Optimizing Software Data Prefetches with Rotating Registers

Gautam Doshi
Intel Corporation
2200, Mission College Blvd
Santa Clara, CA 95052
+1 (408) 765-4601
gautam.doshi@intel.com

Rakesh Krishnaiyer
Intel Corporation
2200, Mission College Blvd
Santa Clara, CA 95052
+1 (408) 765-7948
rakesh.krishnaiyer@intel.com

Kalyan Muthukumar
Intel Technology India Pvt Ltd
17, Mahatma Gandhi Road
Bangalore 560 001, India
+91 (80) 555-0940
kalyan.muthukumar@intel.com

Abstract

Software data prefetching is a well-known technique to improve the performance of programs that suffer many cache misses at several levels of memory hierarchy. However, it has significant overhead in terms of increased code size, additional instructions, and possibly increased memory bus traffic due to redundant prefetches. This paper presents two novel methods to reduce the overhead of software data prefetching and improve the program performance by optimized prefetch scheduling. These methods exploit the availability of rotating registers and predication in architectures such as the Itanium™ architecture. The methods (1) minimize redundant prefetches, (2) reduce the number of issue slots needed for prefetch instructions, and (3) avoid branch mispredict penalties – all with minimal code size increase. Compared to traditional data prefetching techniques, these methods (i) do not require loop unrolling, (ii) do not require predicate computations and (iii) require fewer machine resources. One of these methods has been implemented in the Intel Production Compiler for the Itanium™ processor. This technique is compared with traditional approaches for software prefetching and experimental results are presented based on the floating-point benchmark suite of CPU2000.

1. Introduction

Modern microprocessors run an order of magnitude or more faster than main memory clock speeds. This results in high access times for data in memory. Caches at several levels are used to alleviate this problem. Typically, the closer the cache is to the processor in the memory hierarchy, data access latencies and the cache sizes are smaller. As a result, accesses to data in caches are faster than accesses to main memory, even though the capacities of these caches are smaller. Furthermore, programs typically exhibit temporal and spatial locality of accesses - hence a small storage can service a large fraction of the data requests.

Unfortunately, caches are not as effective in programs that access large amounts of data in loops. Such programs suffer many cache misses at several levels of memory hierarchy. This significantly reduces their performance. Software Data Prefetching [1] is a well-known technique to improve the performance of such programs. In this technique, a special prefetch instruction is issued to a cache line well before accessing the data from that cache line. When the load instruction is issued to access the data, the cache line containing the data is already in the cache - since the prefetch instruction brought the cache line from memory into the cache. Thus data prefetching hides the cache miss latencies for data accesses during program execution and improves application performance.

However, there are some costs associated with issuing data prefetches:

(i) Code size - Since the unit of data addressed by a prefetch instruction (a cache line) is typically larger than the unit of data accessed by individual data accesses, a data prefetch is needed only once for many iterations of a loop. To minimize the issuance of prefetches, loops are unrolled, increasing code size.

(ii) Instruction overhead – As an alternative to loop unrolling, when the architecture supports predication, data prefetches can be scheduled in the loop, but turned on or off for any given iteration by calculating and using appropriate predicates. This results in increased instruction overhead for computing such predicates. In the absence of predication, branches can be inserted in the loop that periodically effect control transfer to code that

prefetches the required data. These branches would then make other optimizations, such as software pipelining, much more difficult.

(iii) Increased resource requirements – Additionally, when multiple prefetches to different arrays are scheduled in a given loop iteration, they all tend to miss in the cache at the same time (since arrays are typically aligned to cache line boundaries). This usually causes a sudden increase in machine resources required to handle those misses. When the arrays being prefetched are multidimensional the mutual address alignment between arrays changes from iteration to iteration – making it difficult for the compiler to proactively arrange the array starts. Such sudden resource requirements often result in additional execution delays, thus lowering performance. Staggered issue of the prefetches that miss in the cache is desirable.

This paper presents two new methods to reduce the overhead of software data prefetches. These methods also improve the program performance by optimized prefetch scheduling. These methods exploit the availability of rotating registers and predication to reduce the instruction overhead associated with data prefetches. They do not require unrolling of the loop thereby reducing the instruction overhead. At the same time, they do not require expensive predicate computations. These methods are applicable to loops that can be software-pipelined.

The methods employ two key architectural features – register rotation (a form of hardware register renaming) and predication. Register rotation can be used to transform the behavior of an instruction across multiple executions and predication enables conditional instruction execution. In one method, the target address prefetched by a single prefetch instruction is altered using rotating registers. This functionality maps precisely to the prefetching requirements of loops that access multiple arrays. A single prefetch instruction, over successive loop iterations, cycles through the task of prefetching data for multiple arrays. In the second method, a set of rotating predicates is used to guard the execution of prefetch instructions to multiple arrays. By initializing these predicates appropriately each prefetch instruction can be selectively and successively "turned on" during different loop iterations.

One of these methods has been implemented in the Intel Production Compiler for the Itanium™ processor. Experimental results by using this method show significant speedups for floating-point programs in the CPU2000 benchmark suite. They compare very favorably with other software prefetching techniques that either use unconditional prefetches or conditional prefetches using predication.

