Multi-Stage Coordinated Prefetching for Present-day Processors

Sanyam Mehta  
Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN USA  
sanyam@cs.umn.edu

Zhenman Fang  
School of Computer Science  
Fudan University  
Shanghai, China  
fangzhenman@gmail.com

Antonia Zhai  
Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN USA  
zhai@cs.umn.edu

Pen-Chung Yew  
Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN USA  
yew@cs.umn.edu

ABSTRACT

Latest microarchitectures provide support for both hardware and software prefetching. However, the architectural features supporting either are different. In addition, these features can vary from one architecture to another. As a result, the choice of the right prefetching strategy is non-trivial for both the programmers and compiler-writers.

In this paper, we study prefetching techniques in the context of different architectural features that support prefetching on existing hardware platforms. These features include, the size of the line fill buffer or the Miss Status Handling Registers servicing prefetch requests at each level of cache, the effectiveness of the hardware prefetchers and their interaction with software prefetching, the nature of the instruction pipeline (in-order/out-of-order execution), etc. Our experiments with two widely different processors, a latest multi-core (SandyBridge) and a many-core (Xeon Phi), show that these architectural features have a significant bearing on the prefetching choice in a given source program, so much so that the best prefetching technique on SandyBridge performs worst on Xeon Phi and vice-versa. Based on our study of the interaction between the host architecture and prefetching, we find that coordinated multi-stage prefetching that brings data closer to the core in stages, yields best performance. On SandyBridge, the mid-level cache hardware prefetcher and L1 software prefetching coordinate to achieve this end, whereas on Xeon Phi, pure software prefetching proves adequate.

Categories and Subject Descriptors

D.3.4 [Processors]: Compilers, Optimization

Keywords

Coordinated prefetching; SandyBridge; XeonPhi

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1. INTRODUCTION

Data prefetching proves very effective for hiding the large memory latency. It is for this reason that latest processors are equipped with improved hardware prefetching logic, and their instruction sets provide support for software prefetch instructions that can selectively prefetch data to different levels of cache. However, while such an extensive support exists in the form of hardware, software, or coordinated hardware-software prefetching, the choice of the right prefetching strategy among the various options continues to remain a mystery for programmers and compiler writers.

The choice of the right prefetching strategy is a function of the various features supporting prefetching in the host architecture. One such feature is the nature of hardware prefetcher - its effectiveness, aggressiveness and behavior in the presence of software prefetch instructions. For example, on SandyBridge, the Mid-level cache (MLC) streamer hardware prefetcher is very effective and prefetches data to the L2 cache. Unlike software prefetching, it is not limited by the small size of the Line Fill Buffer at the L2 cache that allows very few outstanding prefetch requests, but can effectively prefetch data for 32 streams. Further, it can be trained by the software prefetch requests to the L2 cache for prefetching data to the L1 cache. This makes for an ideal scenario for coordination between the MLC streamer prefetcher and the L1 software prefetches that together fetch the data all the way to the L1 cache and make up for the not-so-effective L1 hardware prefetcher. The L1 software prefetching also provides other advantages: (1) Using a larger prefetch distance helps to increase the limit of look-ahead prefetch distance of the hardware prefetcher as the existing limit of 20 cache lines proves insufficient for small loops (as in matrix multiplication). (2) The hardware prefetcher stops at page boundaries. However, using a large prefetch distance in the software prefetch instructions helps to re-trigger the hardware prefetcher in time to prevent incurring any stalls at page boundaries. (3) The hardware prefetcher can track a maximum of 32 streams. Thus, for programs with more than 32 streams, software prefetch instructions prove particularly helpful to improve the prefetch coverage by prefetching data for the remaining streams directly to the L1 cache.

