A Practical Stride Prefetching Implementation in Global Optimizer

Hucheng Zhou
Tsinghua University
zhou-hc07@mails.tsinghua.edu.cn

Xing Zhou
Tsinghua University
zhoux07@mails.tsinghua.edu.cn

Tianwei Sheng, Dehao Chen, Jianian Yan
Tsinghua University
{ctw04, chendh05, yanjn03}@mails.tsinghua.edu.cn

Shinming Liu
Hewlett-Packard Company
shin@cup.hp.com

Wenguang Chen, Weimin Zheng
Tsinghua University
{cwg, zwm-dcs}@tsinghua.edu.cn

Abstract

Software data prefetching is a key technique for hiding memory latencies on modern high performance processors. Stride memory references are prime candidates for software prefetches on architectures with, and without, support for hardware prefetching. Compilers typically implement software prefetching in the context of loop nest optimizer (LNO), which focuses on affine references in well formed loops but miss out on opportunities in C++ STL style codes.

In this paper, we describe a new inductive data prefetching algorithm implemented in the global optimizer. It bases the prefetching decisions on demand driven speculative recognition of inductive expressions, which equals to strongly connected component detection in data flow graph, thus eliminating the need to invoke the loop nest optimizer. This technique allows accurate computation of stride values and exploits phase ordering. We present an efficient implementation after SSAPRE optimization, which further includes sophisticated prefetch scheduling and loop transformations to reduce unnecessary prefetches, such as loop unrolling and splitting to further reduce unnecessary prefetches.

Our experiments using SPEC2006 on IA-64 show that we have competitive performance to the LNO based algorithms.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

General Terms Compiler, Data Prefetching

Keywords Software Data Prefetching, Induction Variable, Strength Reduction, Partial Redundancy Elimination

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Open64 Workshop of CGO'08 April 6, 2008, Boston, Massachusetts.

Copyright © 2008 ACM [to be supplied]...$5.00

1. Motivation

The performance of microprocessors has been increasing at a much faster rate than the main memory (DRAM)'s over the past decades. The introduction of caches has proven effectiveness to reduce memory access latency by exploiting spatial and temporal locality in programs. In addition, data prefetching, which brings data into caches ahead of time has been proposed to hide the access latencies for cases that caching strategies can not handle (Patterson 1990; Gupta et al. 1991).

Data prefetching can be implemented with either software or hardware, while software data prefetching provides more flexibilities than hardware prefetching (Berg 2002; Callahan et al. 1991; Chen and Baer 1995). There are many cache misses caused by massive consecutive memory references in loop nest. A *stride-n reference pattern* (Berg 2002; Vanderwiel and Lilja 2000) is recognized in loops where the stride value of the accessed addresses for a reference is constant between two consecutive iterations. In this paper, we focus on compiler-based software data prefetching on strided references, a technique we call strided prefetching.

C++ has been very popular in the past years and the use of Standard Template Library (STL) is prevalent. STL vector traversing is a typical strided reference. Traditional LNO based stride prefetching algorithms (Mowry et al. 1992) cannot issue prefetches for STL vector references, mainly because of unsupported and unexpected code patterns. LNO is modeled as a sophisticated vector space spanned by a system of affine equations and is resolved through complex time/space consuming mathematical algorithms. This model is generally effective for array subscripting based memory references with affine access function in DO loop nest. STL vector variables don't fall into this category. Due to the lack of shape information, it is hard for LNO to handle such strided references. We want to materialize these potential prefetching opportunities even without LNO.

In contrast, global optimizers provides normalized intermediate representation (IR) on data flow and control flow information, canonicalizing different control structures and different memory references. This can be exploited for covering different strided prefetching. Additionally, other optimizations such as partial redundancy elimination (PRE) and strength reduction in global optimizer have provided enough supports and facilities for stride prefetching, which is shown in the following sections.

In this paper, we present an alternative stride prefetching algorithm in global optimizer, which is based on demand driven speculative recognition of inductive expressions implemented after PRE and strength reduction.

2. Background

2.1 Software Data Prefetching

Prefetching coverage and prefetching accuracy were defined by Joseph and Grunwald (Joseph and Grunwald 1999) to measure the effectiveness of data prefetching algorithm, where prefetching coverage is "the fraction of missing references that are removed by prefetching" and prefetching accuracy is "the fraction of prefetches that are useful, r.e., the prefetched data would be referenced at least once before it is replaced".

Thus, these are some important features what compiler controlled prefetching algorithm must take into consideration.

1. Prefetching candidate identification
Prefetching is possible only if the corresponding memory address can be determined ahead of the access. The more candidates are identified, the broader the prefetching coverage.
2. Prefetching timing determination
Prefetching timing is important as prefetches issued too early and too late will counteract the benefits.
3. Unnecessary prefetching avoidance
Prefetch is unnecessary if the prefetched data is already located in the cache. Too many prefetches have negative effects as well.
4. Fine grained prefetch tuning.
The compiler should set the proper read/write flag, if supported by the hardware. It also determine the optimal cache level data should be prefetched into, as well as the locality hint selection, again, if the target machine has such features.

2.2 Related Work

There is a multitude of previous work on data prefetching. Stefan and Vanderwiel et. al(Berg 2002; Vanderwiel and Lilja 2000) wrote good surveys on data prefetching mechanisms which cover both hardware and software prefetching.

Mowry et al (Mowry et al. 1992) have presented a stride prefetching technique in the context of LNO to reduce the number of prefetches based on the locality analysis. It includes three standard steps. First, compiler analyses the locality information in the loop nests, based on the resolution of locality equations(Wolf and Lam 1991). Second, the compiler computes the prefetching distance according to the cache model Third, the compiler transforms the loop nest to avoid the unnecessary prefetches due to locality, avoiding insertion of additional predicates to control additional prefetches.

Vatsa et al (Santhanam et al. 1997) have presented the prefetch generation algorithm for numerical programs in HP-PA production compiler for HP PA-8000.

