CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **00**:1–7 *Prepared using* `cpeauth.cls` *[Version: 2002/09/19 v2.02]*

Recurrence analysis for effective array prefetching in Java

Brendon Cahoon[1] Kathryn S. McKinley[2*],

[1]*IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10533 brendon@us.ibm.com*
[2]*Department of Computer Sciences, University of Texas at Austin, Austin, TX 78757 mckinley@cs.utexas.edu*

SUMMARY

Java is an attractive choice for numerical, as well as other, algorithms due to the software engineering benefits of object-oriented programming. Because numerical programs often use large arrays that do not fit in the cache, they to suffer from poor memory performance. To hide memory latency, we describe a new unified compile-time analysis for software prefetching arrays and linked structures in Java. Our previous work uses data-flow analysis to discover linked data structure accesses. We generalize our prior approach to identify loop induction variables as well, which we call recurrence analysis. Our algorithm schedules prefetches for all array references that contain induction variables. We evaluate our technique using a simulator of an out-of-order superscalar processor running a set of array-based Java programs. Across all our programs, prefetching reduces execution time by a geometric mean of 23%, and the largest improvement is 58%. We also evaluate prefetching on a PowerPC processor, and we show that prefetching reduces execution time by a geometric mean of 17%. Because our analysis is much simpler and quicker than previous techniques, it is suitable for including in a just-in-time compiler. Traditional software prefetching algorithms for C and Fortran use locality analysis and sophisticated loop transformations. We further show that the additional loop transformations and careful scheduling of prefetches from previous work are not always necessary for modern architectures and Java programs.

KEY WORDS: Memory optimization, Array prefetching, Static analysis

1. Introduction

The flexibility and expressiveness of object-oriented languages, such as Java, provide software engineering benefits over traditional scientific languages. Because of these advantages, programmers are increasingly using them to solve problems that also require high performance. Scientific and more general purpose Java applications frequently use array data structures. Traditional approaches improve performance in array-based applications using loop transformations, such as loop tiling and unrolling.

*Correspondence to: Department of Computer Sciences, University of Texas at Austin, Austin, TX 78757 mckinley@cs.utexas.edu

Received 15 January 2003
Revised 26 August 2003

Implementing loop transformations in Java compilers is challenging due to Java array requirements and exceptions [4].

Memory latency is a barrier to achieving high performance in Java programs, just as it is for C and Fortran. To solve this problem, we investigate software controlled data prefetching to improve memory performance by tolerating cache latency. The goal of prefetching is to bring data into the cache before the demand access to that data. Software prefetching techniques identify appropriate data access patterns, and generate a prefetch instruction for data that will be accessed in the future. Prior research shows that software controlled prefetching is effective in array-based Fortran programs [6, 9, 22, 26]

We describe a new data-flow analysis to identify monotonic loop induction variables, and a method to schedule prefetches for array references that contain induction variables in the index expression. We rely on a simplified form of common subexpression elimination to remove redundant prefetches. Our new recurrence analysis unifies the identification of induction variables and linked data structure traversals, and thus subsumes our previous work that discovers regular accesses to linked structures [8]. In this article, we focus on arrays.

We implement our data-flow analysis and software prefetching technique in Vortex, an optimizing (ahead-of-time) compiler for object-oriented programs [13]. We evaluate prefetching using benchmark programs from the Jama library [16] and the Java Grande benchmark suite [7]. We run the programs on RSIM, a simulator for an out-of-order superscalar processor [27], and on an actual PowerPC processor [24].

Our results show that our simple prefetching implementation is very effective on array-based Java programs on an aggressive out-of-order processor. Prefetching improves performance by a geometric mean of 23% on a simulated processor and by 17% on a PowerPC processor. We see even larger improvements on several kernels, including matrix multiplication, LU factorization, SOR, and Cholesky factorization. In SOR, prefetching eliminates all memory stalls and reduces execution time by 46%. Performance degrades in one program, FFT, due to a large number of conflict misses caused by a power of two data layout and access of a large 1-D array that make prefetching counterproductive. We also include a case study of matrix multiply to explore the utility of additional loop transformations to more carefully schedule prefetches in the spirit of Mowry et al. [26]. In matrix multiplication, we show that locality analysis and loop transformations do improve performance on a simple, in-order processor. On modern architectures with multiple functional units and out-of-order execution, we find that prefetching in Java programs is less sensitive to precise scheduling via loop transformations, but loop transformations will provide further improvements in some cases.

Our technique is much simpler and faster than existing array software prefetching techniques because it does not require array dependence testing or loop transformations. These characteristics make it suitable for a just-in-time (JIT) compiler, but we leave that evaluation for future work. Existing compilers that find induction variables can implement the prefetch scheduling technique with little effort.

We make the following contributions.

1. A new method for detecting monotonic induction variables. Our approach uses data-flow analysis and does not require explicit def-use chains.
2. A unified analysis for discovering recurrences in linked structures and arrays. We use the same analysis for identifying prefetch opportunities in linked structures and arrays.

3. A new technique for prefetching arrays. We prefetch array references that contain induction variables. We do not require array dependence analysis or loop transformations.
4. An evaluation of prefetching on scientific Java programs from the Jama library and Java Grande benchmark suite. We obtain detailed performance statistics using a simulator and validate the results on a PowerPC processor.

We organize the rest of the article as follows. Section 2 briefly describes the related work. In Section 3, we present our recurrence analysis that discovers loop induction variables. Section 4 describes our algorithm for scheduling prefetch instructions for array references. We present experiments showing the effectiveness of our prefetching algorithm in Section 5. The results include overall performance, prefetch instruction effectiveness, and cache statistics. Section 6 presents a case study that compares the performance effect of different loop transformations and analysis on prefetching in matrix multiplication. We discuss how our prefetching technique works on arrays of objects, true multidimensional arrays, and `Enumeration` types in Section 7.

2. Related Work

In this section, we describe the related work on software prefetching in array-based programs. VanderWeil and Lilja provide a detailed survey of hardware and software data prefetching techniques [33]. We also describe previous approaches for discovering loop induction variables. Other researchers investigate optimizing high performance Java applications using traditional loop optimizations [4, 10]. Their work is complimentary to our work.

2.1. Software Prefetching for Arrays

Mowry, Lam, and Gupta describe and evaluate compiler techniques for data prefetching in array-based codes [25, 26]. Their paper is one of the first to report execution times for compiler inserted prefetching. The algorithm works on affine array accesses, and involves several steps. First, the compiler performs locality analysis to determine array accesses that are likely to be cache misses. The compiler uses loop splitting to isolate references that may be cache misses, and software pipelining to schedule prefetch instructions effectively. Bernstein, Cohen, Freund, and Maydan implement a variation of Mowry et al.'s algorithm for the PowerPC architecture [6]. The only loop transformation they use is unrolling. McIntosh extends Mowry et al.'s work by focusing on the compiler support necessary for software prefetching [22]. He develops several new compiler techniques to eliminate useless prefetches and to improve prefetch scheduling for array-based codes. Our prefetching algorithm is effective without locality analysis and loop transformations. We focus on Java arrays that contain features that make code and data transformations challenging compared to transforming Fortran arrays.

Callahan, Kennedy, and Porterfield present and evaluate a simple algorithm for prefetching array references one loop iteration ahead [9]. Their paper illustrates the potential for software prefetching, but they present cache miss rate results only. Santhanam, Gornish, and Hsu evaluate software prefetching for Fortran programs on the HP PA-8000, a 4-way superscalar processor [28]. The compiler creates symbolic address expressions for array references that contain induction variables. The algorithm requires analysis of the symbolic addresses to generate prefetch instructions. The compiler also performs loop optimizations. Selvidge presents profile-guided software data prefetching as a scheduling algorithm [29]. Selvidge uses profile information to identify regular and irregular data references. We propose a compile-time analysis for prefetching array references and linked structures. We prefetch all array references containing loop induction variables, and we eliminate redundant prefetches using common subexpression elimination.

2.2. Hardware Prefetching for Arrays

Hardware prefetching schemes add prefetching functionality without explicit programmer or compiler assistance. The main benefit of hardware schemes is the ability to run existing program binaries, which enables prefetching without recompiling the program. A simple hardware scheme prefetches the next cache line when a program loads a cache line from memory [30]. This simple mechanism often results in cache pollution because it is too aggressive. More sophisticated hardware prefetching mechanisms predict and prefetch data access streams based upon past references [5, 19].

Several existing high performance architectures implement hardware prefetch mechanisms, but there is a large variation in the functionality among these architectures. Some architectures prefetch into the 1st level cache (*e.g.*, the POWER4 [32]), some prefetch into the 2nd level cache (*e.g.*, the Pentium 4 [18]), others prefetch into a special buffer (*e.g.*, the UltraSPARC III [31]), and some prefetch floating point data only (*e.g.*, the UltraSPARC III).