The rest of this paper is organized as follows. Section 2 describes Software Prefetching and presents the traditional approaches to software prefetching. Section 3 discusses the features of the Itanium™ architecture (such as rotating registers, predication, and software-pipelined branches) that are useful for our methods. Section 4 presents the two methods for software prefetching – one that uses rotating integer registers and another that uses rotating predicate registers. Section 5 presents experimental results of using the software-prefetching method on the floating-point programs in the CPU2000 suite of benchmarks. These are compared with results from other (traditional) prefetching techniques. Section 6 provides a summary and directions for future work.

2. Software Data Prefetching

Software Data Prefetching ([13],[1],[9]) works by bringing in data from memory well before it is needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves performance. This works best for programs that have array accesses in which data access patterns are regular. In such cases, it is possible to predict ahead of time the cache lines that need to be brought from memory.

Some work has been done on data prefetching for non-array accesses as well ([7],[8]). However, in this paper, we consider prefetching only for array accesses.

Typically, when a cache line is brought in from memory, it contains several elements of an array that is accessed in the loop. This means that a new cache line is needed for this array access only once in several iterations. This depends, of course, on several factors such as the stride of the array access pattern, the cache line size, etc. The key is that, a prefetch instruction is needed only once in a few iterations for a given array access. In fact, if redundant prefetch instructions are issued in every iteration of the loop, it may hurt performance. There are two reasons for this:

- Prefetch instructions require extra issue slots and thereby increase the cycle time per iteration, if they are executed every iteration of the loop.
- Redundant prefetch instructions can overload the memory subsystem and thereby adversely affect the performance.

Redundant prefetches can be avoided by using one of three techniques: (1) unroll the loop and issue a prefetch only once in several original loop iterations, (2) have branch instructions that branch to prefetch code only once in a few iterations, or (3) have conditional prefetch instructions whose predicates are True only once in a few iterations. Below, we describe these methods using the following running example:

```
Orig_loop:
     Load A(I)
     Load B(I)
     Load C(I)
     Load D(I)
     Load E(I)
     ...
     I = I + 1
     Br Orig_loop if more iterations
```

Figure 1. Running Example Loop

Suppose that these arrays all have element sizes of 8-bytes and the machine has a cache line size of 64-bytes. Each array needs one prefetch instruction per 8 iterations. However, there are five different arrays, which implies that we need 5 prefetch instructions per 8 iterations.

2.1 Prefetch After Unroll

This method unrolls the example loop 8 times and issues 5 prefetches in every iteration of the unrolled loop. The loop is transformed as follows:

```
Unr_Loop:
     prefetch A(I+X)
     load A(I)
     load B(I)
     load C(I)
     load D(I)
     load E(I)
     ...
     prefetch B(I+X)
     load A(I+1)
     load B(I+1)
     load C(I+1)
     load D(I+1)
     load E(I+1)
     ...
     prefetch C(I+X)
     load A(I+2)
     load B(I+2)
     load C(I+2)
     load D(I+2)
     load E(I+2)
     ...
     prefetch D(I+X)
     load A(I+3)
     load B(I+3)
     load C(I+3)
     load D(I+3)
     load E(I+3)
     ...
     prefetch E(I+X)
     load A(I+4)
     load B(I+4)
     load C(I+4)
     load D(I+4)
     load E(I+4)
     ...
     load A(I+5)
     load B(I+5)
     load C(I+5)
     load D(I+5)
     load E(I+5)

     load A(I+6)
     load B(I+6)
     load C(I+6)
     load D(I+6)
     load E(I+6)

     load A(I+7)
     load B(I+7)
     load C(I+7)
     load D(I+7)
     load E(I+7)

     I = I + 8
     Br Unr_Loop if more iterations
```

Here, the value of X is determined by (a) the cycle time per iteration of the loop, (b) strides for the array memory accesses, and (c) the latency of a cache miss. This approach minimizes redundant prefetches and minimizes the number of prefetch instructions required to hide the effect of cache misses. However, this has the disadvantage of increasing the code size due to unrolling and increased register pressure. This can also adversely affect performance due to increased instruction cache misses, address translation misses, and page faults.

2.2 Branches to Prefetch Code

Another approach to minimize redundant prefetch instructions is to have branches to prefetch code. These branches are taken only once in a few iterations, thereby issuing prefetches only when necessary. The example is transformed as follows in this method:

```
Orig_loop:
     Load A(I)
     Load B(I)
     Load C(I)
     Load D(I)
     Load E(I)
     Cmp pA = (I mod 8 == 0)
If(pA) Br to code that prefetches A
     Cmp pB = (I mod 8 == 1)
If(pB) Br to code that prefetches B
     Cmp pC = (I mod 8 == 2)
If(pC) Br to code that prefetches C
     Cmp pD = (I mod 8 == 3)
If(pD) Br to code that prefetches D
     Cmp pE = (I mod 8 == 4)
If(pE) Br to code that prefetches E
     ...
     I = I + 1
     Br Orig_loop if more iterations
```

The main problem with this approach is the addition of five branches that could be mispredicted by the processor leading to increased cycle times per iteration of the loop. Also, five compare instructions have been added to the loop, which could also reduce performance. Furthermore, such a transformation makes it harder to software pipeline

this loop relative to the original or unrolled loop.