Luk and Mowry (Luk and Mowry 1996; Mowry and Gupta 1991) have investigate the compiler-based prefetching for pointer-based applications, in particular for those containing recursive data structures, and proposed three different prefetching schemes: greedy prefetching, history-pointer prefetching and data-linearization prefetching. Artour et. al(Stoutchinin et al. 2001) have presented speculative stride prefetching for linked list prefetching by computing the stride value in run time.

The latest research shows the effectiveness of recognizing stride references dynamically(Roth and Sohi 1999). Wu(Wu 2002) used profiling techniques to integrate stride reference information to cover the situations that compiler cannot recognize statically. Chilimbi et.al(Chilimbi and Hirzel 2002) proposed a three-phase profiling technique to automatically insert prefetches for hot stream data references. Beyer and Clauss(Beyler and Clauss 2007) presented a prefetching framework based on the finite state machine to determine the prefetching, guided by the time spent on attaining the data.

Induction variable recognition(Liu et al. 1996; Gerlek et al. 1995) is very important as it benefits all loop related optimizations, such as dependence testing or strength reduction. There are two different ways to recognize induction variables, pattern matching and Strong Connective Component identification using data flow dependence.

PRE, first developed by Morel and Renvoise (Morel and Renvoise 1979), replaces the partial redundant occurrences with newly introduced temporary variable while finding the optimal code place to insert additional computation as the definition of the temporal variable. Considering the inefficiency of the dense data flow initialization and propagation(Knoop et al. 1992; Khedker and Dhamdhere 1994), Fred. Chow et al (Kennedy et al. 1999; Chow et al. 1997) have presented a sparse technique called SSAPRE, which regards the expressions as variables and creates a sparse factored redundancy graph to process sparsely data flow propagation. SSAPRE has proven to be highly efficient, has low compilation overhead, and is widely used in modern optimizing compiler.

Strength reduction refers to program optimization techniques in which expensive or slow operations are replaced by more efficient and faster ones(Dhaneshwar and Dhamdhere 1995). It benefits any inductive expressions, through creating more opportunities for the PRE on these strength reduction candidates (Dhamdhere 1989; Kennedy et al. 1998). Robert Kennedy et.al. (Kennedy et al. 1998) has presented a PRE-based techniques that allow strength reduction to be performed concurrently with PRE in SSAPRE framework.

3. Introduction

3.1 Overview of compiler components

Existing data prefetching algorithms are implemented as loop nest optimizations. We instead provide an alternative stride prefetching algorithm implemented as a scalar optimization in the context of the global optimizer.

Our algorithm is implemented in the production quality Open64 compiler(ope). For convenience, we give a brief introduction of the components in Open64 to illustrate the requirements of our algorithm. Note that our approach is not restricted to any specific compiler.

Figure 1 gives the rough compilation components in Open64 compiler, and shows the relation of pre-optimizer (PRE-OPT) to the global optimizer (WOPT), inter-procedural analysis (IPA) and loop nest optimizer (LNO). One of the goals of PRE-OPT is to normalize the high level WHIRL IR in a tree form translated from the FE, including canonicalization the induction variables written in any form in the original source code(Liu et al. 1996). This normalized IR is then passed as the input to LNO, IPA and WOPT, greatly simplifying the latter optimization phases.

In our intermediate representation, which is called WHIRL, there are five levels of abstraction, going from very high level to lower levels. The various optimization phases works on a specific levels of WHIRL and optimization proceeds along the process of continuous lowering.

Open64 has implemented the stride prefetching in LNO based on Mowry's algorithm(Mowry et al. 1992).

3.2 The necessity of our algorithm

Generally, there are four ways to traverse STL vector, shown in Figure 2, including subscripting style, iterator, const iterator, and at() references, named ACCESS1, ACCESS2, ACCESS3, ACCESS4

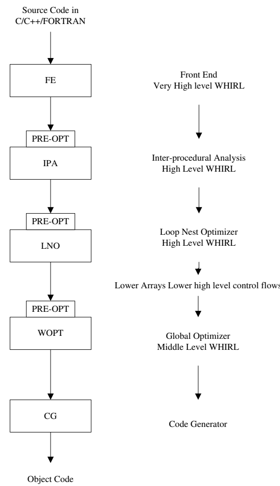

Figure 1. The components flow of Open64 compiler

```
typedef double Type;
#define LEN 10000
vector<Type> V(LEN);
Type sum;
#define ACCESS1 {\
    sum = 0.0; \
    for(int i = 0; i < LEN; i++) { \
        sum += V[i]; \ } \
}
#define ACCESS2 {\
    sum = 0.0; \
    vector<Type>::iterator it;\
    for(it = V.begin(); it != V.end(); it++) { \
        sum += *it; } \
}
#define ACCESS3 {\
    sum = 0.0; \
    for(int i = 0; i < LEN; i++) { \
        sum += V.at(i); \ } \
}
#define ACCESS4 {\
    sum = 0.0; \
    vector<Type>::const_iterator cit; \
    for(cit = V.begin(); it != V.end(); it++) { \
        sum += *cit; } \
}
```

Figure 2. Four different ways to traverse vector

respectively. Figure 3 shows the internal representations in high level WHIRL for the ACCESS1 and ACCESS2 respectively. For comparison, Figure 4 shows the high level WHIRL for simple array references. The high level WHIRL representations are translated to a C-style ASCII format for perusal. We omit the high level WHIRL of ACCESS3 and ACCESS4 since they are the same as ACCESS 2 and ACCESS1 respectively.