A software prefetch mechanism requires less complexity than a hardware mechanism. Software prefetching is also more flexible by allowing the compiler to determine what and when to prefetch. Although hardware mechanisms work well on array-based programs that access data using a predictable unit stride, software prefetching may be able to identify different stride patterns and still produce similar performance improvements for the simple cases. Our software prefetching technique does not rely upon the order of the arrays in memory and is able to detect accesses to arrays of objects.

2.3. Induction Variables

The initial use for induction variable detection in compilers was operator strength reduction [1, 21]. These algorithms typically require reaching definitions and loop invariant expressions. The algorithms are conservative and find simple linear induction variables. The PTRAN compiler uses an optimistic approach and assumes variables in loops are induction variables until proven otherwise [2]. Gerlek, Stoltz, and Wolfe present a demand driven static single assignment (SSA) approach for detecting general induction variables by identifying strongly connected components in the SSA graph [14]. Gerlek et al. present a lattice for classifying different types of induction variables. Haghighat and Polychronopoulos also categorize different types of induction variables for use in parallelizing compilers [15]. Ammarguellat and Harrison describe an abstract interpretation technique for detecting general recurrence relations, which includes induction variables [3]. The approach requires a set of patterns, which they call templates, that describe the recurrences.

Wu, Cohen, and Padua describe a data-flow analysis for discovering loop induction variables [34]. The analysis computes whether a variable increases or decreases along a given execution path, and the minimal and maximal change in a value. The authors compute closed form expressions from the distance intervals to perform array dependence testing. We present a different data-flow analysis for identifying induction variables, and we use the analysis to discover both induction variables and linked structure traversals. We use the analysis for prefetching rather than array dependence testing.

3. Detecting Loop Induction Variables

In this section, we present our data-flow analysis for discovering loop induction variables. Traditional algorithms for finding induction variables are either loop based [1], or use static single assignment (SSA) form [14]. Both techniques require def-use or use-def chains.

A basic type of an induction variable is one that is incremented or decremented by the same value during each loop iteration, for example the expression `i=i+c` occurring in a loop. During each iteration of the loop, the variable `i` is incremented by a loop invariant value, `c`.

Gerlek et al. define an induction variable as a specific kind of *sequence* variable [14]. An expression `v=e` that occurs at statement `s` in a loop is a sequence variable if the expression `e` contains a reference to variable `v` or the expression `e` contains a reference to a difference sequence variable. There are several classifications of induction variables. A *linear* induction variable changes by the addition or subtraction of a loop invariant value in every iteration. A *polynomial* induction variable changes by a linear induction variable using addition or subtraction in each iteration. An *exponential* induction variable changes by the multiplication of a loop invariant expression in each iteration. Our analysis discovers each of these types of induction variables. A *monotonic* induction variable is one that is conditionally executed in the loop body.

We first define our intraprocedural analysis, and we present a simple example to illustrate the analysis. We then describe another technique for discovering induction variables that uses explicit data flow edges [14]. Finally, we describe the similarities between the induction variable and linked structure analysis. We propose a unified analysis, called recurence analysis, that detects both types of traversal patterns.

3.1. Intraprocedural analysis

The intraprocedural recurrence analysis detects induction variables that are due to loops. We base the loop induction variable detection analysis on our similar analysis that discovers linked structure traversals [8]. We use the same analysis framework that requires only minor extensions to discover loop induction variables.

We assume the reader is familiar with the fundamentals of a monotone data-flow analysis framework [20]. Our data-flow analysis is a *may* analysis, which means if a variable is incremented or decremented along at least one path through a loop, then we classify it as an induction variable. The traditional induction variable analysis is a *must* analysis. Our analysis is *pessimistic*, which means the analysis starts with a small set of facts and adds information until reaching a fixed point [11]. An *optimistic* analysis starts with a large set of facts and removes information until reaching a fixed point.

We define the following sets in our data-flow analysis. Let V be the set of integer variables in a method, E be the set of expressions, S be the set of statements in the method, and IS be the induction status that we describe below. We restrict V to integers because we are not interested in floating point induction variables. The analysis associates a set of tuples with each statement:

$$I \subseteq \mathcal{P}(V \times E \times S \times IS)$$

We define a function, IA, that maps program statements to the analysis information, $IA : S \rightarrow I$. The induction expression (E) contains the computation that the program performs during each loop iteration. The analysis begins by creating a simple binary expression. The analysis creates larger expressions from the smaller expressions to form complex expressions. We use the statement number (S) to handle the case when there are multiple assignments to a variable. For example, if the sequence `j=j+1; j=j+1` is not in a loop, we do not want to mark variable `j` as an induction variable. If the two increment instructions are in a loop, then the analysis identifies variable `j` as an induction variable.

The induction status (IS) indicates when a program uses an induction variable. Let $is \in IS = \{$ `ni`, `pi`, `i` $\}$. We order the elements of IS such that `ni` $\prec$ `pi` $\prec$ `i`. We define the element values as follows:

Not induction (`ni`). The initial value indicates a variable is not an induction variable.

Possibly induction (`pi`). The first time we process an expression it is potentially an induction expression.

Induction (`i`). A variable is an induction variable.

The first time we analyze a loop, a variable occurring on the left hand side of a binary expression becomes *possibly induction* (*e.g.*, `t=j+1`). On the second iteration of the analysis, the variable on the left hand side becomes *induction* if the variable in the expression (*i.e.*, `j`) is *possibly induction*. If the variable is *not induction*, then `t`'s value remains the same.

Informally, the data-flow join function is set-union except for tuples that share the same variable name, expression, and statement number. The induction status values and the join operator define *induction* (`i`) as the conservative value. Since it is a may analysis, a variable is a monotonic induction variable if it contains the *induction* (`i`) status along at least one path. Our definition enables the analysis to discover induction variables in the presence of conditional execution. Our analysis also detects if a variable changes by different expressions along different paths through the same loop.

Formally, we define the data-flow join operation, $I_1 \sqcup I_2$, as follows. Let $t \in I$ be a tuple in the set I. Given our ordering of the elements $is \in$ IS, $is \sqcup \texttt{ni} = is$, $\texttt{pi} \sqcup \texttt{pi} = \texttt{pi}$, and $is \sqcup \texttt{i} = \texttt{i}$. Then,

$$\begin{aligned} I_1 \sqcup I_2 &= \{t | t \in I_1 \wedge t \notin I_2\} \cup \{t | t \notin I_1 \wedge t \in I_2\} \cup \\ &\quad \{(\mathrm{v,e,s,is_1} \sqcup \mathrm{is_2}) | (\mathrm{v,e,s,is_1}) \in I_1 \wedge (\mathrm{v,e,s,is_2}) \in I_2\} \end{aligned}$$

The data-flow equations for the induction analysis are:

$$\begin{aligned} IA_{in}(s) &= \bigsqcup_{p \in pred(s)} IA_{out}(p) \\ IA_{out}(s) &= (IA_{in}(s) \setminus \textsc{kill}_{IA}(s, IA_{in}(s))) \sqcup \textsc{gen}_{IA}(s, IA_{in}(s)) \end{aligned}$$

The functions GEN and KILL operate on the set of tuples for each statement and produce a new set of tuples, $\textsc{gen}_{IA}, \textsc{kill}_{IA} : S \times I \rightarrow I$.

At the initial statement, init(S), the value for the function IA_{in} is $\{(v, \emptyset, \emptyset, \texttt{ni}) | v \in V\}$, which means that the set of expressions and statements for each variable is the empty set and the status is *not induction*.

We define data-flow transfer functions for binary expressions and assignments. Our intermediate representation only requires statements that contain binary expressions and assignments. For each statement, the representation includes the set of predecessors. Our analysis does not require that loops be identified explicitly. We describe the details of our GEN and KILL functions for each interesting program statement below. In the definitions below, e' $\in E$, s' $\in S$, and is' $\in IS$.