On the other hand, on Xeon Phi, there is similarly an effective L2 streamer prefetcher but it does not prefetch data in the presence of L1 software prefetch instructions. Thus, the advantages of coordinated hardware-software prefetching cannot be realized on
tested our algorithm using various memory-intensive benchmarks compiled using the Intel compiler as the back-end vendor compiler. We sent to achieve the desired coordination. We have implemented instructions and the prefetch distance at each level is carefully cho-
prefetch distance to use at each level. The choice of prefetch in-
identifying what to prefetch, (2) identifying where to insert the prefetch instructions, and (3) identifying when to prefetch, or what prefetch distance to use at each level. The choice of prefetch in-
s and the prefetch distance at each level is carefully chos-
so as to achieve the desired coordination. We have implemented our prefetching algorithm in the ROSE source-to-source compiler framework that transforms the original source code to include the prefetch instructions. The generated transformed code is then compiled using the Intel compiler as the back-end vendor compiler. We tested our algorithm using various memory-intensive benchmarks from the SPEC CPU2006 and OMP2012 suites on Intel Sandy-Bridge and Xeon Phi processors. Experimental results show that our multi-stage prefetching achieves a speedup of 1.55X over the Xeon Phi hardware prefetcher, and that of 1.3X over the state-of-
the-art Intel compiler for Xeon Phi. On SandyBridge that employs an effective hardware prefetcher, an improvement of 1.08X is ob-
tained.

The rest of the paper is organized as follows. Section 2 provides the background for this work and reinforces our motivation through an example. We describe our multi-stage prefetching algorithm with its three phases in Section 3. The interaction of our prefetching algorithm with the multithreading technique is also discussed in this section. Section 4 describes our compiler framework for implementing the prefetching algorithm. We present experimental results and discuss the results obtained using different prefetching strategies on individual benchmarks in Section 5. Related work is presented in Section 6 and we conclude in Section 7.

2. BACKGROUND AND MOTIVATION

In the literature, there have been many schemes proposed, both for hardware-based and software-directed prefetching. In hardware-based prefetching, some special hardware monitors data access pat-
ters to a particular cache and identifies data suitable for prefetch-
based on obtained information. Current processors employ multiple hardware prefetchers for streaming as well as strided ac-
cesses. Software-directed prefetching, on the other hand, involves the insertion of prefetch instructions into the original code by the programmer or the compiler that request data needed a few itera-
tions later. This distance in the number of loop iterations is called the prefetch distance. Like hardware prefetchers that sit on mul-
tiple levels of cache and can prefetch data to those levels, the lat-
est instruction sets provide prefetch intrinsics for prefetching data at different levels of cache. In software-directed prefetching, it is the responsibility of the programmer to ensure timeliness and pre-
vent redundant prefetches by deciding the data to prefetch and the prefetch distance. The hardware prefetchers usually ensure time-
liness through aggressive prefetching, i.e. maintaining a prefetch degree of more than one. For example, the streamer hardware prefetchers on SandyBridge and Xeon Phi have prefetch degrees of 2 and 4, respectively, and can maintain a prefetch distance of a maximum of 20 cache lines. Depending on implementation, soft-
ware prefetch instructions can also be used to train and thus control the prefetch distance at which the hardware prefetcher oper-
ates. Comparing software-directed and hardware-based prefetch-
ning, software-directed prefetching has the advantage of being used in a controlled manner, but is associated with an additional instruc-
tion overhead which may compete with the gains.

While the above-mentioned facts about prefetching are better known, the impact of other hardware features influencing prefetching are less well understood. The hardware tracks the outstanding prefetch requests through a buffer or a queue. This hardware structure is called the line fill buffer (LFB) in SandyBridge and MSHR (Miss Status Handling Registers) file in Xeon Phi, and is responsi-
ble for rendering the data prefetch requests non-blocking. The size of the LFB or the MSHR file, among other factors, has a significant bearing on the most appropriate choice of prefetching strategy on a particular architecture. This is because, on Xeon Phi, if the MSHR file is full, the pipeline stalls. On SandyBridge, if the LFB is full, subsequent prefetches/loads enter the load buffer, which when full, stalls the pipeline as well.