We can see that STL vector references are all lowered to indirect references with explicit address arithmetic after type specification, member function inlining, copy propagation and dead code elimination in PRE-OPT. Their address values are incremented by a constant value 8 (*sizeof(double)*) for each iteration. There are slight differences of the address calculation between ACCESS1 and ACCESS2, where in ACCESS1 it is indirectly dependent on primary induction variable anon1, while it is definite incremented by 8 in ACCESS2. Even then their address expressions can be easily recognized as inductive expressions or induction variables illustrated in Section 4 respectively. Whereas, array references are kept as explicitly subscripting access, providing the corresponding information of access function and dimensionality of array.

```
         ACCESS1:
// start = &(*V.begin());
// finish = &(*V.end());
sum = 0;
anon2 = (_IEEE64 *)start;
for(anon1 = 0; anon1 <= 9999; anon1 = anon1 + 1)
{
  anon2 = anon2 + anon1 * 8;
  sum = *anon2 + sum;
}
         ACCESS2:
annon1 =  (_IEEE64 *)start;
annon 2 = (_IEEE64 *)finish;
while(anon1 !=  anon2)
{
  sum = *anon1 + sum;
  anon1 =  anon1 + 8;
}
```

Figure 3. The IR after PRE-OPT for ACCESS1 and ACCESS2

```
          Source Code
double A[1000];
for (int i = 0; i < 1000; i = i + 2)
  A[i] = A[i -1] + 5;

      WHIRL after PRE-OPT
_IEEE64 anon1[1000LL];
for(anon3 = 0; (anon3 * 2) <= 999; anon3 = anon3 + 1)
  anon1[anon3 * 2] = anon1[(anon3 * 2) + -1] + 5.0;
```

Figure 4. High level WHIRL after PRE-OPT for array references

LNO based strided prefetching algorithms can only issue prefetches for array reference in Figure 4, but miss out the opportunities for STL vector traversing in Figure 2, since the construction of locality equations depends on data space and iteration space representations, as well as explicit access function mapping from iteration space to data space. Thus LNO operates on high level WHIRL, where shape information of loop nest and data structures and subscripting functions are available. Without these information for STL vector traversing, LNO hardly recognize and model stride patterns to determine the right prefetching decisions. Instead, inductive expression recognition in global optimizer can identify these stride reference pattern and calculated the accurate stride value for further prefetching scheduling. In the example, all four STL vector traverses are stride reference with stride value 8.

LNO based prefetching algorithm for stride array references is based on the tight affinity with the other analysis and optimization phases in LNO, such as locality analysis, cache model. Conversely, this affinity limits itself only for stride array references in DO loops, and shows no benefits for STL vector references and the similar wrapper-based traversing at all. Instead, the tight coupling with induction variable recognition in global optimizer is amenable to generalized prefetching of different style stride references. This is the focus of our paper.

3.3 The introduction of global optimizer

Our algorithm benefits from the infrastructure provided by the global optimizer, which we summarize in the following.

In Open64's global optimizer, the IR has been translated to SSA form(Cytron et al. 1991). Furthermore, Open64 introduced the Hashed SSA (HSSA) form (Chow et al. 1996) which repre-

sents aliases and indirect references directly in SSA. For the real definition or use of a variable in program, it expresses potential definitions and uses for other variables in a program. To represent these May-Def and May-Use information, HSSA has introduced *chi* and *mu* into SSA form. SSA form greatly most of the global optimization related to data flow analysis, such as loop invariant determination, whose definitions are outside the loop.

In this paper, we show the tight affinity that the stride prefetching candidate identification equals to the recognition of inductive expression in global optimizer. Continuously lowered in global optimizer, the middle level WHIRL benefits the generalized stride prefetching algorithm for all stride references in all kinds of loop structures. Furthermore, there is only one primary induction variable in a loop after induction variable canonicalization phase in PRE-OPT, with other induction occurrences expressed in terms of the primary induction variable. Thus this greatly simplify the inductive expression recognition.

Section 4 describes inductive expression recognition and the algorithm using it to identify prefetching candidates. Section 5 explains phase ordering considerations, providing rationale for implementing our algorithm after SSAPRE. Section 6 presents the implementation details for other prefetching issues, such as prefetching hint selection. Section 7 shows the experiment results. We conclude in Section 8 and describe future work.

4. Approach

The effectiveness of software controlled stride prefetching is dependent on the compiler's ability to recognize as many strided references as possible to cover prefetching candidates. In the following, we first define inductive expressions and stride references, and then prove that all stride references that a compiler can identify statically are references whose addresses are inductive expressions. Thus, the essence of the stride prefetching candidate identification equals the recognition of inductive expression. We also discuss the details of inductive expression recognition in SSA form.

4.1 Basic definition

compiler's ability is to recognize as many strided references as possible to cover prefetching candidates. For the convenience of the following descriptions and proofs, we first define inductive expressions and stride references, and then prove that all stride references that a compiler can identify statically are references whose addresses are inductive expressions. Note that without explicitly declaration, all the definitions are integer type. Thus, the essence of the stride prefetching candidate identification equals the recognition of inductive expression. We also discuss the details of inductive expression recognition in SSA form.

Definition 1. *(Loop Invariant) A variable or an expression is said to be loop invariant if its value remains unchanged during the execution of all iterations in that loop.*

Definition 2. *(Linear Induction Variable) A linear induction variable is a variable that assigned in a loop and incremented by a nonzero loop invariant on every iterations. The corresponding invariant value is the stride value between two consecutive iterations. It the stride value is 1, it is primary linear induction variable.*

Thus, it is a subclass of sequence variable defined in(Gerlek et al. 1995) by Wolfe, which also contains polynomial induction variable, geometric induction variable, wrap-around variable, period sequence variable, and monotonic variable.

Definition 3. *(Linear Inductive Expression) A linear inductive expression is an expression whose value is incremented by a nonzero loop invariant on every iterations. The corresponding invariant value is the stride value between two consecutive iterations.*

Definition 4. *(Linear Inductive Expression) A linear inductive expression is recursively defined as following:*

1. If v is a linear induction variable with stride s, then v is a linear inductive expression with the same stride s;
2. If $expr$ is a linear inductive expression with stride s, then $-expr$ is a linear inductive expression with the same stride $-s$;
3. If $expr$ is linear inductive expression with stride s and invar is a loop invariant, then $expr + invar$ and $invar + expr$ are all inductive expressions with stride s;
4. If $expr1$ and $expr2$ are linear inductive expressions with stride $s1$ and $s2$ respectively, then $expr1 + expr2$ is a linear inductive expression with stride $s1 + s2$;
5. If $expr$ is linear inductive expression with stride s and $invar$ is a loop invariant, then $expr * invar$ and $invar * expr$ are all inductive expressions with stride $invar * s$;
6. If $expr$ is linear inductive expression with stride s and $invar$ is a loop invariant, then $expr/invar$ is a linear inductive expression with stride $s/invar$.