`v = j` *op* $\texttt{c}_k$ An integer binary expression *op* involving two variables `j` and `c` at statement `k` may create an induction variable when it occurs in a loop. The value `c` is loop invariant for linear induction variables only. For polynomial and exponential induction variables, the value `c` is an induction variable. Informally, the expression is an induction variable when the value assigned to variable `v` is propagated to `j`, the variable on the right hand side. The canonical example is `j=j+1` in a loop with no other assignments to `j`. The KILL$_{IA}$ and GEN$_{IA}$ functions for a binary expression are:

$$\begin{aligned}
\textsc{kill}_{IA}(\text{v=j op c}_\text{k}, I) &= \{(\text{v,j op c,k},\texttt{pi})\ ,(\text{v},\emptyset,\emptyset,\texttt{ni})\} \\
\textsc{gen}_{IA}(\text{v=j op c}_\text{k}, I) &= \begin{cases} \{(\text{v,j op c,k},\texttt{pi})\} & : \text{ if } (\text{j},\emptyset,\emptyset,\texttt{ni}) \in I \\ \{(\text{v,j op c,k},\texttt{i})\} & : \text{ if } (\text{j, j op c,k},\texttt{pi}) \in I \\ \emptyset & : \textit{otherwise} \end{cases}
\end{aligned}$$

The first time we process a binary expression, we create a tuple containing variable `v`, the expression `j op c`, with the *possibly induction* (`pi`) status. If the expression occurs in a loop, the analysis processes the statements again because the data-flow information has not yet reached a fixed point. In the next iteration of the analysis, if there exists a tuple containing variable `j` with the *possibly induction* (`pi`) status, then there is no intervening assignment to variable `j` in the loop body. In this case, we create a tuple containing variable `v`, the expression `j op c`, with the *induction* (`i`) status.

Binary expressions require an additional GEN and KILL function for the *operands*. Propagating information about the operands enables the analysis to create complex induction variable expressions, such as mutual induction variables.

$$\begin{aligned}
\textsc{kill}_{IA}(\text{v=j op c}_\text{k}, I) &= \{(\text{v,e' op c,l,is'}) | \text{l} \neq \text{k}\} \\
\textsc{gen}_{IA}(\text{v=j op c}_\text{k}, I) &= \{(\text{v,e' op c,l,is'}) |\ (\text{j,e',l,is'}) \in I \wedge \text{l} \neq \text{k}\}
\end{aligned}$$

The analysis only creates a complex induction variable `e'` once at a statement `k`. If the analysis evaluates the statement again, the complex induction expression is not changed. Otherwise, the analysis would add the variable `c` to the expression `e'` each time it processes the statement so the analysis would never terminate.

`v=u` A variable assignment expression copies the induction information associated with variable `u` to variable `v`. For each tuple containing a variable `u`, we create a new tuple containing variable `v` with the same expression, statement, and induction status as variable `u`. We kill the old information associated with variable `v`. The KILL$_{IA}$ and GEN$_{IA}$ functions for an assignment are:

$$\begin{aligned}
\textsc{kill}_{IA}(\text{v=u}, I) &= \{(\text{v,e',s',is'})\} \\
\textsc{gen}_{IA}(\text{v=u}, I) &= \{(\text{v,e',s',is'}) |\ (\text{u,e',s',is'}) \in I\}
\end{aligned}$$

`v=`*expr* Any other assignment of an expression *expr* to a variable `v` kills the analysis information for variable `v`. Our analysis sets the induction status of any tuple containing variable `v` to *not induction* (`ni`). The KILL$_{IA}$ and GEN$_{IA}$ functions for all other assignments are:

$$\begin{aligned}
\textsc{kill}_{IA}(\text{v=}\textit{expr}, I) &= \{(\text{v,e',s',is'})\} \\
\textsc{gen}_{IA}(\text{v=}\textit{expr}, I) &= \{(\text{v},\emptyset,\emptyset,\texttt{ni}) | (\text{v,e',s',is'}) \in I\}
\end{aligned}$$

3.1.1. *Object Fields*

The analysis tracks the data-flow information for variables assigned to object fields. An example of an induction variable that is stored in an object field is `o.f=o.f+1` when occurring in a loop. For example, an induction variable in an object field occurs when using the `Enumeration` class for a `Vector` object as we show in Figure 16.

For object fields, we associate the analysis information with the field name, and we ignore the base object. The implication of ignoring the base object is that all objects of the same class are aliases. We prepend the name with its class name to avoid ambiguity between fields from different classes. We could improve the precision by tracking the aliases between base objects, which would increase the analysis complexity cost. Our definition also ignores the effects of other threads, which means that we may indicate that a field contains an induction variable when another thread assigns a different value to the field.

We define GEN_{IA} and KILL_{IA} for object field assignments.

`o.f = v` Create data-flow information for a field definition. The GEN_{IA} and KILL_{IA} functions are similar to a variable assignment.

$$\begin{aligned}\text{KILL}_{IA}(\text{o.f=v},I) &= \{(\text{f,e',s',is'})\}\\ \text{GEN}_{IA}(\text{o.f=v},I) &= \{(\text{f,e',s',is'})|(\text{v,e',s',is'})\in I\}\end{aligned}$$

`v = o.f` For any tuple containing `o.f`, we create a new tuple containing `v`, which includes the field, statement, and induction status. The GEN_{IA} and KILL_{IA} functions are similar to a variable assignment.

$$\begin{aligned}\text{KILL}_{IA}(\text{v=o.f},I) &= \{(\text{v,e',s',is'})\}\\ \text{GEN}_{IA}(\text{v=o.f},I) &= \{(\text{v,e',s',is'})|(\text{f,e',s',is'})\in I\}\end{aligned}$$

`o.f = `*`expr`* Any other assignment to a field kills the data-flow information for `o.f`. The GEN_{IA} and KILL_{IA} functions are:

$$\begin{aligned}\text{KILL}_{IA}(\text{o.f=}\mathit{expr},I) &= \{(\text{f,e',s',is'})\}\\ \text{GEN}_{IA}(\text{o.f=}\mathit{expr},I) &= \{(\text{f},\emptyset,\emptyset,\texttt{ni})|(\text{f,e',s',is'})\in I\}\end{aligned}$$

3.1.2. *Analysis example: Mutual induction variables*

Figure 1 illustrates the how the data-flow analysis processes mutual induction variables. In the example, the value for variable `j` depends upon variable `i`, and the value for variable `i` depends upon variable `j`. Since the loop increments each variable by one, both variables increase by two in each iteration. Our induction analysis detects the mutual induction variables, and correctly computes the increment value. The analysis would also identify the variables as induction variables if two increment statements were conditionally executed.

3.2. Using SSA form to identify induction variables

Although we propose a new data-flow technique, which does not require an explicit data-flow graph, many compilers explicitly compute the relationships between definitions and uses of variables to

```
1   int sum=0; int i=0; int j=0;
2   while (i < n) {
3     sum += arr[i];
4     j = i + 1;
5     i = j + 1;
6   }
```

stmt	IA	Iteration 1	Iteration 2	Iteration 3
2	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,5,pr), (j,i+1,4,pr) (i,i+2,4,pr)	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), (j,j+2,5,pr)
	out	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,5,pr), (j,i+1,4,pr) (i,i+2,4,pr)	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), (j,j+2,5,pr)
4	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,5,pr), (j,i+1,4,pr) (i,i+2,4,pr)	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), (j,j+2,5,pr)
	out	(i,∅,∅,nr), **(j,i+1,4,pr)**	(i,j+1,5,pr), **(j,i+1,4,r)** (i,i+2,4,pr),**(j,j+2,5,pr)**	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), **(j,j+2,5,r)**
5	in	(i,∅,∅,nr), (j,i+1,4,pr)	(i,j+1,5,pr), (j,i+1,4,r) (i,i+2,4,pr), (j,j+2,5,pr)	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), (j,j+2,5,r)
	out	**(i,j+1,5,pr)**, (j,i+1,4,pr) **(i,i+2,4,pr)**	**(i,j+1,5,r)**, (j,i+1,4,r) **(i,i+2,4,r)**, (j,j+2,5,pr)	(i,j+1,5,r), (j,i+1,4,r) (i,i+2,4,r), (j,j+2,5,r)

Figure 1. Detecting mutual induction variables. The interesting points occur at lines 4 and 5. For example, in the first iteration, at line 5, the analysis creates two new tuples. The first tuple, `(i,j+1,5,pr)`, indicates that variable `i` is assigned the value `j+1`. Since $IA_{in}(5)$ contains a tuple for variable `j`, the analysis creates a second tuple, `(i,i+2,4,pr)`, which builds a complex expression using the expression from variable `j`'s tuple. At line 5, the analysis represents variable `i` as `j+1` and `i+2`. After reaching a fixed point, the analysis indicates that variables `i` and `j` are induction variables and they increase by two in each iteration.

perform optimizations. Instead of using our data-flow analysis, compilers that represent data flow explicitly can discover induction variables by identifying the strongly connected components (SCCs) in the data-flow graph.

One popular representation that many compilers use is static single assignment (SSA) form [12]. The key feature of SSA form is that each use of a variable has a single definition. SSA form introduces a special ϕ operator to combine definitions at merge points and produce a new definition. A main benefit of SSA form is that it is a sparse representation of the definitions and uses of variables.