On all hardware platforms, the size of the LFB or MSHR file at the L1 cache is usually small. This is because any arriving data requests initiates a fully associative search across the structure to eliminate redundant requests, which is costly in terms of power us-
age and time delay, restricting its size. Chip area concerns also limit their size since these are located close to the core. It is for these reasons that both SandyBridge and Xeon Phi allow for only 8 outstanding prefetch requests at the L1 cache. However, clearly, such a small number of requests are insufficient to hide the large memory latencies on modern processors as pipeline stalls will result once the MSHR file or the load buffers are full. As a result, a multi-stage prefetching algorithm that brings the data from the memory in stages being cognizant of the resource availability at each level, is necessitated. This is more clearly illustrated through the following example of a well known memory-intensive weather prediction benchmark from SPEC OMP2012 suite, swim. The Xeon Phi processor is considered below, and similar arguments hold for a multicore processor as well.

```plaintext
for (j=0; j<M; j++) {
    for (i=0; i<N; i++) {
        S1: UNRVW[i][j] = UNVWL[i][j] + CV[i][j] + C1*CV[i][j+1] + C2*CV[i][j+2]
        S2: UNSND[i][j] = VNRLD[i][j] + CV[i][j] + CV[i+1][j] + CV[i+2][j] + CV[i+3][j]
        S3: UNRVW[i] = FOLD[i][j] - (C1*CV[i][j+1] + C2*CV[i+1][j] + C3*CV[i+2][j] + C4*CV[i+3][j])
    }
}
```

Figure 1: An example loop nest in the `swim` benchmark.

Figure 1 shows one of the three computationally (and memory) intensive loop nests in `swim`. The loop nest shown has 14 array references that access different cache lines, or in other words 14 data streams that need to be prefetched. When testing on Intel Xeon Phi that has a maximum memory latency of 1000 cycles, we observe that a minimum prefetch distance of 6 cache lines is needed to hide the memory latency. That is, there can be a maximum of $14 \times 6 = 84$ outstanding prefetch requests per thread for `swim`. Thus, for the data to be prefetched directly to the L1 cache, the L1 cache should provide 84 MSHRs per thread (in the ideal scenario when there is enough off-chip memory bandwidth available), or at least much more than the currently available 8 MSHRs per thread. Such a large MSHR file at L1 cache is not feasible. To make things worse, the 8 MSHRs at L1 cache are shared by the 4 SMT threads on each core on Xeon Phi.

Thus we implement a coordinated multi-stage data prefetching strategy, where data is first brought to the lower-level cache, e.g., L2 cache (that on Xeon Phi, has more MSHRs to allow holding more requests for hiding larger memory latencies) using a large prefetch distance (6 cache lines in case of `swim`). Subsequently, data is brought from the lower-level cache (e.g., L2 cache) to the higher-level cache (e.g., L1 cache) using a smaller prefetch distance (1 cache line in case of `swim`) since the data is already in the next-level cache. As a result, a small MSHR file at the L1 cache proves sufficient to hide the small L2-to-L1 latency, and prevent stalls due to contention. On SandyBridge, the same coordinated multi-stage prefetching strategy is implemented, but the coordination is between the L1 software prefetches and the L2 hardware prefetcher, which is more effective than its software counterpart due to its ability to hold many more outstanding prefetch requests.

### 3. Coordinated Multi-stage Data Prefetching

As stated in Section 1, our coordinated multi-stage data prefetching algorithm works in three phases, namely, **what** to prefetch, **where** to insert prefetch instructions, and **when** to prefetch. For each of these phases, we discuss our specific choices for the two hardware platforms considered and reasons behind those specific choices.

#### 3.1 What to Prefetch

This is the first of the three phases in which the references that should be prefetched are identified. Such references include (1) those whose future memory accesses can be determined, and (2) those when prefetched will benefit application performance.

A recent work [12] classified memory references into 5 types - (1) direct-indexed streaming array references, (2) direct-indexed striped array references, (3) indirect-indexed array references, (4) recursive data structures (RDS), and (5) hashing data structures. In our compiler framework, we only handle memory references of types (1) through (3), as references of types (4) and (5) need to be handled differently.

