit in the L1 cache. Our prefetching results are similar to those reported in related work for linked structures in C programs [18, 24].

We present prefetching effectiveness and cache statistics next. Then, we discuss the performance of the individual programs in more detail.

Prefetching effectiveness: Figure 7 divides the dynamic prefetches into several categories. A *useful* prefetch arrives on time and is accessed. The latency of a *late* prefetch is only partially hidden because a demand request occurs while the memory system retrieves the cache line. The cache replaces an *early* prefetch before the cache line is used, or when the line is never used. An *unnecessary* prefetch hits in the cache, or is coalesced into an MSHR. Figure 7 categorizes the L1 cache prefetches for greedy (G) and jump-pointer (J) prefetching. Useful, late, and early prefetches require accesses to the next level in the memory hierarchy.

Table 2 lists the number of static and dynamic prefetches that our benchmarks generate. The static value represents the number of compiler generated prefetches. The dynamic value is the number of prefetches issued at run-time. The value in parentheses represents the number of prefetches as a percentage of the demand reads. Finally, the third value is the percentage increase in bus bandwidth due to prefetching. Although bus traffic increases due to prefetching, the maximum bus utilization with and without prefetching is 46% and 25%, respectively. The dynamic prefetching counts show why prefetching is ineffective on `em3d` and `power`; these programs do not frequently access linked structures.

Cache statistics: Figures 8 and 9 display cache hit and miss statistics in the L1 and L2 caches, respectively. From

Table 1. Current benchmark suite

Name	Main Data Structure(s)	LOC	Methods	Inputs	Inst. Issued	Total Memory	Bytes /Obj.
bh	oct-tree, linked list	487	74	4096 bodies, 2 iters.	731 M	415 MB	28
bisort	binary tree	164	14	100,000 numbers	1292 M	1.5 MB	24.1
em3d	linked list	182	22	2000 nodes, 100 degree, 4 iters.	2120 M	6.5 MB	413
health	quad-tree, linked list	279	36	5 levels, 500 iters.	366 M	22 MB	19.4
mst	hashtable	183	26	1024 nodes	955 M	13 MB	25.9
perimeter	quad-tree	242	47	4K x 4K image	188 M	3.4 MB	32
power	tree	347	30	10K customers	2086 M	24 MB	32
treeadd	binary tree	81	11	20 levels	168 M	24 MB	24
tsp	binary tree, linked list	289	15	60,000 cities	787 M	14 MB	37
voronoi	binary tree	526	70	20,000 points	848 M	35 MB	25

Table 2. Prefetch statistics

Program	Greedy			Jump		
	static	dynamic (10^3)	band. (%)	static	dynamic (10^3)	band. (%)
health	14	10518 (20%)	8.0	10	10204 (14%)	40.3
mst	3	2949 (8%)	9.2	3	2950 (9%)	19.9
perimeter	17	2523 (9%)	8.3	8	1001 (4%)	33.5
treeadd	2	2178 (23%)	21.1	1	845 (8%)	93.8
bh	34	5719 (4%)	23.2	3	759 (.6%)	11.9
bisort	8	7296 (31%)	19.6	4	3601 (13%)	77.2
tsp	20	12201 (20%)	46.9	18	21837 (32%)	76.9
voronoi	15	1155 (1%)	2.5	13	724 (.9%)	22.9
em3d	20	56 (.06%)	0.4	20	56 (.06%)	4.7
power	4	53 (.02%)	0.2	4	53 (0.2%)	1

bottom to top, we categorize each reference as a cache *hit*, *coalesce* miss, *capacity* miss, or a *conflict* miss. A coalesce miss hits in the MSHR due to a prior miss that the memory subsystem is transferring into the cache. The cache statistics mirror the overall results such that when prefetching improves overall performance, the combined cache hit and coalesced reference rate increases as well. In `treeadd`, for example, prefetching is able to eliminate almost all capacity and coalesce misses in the L1 cache, and also significantly improve the L2 cache utilization.

`health`: Both prefetching schemes significantly improve `health` which is not surprising considering its poor locality. `Health` is the only program that builds jump-pointers while traversing its linked structures, and contains indirect recurrent pointer variables. The jump-pointers eliminate all of the late prefetches, but we do not see larger improvements due to the overhead of updating the jump-pointers at traversal time. It is important that the compiler generate the jump-pointers correctly. If the jump-pointers are added during object creation, performance degrades by 13%.