Gerlek et al. describe an efficient algorithm for discovering different types of induction variables using SSA form [14]. They introduce a variant of SSA form that uses use-def chains exclusively. The algorithm works by computing the SCCs in the SSA data flow edges. Finding the SCCs in the use-def chains is not enough to classify the induction variables appropriately. The compiler must analyze the statements in the SCCs to determine how the induction variables change during each loop iteration. To aid the analysis, Gerlek et al. distinguish data merges at the beginning of loops using a μ operator rather than a ϕ operator. The μ operator provides a convenient mechanism to classify the loop induction variables. For example, a SCC that contains one μ, no ϕ, and an addition of a loop invariant value is a linear induction variable.

```
int i = 0;                   List o = getList();
while (i<n) {                while (o != null)  {
  sum += arr[j];               sum += o.value();
  j=j+1;                       o=o.next;
}                            }
```

Figure 2. Similarity between induction variables and list traversals

Although SSA form provides explicit data flow edges, Gerlek et al. show that finding induction variables requires more than just identifying SCCs in the flow edges. An induction variable detection algorithm must process the statements in the SCC to classify the induction variables. A compiler that uses SSA form may choose the approach that Gerlek et al. describe, but our approach also works and does not require a specialized intermediate representation.

3.3. Relationship to linked structure analysis

An induction variable and a linked structure traversal are examples of general recurrences. Figure 2 illustrates the similarities between discovering induction variables and linked structure traversals. The first loop in Figure 2 updates variable `j` by incrementing the value by 1. The second loop in Figure 2 updates object `o` by referencing the next element in the list. We propose a unified recurrence analysis that detects both of these traversal patterns.

In the linked structure analysis, the data-flow information is a set of tuples $R \subseteq \mathcal{P}(PV \times F \times S \times RS)$, where PV includes pointer variables and field references, F is a set of field objects, S is a statement, and RS is the recurrent status. RS is a value from the partially ordered set $\{$ `nr`, `pr`, `r` $\}$, which denote *not recurrent*, *possibly recurrent*, and *recurrent*, respectively [8].

Although we use different names, the RS values are equivalent to the IS values in the induction analysis. In the unified analysis, `nr` = `ni`, `pr` = `pi`, and `nr` = `ni`. The ordering of the elements in RS and IS are the same, and the join operator $\sqcup$ produces equivalent results.

A unified recurrence analysis combines the information in the tuples for the induction variable analysis and the linked structure analysis. The data-flow information in the unified analysis is a set of tuples, $R \subseteq \mathcal{P}(V \times FE \times S \times RS)$, where V is the set of variables and object fields, FE is the set of object fields and binary expressions, S is a statement number, and RS is the recurrence status. The unified analysis requires the GEN and KILL functions from the induction variable analysis in Section 3.1 and from the linked structure analysis in our prior work [8].

We implement the unified recurrence analysis in our compiler to identify prefetching opportunities in array references and linked structures. In prior work, we implemented and evaluated compile-time methods for prefetching linked structures. Although Java programs frequently allocate objects and create linked structures, we find that many Java programs also use arrays. In this article, we focus on evaluating array prefetching in Java.

```
for (int j=0; j<n; j++)
{
  prefetch(&arr[j+d]);
  sum += arr[j];
}
```

(a) Simple index expression

```
for (int j=0; j<n; j++ )
{
  prefetch(&arr[2*(j+d)]);
  sum += arr[2*j];
}
```

(b) Complex index expression

```
for (int j=0; j<n; j++ )
{
  prefetch(&arr[j+(d*2)]);
  prefetch(arr[j+d]);
  sum += arr[j].value;
}
```

(c) Array of Objects

Figure 3. Examples of prefetching arrays

4. Array Prefetching

In this section, we describe our algorithm to insert array prefetch instructions. The prefetch algorithm must identify an array access pattern, and generate a prefetch for an element that will be accessed in the future. We illustrate a simple array prefetching example in Figure 3(a). During each iteration, the program references the `j`th element, and we prefetch element `j+d`, where `d` is the prefetch distance. Prefetching is most effective when the prefetch distance value, `d`, is large enough to move the `j+d`th array element into the L1 cache prior to completing `d` iterations of the loop.

We propose a prefetch algorithm that does not require array locality analysis or loop transformations. Our compiler generates a prefetch instruction for array references that contain a linear induction variable in the index expression. The compiler generates the prefetch only if the array reference is enclosed in the loop that creates the induction variable.

Figure 4 summarizes the pseudo-code for the prefetch scheduling process. We restrict the algorithm to linear induction variables only because they generate arithmetic sequences. Since the induction variable value changes by the same constant expression each time it is executed, the prefetch distance remains the same during each iteration. Polynomial and exponential induction variables generate geometric progressions, which require a new prefetch distance to be computed during each iteration, and the distance depends upon the loop index value. We also allow conditional expressions to guard the execution of the induction variables and array accesses. In this case, the variable must monotonically increase or decrease by the same loop invariant value. The compiler generates the prefetch instruction immediately prior to the array reference. If the array reference is conditionally executed then so is the prefetch instruction.

An array index expression may contain other terms besides the induction variable. For example, in Figure 3(b), the array index expression is `2*j`, and the induction variable is `j`. We generate a prefetch in this example because the induction variable is linear. The compiler generates code to add the prefetch distance to `j` before the multiplication.

Our compiler generates prefetches for array elements and objects referenced by array elements, as appropriate. We generate multiple prefetch instructions if the size of the referent object is greater than a cache line. Prior prefetching algorithms focus on Fortran arrays and prefetch array elements only. In Java, arrays may contain object references as well as primitive types. For an array of objects, we want to hide the latency of accessing the array element and the referent object. Figure 3(c) illustrates array

```
I = $IS_{out}$(exit(S)); exit(S) is the last statement
for each assignment, t = arr[v]
  if (v, e, s,i) $\in$ I
    lp = set of statements in current loop
    if s $\in$ lp and e is linear
      c = increment/decrement value of e
      d = prefetch distance $*$ c
      generate prefetch (&arr[v + (d*2)])
      if array of objects
        generate prefetch (arr[v + d])
```

Figure 4. Pseudo-code for scheduling prefetch instructions

object prefetching. The first prefetch instruction is for the array element, and the second prefetch is for the referent object. The second prefetch must load the array element to get the address of the object. The prefetch distance for the array element is twice the distance of the prefetch distance for the referent object to ensure the array element is in the cache.

We use common subexpression (CSE) analysis to eliminate redundant prefetches. A prefetch is redundant if the compiler has already generated a prefetch for the cache line that contains the data. For example, if a loop accesses the same array element multiple times in the same iteration of a loop, our algorithm generates a prefetch instruction for each of the references. The CSE phase eliminates all but the first prefetch because the others are redundant. We leverage an existing implementation of CSE, but eliminating redundant prefetches is simpler than a complete CSE analysis. A very simple redundant prefetch elimination algorithm needs to analyze prefetch statements only, and eliminate prefetches that are redundant in the same loop iteration only. Using a CSE algorithm may eliminate more redundant prefetches by considering loads as well as prefetch instructions.

Generating prefetch instructions requires several steps using the algorithm by Mowry et al. [26]. First, the compiler uses array data dependences to perform locality analysis on the array references in a loop to approximate the cache misses. Then, the compiler performs loop unrolling and loop peeling to prefetch the references causing cache misses. Finally, the compiler attempts to improve prefetch effectiveness by software pipelining loops.

Mowry et al. use loop transformations to improve prefetch effectiveness by prefetching the first array elements prior to starting the loop, eliminating prefetches that hit in the L1 cache, and eliminating useless prefetches past the end of the last array element. Figure 5 shows a transformed version of the code in Figure 3(a). The prefetch distance is ten array elements. Software pipelining generates prefetches for the initial ten array elements, and loop unrolling improves prefetch effectiveness by prefetching whole cache lines instead of individual array elements.

Our prefetch algorithm does not perform loop transformations, which reduces the complexity of our approach. Our experimental results suggest that loop transformations are useful but, in the benchmarks that we use, they are not required to achieve significant performance improvements. We take advantage of available instruction level parallelism (ILP) in modern processors to reduce the effect of unnecessary

```
for (int i=0; i<10; i=i+4) {
  prefetch(&arr[i]);
}
int i=0;
for ( ; i<n-13; i=i+4) {
  prefetch(&arr[i+d]);
  sum += arr[i];
  sum += arr[i+1];
  sum += arr[i+2];
  sum += arr[i+3];
}
for (; i<n; i++) {
  sum += arr[i];
}
```

Figure 5. Traditional loop transformations for improving prefetch effectiveness on the code in Figure 3(a)

prefetches that hit in the L1 cache. When a processor has available ILP, an unnecessary prefetch instruction is very cheap, and the cost is much less than the benefit from prefetching useful data.