Further, even among references whose future access patterns can be statically determined, prefetching all of them is not always beneficial. For example, references that have temporal locality in the inner loops such as references $A$ and $C$ in `matmul` (shown in Figure 2) have small reuse distances and thus, the data referenced by them stays in the cache with a high probability. Such references occur often in SPEC benchmarks and lead to redundant prefetches that reduce performance gains, especially when the amount of computation in loop-nests is small (as in `matmul`). Although Mowry et al. in [13] proposed prefetch predicates in the form of IF statements (or its equivalent) to eliminate redundant prefetches, we empirically observe that the instruction overhead of such predicates usually offset the performance gain from data prefetching. These predicates also hinder automatic vectorization by the compiler. It is for these reasons that we do not consider such references as profitable candidates for prefetching. In addition, for references that have group reuse, such as $CV[i][j]$ and $CV[i][j+1]$ in statement S1 in `swim` (shown in Figure 1), we prefetch only for the leading reference $CV[i][j+1]$ as it is the one that accesses new data first and caches it for later use.

```plaintext
for (i = 0; i < N; i++)
for (k = 0; k < N; k++)
for (j = 0; j < N; j++)
    C[i][j] = A[i][k] * B[k][j];
```

Figure 2: The `matmul` kernel.

For all references that are thus marked for prefetching, our prefetching algorithm inserts prefetch instructions for just the L1 cache in case of SandyBridge since it relies on the hardware prefetcher for prefetching data to other levels. In case of Xeon Phi, however, the algorithm inserts prefetch instructions for all levels of cache to implement pure software-directed coordinated multi-stage prefetching.

#### 3.2 Where to Insert Prefetch Instructions

Having identified the array references to be prefetched, the next phase is to determine where prefetch instructions for the identified references should be inserted. For all identified references, we insert prefetch requests in the innermost loop. This simplifies the insertion of prefetch instructions and prefetch distance calculation by the compiler. However, even in the innermost loop, the placement of prefetch instructions is important. For example, in the loop nest from the `swim` benchmark shown in Figure 1, if prefetch requests for all 14 array references are placed contiguously in the source code without any intermittent computation, then it might lead to pipeline-stalls because prefetches will be blocked waiting for the availability of MSHRs. This is particularly true for L1 prefetch instructions given the small size of the L1 LFB or MSHR file. Such stalls are particularly visible on Xeon Phi where the pipeline is stalled immediately upon unavailability of MSHRs, whereas incoming requests could be buffered in the load buffer in case of dy-
naturally scheduled SandyBridge processor. Thus, our compiler framework inserts prefetch requests between individual statements requesting data needed by the array references in that statement - this introduces computation between a batch of prefetch requests providing adequate time for them to finish and free MSHRs. Further, prefetch requests for different levels of cache are intermingled, creating additional cycles for an L1 prefetch request to prefetch data from a lower-level cache.

3.3 When to Prefetch

This is the last phase of our prefetching algorithm that determines when to prefetch data for a memory reference. In particular, this phase determines for each array reference, the prefetch distance to be used when prefetching data to a particular level of cache. Prefetch distance calculation involves calculation of the loop iteration time, which is hard to precisely determine for processors with out-of-order execution capability. Thus, in this section, we treat the prefetch distance calculation for the in-order many-core Xeon Phi and the out-of-order multi-core SandyBridge separately.

3.3.1 Calculating Prefetch Distance in Xeon Phi

Mowry et al. in one of the earliest works on software prefetching [13] defined prefetch distance as $\frac{\text{Lat}_i}{\text{LIT}}$, where Lat and LIT are the prefetch Latency and the Loop Iteration Time, respectively. Of the two parameters, Lat is a machine specific parameter and can be known from vendor’s data sheets or measurements. LIT, on the other hand, must be estimated by the compiler. In our framework, we estimate LIT using Equation 1 below. This estimated value of LIT is then used to calculate the prefetch distances for prefetches to any level of cache, using measures of the corresponding prefetch latencies.

$$\text{LIT} = n\text{refs} \times \text{Lat}_{\text{L1}} + n\text{refs} \times C_p \times i + \sum_{\text{comp}=1}^{n} C_{\text{comp}} \quad (1)$$

where $n\text{refs}$ is the number of distinct array references in the loop nest, $\text{Lat}_{\text{L1}}$ is the latency of accessing a data item in the L1 cache, $C_p$ is the number of cycles spent in executing a prefetch instruction, and $C_{\text{comp}}$ is the cost of one of $n$ computations in the loop nest measured in the number of cycles.