2.3 Conditional Prefetch Instructions

The third approach to minimize redundant prefetches is to have conditional prefetch instructions. This is possible in architectures that support predication or conditional/guarded execution. The example loop is transformed as follows by this method:

```
Orig_loop:
     Load A(I)
     Load B(I)
     Load C(I)
     Load D(I)
     Load E(I)
     Cmp pA = (I mod 8 == 0)
If(pA) prefetch A(I+X)
     Cmp pB = (I mod 8 == 1)
If(pB) prefetch B(I+X)
     Cmp pC = (I mod 8 == 2)
If(pC) prefetch C(I+X)
     Cmp pD = (I mod 8 == 3)
If(pD) prefetch D(I+X)
     Cmp pE = (I mod 8 == 4)
If(pE) prefetch E(I+X)
     ...
     I = I + 1
     Br Orig_loop if more iterations
```

This method avoids the branch misprediction penalties associated with the previous method. However, it needs extra issue slots for the five prefetch instructions that are issued every iteration of the loop. Alternatively, a single prefetch instruction could be used and each compare could guard a register move instruction to setup the register used by the single prefetch instruction with the desired array address. In either case, the extra compares and arithmetic instructions (for predicate computation) would be required.

It can thus be seen that the traditional approaches to Software Prefetching run into one problem or other and thus reduce the effectiveness of prefetching due to their increased overhead. The methods that we describe in this paper solve these problems by cleverly making use of rotating registers to (1) minimize redundant prefetches, (2) reduce the number of issue slots needed for prefetch instructions, and (3) avoiding branch mispredict penalties – all with minimal increase of code size or additional instructions for prefetch control. The next section describes some of the relevant features of the Itanium™ architecture that we use in our methods.

3. Architectural Features

Itanium™ architecture ([4],[5]) provides many features to aid the compiler in enhancing and exploiting instruction level parallelism (ILP). These include an explicitly parallel (EPIC) instruction set, register rotation, software pipelined loop branches, predication, and data prefetching instructions. Such features were first seen in the Cydrome Cydra-5 ([2]) and HPL PlayDoh ([10]). All these features are used in the methods that enable efficient data prefetching.

Register rotation provides a hardware renaming mechanism that eliminates the need to unroll loops for the purpose of software renaming of registers. Renaming is accomplished by adding the register specifier in the instruction to the register rename base (RRB) to determine the physical register to be accessed. The RRB is decremented when a software-pipelined loop branch is executed. Decrementing the RRB makes the value in register X during iteration I, appear to move to register X+1 in iteration I+1. By taking modulo the size of the rotating register file, the highest numbered register wraps to the lowest numbered register after renaming, thus exhibiting rotation. Explicit register move/add instructions can then be used to create multiple smaller rotating regions. In the Itanium™ architecture, a subset of the register files (General registers r32-r127, floating-point registers f32-f127, and predicate registers p16-p63) can rotate. Registers r0-r31, f0-f31 and p0-p15 do not rotate and are referred to as **static** registers.

Below is an example of register rotation:

```
L1:  ld4 r32 = [r4],4 //load to r32
     add r34 = r34,r9 //use ld value
     st4 [r5] = r35,4 //store result
     br.ctop L1 ;;
```

The value that the load writes to rotating register r32 is read by the add, two iterations (and hence two rotations) later, as r34. In the meantime, two more instances of the load are executed. However, because of register rotation, those instances write to different physical registers and do not destroy the value needed by the add. Br.ctop is a software pipelined loop branch whose execution triggers the register rotation.

Predication refers to the conditional execution of an instruction based on a boolean source operand called the qualifying predicate. Almost all the Itanium™ architecture instructions have a qualifying predicate. If the qualifying predicate is *true* (one), the instruction is executed. If the qualifying predicate is *false* (zero), the instruction generally behaves like a no-op. Predication helps remove branches (by converting control dependences into data dependences) and the misprediction penalties associated with them, and also exposes more ILP [10].

Predicates are assigned values by compare, test-bit, or software-pipelined loop branch instructions. Compare instructions generally write two predicate registers with

complementary boolean values based on the evaluation of the compare condition.

Itanium™ architecture provides 64 predicate registers (p0-p63) of which 48 predicate registers (p16-p63) rotate. The rotation of predicate registers serves two purposes. The first, similar to the rotating general and floating-point registers, is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of a software pipeline.

Software-pipelined loop branches in the Itanium™ architecture enable efficient counted (br.ctop, br.cexit) and while (br.wtop, br.wexit) loops. They manage registers that maintain the loop count (LC) and epilog count (EC). The behavior of counted loop type branches is depicted in Figure 2.