This recursive definition is useful for the linear inductive expression recognition. Mathematically, it equals to the linear combination of linear induction variables and loop invariants, with the form:

$$E = c_1 i_1 + c_2 i_2 + \ldots + c_n i_n + invar \quad (1)$$

where $c_1 \ldots c_n$ and $invar$ are loop invariants, $i_1 \ldots i_n$ are linear induction variables.

Lemma 1. *Linear inductive expression in Definition 3 can be and can only be in recursive form in Definition 4.*

Proof. The proof of lemma 1 equals to the proof that the linear inductive expression must be of the form (1).

Sufficient Condition: Suppose E is the linear combination of linear induction variables and loop invariants with the form (1), the value of E in two consecutive iterations is V_1 and V_2 respectively, then

$$V_1 - V_2 = c_1(i'_1 - i''_1) + c_2(i'_2 - i''_2) + \ldots + c_n(i'_n - i''_n) \quad (2)$$

Since $i_1 \ldots i_n$ are induction variables, $invar$ is a loop invariant, thus the value of formula (2) is a loop invariant. The stride s_j of i_j $(1 \leq j \leq n)$ is $(i'_j - i''_j)$, and the stride of E is $\sum s_i c_i$.

Necessary condition: We prove the necessary condition by contradiction. If an expression is not represented as the linear combination of derived induction variables and loop invariants, then there are two situations: (1). the $i_1, \ldots, i_n, invar$ are not linear inductive variables, such as common variables and pointer dereferences; (2). The expression is represented as non-linear combination of variables, such as $i_1 \times i_2$. We can get the conclusion that $V_1 - V_2$ in situation both (1) and (2) is not guaranteed to be constant. □

Without ambiguity, we call linear induction variable and linear inductive expression as induction variable and inductive expression respectively for short.

Definition 5. *(Stride Reference) A stride reference is the reference in a loop whose accessed memory address is incremented by a loop invariant on every iterations. The corresponding invariant value is the stride value between two consecutive iterations.*

Lemma 2. *If a reference in loop whose accessed memory address is represented as an inductive expression, then it is a stride reference. Obviously, Definition 3 and Definition 5 are identical.*

Therefore, the identification of stride prefetching candidate equals to the recognition of inductive expression with the recursive definition form.

Recent studies(Wu 2002; Stoutchinin et al. 2001) show that there are irregular references whose stride of accessed address for two consecutive iterations appears as a constant value, such as `p = p->next` if the list is allocated in a sequential memory space. Apparently, this implicit stride information is not guaranteed in run time and just a probability. A compiler cannot detect them statically, only with some tricks such as the help of feedback information, by dynamic instrumentation, or sampling techniques. This scenario is not covered by our static prefetching algorithm, but it can be seamlessly integrated. We discuss this in the section on future work.

4.2 Algorithm in detail

In the following, we present an algorithm for demand driven speculative recognition of prefetching candidates. It is shown by Figure 5 and Figure 6.

Function `Collect_Indirect_Reference` creates a work-list by collecting all of the occurrences of indirect references in loop nests, and store the needed information of the enclosing statement and loop. Function `Sort_Worklist` sorts all the collected occurrences in ascending order according to the number of the contained leaf operands. This avoids repetitively recognize the inductive expression.

Function `Inductive_Expression_Recognition` determines whether the input expression is an inductive expression, which has the recursive form as in the Definition 4 in section 4.1. For simplicity, we only list the pseudo code corresponding to (1), (3) and (4) in Definition 4. It will call the function `Induction_variable_Recognition` to speculative determine whether one variable is induction variable.

The method used to recognize induction variables, demonstrated by Figure 6, exploits the fact that the use-def chains of the SSA versions of induction variable must form a strongly connected component (SCC), presented by(Gerlek et al. 1995). Our algorithm is different from theirs as we use speculative symbolic evaluation in the recognition process. Our algorithm is also different from the method presented at(Liu et al. 1996), since we do demand driven recognition by creating a sorted work list.

Induction variables in SSA form must satisfy the following condition: (1). there must be a live phi in the corresponding loop header BB; (2). among the two operands of the phi, the loop invariant operand must point to the initialization of the induction variable out of the loop, while the other operand must be defined within the loop body. We call them init and increment respectively; (3). After expanding the increment operand of phi by copy propagation, the expanding result must contain the result of that phi, with a loop invariant expression as stride of the induction variable.

Our optimizer will control the copy propagation such that an expression is not propagated if it causes the live range of any variable versions in the expression to overlap with another version of the same variable. Aliasing will also block the copy propagation as well. Thus we must extend the expanding algorithm in `Induction_Varaible_Recognition` to resolve the variables through symbolic interpretation, which does not perform real copy propagation. This is implemented as function `Expand_Conservative` and `Expand_Aggressive`, which do speculative copy propagation backwards to expand the expressions to be comprised of phi or loop invariant operands.

The other difference with existing induction variable recognition algorithm is that we do speculative determination by aggressive copy propagation across chi nodes in HSSA. A chi node is attached to store statements, representing the May-Define relation introduced by alias. A variable is defined by chi, implying that it may be aliased with the variable really defined by the statement which the chi is attached to. This will block the copy propagation, thus block the following induction variable recognition. Since data prefetching are presumed to not cause correctness issues on most architectures, we can identify prefetching candidates more aggressively than existing algorithms. The corresponding details are shown in function `Expand_Conservative` and `Expand_Aggressive`, In `Expand_Conservative`, we pessimistically expand the expressions by symbolic interpreting the result of chi as the right hand side value of the statement chi attached speculatively. Here we pessimistically assume that the alias statically analyzed by compiler is a MUST alias relation in runtime. In contrast, `Expand_Aggressive` optimistically assume the alias result is wrong in runtime, thus speculatively copy propagate the value of the operand of chi to the result of that chi. As a result, we can recognize more inductive expression as stride prefetching candidates. Note that we apply on the same pessimistic or optimistic assumption for all of the chi nodes, either MUST or MUST NOT alias. This will reduce the complexity and avoid too many prefetching candidates by false stride references. The function `Insert_Prefetch_For_Ref` will be illustrated in section 5.