The above formula for calculating the loop iteration time (LIT) assumes that all array references in the loop nest are accessed from the L1 cache (i.e. L1 cache hits). This is because we accomplish a multi-stage data prefetching where the data is prefetched all the way up to the L1 cache. This assumption holds for all but the initial few iterations of the loop nest, which can be considered as the warm-up phase. Also, since our framework inserts one prefetch instruction for each level of the cache memory per array reference, the instruction overhead in each loop iteration is calculated as $n\text{refs} \times C_p \times i$, where $i$ is the number of levels in the cache memory hierarchy. To facilitate further discussion in this section, we assume a 2-level cache memory hierarchy as in the Xeon Phi processor.

Once LIT is determined, the prefetch distance at each of the two levels is calculated as,

$$\text{PD}_{\text{L1}} = \left(\frac{\text{Lat}_{\text{L2}}}{\text{LIT}}\right) \ast \alpha \ast \beta \quad (2)$$

and

$$\text{PD}_{\text{L2}} = \left(\frac{\text{Lat}_{\text{memory}}}{\text{LIT}}\right) \ast \alpha \ast \beta \quad (3)$$

where $\alpha$ and $\beta$ are program dependent constants as explained below.

When calculating the prefetch distance, it is important to consider whether or not the innermost loop is vectorized. This is because, the prefetch distance computed using the formula, $\frac{\text{Lat}}{\text{LIT}}$, gives the prefetch distance in the number of loop iterations. However, if the innermost loop is vectorized, the same formula (using appropriate computation costs for vector operations) gives the prefetch distance in the number of vector iterations. We accommodate this in our algorithm by the parameter $\alpha$, where $\alpha$ is size of the vector (in the number of data elements) when the innermost loop is vectorized and 1, otherwise. The prefetch distance is further multiplied by the constant stride, $\beta$, for an array reference that exhibits stride access pattern. The prefetch distance is also rounded to the next higher multiple of the cache line size in cases when the innermost loop is not vectorized.

From Equations 2 and 3, we observe $\text{PD}_{\text{L2}}$ is much larger than $\text{PD}_{\text{L1}}$ (as $\text{Lat}_{\text{memory}} \gg \text{Lat}_{\text{L2}}$). As a result, data is prefetched first to the larger L2 cache using a larger prefetch distance, and this data is then prefetched to the L1 cache a few iterations ahead. Thus, the 2 stages of data prefetching coordinate to timely fetch the data to the L1 cache. As discussed earlier, using a smaller L1 prefetch distance facilitated by this strategy minimizes contention (and also prevents a possible cache pollution due to early prefetches).

3.3.2 Calculating Prefetch Distance in SandyBridge

Unlike the many-core Xeon Phi that has single-wide in-order issue and in-order execution, the multi-cores are usually superscalar processors with dynamic scheduling capability. As a result, it is non-trivial to statically determine the iteration time of loops on a multi-core processor. It is for this reason that Lee et al. [12] use the IPC values from benchmark profiling in their prefetch distance calculation. Their calculation gives a lower bound on the prefetch distance. The prefetch distance can, however, be increased without any negative effects until the newly prefetched data begins to replace the previously prefetched but unused data. A larger prefetch distance for L1 prefetches is, in fact, required as this helps to increase the prefetch distance for the hardware prefetcher. This, as explained earlier, helps to prevent stalls at page boundaries and in particular, helps small loops that need large prefetch distances. In our algorithm we prevent the replacement of useful data from the L1 cache by ensuring that the amount of unused prefetched data does not exceed a fourth of the L1 cache size. We restrict ourselves to a fourth of the L1 cache size as misses due to cache interference can set in much before the cache capacity is reached. The prefetch distance is thus calculated as