Each time a software-pipelined counted loop branch is executed, the following actions take place:

- Based on LC and EC, a decision is made on whether or not to continue loop execution.
- Loop count and Epilog count (LC and EC) registers are selectively decremented.
- Predicate register (p63) is set to a value to control the initiation of new iterations.
- The registers (r32-r127, f32-f127, p16-p63) are rotated by decrementing RRB

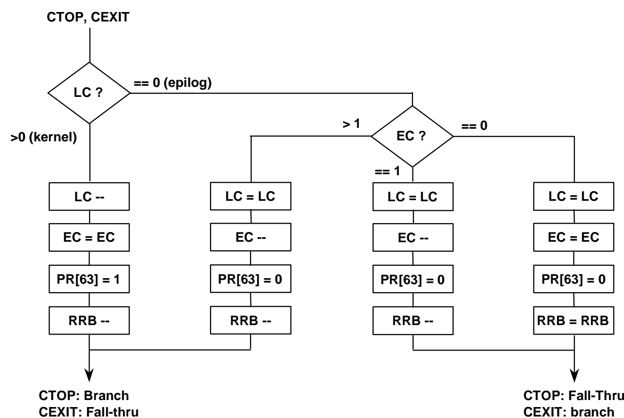

Figure 2. Behavior of counted loop branches in the Itanium™ architecture

During the prolog and kernel phases of a software-pipelined loop, a decision to continue kernel loop execution means that a new iteration is started. LC is decremented to update the count of remaining iterations. EC is not modified. P63 is set to one, registers are rotated (RRB--) and the branch (br.ctop) is taken so as to continue the kernel loop execution.

Once LC reaches zero, all required source iterations have been started and the epilog phase is entered. During this phase, a decision to continue kernel loop execution means that the software pipeline has not yet been fully drained. P63 is now set to zero because there are no more new source iterations to start and the instructions that correspond to non-existent source iterations must be disabled. EC is decremented to update the count of the remaining stages for the last source iteration. Registers are rotated and the branch is executed so as to continue the kernel loop execution. When EC reaches one, the pipeline has been fully drained, and the branch is executed so as to exit the kernel loop execution.

The particular feature of the software-pipelined loop branches employed in our methods is their ability to trigger the rotation of registers at the end of each iteration.

The next section describes the data prefetching methods that employ these architectural features to achieve optimized code with minimal overhead.

4. Optimized Data Prefetching

To effectively perform the task of data prefetching, an efficient scheme should possess the following attributes:

1. Issue only the required prefetches, i.e. one prefetch per cache line required.
2. Utilize minimal prefetch instruction slots, thus minimizing the overhead introduced by prefetching.
3. Avoid the need to unroll loops, thus minimizing increases in code size.
4. Require minimal code for prefetch control.
5. Distribute prefetches over time to not cause a surge in resource requirements for data accesses.

The two methods detailed below, utilize the architectural features described in section 3, to deliver prefetching schemes that possess these attributes.

4.1 Using Rotating General Registers

A prefetch instruction utilizes a general register to specify the address of the cache line to be prefetched. Since this address register can be renamed in hardware (using rotation of general registers), it can be mapped to different physical registers during different iterations of a loop's execution. Furthermore, since the registers rotate, a single address register can cycle through a set of physical registers, periodically. Each physical register in this set can be initialized to a different address so that a single prefetch instruction can be used to prefetch cache lines from different locations. A single prefetch instruction can thus accomplish the task of data prefetching from multiple locations (corresponding to different arrays in the program). Using this scheme, our running example code

(from Figure 1) would be modified into:

```
      r33 = address of E(1+X)
      r34 = address of D(1+X)
      r35 = address of C(1+X)
      r36 = address of B(1+X)
      r37 = address of A(1+X)
  Method1Loop:
      prefetch [r37]
      r32 = r37 + INCR
      . . .
      load A(I)
      load B(I)
      load C(I)
      load D(I)
      load E(I)
      . . .
      I = I + 1
      Br Method1Loop if more iters
```

Outside the loop, the registers (r33-r37) are initialized with the address locations of the various arrays (A,B,C,D,E) that need to be prefetched. Within the loop, a single prefetch instruction is used to issue the prefetch at the address indicated by the register r37 (initialized to A(1+X)). After the prefetch is issued, r37 is incremented by INCR and placed in r32 thus effectively creating a smaller rotating region of 5 integer registers (r33-r37).. When the branch is executed, the set of registers r32-r36 is renamed to r33-r37. In the next iteration, r33 contains the address of A(1+X) plus INCR, and r37 contains the address of B(1+X). Thus the next execution of the prefetch instruction will issue a prefetch to the address of B(1+X). Similarly, in three subsequent iterations the prefetch instruction will issue prefetches to the addresses of C(1+X), D(1+X), and E(1+X) in that order. The sixth execution of the prefetch instruction will issue a prefetch to the address of A(1+X) plus INCR. The value of X is chosen so as to cover the latency of access of the last array prefetched (in our case E). The value of INCR can be determined as follows:

```
      INCR = N * S
  where,
    S = the access stride in the array,
    N = the #arrays to be prefetched
```

This choice of INCR ensures that the prefetches are issued at a fixed distance from the datum being consumed. For the example (where S=8 bytes and N=5) INCR is 40-bytes. Hence over 40 iterations, 8 prefetches will be issued to 5 unique contiguous lines of the array A. Thus 3 out of 8 prefetches to the array A would have been redundant.