`mst`: The linked structure is a hashtable. We see performance improvements because the hashtable is small and each hash entry contains several objects. Jump-pointer prefetching increases the number of useful prefetches, but the number of early prefetches also increases because the objects at the end of each list are not useful jump-pointers. We create the jump-pointers in the reverse direction because new elements are added to the beginning of each list. We must limit the jump-pointer prefetch distance to see an improvement because each linked list is small.

`perimeter`: We obtain performance improvements even though most of the prefetches hit in the L1 cache. Adding the jump-pointer field increases the object size from 32B to 40B, so we prefetch two cache lines instead of just one. Prefetching the extra cache line reduces the L2 hit rate relative to the greedy prefetching results. Prefetching the extra cache line does help; if we only prefetch one cache line, then performance only improves by 1% instead of 5.5%.

`treeadd`: We see the largest performance improvements for both prefetching schemes. `Treeadd` is a very simple program that creates a binary tree and traverses the tree in the same order. Figure 7 illustrates that only a small number of useful prefetches are necessary to obtain improvements. Increasing the prefetch distance to 16 objects almost eliminates the late prefetches with a small run-time improvement.

`bh,bisort,tsp,voronoi`: Performance slightly degrades in `bh` using greedy prefetching because our compiler issues too many prefetches. `Bh` contains an oct-tree and each node is larger than a cache line, resulting in 16 prefetches for each node. The performance slightly improves when we restrict the number of prefetches generated for each node. Using jump-pointers reduces the number of prefetches to 2 and results in a slight improvement. Performance decreases in `bisort`, `tsp`, and `voronoi` using jump-pointers because these programs either update their linked structures or contain conflicting traversal patterns. For example, `bisort` first traverses a binary tree in one direction, and then traverses the tree in the opposite direction. Obtaining further improvements in `bisort` is difficult because the read stall percentage is low. In `tsp`, the number

Table 3. Static prefetch statistics

Program	IP M	IP P	Fields (F)	Intra (I)	Inline	Array size	Total
health	8		5	1	5 (F)	4 (IP)	14
mst	3				3 (I)		3
perimeter	9	8					17
treeadd	2						2
bh	16	8	10		10 (F)	8 (IP)	34
bisort	4			4			8
tsp	6		14				20
voronoi	14		1		14 (I)		15
em3d			20		20 (F)		20
power	4						4

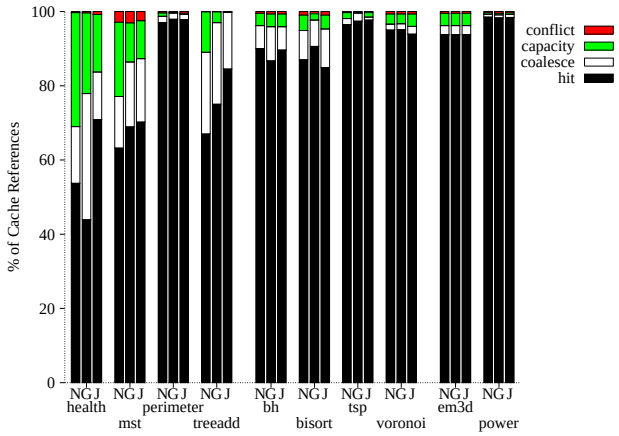


Figure 8. L1 cache statistics

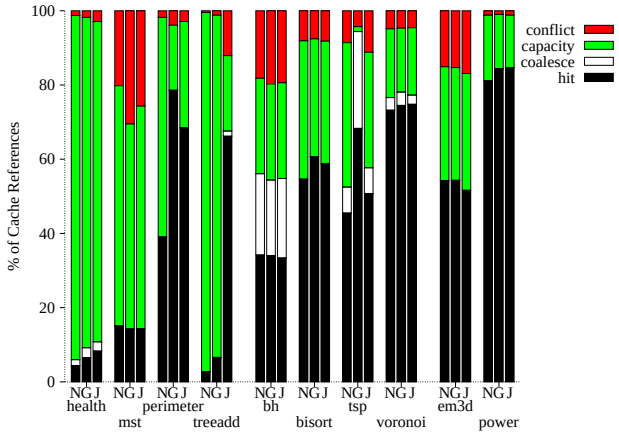


Figure 9. L2 cache statistics

of unnecessary prefetches is high because the hit rate in this program is high. Greedy prefetching improves `voronoi` slightly even though the number of useful prefetches is low.