$$\text{PD}_{\text{L1}} = \left(\frac{\text{Size}_{\text{L1}}}{4 \times n\text{refs} \times \text{Size}_{\text{elem}}}\right) \quad (4)$$

where $\text{Size}_{\text{L1}}$ is the L1 cache size, and $\text{Size}_{\text{elem}}$ is the size of each data element. The L1 software prefetch instructions use this prefetch distance and in turn trigger the L2 hardware prefetcher which in time runs sufficiently ahead of the L1 prefetches to timely bring the data to the L2 cache. Thus, like Xeon Phi, coordinated multi-stage prefetching that prefetches the data all the way to the L1 cache with minimal contention in L1 LFB, is accomplished on SandyBridge.

3.3.3 Prefetch Distance for Tiled Code

While the formula for calculating the prefetch distances as derived above proves applicable to most programs, it nevertheless assumes long contiguous streaming/strided access patterns, i.e. array references where the entire array is referenced in all loop nests within an application. Such access patterns are not observed, however, in tiled codes and certain other scientific codes where the computational domain is split into parts, as in gemsfdd and mgrid benchmarks from the SPEC suite.

We use a simple example of tiled matmul to illustrate such a case and the prefetching strategy applied therein. Figure 3(b) shows
the tiled matmul code with prefetch instructions inserted assuming contiguous streaming access patterns - \( B[k][j] \) is identified as suitable for prefetching and prefetch instructions are inserted in the innermost loop as shown. However, as a result of tiling, the access pattern is no longer contiguous - only \( J \) elements of array \( B \) are accessed contiguously, followed by another \( J \) elements from the next row of \( B \). Thus, if a prefetch distance, \( PD_{L2} \) (that is likely to be greater than \( J \)), as calculated using Equation 3 is used, useless prefetches could result. Since the data does not arrive at L2 cache in time for L1 prefetches, 2-stage prefetching becomes ineffective. Our framework identifies such cases of accesses to non-contiguous data chunks (where the size of the innermost loop is a fraction of the array size) as tiled matmul, and inserts prefetches as shown in Figure 3(c). It prefetches one of the next array rows instead of prefetching elements that are a few cache lines (= prefetch distance) away. Thus, useless prefetches are reduced to only those that are prefetched while processing the last few rows of a tile of array \( B \). Since correct data does arrive in time in L2 cache, 2-stage prefetching again becomes very effective. In such cases, the prefetch distance is calculated in the number of array rows, instead of elements as in the general case above discussed. It is given as follows.

\[
PD_{L1}^* = \max \left( 1, \frac{PD_{L1}^0}{lc} \right) \quad (5)
\]

and

\[
PD_{L2}^* = \max \left( 2, \frac{PD_{L2}^0}{lc} \right) \quad (6)
\]

where \( PD^* \) is the prefetch distance in the special case, \( PD^0 \) is the prefetch distance in the general case (calculated in the number of elements) and \( lc \) is the loop count of the innermost loop, or tile size in case of tiled codes. In the formula, the lower bounds of 1 and 2 for L1 and L2 prefetch distance is to ensure sufficient time for the data to arrive in both L1 and L2 cache. In SandyBridge, our framework inserts prefetch requests for just the L1 cache using the above formula, and relies on the hardware to prefetch the data to L2 cache.

3.4 Multi-stage Prefetching in multi-threaded environment

Chip multiprocessing (CMP) and simultaneous multipthreading (SMT) can both be used to extract the inherent instruction-level parallelism in programs run in a multithreaded environment. Both these techniques interact closely with prefetching. In CMP, threads on each core simultaneously place demands for data from the memory, and pronounced bandwidth contention results. The net effect of this is an increase in the effective memory latency. As a result, pipeline-stalls due to unavailability of slots in the load/store buffer in multi-cores tend to be more prominent in such cases. Data prefetch requests that are additional requests to the regular loads, also occupy slots in the load buffer and thus end up contributing to these stalls. It is for this reason that we observe that for certain memory-intensive benchmarks such as swim, bwaves and others, the performance advantage due to software-directed prefetching to L1 over the hardware prefetcher, is lost. For other benchmarks that are less memory-intensive such as bt, cactus and matmul, the performance advantage persists. In a many-core processor such as Xeon Phi that has blocking loads, there is no load buffer and thus data prefetch requests and regular loads do not share common resources. As a result, prefetching does not cause a side-effect to parallel performance as in multi-cores as long as useful data is timely prefetched.