5. Experiments

In this section, we evaluate the effectiveness of our array prefetching technique. We first describe our experimental methodology. We present simulation results that show the benefits of compile-time prefetching. We use the simulator to understand the overall results better by evaluating the effectiveness of the dynamic prefetch instructions and the effect on cache performance. We also discuss the negative impact of conflict misses on prefetching. Finally, we validate the simulation results with execution times on a PowerPC processor.

5.1. Methodology

We implement the induction variable analysis and prefetching algorithm in Vortex, an optimizing compiler for object-oriented languages [13]. We use Vortex to compile our Java programs, perform object-oriented and traditional optimizations, and generate Sparc assembly code. We specify a prefetch distance of twenty elements as a compile-time option. We use a twenty element prefetch distance since it is small enough to cover the latency of accessing memory in loops that contain a reasonable amount of work.

We use RSIM to perform a detailed cycle level simulation of our programs [27]. RSIM simulates an out-of-order processor that issues up to four instructions per cycle and has a 64 entry instruction window. The functional units include two ALU, two FP, one branch, and two address units. Table I lists the memory hierarchy parameters. We use the default values for other simulation parameters. In RSIM, a prefetch instruction causes one cache line to be brought into the L1 cache. We have also

Table I. Simulation parameters

L1 cache	32KB, 32B line
	direct, write through, 2 ports
L2 cache	256KB, 32B line
	4-way, write back, 1 port
Write buffer size	8 entries
L1 hit time	1 cycle
L2 hit time	12 cycles
Memory hit time	60 cycles
Miss handlers (MSHR)	8 L1, 8 L2
Memory bandwidth	32B/cycle

Table II. PowerPC configuration

L1 cache	32KB, 32B line
	8-way, split
L2 cache	256KB, 64B line
	8-way, unified
L3 cache	1MB, 64B line
	8-way, unified
L1 hit time	3 cycles
L2 hit time	9 cycles
L3 hit time	38 cycles
Miss handlers	5 L1

Table III. Array-based benchmark programs

Name	Description	Inputs	Inst. Issued
Jama library			
cholesky	Cholesky decomposition	300x300 matrix	1381 M
eigen	Eigenvalue decomposition	250x250 matrix	1675 M
lufact1	LU decomposition	300x300 matrix	1570 M
matmult	Matrix multiply	400x400 matrix	1744 M
qrd	QR factorization	400x400 matrix	1811 M
svd	Singular value decomposition	300x300 matrix	5733 M
Java Grande			
crypt	IDEA Encryption	250000 elements	2500 M
fft	FFT	262144 elements	1828 M
heapsort	Sorting	1000000 integers	2916 M
lufact2	LU factorization	500x500 matrix	1167 M
sor	SOR relaxation	1000x1000 matrix	6972 M
sparse	Sparse matrix multiply	12500x12500 matrix	815 M

run a limited set of experiments with larger cache configurations. In general, changing the cache size and associativity parameters does not significantly affect the relative performance results. When we increase the cache size or set associatively, our prefetching results often improve slightly, until the cache grows large enough to eliminate most capacity misses.

To validate our simulation results, we run experiments on a PowerMac G4 running Yellow Dog Linux. The processor is a PowerPC MPC7450 running at 733MHz [24]. The MPC7450 is a superscalar out-of-order processor. Table II lists the important characteristics of the memory hierarchy.

We evaluate array prefetching using scientific library routines in the Jama package [16], and programs from section 2 of the Java Grande benchmark suite [7]. Table III lists the benchmarks we use in our experiments, along with characteristics about each program. We use input sizes that enable

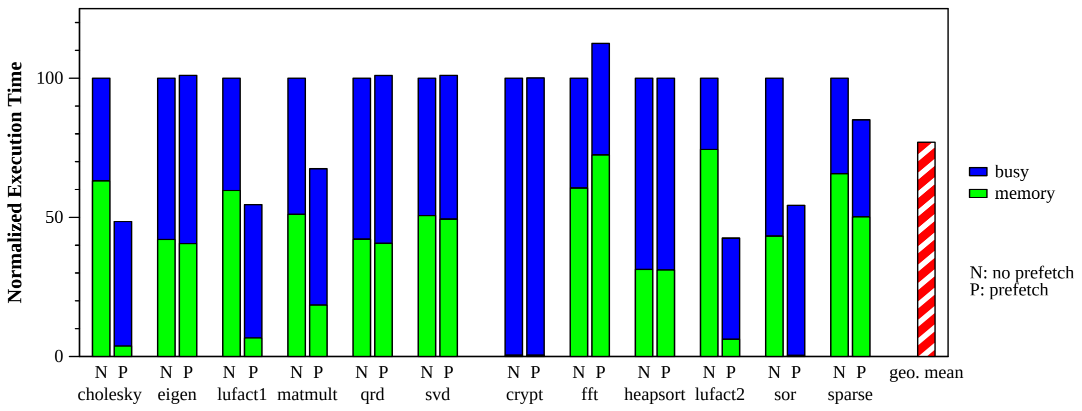

Figure 6. Overall prefetching performance

our simulations to complete in a reasonable time. We exclude `series` because it does not use an array as a main data structure.

We made a change to our garbage collector to reduce the occurrence of conflict misses in two of the programs. Vortex uses the UMass Garbage Collector toolkit for memory management [17]. The generational collector allocates memory in fixed sized 64KB blocks. Each generation may contain multiple blocks. The collector contains a large object space for objects larger than 512 bytes. The initial large object space implementation allocated very large arrays in new blocks aligned on a 64KB boundary. This allocation strategy results in many unnecessary conflict misses when programs access multiple large arrays at the same time. We fix the problem by adding a small number of pad bytes to the beginning of each large object. The pad bytes eliminate conflict misses and improve the performance of `sparse` and `qrd` with and without prefetching. Without the pad bytes, prefetching actually degrades performance in these programs by a few percent. The pad bytes do not help in `fft` because `fft` allocates a single array.

5.2. Overall performance

Figure 6 presents the results of array prefetching (P) on our programs. We normalize the results to those without prefetching (N). The six programs on the left side are part of the Jama library, and the other six programs are Java Grande benchmarks. We divide run time into the number of cycles spent waiting for memory requests, and the number of cycles the processor is busy. The memory requests include both read and write instructions, but almost all of the memory stall time is due to read instructions. The busy time includes all other execution cycles, including branches and multi-cycle arithmetic operations. In a processor with ILP, dividing the run time is not straightforward because instructions may overlap. RSIM counts a cycle as a memory stall if the first instruction that the processor cannot retire in a cycle is a load or store.

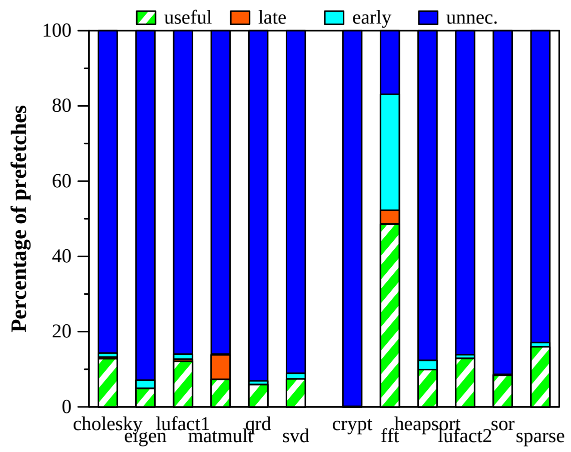

Figure 7. Prefetch effectiveness

Figure 6 shows that the programs spend a large fraction of time waiting for memory requests. Seven of the twelve programs spend at least 50% of run time stalling for memory requests. Clearly, these programs have substantial room for improvement. We see substantial improvements in six of the twelve programs. Across all programs, prefetching reduces the run time by a geometric mean of 23%.

The largest improvement occurs in `lufact2` where prefetching reduces the run time by 58%. In five of the programs, prefetching reduces the run time by more than 30%. Prefetching either slightly degrades performance by less than 1% or has little effect on five programs. Prefetching increases run time in `fft` by 13%. In Section 5.5, we show that prefetching improves the performance of a different FFT implementation, which is faster than the Java Grande version on our architecture model.

Prefetching contributes to large improvements in `cholesky`, `lufact1`, `matmult`, `lufact2`, and `sor`. The performance improvement is due to memory stall reduction. In the programs that improve significantly, the amount of time spent stalling due to memory requests decreases substantially. Prefetching eliminates almost all the memory stalls in `sor`.

Prefetching does not have any effect in `crypt` or `heapsort`. The time spent on memory operations in `crypt` is less than 1% of total run time, so we do not expect to see any performance improvement. The access pattern in `heapsort` is not regular and is data dependent, so it is difficult to improve using prefetching.