The use of a single prefetch for prefetching data for multiple arrays and the absence of the need for loop unrolling demonstrate that this scheme has minimal impact on code size. Using a single prefetch instruction automatically distributes the prefetches over time and eliminates the possibility of causing a surge in the requirement of the data access resources. Using the hardware register rotation mechanism triggered by the loop-closing branch ensures that prefetch control is accomplished with minimal overhead. The add of INCR is the only additional instruction. Thus, this scheme provides an effective method for data prefetching.

This scheme does cause some redundant prefetches to be issued, a shortfall that is addressed by the method described next. However, since the penalty for redundant prefetches is negligible on the Itanium™ processor, we have implemented this scheme in the Intel Production Compiler for the Itanium™ processor – the experimental results for which are described in Section 5.

4.2 Using Rotating Predicate Registers

While the first method utilizes rotating general registers to alter the address location prefetched by a single prefetch instruction as it gets executed over multiple iterations, the second method employs rotating predicate registers to selectively control the execution of different prefetch instructions.

The running example code (from Figure 1) using this second scheme would be modified into:

```
      p41 = true
      p42 = false
      p43 = false
      p44 = false
      p45 = false
      p46 = false
      p47 = false
      p48 = false
      r4 = address of A(1+X)
      r5 = address of B(1+X)
      r6 = address of C(1+X)
      r7 = address of D(1+X)
      r8 = address of E(1+X)
  Method2Loop:
  (p41)      prefetch [r4], 64
  (p42)      prefetch [r5], 64
  (p43)      prefetch [r6], 64
  (p44)      prefetch [r7], 64
  (p45)      prefetch [r8], 64
      p40 = p48 // Copy to rotate
      load A(I)
      load B(I)
      load C(I)
      load D(I)
      load E(I)
      . . .
      I = I + 1
      Br Method2Loop if more iters
```

As before, outside the loop, registers are initialized with the prefetch locations of the different arrays. The registers used, however, are static registers not rotating registers. Instead of using a single prefetch instruction for multiple arrays, this scheme requires a prefetch instruction for each array. These prefetch instructions, however, are

predicated using rotating predicate registers that are so initialized that one and only one predicate register is true during any iteration. This guarantees that at most one prefetch instruction will be "active" during any iteration. To achieve the rotation of 8 predicate registers, a predicate copy instruction (p40 = p48) is inserted into the loop. (In the Itanium™ architecture a single instruction can achieve this result).

Since this method is largely a dual of the first method, it realizes all the benefits of the first method such as : no loop unrolling, minimal prefetch control overhead, distributed prefetches over time, etc. However, it trades off the use of multiple prefetch instruction issue slots for reduced general purpose rotating register requirements. A distinct advantage of this method is that it can accommodate array accesses with different strides, since the prefetch address increment for an array is realized through the post-increment on its prefetch instruction.

It is also possible to devise blended methods that employ both rotating general registers and rotating predicate registers. As an example consider the code of our running example modified thus:

```
      p41 = false
      p42 = false
      p43 = false
      p44 = true
      p45 = true
      p46 = true
      p47 = true
      p48 = true
      r44 = address of E(1+X)
      r45 = address of D(1+X)
      r46 = address of C(1+X)
      r47 = address of B(1+X)
      r48 = address of A(1+X)
  Method12Loop:
   (p48)     prefetch [r48], 64
      r40 = r48  // Copy to rotate
      p40 = p48  // Copy to rotate
      . . .
      load A(I)
      load B(I)
      load C(I)
      load D(I)
      load E(I)
      . . .
      I = I + 1
      Br Method12Loop if more iters
```

Here, the first five executions of the single prefetch instruction are predicated ON and enable the prefetching of the addresses of A(1+X), B(1+X), C(1+X), D(1+X), and E(1+X). The three subsequent executions of the prefetch instruction are predicated OFF. And this pattern (of five ON and three OFF) repeats itself. The post-incrementing feature of the prefetch instructions enables preparing the address register for the next execution.

The broad applicability of this optimized prefetching scheme is detailed by the implementation discussion in the next section. The performance measurements presented in the next section lend credence to its usefulness.

5. Compiler Implementation and Results

We have implemented the rotating-register prefetching technique described in Section 4.1 in the Intel Production Compiler for the Itanium™ processor. In this section, we present an overview of the algorithm used and results comparing this technique with more traditional prefetching methods.

5.1 Implementation Overview

Prefetching is done as part of the high-level optimizations within our compiler. Figure 5 gives relevant details of the algorithm used to prefetch array accesses.

Step 1 identifies the set of candidate loops for inserting prefetch instructions. This selection uses the compiler estimate of the average trip count of the loop and heuristics such as whether a loop is an innermost loop. In Step 2, the locality analysis attempts to discover those instances of array accesses that refer to the same cache line ([9],[14]). Spatial locality refers to accesses to different locations within the same cache line. Since each prefetch instruction brings in an entire cache line, only one prefetch instruction is required per cache line. The prefetching scheme described in Section 4.1 exploits this spatial locality of array references to prefetch for multiple arrays in adjacent iterations. Group locality occurs when different array references access the same cache line. Only the leading reference needs to be prefetched in this case.