In SMT, multiple SMT threads on a core share the LFB or the MSHR file. As a result, the increased data access rate puts pressure on the shared MSHR file, resulting in potential stalls due to contention. Precisely, a contention results when the maximum number of outstanding prefetches are less than the available MSHRs. Since MSHRs are already scarce for a single thread, we observe that prefetching generally adversely affects performance of programs run in an SMT environment. In case of Xeon Phi, however, prospects of an interesting compiler optimization to reduce contention for MSHRs emerge for several reasons. Xeon Phi employs GDDR5 memory that has significantly higher latency and the latency increases when there are multiple active data streams or the number of distinct array references in a loop. It is for this reason that our framework detects contention for MSHRs and performs loop distribution prior to the insertion of prefetch instructions. This reduces the number of active streams in a loop by distributing statements in the original nest to multiple nests and thus benefits from reduced contention for MSHRs due to reduced memory latency. It is important to note that in cases where there is reuse among array references in original nest, loop distribution hurts temporal locality and thus reuse. However, since we distribute at the innermost loop, only L1 cache locality is hurt, which is more than compensated by effective data prefetching rendered possible by reduced contention. In multi-cores that employ a DDR3 memory, the latency does not increase significantly with the number of active streams, and thus loop distribution is not beneficial.

4. THE COMPILER FRAMEWORK

Figure 4 gives an overview of our compiler framework for coordinated multi-stage data prefetching. Our prefetching strategy is implemented in the open-source ROSE [14] compiler, which performs source-to-source transformations and provides many APIs for program analysis and transformations based on its Abstract Syntax Tree (AST). As shown in Figure 4, there are mainly five steps to insert multi-stage prefetches to the input source code after it is parsed to the AST. Steps 1 through 3 determine what to prefetch - all array references in the innermost loops are recognized and those with either temporal or group locality are discarded for prefetching. Step 4 is the key step to determine the prefetch distances or, when to prefetch, for multi-stage prefetching. As stated in Section 3,
prefetch distance for each cache level is calculated using Equations (2)-(6). This requires information such as the total number of array references as well as the number and type of computations involved in the loop nest, which are obtained by querying the AST. The last step is to insert those prefetches with the calculated prefetch distance into the AST. After our multi-stage prefetching transformation, ROSE unparses the AST to the transformed source code with prefetch instructions. Finally, a back-end vendor compiler is used to compile the transformed source code to the final executable.

![Compiler framework for coordinated multi-stage prefetching](image)

Figure 4: Compiler framework for coordinated multi-stage prefetching.

5. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present the results of our coordinated multi-stage data prefetching and compare it with other prefetching strategies in both SandyBridge and Xeon Phi processors. Results comparing multi-stage prefetching and other strategies in a multithreaded environment are also presented.

5.1 Experimental Environment

We ran our experiments on the two different hardware platforms discussed in the paper - the multi-core Intel SandyBridge and the recently released many-core Intel Xeon Phi. The micro-architectural details of the two processors are compared in Table 2. Also, Xeon Phi contains instructions for prefetching data as ‘exclusive’ into either of the two caches, whereas SandyBridge does not provide this functionality. On Xeon Phi, we use the ‘exclusive’ prefetches to prefetch writes. Since Xeon Phi has 2-level cache hierarchy and SandyBridge (with 3-level cache hierarchy) also allows to prefetch only to the L1 and L2 caches, the coordinated prefetching strategy is implemented in 2 stages on both platforms. After generating the transformed source code that contains the prefetch instructions, we use the Intel compiler (v13) [1] as the back-end compiler to generate the executable.