Software prefetching increases the number of executed instructions. The amount of busy time in the programs tends to increase slightly in Figure 6, although the increases are difficult to see because they are so small. The additional functional units in a superscalar processor are able to hide most of the cost of the additional instructions.

Table IV. Static and dynamic prefetch statistics

Program	Static	Dynamic		Bus Util.	
		total pfs	pfs/reads	N	P
cholesky	32	97 374 473	10 %	13 %	28 %
eigen	153	47 214 494	10 %	8 %	8 %
lufact1	36	48 721 596	12 %	13 %	24 %
matmult	22	77 693 667	38 %	21 %	31 %
qrd	20	40 771 107	9 %	9 %	9 %
svd	74	134 700 673	9 %	10 %	10 %
crypt	31	500 794	3 %	0.1 %	0.1 %
fft	37	7 159 698	12 %	16 %	16 %
heapsort	27	1 442 004	0.3 %	9 %	9 %
lufact2	50	55 797 568	25 %	17 %	39 %
sor	30	224 896 104	16 %	11 %	20 %
sparse	29	24 234 710	19 %	20 %	24 %

5.3. Prefetch effectiveness

Figure 7 divides the L1 dynamic prefetches into several categories. A *useful* prefetch arrives on time and is accessed. The latency of a *late* prefetch is partially hidden because a demand request occurs while the memory system is retrieving 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 outstanding miss. Useful, late, and early prefetches require accesses to the second level cache at least.

Figure 7 shows that the percentage of useful prefetches does not need to be large to improve performance. The number of useful prefetches is less than 16% in each program, except for `fft`. In the program with the largest improvement, `lufact2`, the number of useful prefetches is just 13%.

Only `matmult` has a noticeable number of late prefetches. We can slightly improve performance in `matmult` by increasing the prefetch distance, which reduces the late prefetches. The program with the largest number of useful prefetches, `fft`, is also the program with the worst overall performance. The large number of early prefetches results in poor performance. The early prefetches are due to conflict misses, which we discuss in more detail in Section 5.5. Another potential source of early prefetches is a prefetch distance that is too large, but we do not see this effect in our programs.

Figure 7 shows that a twenty element compile-time prefetch distance is effective in achieving most of the performance gains. The prefetch distance is large enough to bring data into the cache when needed, and small enough so that the data is not evicted prior to the demand request, which our results illustrate since the number of late prefetches is small. We also vary the prefetch distance to examine the impact on the results. Using a prefetch distance of five, ten, and thirty elements reduces the run time by a geometric mean of 20%, 23%, and 23%, respectively. The only noticeable difference occurs when the prefetch distance is five elements. Although the performance is stable when we aggregate the run times, we see some small differences among the individual programs. We do not see a compelling reason to use more sophisticated analysis to determine the appropriate prefetch distance automatically.

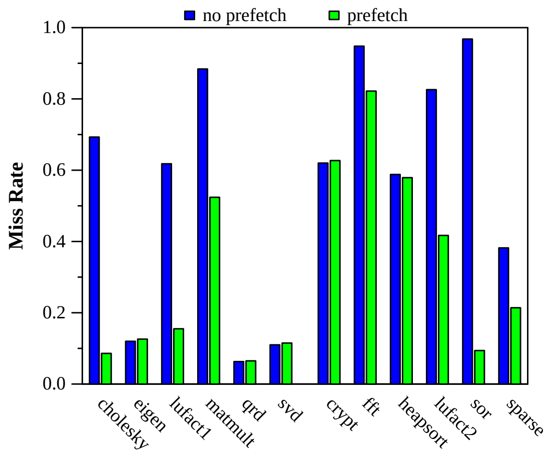

Figure 8. L1 cache miss rate

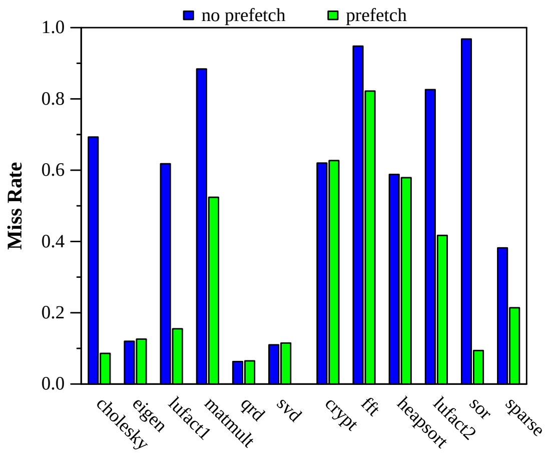

Figure 9. L2 cache miss rate

Table IV lists statistics about the static and dynamic prefetches. The static prefetch value is the number of prefetch instructions that the compiler generates. We present the number of dynamic prefetch instruction executed, and the percentage of dynamic prefetches relative to the number of dynamic load instructions. The largest percentage of dynamic prefetches occurs in `matmult` because most of the run time is spent in a short inner loop. The small number of prefetches in `heapsort` is one reason that prefetching is ineffective.

Table IV also shows the bus utilization values with (P) and without prefetching (N). Prefetching does increase the bus utilization. In some cases, the utilization percentages double (*e.g.*, `cholesky`, `lufact1`, `lufact2`, and `sor`), but the bus utilization in these programs remains under 40% even with prefetching. The main reason for the utilization increase is that the run time decreases substantially. Except for `fft`, prefetching uses the data brought into the cache effectively.

5.4. Cache statistics

The cache miss rate is also a useful metric to illustrate the benefits of prefetching. Figures 8 and 9 show the L1 and L2 cache miss rates, respectively. Effective prefetching improves the miss rate by moving data into the cache prior to the demand request.

The miss rates vary considerably in our benchmark programs. The L1 miss rates are much better than the L2 miss rates for most programs. When computing the L2 miss rate we count only the references that miss in the L1 cache. The number of references to the L2 cache is far less than the number of references to the L1 cache.

The L1 miss rate varies from almost 0% to just under 50%. The L2 miss rate varies from 6% to 98%. Over 50% of the references miss in the L2 cache in 8 programs. It is possible to increase the cache

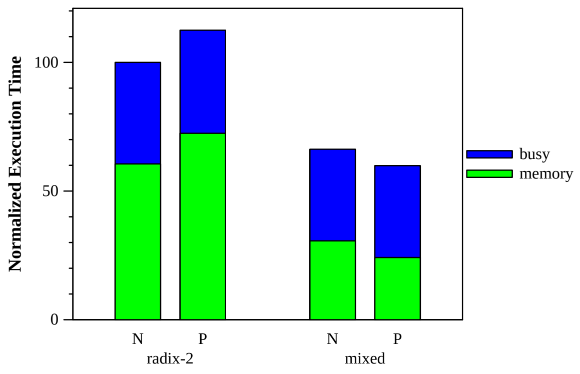

Figure 10. Comparing FFT implementations

sizes to improve the miss rates. We find that increasing the cache size tends to improve prefetching slightly, until the data fits in cache.

Prefetching effectiveness does not correspond to high or low miss rates. Prefetching improves or does not affect the L1 miss rate in each program. In the programs with the largest execution time improvements, we see significant miss rate reductions. Prefetching almost completely eliminates L1 cache misses in `sor` by reducing the miss rate from 38% to 1%. Prefetching improves the L1 miss rate in each program, and the L2 miss rate in most programs. The L2 miss rate is slightly worse in several programs because there are fewer L2 references, but the *percentage* of references that miss is higher.

Improving the miss rate, however, does not always correspond to run time improvements. We see a significant improvement in the L1 and L2 miss rate for `fft` even though prefetching increases the run time. The problem is due to conflict misses, which we discuss in the next section.

5.5. Conflict misses

Performance degrades by 13% in `fft` due to a large number of cache conflict misses. The conflict misses in the L2 cache cause the biggest problem, and increasing the set associatively to eight or sixteen does not eliminate the problem. The implementation uses the radix-2 algorithm, which computes results in-place using a single dimension array. The size of the array and the strides through the array are powers of two. For large arrays, the power of two stride values cause the conflict misses. Without prefetching, 7% of the read references cause conflict misses in the L1 cache, and 37% of the read references in the L2 cache cause conflict misses. Prefetching exacerbates the problem by increasing the number of conflict misses. With prefetching, 8% and 34% of the read references cause conflict misses in the L1 and L2 cache, respectively. Due to a power of two prefetch distance, the prefetches evict data that are prefetched in prior iterations. Although it might be possible for a compiler to detect this situation using sophisticated and expensive array dependence analysis, a more effective solution is to use a better FFT algorithm.