Step 3 computes the prefetch frequency for each memory reference that requires to be prefetched. Prefetch frequency for a memory reference is the number of iterations that can be executed before the next cache line of that reference has to be prefetched.

In the loop in Figure 3, the access stride for array A is 16 bytes and that for array B is 32 bytes assuming that these arrays are of type double precision (8 bytes). Using a cache line size of 64 bytes, the prefetch frequency for A and B are 4 and 2 respectively.

```
For (I=0; I<n; I++) A[2*I]=B[4*I]+2;
```

Figure 3. A simple example loop

Step 4 computes the prefetch distance that determines how many iterations ahead a prefetch is issued for an

array reference. This parameter is computed taking into account the memory latency and the resource, recurrence, and memory bandwidth constraints within the loop. Since the prefetch instructions are inserted as part of the high-level optimizations, it is not possible to accurately calculate the distance required, so the compiler estimates the distance based on the high-level information available. The distance computed should be enough to cover the memory latency so that the cache line is brought to the nearest level cache prior to its actual use. But at the same time, it should not be too large, since there is the possibility of it getting replaced from the cache before its use.

The actual prefetch instruction (named *lfetch* in the Itanium™ architecture) is inserted within the loop under a prefetch condition (if prefetch frequency > 1) in step 5 using the locality analysis information from step 2. For example, for the array reference A in Figure 3, a prefetch is issued as follows:

```
For (I=0; I<n; I++){
     A[2*I] = B[4*I] + 2;
     If (mod (I,4) == 0)
        lfetch (&A(2*(I+dist))
     . . .
   }
```

Figure 4. Prefetch for reference A in Figure 3

Here dist is the prefetch distance computed by the compiler. The effect of the *lfetch* instruction is to move the cache line containing the address to a higher level of the memory hierarchy. The address itself has no cache alignment requirement.

In Step 7, the grouping also takes into account the maximum number of conditional prefetches that can be prefetched using a single instruction. For example, if there are 10 uniform array references (of type double precision and unit stride access) to be prefetched, these will be divided into two clusters, where the first cluster contains 8 references and the second one contains 2 references. This is because with a cache line size of 64-bytes, the prefetch frequency will be 8 limiting the maximum number of prefetches in each cluster to 8.

```
Algorithm Prefetch_array_accesses
{
1.For each candidate loop
  {
2.  Perform spatial and group locality
    analysis for all memory references
    that appear within the loop
3.  For each memory reference that has to be
    prefetched, calculate prefetch frequency
    based on cache line size, data size and
    access stride
4.  Calculate the prefetch distance to be
    used for each memory reference based on
    memory latency and the resource,
    recurrence and bandwidth constraints
    within the loop
5.  Generate a conditional prefetch for each
    memory reference and insert within the
    loop
  } // end for each candidate loop
6.For each loop
  {
7.  Group the conditional prefetches into
    multiple clusters, each cluster having
    the same prefetch frequency
8.   For each cluster
     {
9.  Insert the address initialization
    statements in the preheader
10.Insert a rotational address  assignment
    stmt using the computed   value of INCR
11.Delete the conditional prefetches and
    insert a single  prefetch statement
12.Insert the assignment statements that
    rotate the  addresses
     } // end for each cluster
  } // end for each loop
```

Figure 5. Algorithm for prefetching array accesses

In step 7, we also make sure that each cluster contains only prefetches of the same frequency. This is required since we replace these multiple prefetches with a single *lfetch* instruction.

Steps 8-12 convert each cluster formed in Step 7 into rotating register mode as shown in the example given in Section 4.1. The address calculation part in step 10 uses the equation given in Section 4.1. In step 12, the assignment statements that achieve the rotation of address are also annotated appropriately. This annotation is later used by the scheduler to convert these explicit assignments to rotating registers.

5.2 Experimental Results

We used the 14 benchmarks in the CPU2000 (SPEC) FP suite for these experiments. This suite contains a collection of six Fortran77 programs, four Fortran90 programs, and four C applications. The measurements were performed on a 733 MHz prototype Itanium™ processor with 4MB L3 cache.

One of the metrics to gauge the "goodness" of the different prefetching schemes is to measure the increase in the initiation interval of a software-pipelined loop using each prefetching scheme. Initiation interval (II) of a software-pipelined loop is the delay in cycles between the initiation of adjacent iterations of the loop. In other words, II denotes the average time one iteration takes to complete. For each benchmark, we calculate the average II as the sum total of the IIs of all loops that get software-pipelined divided by the number of software-pipelined loops. Instead of comparing the average IIs using the

different methods, we compare the increase in average II of each scheme when compared to the base configuration. This comparison is shown in Table 1.