5. Phase Ordering Consideration

The global optimizer generally includes several important optimizations such as partial redundancy elimination, strength reduction, register variable identification and dead code elimination. They are tightly related to each other, and share some common utilities such as induction variable recognition, loop invariant determination, dead code elimination, and expression simplification, constant-folding and so on. From the view of both engineers and researcher, the right ordering between different optimizations will result in an elegant solution and reduce the complexity of implementation. In this section, we will discuss the tight relations between inductive expression recognition and SSAPRE in our global optimizer.

Our global optimizer has integrated strength reduction within PRE optimization in SSAPRE framework. For distinguishing killing definition, it defines the injuring definition with the form `a = a + invariant` to identify the candidates for strength reductions. This benefits the optimization on all inductive expressions in loops, whose incremental update is injuring definition.

Strength reduction will introduce a new secondary induction variable to replace injured inductive expression. This greatly simplifies the inductive expression recognition if we implement our inductive prefetching algorithm after SSAPRE, since almost all of the address expressions for array references are converted to induction variables by strength reduction.

The existence of temporal locality for a memory reference in a loop implies the corresponding address expression is the corresponding loop invariant. The loop invariant hoisting inherently in SSAPRE will hoist all the loop invariant expression outside of the loop, including indirect memory reference, which are well known as load and store PRE or register promotion(Lo et al. 1998). As a result, if we implement our inductive prefetching algorithm after SSAPRE, the occurrences of memory references which appear locality will be hoisted outside the loop, thus will not be collected in the prefetching worklist. This can automatically avoid the unnecessary prefetches in inner loop due to temporal locality.

These characteristics can be shown in Figure 7. The corresponding source code is shown by (a), and (b) is as the input of SSAPRE, where the address expression of $A[i][0]$ is $\&A + i * 400$ in inner loop level j. It is loop invariant in loop j, then PRE will hoist it outside of loop j and replace the real occurrence with $preg1$, shown by (c). Then SSAPRE will do strength reduction on injured expression

```
int Induction_Variable_Recognition (var)
{
  phi = Get_phi (var);
  if (phi->Bb() != Header()) return 0;
  for each opnd of phi {
    if (Invariant(opnd))
      { init = opnd; continue; }
    expr1 = Expand_Conservative (opnd);
    expr2 = Expand_Aggressive (opnd);
    if (expr1 contains phi result)
      step = Generate_step (expr1);
    else if (expr2 contains phi result)
      step = Generate_step (expr1);
    if (Loop_invariant(step))
      { incr = opnd; return step; }
  }
}
int Inductive_Expression_Recognition (ref)
{
  if ref is a vairiable {
    return Induction_Varaible_Recognition (ref);
  } else if ref is (expr1 + expr2) {
    stride1 = Inductive_Expression_Recognition (expr1);
    stride2 = Inductive_Expression_Recognition (expr2);
    if (stride1 && stride2) stride = stride1 + stride2;
    if (stride1 && expr2 is loop invariant)
      stride = stride1;
    if (stride2 && expr1 is loop invariant)
      stride = stride2;
  } else if ref is (expr1 - expr2) {
    stride1 = Inductive_Expression_Recognition (expr1);
    stride2 = Inductive_Expression_Recognition (expr2);
    if (stride1 && stride2) stride = stride1 - stride2;
    if (stride1 && expr2 is loop invariant)
      stride = stride1;
    if (stride2 && expr1 is loop invariant)
      stride = -stride2;
  } else if ref is (expr + invar) { ... }
    else if ref is (expr1 * expr2) { ... }
    else if ref is (expr1 / expr2) { ... }
    else if ref is (-expr) { ... }
    else return 0;
  return stride;
}
Stride_Prefetching (loop)
{
  Candidate_List_Clear ();
  Collect_Indirect_Reference ();
  Sort_Worklist ();
  for each reference ref ()
    if (Inductive_Expression_Recognition ())
      Insert_Prefetch_For_Ref (ref);
}
```

Figure 5. Inductive Expression Recognition

```
Get_phi (var)
{
  if (var is defined by phi) return var;
  def = Get_def (var); // use-def chain
  return Get_phi (def);
}
Expand_Conservative (expr)
{
  if (expr is variable) {
    if (Invariant(expr)) return NULL;
      if (expr is real defined by statement)
        expr = defstmt->Rhs();
        // conservatively assume MUST alias
      else if (expr is defined by chi)
       expr = defstmt->Rhs();
      else if (expr is defined by phi) return NULL;
       return Expand_Conservative (expr);
    } else if (expr is operator)
        for each opnd of expr
        expr->Set_opnd (Expand_Conservative (opnd));
}
Expand_Aggressive (expr)
{
  if (expr is variable) {
    if (Invariant(expr)) return NULL;
      if (expr is real defined by statement)
        expr = defstmt->Rhs();
     // aggressively assume MUST alias
    else if (expr is defined by chi)
      expr = defchi->opnd;
    else if (expr is defined by phi) return NULL;
      return Expand_Conservative (expr);
  } else if (expr is operator)
     for each opnd of expr
      expr->Set_opnd (Expand_Conservative (opnd));
}
```

Figure 6. Induction Variable Recognition

$i*400$, by introducing a second induction variable $preg2$, replacing `&A` $+ i*400$ as `&A` $+ preg2$, and inserting $preg2 = preg2 + 400$ at the end of the exit bb of loop I, shown by (d). Afterwards, SSAPRE will do the same process on `&A` $+ preg2$, which is injured by $preg2$, by introducing another secondary induction variable $preg3$, replacing `&A` $+ preg2$ with $preg3$, and inserting $preg3 = preg4 + 400$ at the corresponding exit bb, shown by (e). Finally, SSAPRE will do linear function test replacement(Kennedy et al. 1998) by replacing loop-exit condition $i < 100$ with $preg3 <$ `&A` $+ 40000$, then eliminating the dead variable i, shown by (f).