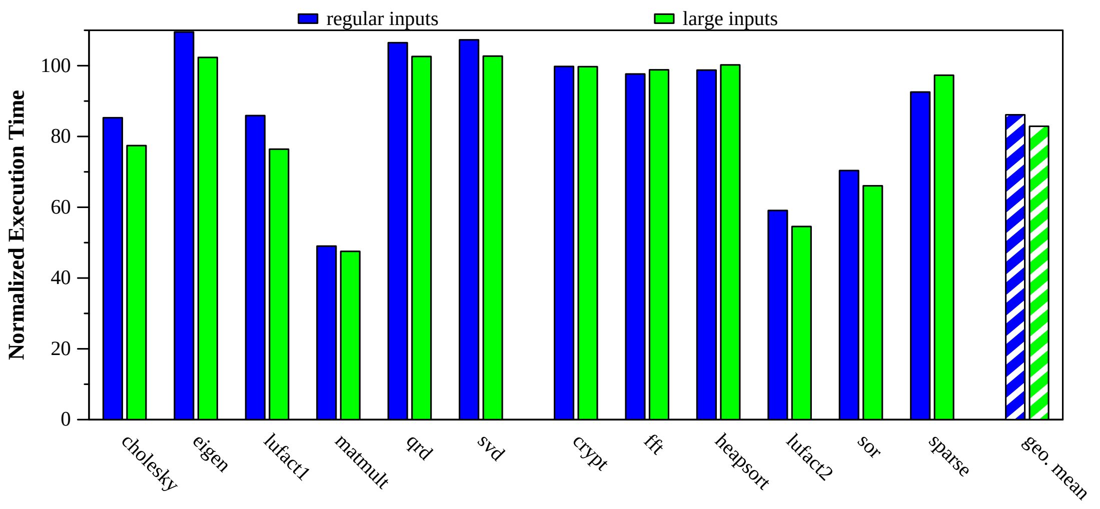

Figure 11. Prefetching on the PowerPC

We evaluate prefetching using a mixed radix implementation of FFT and compare the performance to the radix-2 implementation. The mixed radix version is a more complex algorithm, which requires additional storage. We compare the two versions of FFT in Figure 10. We present results with and without prefetching for the two implementation. The `radix-2` bars on the left side are the same results from Figure 6. The `mixed` FFT version is 34% faster than the `radix-2` version, and prefetching further improves the performance of `mixed` by 10%.

5.6. Prefetching on the PowerPC

In this section, we evaluate the effectiveness of prefetching on a PowerPC processor in order to validate our simulation results. We use Vortex to generate C code instead of SPARC assembly code, and we compile the C code using gcc. Vortex inserts a prefetch instruction using gcc's inline assembly mechanism. Table II describes the PowerPC memory hierarchy configuration.

Figure 11 shows the results for prefetching on the PowerPC. We normalize the run times to those without prefetching. Since we run the programs on the machine instead of the simulator, we are unable to divide the run time into memory and busy time. The figure presents results using two inputs sizes. The *regular* inputs are from Table III. For the *large* inputs, we increase the input sizes. For the Jama programs, we double the array sizes in Table III. For the Java Grande programs, we use the size C (large) input values.

Prefetching reduces the run time by a geometric mean of 14% for the regular inputs and 17% for the large inputs. Prefetching reduces the run time of `matmult` by 52%. One reason that the PowerPC improvements are not as large as the simulation improvements is that the simulated processor uses larger cache access latencies, which future processors will certainly experience and they will continue

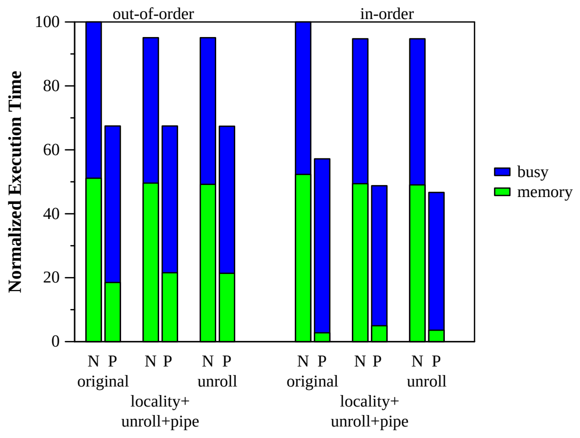

Figure 12. Applying different loop transformations matrix multiplication

to increase. Even with the fast cache access on the PowerPC, prefetching improves several programs significantly including `matmult`, `lufact2`, and `sor`.

In summary, compile-time data prefetching is effective on array-based programs, even without loop transformations and array dependence information. Our results show that generating prefetches for array references that contain induction variables improves performance substantially. We now explore in more detail how prefetching achieves its improvements.

6. Case Study: Matrix Multiplication

In this section, we show the effects of loop transformations and additional analysis on performance by applying loop unrolling and software pipelining on matrix multiplication. We use a case study to help explain why our simple prefetch algorithm is effective on a modern processor.

Figure 12 presents results for three versions of matrix multiplication with different code and data transformations on an in-order and out-of-order processor. We provide results for each version with and without prefetching. We normalize all times to `original`, the Jama library version from Section 5, without prefetching on either an in-order or out-of-order processor. We perform the transformations by hand starting with the code in `original`.

We obtain the out-of-order results using the processor configuration from Section 5. To obtain the in-order results, we simulate a single issue processor with blocking reads. The in-order processor allows the prefetch instructions to complete independently of reads. We apply Mowry et al.'s [26] prefetch algorithm to matrix multiplication in `locality+unroll+pipe`. We present results for loop unrolling only in `unroll`.

Transforming `locality+unroll+pipe` requires several steps. We unroll the innermost loop four times to generate a single prefetch instruction for an entire cache line. We perform software pipelining on the innermost loop to begin prefetching the array data prior to the loop. We generate a prefetch for one of the arrays only. Matrix multiplication operates on the same portion of the second array during the two innermost loops. Prefetching the second array results in many unnecessary prefetches.

Figure 12 shows that a state of the art prefetching algorithm does not provide much benefit for matrix multiplication on the out-of-order processor. Without prefetching, both `locality+unroll+pipe` and `unroll` improve performance by 5%. The run time of `original` with prefetching is the same as the run time after applying loop transformations, locality analysis, and prefetching. But, the transformations and locality analysis do improve prefetch effectiveness. Only 3% of prefetches in `locality+unroll+pipe` are unnecessary compared to 86% in `original`.

The loop transformations do have an impact on the in-order processor. In `original`, prefetching improves performance by 43%, which is larger than the performance improvement on the out-of-order processor. The better scheduling methods improve performance by an additional 18% over `original` with prefetching on the in-order processor. The improvement occurs because the locality analysis and loop transformations reduce the number of dynamic instructions. These results show that careful scheduling is more important on the in-order processor than the out-of-order processor for matrix multiplication.

The overall results in Section 5 suggest that advanced prefetch algorithms are not necessary to achieve benefits from prefetching on modern processors. This case study argues that the loop transformations and additional analysis may not provide benefits on modern processors. In a superscalar out-of-order processor, the cost of checking if data is already in the cache is cheap. The additional functional units and out-of-order execution hide the effects of issuing unnecessary prefetches that hit in the L1 cache. Loop transformations can reduce the number of unnecessary prefetches, but the resulting performance gain may be negligible. Furthermore, transformations may not be possible due to exceptions or inexact array analysis information.

7. Additional Prefetching Opportunities

In this section, we illustrate that our analysis and prefetching technique work on other program idioms. Although these idioms do not appear in any of our benchmarks, we believe they are used in other Java programs.

7.1. Arrays of objects

Java allows arrays of references as well as arrays of primitive types such as `double`. In an array of references, the array element contains a pointer to another object instead of the actual data. As we describe in Section 4, our compiler generates prefetches for the array element value and referent object. The prefetch distance for the array element is twice as long as the prefetch distance for the array element referent object.

Since none of our benchmarks use arrays of objects, we change the Jama version of matrix multiplication to use `Complex` objects instead of `double` values. The `Complex` object contains two fields, which contain the real and imaginary parts of a complex number. The class implements methods to operate on complex numbers.

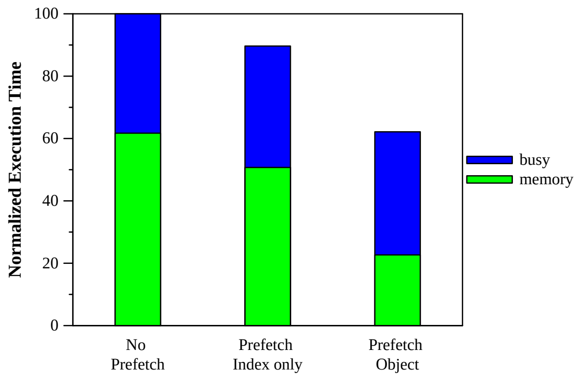

Figure 13. Prefetching arrays of objects

Figure 13 shows the results for matrix multiplication with complex numbers. We generate results for no prefetching, prefetching just the element values, and prefetching both the element and the referent object. Prefetching just the array elements reduces run time by just 10%. When we prefetch both the array elements and the referent objects, prefetching reduces run time by almost 38%. The large improvement occurs even though we increase the number of instructions by generating two prefetches and an additional load for each array reference.