For the base case (column 2), we used compilation options that yielded highly optimized code without issuing any prefetches. These options include profiling feedback, inter-procedural analysis, loop transformations, and all traditional optimizations such as Partial Redundancy Elimination (PRE) ([3],[11]). The base case does not issue any prefetches. We compare three different prefetching methods (with all other options remaining exactly the same):

a) *Unconditional Prefetches*: Issue unconditional prefetches in every iteration without taking spatial locality into account (Column 3). This follows Steps 1-4 given in Figure 5 and Step 5 is replaced with the generation of an unconditional prefetch. Steps 6 through 12 are skipped.

b) *Conditional Prefetches*: Prefetch with conditions to make sure every cache line is prefetched only once (Column 4). This algorithm follows Steps 1-5 in Figure 5 and skips Steps 6 through 12. The conditions are if-converted into predicates by a later phase of the compiler.

c) Rotating Register Prefetches: Convert the conditional prefetches into rotating register-mode (Column 5). This follows the algorithm given in Figure 5.

Table 1. Increase in Average II (cycles) compared to Base

Benchmark	Base (No Prefetch)	Unconditional Prefetch	Conditional Prefetch	Rotating reg pfetch
Wupwise	0	+1	+1	0
Swim	0	+4	+6	+2
Mgrid	0	+2	+2	0
Applu	0	+1	+1	+1
Mesa	0	0	0	+1(*)
Galgel	0	+2	+2	+1
Art	0	+1	+2(*)	+2(*)
Equake	0	0	0	0
Facerec	0	+1	+2	0
Ammp	0	0	0	0
Lucas	0	+1	+1	+1
fma3d	0	+1	+1	+1
Sixtrack	0	+1	0	+1
Apsi	0	+1	+1	+1

Schemes **a** and **b** are chosen to represent traditional data-prefetching methods. Scheme **c** corresponds to the method described in Section 4.1. The algorithms ensure that the same references are prefetched using the three techniques and the same prefetch distance value is used for each reference. The prefetching scheme that inserts the required prefetches without conditionals through loop unrolling is not currently implemented in the compiler.

Each column in Table 1 lists the increase in average II (compared to Base – column 2) of the loops in each benchmark using a particular prefetching technique. Increase in II is a measure of the overheads associated with introducing prefetches, and this overhead is the least using rotating registers. The total number of dynamic instructions that get executed go down as we decrease the average II. The static code size is also smaller for the rotating register prefetching technique.

In some cases(*), the same set of loops may not be getting pipelined under different prefetching techniques. We can see that when compared with base, the average II increases with the introduction of prefetches. With no predication, the increase comes from the extra memory slots used for the prefetch instructions (and related address increments) themselves. For conditional prefetching technique, we have the additional compare instructions leading to a further increase in II.

The increase in II over the base configuration is the least when we switch to rotating register mode. The efficacy of the rotating register prefetch method is illustrated by the following loop extracted from the psinv function in the floating-point benchmark mgrid of the CPU2000 suite.

```
    DO 600 I3=2,N-1
    DO 600 I2=2,N-1
    DO 600 I1=2,N-1
600 U(I1,I2,I3)=U(I1,I2,I3)
    +C(0)*
     (R(I1,I2,I3))
    +C(1)*
      (R(I1-1,I2,I3)+R(I1+1,I2 ,I3)
      +R(I1,I2-1,I3)+R(I1,I2+1,I3)
      +R(I1,I2,I3-1)+R(I1,I2,I3+1))
    +C(2)*
      (R(I1-1,I2-1,I3)+R(I1+1,I2-1,I3)
      +R(I1-1,I2+1,I3)+R(I1+1,I2+1,I3)
      +R(I1,I2-1,I3-1)+R(I1,I2+1,I3-1)
      +R(I1,I2-1,I3+1)+R(I1,I2+1,I3+1)
      +R(I1-1,I2,I3-1)+R(I1-1,I2,I3+1)
      +R(I1+1,I2,I3-1)+R(I1+1,I2,I3+1))
    +C(3)*
      (R(I1-1,I2-1,I3-1)
      +R(I1+1,I2-1,I3-1)
      +R(I1-1,I2+1,I3-1)
      +R(I1+1,I2+1,I3-1)
      +R(I1-1,I2-1,I3+1)
      +R(I1+1,I2-1,I3+1)
      +R(I1-1,I2+1,I3+1)
      +R(I1+1,I2+1,I3+1))
```

There are 10 unique vectors that need to be prefetched in this loop: Nine specified by R(I1, {I2-1,I2,I2+1}, {I3-1,I3,I3+1}), and one specified by U(I1,I2,I3). (Accesses

to elements with the leading index I1+1 and I1-1 are contiguous and hence subsumed in these prefetches).

Using the optimized prefetching scheme **c**, the 10 unique prefetches were coalesced into 2 prefetches with rotating registers. The resulting software pipeline initiation interval for this loop on the Itanium™ processor was 43% lower than the II obtained using scheme **a**. Since this loop represents a significant portion of the total program execution time, attendant performance benefits accrue. Considering all loops in mgrid, we can see that the increase in average II is 0 using scheme **c** whereas we get an increase in average II of 2 using schemes **a** and **b**.