It benefits to implement our demand-driven recognition of inductive expression after SSAPRE. Shown from Figure 7, the worklist construction phase of our algorithm will only collect $*preg3$ in loop i as the prefetching candidate to automatically avoid inserting the prefetches in inner loop j since temporal locality exists in that loop. Another important character is that the inductive expression before SSAPARE `&A` $+ i*400$ was automatically replaced as a new secondary induction variable $preg3$ by strength reduction in SSAPRE. Thus it can automatically simplify the inductive expression recognition in our algorithm.

By making full use of the optimization result of SSAPRE, our algorithm based on inductive expression recognition can be implemented efficiently to be integrated as scalar optimization in global optimizer.

6. Implementation

Due to the phase ordering considerations elaborated on in section 5, we have implemented our algorithm after SSAPRE in the global optimizer of the Open64 compiler. The following will describe the remaining task to generate prefetches effectively and efficiently after recognizing the stride references.

Besides improving the prefetching coverage, it is beneficial to avoid unnecessary prefetches and to schedule prefetches on time to improve cache efficiency. Target dependent abstraction and tuning can be well integrated into our framework by the cache model. Thus precise information on prefetch distances, locality hints, and

```
int A[100][100];

for (int i = 0; i < 100; ++i)
   for (int j = 0; j < 100; ++j)
      A[i][0] +=j;
```
(a)

```
loop_1:
   loop_2:
      *(&A + i * 400) = *(&A + i * 400) + j;
      j = j + 1;
   if (j < 100) goto loop_2;
   i = i + 1;
if (i < 100) goto loop_1;
```
(b)

```
loop_1:
   preg1 = *(&A + i * 400)
   loop_2:
      preg1 = preg1 + j;
      j = j + 1;
   if (j < 100) goto loop_2;
   i = i + 1;
if (i < 100) goto loop_1;
```
(c)

```
preg2 = 0;
loop_1:
   preg1 = *(&A + preg2);
   loop_2:
      preg1 = preg1 + j;
      j = j + 1;
   if (j < 100) goto loop_2;
   preg2 = preg2 + 400;
   i = i + 1;
if (i < 100) goto loop_1;
```
(d)

```
Preg3 = &A;
preg2 = 0;
loop_1:
   preg1 = *preg3;
   loop_2:
      preg1 = preg1 + j;
      j = j + 1;
   if (j < 100) goto loop_2;
   preg2 = preg2 + 400;
   preg3 = preg3 + 400;
   i = i + 1;
if (i < 100) goto loop_1;
```
(e)

```
preg3 = &A;
loop_1:
   preg1 = *preg3;
   loop_2:
      preg1 = preg1 + j;
      j = j + 1;
   if (j < 100) goto loop_2;
   preg3 = preg3 + 400;
if (preg3 < A + 40000) goto
loop_1;
```
(f)

Figure 7. The demonstration of strength reduction with SSAPRE

prefetch scheduling should be determined, according to both the cache model and the collected memory references.

Our prefetching algorithm includes the following steps to optimize prefetching effects.

1. Prefetching candidate identification. All the indirect references are the prefetching candidates if their address expressions are recognized as inductive expressions. This is accomplished by demand driven speculative recognition of inductive expression as discussed before.
2. Leading reference recognition. A leading reference is the first reference accessing a memory location among all references. For example, among references $A[i]$ and $A[i+1]$, $A[i]$ is the leading reference if i steadily decrease, otherwise, $A[i+1]$ is the leading reference. This can be done through simple arithmetic comparison between the corresponding init and stride values of the recognized induction variables.
3. Prefetch information collection. This includes information of reference stride, the corresponding data and loop structure, as well as the target cache model. This data is collected while creating the prefetching worklist. If the corresponding accurate information cannot be determined at compile time, we define them with help of heuristics.
4. Prefetch determination based on the collected information. We don't need to generate prefetches for all the prefetch candidates to avoid too much prefetching. There are some heuristic conditions used to exclude the prefetch candidates: (1) Not leading reference; (2) very small array less than one cache line and very small loop trip counts; (3) Current loop contains more than N prefetches (N currently set to 10); (4) For small arrays, generally if smaller than 3 cache lines, we perform outer-loop prefetching to hoist prefetch outside the current loop, and don't generate the inner loop prefetch any more. If the total data size accessed in the loop is smaller than the effective cache size of computed cache level, we do the aggressive outer-loop prefetch by generating a new loop consisting only of prefetch statements.
5. Computation of prefetch distance. Prefetching distance is the address distance between the really accessed address and the prefetched address in current iteration. Too late or too early prefetch would make prefetching ineffective. In general, the number of iterations prefetched ahead of use is calculated as the division of memory latency and the estimated time per iteration in that loop; prefetch distance equals to the multiplication of the computed number of iterations ahead and the reference stride.
6. Loop transformations based on locality information to further reduce the number of prefetches. For spatial locality, loop unrolling has precedence to prefetch predicate. In practice, the unroll factor equals to the division of cache line size and the reference stride. The temporal locality of prefetched occurrences only exists in its parent loop since PRE has exploits the locality in inner loop, shown in Figure 7. However, if the array references is $A[0][j]$ in , PRE cannot hoist the address expression of $\&A + j * 4$, which is killed by the update of j in loop level j. We will do loop splitting on the outmost loop level where locality exists, which based on the same strategy in LNO based prefetching algorithm(Mowry et al. 1992).
7. Prefetch statement insertion. Since we will insert prefetches about n iterations ahead of the real references, thus the first n data would not be prefetched and the last n data prefetches are unnecessary. We will add the prologue and epilogue sections as presented at(Mowry et al. 1992).