7.2. True multidimensional arrays

Java implements multidimensional arrays as arrays-of-arrays, unlike languages such as Fortran that implement true multidimensional arrays. Java allocates each array dimension separately, so there is no guarantee that the memory allocator allocates a contiguous region of memory for the entire array. True multidimensional arrays allocate a single contiguous region of memory for the entire array. Using a single contiguous region of memory simplifies compile-time analysis and optimization because the compiler can compute the address of any element relative the start of the array. The array specification in Java makes it challenging to apply existing array analysis and loop transformations.

In this section, we examine the performance of prefetching on true multidimensional arrays. We simulate true multidimensional arrays using a single array with explicit index expressions. We also use the *multiarray* package from IBM, which is a Java class that contains an implementation of true multidimensional arrays [23]. The underlying structure is a one dimensional array, and the class provides methods that mimic Fortran style array operations. We compare the performance of standard Java arrays, simulated true multidimensional arrays, and the IBM multiarrays using matrix multiplication.

We can simulate a true multidimensional array by allocating a single array and using explicit index expressions to treat the array as a multidimensional array. Figure 14 shows the implementation of matrix multiplication when we implement a two-dimensional array using a single dimension. The

```
for (int i=0; i<rows; i++) {
  for (int j=0; j<cols; j++) {
    double val = 0;
    for (int k=0; k<cols; k++) {
      val += m1[i*cols+k] * m2[k*cols+j];
    }
    m3[i*cols+j] = val;
  }
}
```

Figure 14. Matrix multiplication with a single array

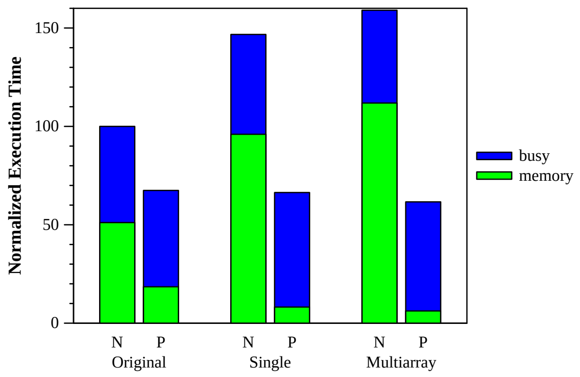

Figure 15. Performance of prefetching on true multidimensional arrays

code multiplies array `m1` by array `m2` and places the result in array `m3`. The array index expression `m1[i*cols+k]` is equivalent to the expression `m1[i][k]`.

Figure 15 shows the results of prefetching on the standard array representation, the simulated true multidimensional array representation, and the multiarray representation for matrix multiplication. We normalize all times to `Original`, the Jama library version from Figure 6. The `Single` version uses a single array with explicit addressing to simulate a two dimensional array, and the `Multiarray` version uses IBM's multiarray package. The performance of `Single` and `Multiarray` without prefetching is 46% and 59% worse than the performance of `Original` without prefetching. One reason is that the programmer is able to hoist the loop invariant address expressions out of the inner loop in `Original`. `Multiarray` has an additional cost because of more method calls and object allocations. Figure 15 shows that our array prefetching algorithm is able to discover the complex loop induction expression and insert effective prefetch instructions. The performance of `Single` with prefetching is slightly better than the performance of `Original` with prefetching by 1%. Prefetching improves the performance of `Multiarray` further. The performance of `Single` and `Multiarray` with prefetching is better than

```
Enumeration e=o.elements();
while (e.hasMoreElements()) {
  Element m = (Element)e.nextElement();
  sum += m.value();
}
// Inlined Enumeration for a Vector object
VectorEnumerator e;
e.vector=o;
e.count=0;
while (e.count < e.vector.elementCount)
  prefetch &e.vector.elementData[e.count + d];
  Element m = e.vector.elementData[e.count];
  e.count = e.count + 1;
  sum += m.value();
}
```

Figure 16. Using the `Enumeration` Class

`Original` with prefetching because there are fewer late prefetches. Since `Single` and `Multiarray` perform more work in the inner loop, there is more time for the prefetches to bring data into the L1 cache.

7.3. Enumeration class

The `Enumeration` class is a convenient mechanism for encapsulating iteration over a data structure. Figure 16 illustrates code that uses the `Enumeration` class to iterate over elements in a `Vector`. Figure 16 also shows the version after the compiler performs inlining.

Our induction variable algorithm detects that `e.count` is an induction variable even though `e.count` is an object field and not just a simple variable. We discuss the analysis extensions for object fields in Section 3.1.1. Once the analysis detects the induction variable in an object field, the array prefetch algorithm generates a prefetch for a reference that contains the object field in the index expression.

8. Conclusion

Compile-time data prefetching is effective in improving the memory performance of array-based Java programs. Traditional prefetch algorithms focus on Fortran arrays, use sophisticated locality analysis, and transform loops to generate prefetches. We propose a prefetch technique that does not require locality analysis or loop transformations. Our algorithm generates prefetches for all array references containing loop induction variables. We also generate an additional prefetch for array object references. Compilers that detect induction variables can implement our prefetch algorithm easily. We present a new, unified data-flow analysis to detect general recurrences in programs, including induction variables and linked structure traversals. We present results showing the effectiveness of software prefetching on a set of array-based Java programs. Although hardware prefetching is successful on similar programs, compiler analysis enables prefetching on a larger set of access patterns. Our results show that loop transformations and array analysis are not necessary to achieve large performance gains with prefetching in Java programs. Prefetching improves performance in six of the twelve programs by a geometric mean of 23% on a simulated architecture and 17% on a PowerPC processor. The largest improvement is 58%, which occurs in LU factorization.

ACKNOWLEDGEMENTS

This work was performed at the University of Texas at Austin while the first author was a graduate student at the University of Massachusetts. We thank the anonymous referees from the Java Grande/ISCOPE conference for their detailed comments and helpful suggestions. This work is supported by NSF ITR grant CCR-0085792, NSF CCR grant CCR-0311829, DARPA grant F33615-03-C-4106, and IBM. 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. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, Oct. 1988.
3. Z. Ammarguellat and W. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, NY, June 1990.
4. P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 1–10, Santa Fe, NM, May 2000.
5. J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of 1991 Conference on Supercomputing*, pages 176–186, Alburquerque, NM, Nov. 1991.
6. 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, Limassos, Cyprus, June 1995.
7. J. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmarks suite for high performance Java. *Concurrency : Practice and Experience*, 12(6):375–388, May 2000.
8. B. Cahoon and K. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, Sept. 2001.
9. 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, Santa Clara, CA, Apr. 1991.
10. M. Cierniak and W. Li. Just-in-time optimizations for high-performance Java programs. *Concurrency : Practice and Experience*, 9(11):1063–1073, Nov. 1997.
11. C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, TX, Feb. 1995.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
13. 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, San Jose, CA, Oct. 1996.
14. 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):85–122, 1995.
15. M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

16. J. Hicklin, C. Moler, P. Webb, R. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A Java matrix package. URL: http://math.nist.gov/javanumerics/jama, 2000.
17. R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report Technical Report 91-47, University of Massachusetts, Dept. of Computer Science, Sept. 1991.
18. Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2002.
19. D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997.
20. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the Association for Computing Machinery*, 23(1):158–171, Jan. 1976.
21. E. S. Lowry and C. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, Jan. 1969.
22. N. McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.
23. J. Moreira, S. Midkiff, M. Gupta, and P. Artiga. Numerically intensive Java: Multiarrays. URL: http://www.alphaWorks.ibm.com/tech/ninja, 1999.
24. Motorola, Inc. *MPC7450 RISC Microprocessor Family User's Manual*, Dec. 2001. http://e-www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf.
25. T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Department of Electrical Engineering, Mar. 1994.
26. 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.
27. 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.
28. V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, Denver, CO, June 1997.
29. C. W. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Sept. 1992.
30. A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, Dec. 1978.
31. Sun Microsystems. *UltraSparc III Cu User's Manual*, version 1.0 edition, May 2002.
32. J. Tendler, J. Dodson, J. J.S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, Jan. 2002.
33. S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.
34. P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, Cumberland Farms, KY, Aug. 2001.