Analysis features Table 3 shows the features of our recurrent analysis that are responsible for generating static prefetch instructions in our greedy prefetching scheme. We show the contribution from interprocedural analysis (*IP*), analyzing stores into fields (*F*), and intraprocedural analysis only (*I*). The total number of greedy prefetches is *IP*+*F*+*I*. Interprocedural class analysis divides the IP results into monomorphic (*M*) and polymorphic (*P*) recursive method calls. The *Inline* column shows how inlining affects our recurrent analysis. For example, in `health`, we generate the five prefetches in the *Field* column only when inlining is enabled and interprocedural analysis is disabled. In `mst`, the compiler still generates the prefetches if we perform intraprocedural analysis only, and inlining is enabled. The *Array size* column shows the results of our analysis for computing the array sizes. Only two of the programs, `health` and `bh`, use constant size arrays to represent *n*-ary trees.

Table 3 shows that both interprocedural analysis and analyzing stores into fields are important in our Java programs. Intraprocedural analysis rarely discovers recurrent accesses on its own. In `health`, most of the improvement comes from analyzing field stores. When we disable field stores, performance improves by less than 1% instead of 7%. In `perimeter`, the prefetches in the monomorphic and polymorphic recursive calls contribute 2% and 4%, respectively.

7. Related work

In this section, we describe the related work on prefetching linked data structures. We also describe related compile-time analyses for identifying linked structures. We omit prior work on array prefetching.

Luk and Mowry first developed an effective compiler technique for software prefetching recursive data structures in C programs [18, 17]. Their prefetching techniques include *greedy* prefetching, *history-pointer* prefetching, and *data-linearization*. They show that prefetching improves performance on programs that traverse linked data structures. Luk presents a compiler implementation of history-pointer (jump-pointer) prefetching in his dissertation [17]. Our contributions include a new intra and interprocedural data flow analysis for discovering objects to prefetch, and an evaluation on a suite of Java programs. Luk and Mowry do not perform interprocedural analysis, but they do detect self-recursive calls. Our analysis works in the presence of virtual method calls, and when data flow facts are assigned to object fields and arrays. We also detect indirect recurrent pointer variables. We developed our implementation of jump-pointer prefetching simultaneously with Luk. Other researchers present techniques for prefetching linked structures, but they do not implement both prefetching techniques in an optimizing compiler [16, 25, 5, 15, 28].

Roth and Sohi introduce several hardware and software jump-pointer prefetching mechanisms for linked data structures [23, 24]. They manually insert instructions to create jump-pointer structures and issue prefetches. In our work, the compiler automatically generates instructions to create jump-pointers and issue prefetches. Other

hardware approaches for prefetching linked structures include [13, 20, 29, 30].

Our analysis for identifying accesses to linked data structures is related to the work on identifying shapes in heap allocated structures, called *shape analysis* (*e.g.*, see [26, 12]). A main difference is that we analyze traversal sites while shape analysis analyzes creation sites to identify pointer relationships. Our analysis requires less information since we only discover how pointers are used instead of recording all pointer relationships. Benedikt et al. define a logic for describing linked data structures which they briefly mention can be used for identifying recurrent accesses [3].

The work by Hwang and Saltz on identifying def-use expressions in dynamic data structures is most closely related to our data flow analysis [14]. They introduce the term loop induction pointer to identify accesses to linked data structures. Their backward data flow analysis creates full interprocedural def-use chains, and a separate algorithm finds the induction pointers by detecting cycles in the def-use chains. Our forward data flow analysis maintains less information by computing the recurrent access status of objects rather than maintaining complete interprocedural def-use chains. Stroutchinin et al. also prefetch simple linked lists occurring in loops by identifying induction pointers [28]. Their prefetching technique works when list elements are allocated contiguously in memory.

A recurrent pointer variable is similar to a loop induction variable. A loop induction variable is a variable that is incremented by a constant amount during each loop iteration[1, 11]. Existing techniques identify induction variables by discovering cycles in the data dependence graph and explicitly require the loop structure of the program. The classic algorithm for finding induction variables must first compute reaching definitions and loop invariant variables [1]. We formalize our analysis using data flow equations and we do not require a data dependence graph.