7. Experiments

We have conducted experiments against SPEC2006 benchmark on IA64 platform, which has no hardware prefetching support and is the good choice for comparison between different software prefetching algorithms. The results indicate that we have competitive performance to the LNO based algorithm, with the added benefits that the potential prefetching opportunities for STL vector style code can be materialized in many cases without having to invoke the loop nest optimizer.

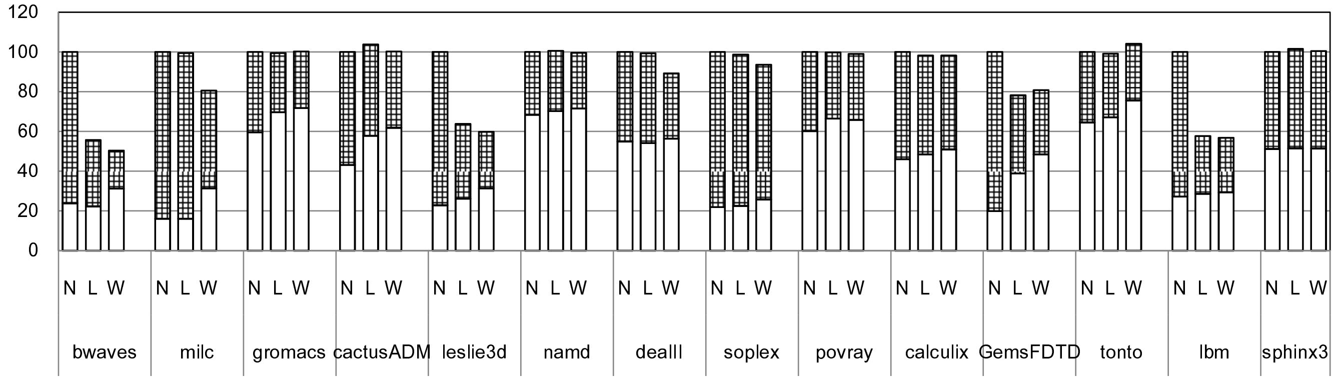

Figure 8. Normalized CPU cycles of SPEC 2006 FP evaluation

Our experiments are performed on a quad-processor server with Redhat Linux Advanced Server 4.0 and 8 GBytes memory installed. The processor is Itanium 2 Madison 1.6GHz with 6MB L3 cache. The base compiler is Open64 4.1, which has implemented stride prefetching algorithm in LNO, and yields much better performance gains than gcc 4.1, while having comparable prefetching result with icc.

We normalized the execution time of benchmarks compiled with LNO prefetching enabled and WOPT prefetching enabled respectively, where the baseline is the running time of program without prefetching, which is normalized to be 100. All benchmarks are compiled at -O3 optimization level. To prove the effectiveness of our approach, we compare our approach with the original LNO based prefetching algorithm in Open64. The experimental results on SPEC2006 FP and INT are shown in Figure 8 and Figure 9 respectively. For each benchmark, the three bars correspond to the cases with no prefetching (N), LNO based prefetching algorithm (L) and our WOPT based algorithm (W). In each bar, the bottom section is the amount of time spent on executing instructions (including instruction overhead of prefetching), and the section above is the memory reference time due to data cache misses. It is well known that data prefetching is used to hide the memory wall time through reducing the cache misses. Thus, the more the memory stall time is eliminated, the better the prefetching algorithm is.

We can attain the following two experiment results:

1. The effectiveness of LNO prefetching algorithm. LNO prefetching can handle almost all of the array references. Compared with the result of benchmarks with prefetching disabled, LNO prefetching has improved about 44.21%, 42.22%, 36.1% and 21.8% for 410.bwaves, 470.lbm 437.leslie3d and 459.GemsFDTD in SPEC2006 FP respectively. Even for non scientific computing benchmarks in SPEC2006 INT, LNO based prefetching still get about 9.1% and 4.8% improvements for 429.mcf and 401.bzip2 respectively.
2. The effectiveness of our global optimizer based prefetching algorithm. Our approach shows 49.65%, 43.13%, 40.13% and 19.40% performance gains for 410.bwaves, 470.lbm 437.leslie3d and 459.GemsFDTD in SPEC2006 FP respectively. And for SPEC2006 INT it gets about 25.9% and 3.4% gains 429.mcf and 401.bzip2 respectively. For these benchmarks benefited from LNO based prefetching, our approach gets about 5.44%, 0.91%, 7.03%, -1.4%, 14.8% and -1.4% further gains respectively compared with LNO prefetching. Besides, our approach has also improved 19.4%, 10.75% and 6.36% for 433.milc, 447.dealII and 450.soplex in SPEC2006 FP respectively, which LNO cannot improve the performance. And in SPEC2006 INT, our approach shows 50.07% and 4.78% for 462.libquantum and 403.gcc respectively.

Note that 447.deall contains many STL vector style traverses which can be handled well by our approach. Even libquantum and milc are not C++ programs and dont include STL vector traversing, our approach still outperforms LNO based algorithms. Because they have irregular loop nests and wrapped compound structure, which cannot be handled by LNO based algorithm, while can be covered by our approach. In theory, we can cover all of the strided prefetching candidates determined statically during compile time. So our WOPT-based approach can be used to replace rather than complement LNO-based algorithms.

In some benchmarks, the issued prefetches even harms performance, e.g. 456.hmmer, due to the overhead introduced by prefetch instructions and related memory impacts.

8. Conclusion and Future work

In this paper, we propose an alternative inductive data prefetching algorithm implemented in global optimizer. Compared with most known stride prefetching algorithms, our algorithm can identify more prefetch candidates. As far as I know, our inductive stride prefetching algorithm is the first static compiler technique proposed to do prefetching for STL vector and other wrapper-based stride references, shown in Abstraction Penalty Benchmark.