In Table 2, we list the percentages of the hardware run-time performance improvements on each benchmark for the three prefetching techniques. In general, we can see that performance gains from prefetching range between a degradation of 18% to an improvement of 161%. Note that these numbers reflect data on entire benchmarks, and not just the loops where prefetching occurred.

Table 2. Run-time performance improvements with different prefetching techniques (% improvements over Base)

Benchmark	Base (No Prefetch)	Unconditional Prefetch	Conditional Prefetch	Rotating reg pfetch
wupwise	0	27	21	26
swim	0	117	120	121
mgrid	0	83	58	118
applu	0	44	41	41
mesa	0	-3	-18	-2
galgel	0	-8	-13	7
art	0	4	4	4
equake	0	159	161	159
facerec	0	14	10	13
ammp	0	-4	-6	-4
lucas	0	66	66	67
Fma3d	0	28	28	28
sixtrack	0	-12	-10	-3
apsi	0	4	1	6
Geomean	0	29	25	34

The last row in Table 2 gives the percentage improvement in the geometric mean of all the 14 SPEC benchmark numbers. We can see that prefetching with no predication improves performance by 29% and using explicit conditionals reduces this to 25%. The rotating register mode prefetching gives the maximum benefit of 34% over base.

Note that most of the improvements from prefetching are obtained by issuing unconditional prefetches in every iteration as shown in column 3. This is because the Itanium™ processor discards redundant prefetch instructions to the same cache line. So the traditional technique of unrolling for exploiting spatial locality is not really required for this processor. Unrolling by large factors may in fact have more detrimental effects due to increasing code size and register pressure.

In general, the relative performance improvement is largest for the technique using rotating registers, which gives all the benefits of the other two techniques, but with a smaller overhead. This is particularly true for mgrid, galgel and apsi which have the most gains from rotating register prefetching strategy as shown in Table 2.

We can see a good correlation between these improvements and the percentage improvements in II shown in Table 1. For galgel, by reducing the overheads associated in prefetching, the rotating register prefetches actually improve performance by 7% whereas the other two techniques show 8% and 13% degradation while prefetching the same data elements.

In cases where prefetching itself does not improve performance as in sixtrack, using rotating registers gives a degradation of only 3%, where as the other two techniques degrade performance by 10% and 12% respectively.

In swim, though we get the least increase in average II using rotating registers (2 cycles versus increases of 4 cycles and 6 cycles using the other two methods), this does not translate into an actual improvement in performance. This is because this application is essentially bandwidth bound, and the dynamic II of the core loops is determined by the memory bandwidth and is always higher than the scheduled II. But as we mentioned earlier, converting prefetches to rotating register mode is beneficial even for these applications as they decrease code size and dynamic instruction counts.

6. Concluding Remarks

The ever-increasing disparity between processor and memory speeds continually add to the importance of latency hiding techniques such as software data prefetching. However, the overhead associated with traditional methods of software data prefetching reduce their effectiveness in terms of realized performance improvement. Utilizing register rotation and predication, the methods described here, significantly reduce that overhead *and* deliver a performance improvement due to optimized prefetch scheduling. Experimental results show that the rotating register prefetching scheme, delivers an overall performance improvement of 5-9% over the traditional prefetching methods on the set of 14 floating-point programs in the CPU2000 benchmark suite.

7. References

[1] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching", in Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems, 1991, 40-52.

[2] J.C. Dehnert, P.Y.Hsu, and J.P. Bratt, "Overlapped Loop Support in the Cydra 5,", in Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989, 26-38.

[3] Dulong, C., Krishnaiyer, R., Kulkarni, D., Lavery, D., Li, W., Ng, J., and Sehr, D., "An Overview of the Intel IA-64 Compiler," Intel Technical Journal, Q4 1999.

[4] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 Architecture,", IEEE Micro, Sept-Oct 2000, 12-23.

[5] Intel IA-64 Architecture Software Developer's Manual, Vol 1-4, 245319-001, January 2000.

[6] M.S. Lam, "Software Pipelining: An Effective Scheduling Technique for {VLIW} Machines," in Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, June 1988, 318-328.

[7] M.H.Lipasti, W.J.Schmidt, S.R.Kunkel, and R.R.Roediger, "SPAID:Software Prefetching in Pointer and Call Intensive Environments", In Proc 28th International Symposium on Micro-architecture, Nov 1995, 231-236.

[8] C.K. Luk and T.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, September 1996, 222-233.

[9] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, 62-73.

[10] V. Kathail, M.S. Schlansker, B.R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0", HP Labs Technical Report, HPL-93-80, March 1994.

[11] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, D. Sehr, "An Advanced Optimizer for the IA-64 Architecture,", IEEE Micro, Nov-Dec 2000.

[12] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in Proceedings of the 27th International Symposium on Microarchitecture, Dec 1994, 63-74.

[13] V. Santhanam, E. Gornish, and W. Hsu,, "Data Prefetching on the HP PA-8000," in Proceedings of the 24th Annual International Symposium on Computer Architecture, June 1997, 264 – 273.

[14] M. Wolfe and M.S. Lam, "A Data Locality Optimizing Algorithm," in Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, 30-44.