To summarize, our work has several unique contributions. We develop a novel interprocedural data flow analysis for discovering linked structure traversals which we use in our compile-time prefetching algorithms. We implement and evaluate two different prefetching techniques for linked data structures, greedy prefetching and jump-pointer prefetching. We evaluate both of them on a suite of Java programs.

8. Conclusion and future work

In this paper, we introduce an effective data flow analysis for identifying linked data structure traversals in object-oriented programs using intra and interprocedural analysis. We use our analysis to implement greedy and jump-pointer prefetching in a Java compiler, and evaluate the effectiveness of prefetching on a suite of pointer intensive, object-oriented programs. Both prefetching techniques improve performance in four of our ten programs. In one program, jump-pointer prefetching improves performance by 48%, but obtaining large improvements consistently is difficult. One reason that prefetching is not more effective is that several of our programs do not spend enough time accessing linked structures. Prefetching is most effective on programs with poor locality during linked structure traversals.

Greedy prefetching often improves the performance our programs even in the presence of object-oriented features, such as encapsulation, that hide accesses to underlying data structures. As memory latency increases, greedy prefetching will become less effective in improving memory performance. Our jump-pointer implementation has the potential for bigger improvements than greedy prefetching, but it is less consistent overall. Better compiler analysis is necessary to improve jump-pointer prefetching. Even with prefetching, our results show there is considerable room for improving the locality of object-oriented programs.

We are currently addressing improvements to our prefetching techniques and experimental methodology. We plan to automate our jump-pointer prefetching implementation to choose the location for building the jump-pointer links, and the distance and direction of the links. Choosing the appropriate location and direction is feasible, but consistently choosing the distance is difficult due to data dependent data structures such as hashtables. We are developing techniques to take advantage of the garbage collector to improve prefetching. We need to change our copying garbage collector to recognize jump-pointer hints, and update the links appropriately when copying linked structures. We are investigating the applicability of prefetching to a wider variety of Java programs, such as the SPECjvm98 benchmark suite. We notice that the SPECjvm98 programs frequently use arrays instead of linked data structures. We are in the process of extending our recurrent analysis to recognize simple array access patterns, and we will prefetch array elements and objects referenced by array elements.

Acknowledgments

Thanks to Sam Guyer and Calvin Lin for reading drafts of this work. We thank the anonymous referees for their helpful comments. This work is supported by NSF grant EIA-9726401, NSF Infrastructure grant CDA-9502639, NSF ITR grant CCR-0085792, and Darpa grant 5-21425. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, pages 19–38, Sept. 1999.

[3] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proceedings of ESOP '99: European Symposium on Programming*, Mar. 1999.

[4] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, June 1995.

[5] B. Cahoon and K. McKinley. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*, Oct. 1999.

[6] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Apr. 1991.

[7] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, July 1995.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[9] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 83–100, Oct. 1996.

[10] J. Dolby and A. A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[11] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1), 1995.

[12] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996.

[13] L. Harrison and S. Mehrotra. A data prefetch mechanism for accelerating general-purpose computation. Technical Report CSRD Technical Report 1351, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1994.

[14] Y.-S. Hwang and J. Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 131–145, Aug. 1997.

[15] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, Jan. 2000.

[16] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, 1995.

[17] C.-K. Luk. *Optimizing the Cache Performance of Non-Numeric Applications*. PhD thesis, University of Toronto, Department of Computer Science, 2000.

[18] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[19] N. McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.

[20] S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic And Numeric Computation*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Apr. 1996.

[21] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–72, Oct. 1992.

[22] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, Aug. 1997.

[23] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[24] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[25] S. Rubin, D. Bernstein, and M. Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 259–273. Springer, Mar. 1999.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[27] O. Shivers. Control flow analysis in scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[28] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In *Proceedings of the 10th International Conference on Compiler Construction*, Apr. 2001.

[29] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[30] L. Zhang, S. A. McKee, W. C. Hsieh, and J. B. Carter. Pointer-based prefetching within the Impulse adaptable memory controller: Initial results. In *Proceedings of the Workshop on Solving the Memory Wall Problem*, June 2000.