There are two main advantages over existing LNO based algorithm: (1). the prefetching decision is regarded as the inductive expression identification issue in global optimizer (-O2 level), rather than as the time/space consuming loop nest optimization (-O3 level); (2). it has higher prefetching coverage and materialize the potential prefetching opportunities in applications containing STL vector and other wrapper-based stride references.

Thus, it provides the possibility that implementing stride prefetching as common scalar optimization fed by data flow information in global optimizer, and shows the tight coupling between stride data prefetching and inductive expression recognition, as well as SSAPRE with strength reduction optimization.

Besides linear inductive expression defined in our paper, we plan to extend the prefetching candidate identification to cover non-linear sequence forms defined in, including periodic, polynomial, geometric, monotonic and wrap-around variables.

We will also try to integrated stride prefetching algorithm with strength reduction optimization in SSAPRE framework. This is partially done, but some inductive expressions cannot be handled by strength reduction due to alias issue. As this paper described, we plan to extend strength reduction by symbolic interpretation to fully integrated stride prefetching into SSAPRE framework. Global

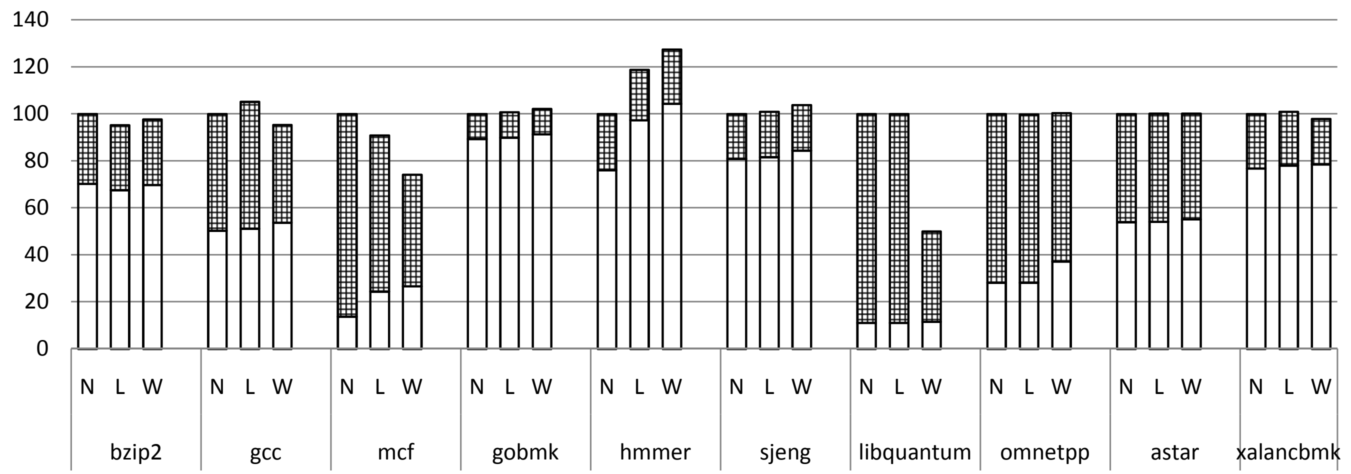

Figure 9. Normalized CPU cycles of SPEC 2006 INT evaluation

optimizer has the combination of loop and dependence analysis to capture the repetitive references; it may be a good choice to implemented pointer chasing style prefetch in global optimizer, hybrid with current algorithm to provide a wider domain covered by global optimizer based prefetching framework.

It is also interesting to coordinate the data prefetch with data layout optimization, to solve the memory wall in a better way. In addition, the analysis of impact of different parameter on prefetching is useful, from which we could get more real heuristics to prefetch decisions. Feed back guided heuristics selection will be more effective way to provide a hybrid prefetching framework integrating static and dynamic information.

Considering that X86 platforms are overwhelmed and provide the hardware prefetching supports, we will further investigate the interaction between software and hardware prefetching according to the static compiler analysis and feedback information.

Acknowledgments

We would like to thank to Robert Hundt and Sun Chan for their comments and revisal.

References

The open64 compiler. URL `http://www.open64.net`.

Stefan G. Berg. Cache prefetching. Technical report, March 05 2002.

Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In Burton J. Smith, editor, *ICS*, pages 202–209. ACM, 2007. ISBN 978-1-59593-768-1.

David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. pages 40–52, 1991.

Tien-Fu Chen and Jean-Loup Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Trans. Computers*, 44(5): 609–623, 1995.

Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. pages 199–209, 2002.

Fred Chow, Sun Chan, Shin ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form, December 24 1996.

Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *PLDI*, pages 273–286, 1997.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation (+ corrigendum). *IJCM: International Journal of Computer Mathematics*, 1989.

Vikram M. Dhaneshwar and Dhananjay M. Dhamdhere. Strength reduction of large expressions. *J. Prog. Lang*, 3(2), 1995.

Michael P. Gerlek, Eric Stoltz, and Michael 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, January 1995. ISSN 0164-0925.

Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 309–318, May 1991.

Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Trans. Computers*, 48(2), 1999.

Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via SSAPRE. In *Computational Complexity*, pages 144–158, 1998.

Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999. ISSN 0164-0925.

Uday P. Khedker and Dhananjay M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, September 1994.

J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

Shin-Ming Liu, Raymond Lo, and Fred Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 228–237, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pages 222–233. ACM SIGARCH/SIGOPS/SIGPLAN, October 1996.

Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.

T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, pages 62–73, 1992.

D. Patterson. *Computer architecture: a quantitative approach*. Morgan-Kaufmann, Los Altos, 1990.

Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA*, pages 111–121, 1999.

Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data prefetching on the hp pa-8000, 1997. ISSN 0163-5964.

Artour Stoutchinin, Jose Nelson Amaral, Guang R. Gao, James C, Suneel Jain, and Alban Douillet. Speculative prefetching of induction pointers, March 07 2001.

Vanderwiel and Lilja. Data prefetch mechanisms. *CSURV: Computing Surveys*, 32, 2000.

Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44, 1991.

Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *ACM SIGPLAN Notices*, 37(5):210–221, May 2002. ISSN 0362-1340.