![Table 2: Details of the microarchitectures](image)

Table 2: Details of the microarchitectures

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Cache</th>
<th>No. of Cores</th>
<th>SMT</th>
<th>Hardware prefetcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>SandyBridge</td>
<td>Non-blocking</td>
<td>8</td>
<td>2-way</td>
<td>Streaming, Strided (L1+L2)</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>Blocking</td>
<td>60</td>
<td>4-way</td>
<td>Streaming (L2 only)</td>
</tr>
</tbody>
</table>

![Table 3: Summary of different prefetching strategies](image)

Table 3: Summary of different prefetching strategies.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware prefetching</td>
<td>The prefetching strategy used by Intel processors</td>
</tr>
<tr>
<td>L1 SW pref.</td>
<td>Data is prefetched only to L1 cache</td>
</tr>
<tr>
<td>L2 SW pref.</td>
<td>Data is prefetched only to L2 cache</td>
</tr>
<tr>
<td>2-stage SW pref.</td>
<td>Data is prefetched to both L1 and L2 cache using carefully chosen prefetch distances</td>
</tr>
<tr>
<td>2-stage HW-SW pref.</td>
<td>Data is prefetched to L1 cache assuming that hardware prefetcher brings the data to the L2 cache</td>
</tr>
<tr>
<td>icc SW pref.</td>
<td>The prefetching strategy used by the Intel compiler</td>
</tr>
<tr>
<td>Baseline pref.</td>
<td>The baseline configuration with all prefetching disabled</td>
</tr>
</tbody>
</table>

Figure 5 compares the performance results obtained on SandyBridge and Xeon Phi, respectively, for the different prefetching strategies summarized in Table 3. In addition to the inherent hardware-based prefetching on Intel processors and the icc-directed prefetching, we implemented four other prefetching strategies for the purpose of comparison. As discussed in the paper, we propose a 2-stage coordinated software prefetching for Xeon Phi and 2-stage coordinated hardware-software prefetching for SandyBridge. Of the seven different strategies listed in Table 3, we could not implement the baseline prefetching on Xeon Phi because the BIOS did not provide the facility to turn off the hardware prefetcher as in SandyBridge. Thus, the hardware-based prefetching serves as the baseline in case of Xeon Phi. On the other hand, the Intel compiler for multi-cores is largely passive in the matter of prefetching and relies primarily on the hardware prefetcher. Thus, in our results for SandyBridge, we do not show the performance results for icc-directed prefetching which performs similar to the hardware-based prefetching for all benchmarks. In our analysis, we explain the behavior of the prefetching strategy adopted by the Intel compiler by studying the assembly code generated for different benchmarks.

Figure 5 compares the performance results obtained on SandyBridge and Xeon Phi, respectively, for the different prefetching strategies summarized in Table 3. In addition to the inherent hardware-based prefetching on Intel processors and the icc-directed prefetching, we implemented four other prefetching strategies for the purpose of comparison. As discussed in the paper, we propose a 2-stage coordinated software prefetching for Xeon Phi and 2-stage coordinated hardware-software prefetching for SandyBridge. Of the seven different strategies listed in Table 3, we could not implement the baseline prefetching on Xeon Phi because the BIOS did not provide the facility to turn off the hardware prefetcher as in SandyBridge. Thus, the hardware-based prefetching serves as the baseline in case of Xeon Phi. On the other hand, the Intel compiler for multi-cores is largely passive in the matter of prefetching and relies primarily on the hardware prefetcher. Thus, in our results for SandyBridge, we do not show the performance results for icc-directed prefetching which performs similar to the hardware-based prefetching for all benchmarks. In our analysis, we explain the behavior of the prefetching strategy adopted by the Intel compiler by studying the assembly code generated for different benchmarks.

According to the data access pattern, we divide the benchmarks into four categories: 1) programs with streaming or strided accesses, 2) programs with indirect array indexes, 3) programs with array of structures (and pointers), and 4) programs with loop sizes that are fractions of the problem size.

1. Programs with streaming or strided accesses. Among the benchmarks used for our experiments, cactus, hmmer, bwaves, swim, and matmul involve streaming array accesses whereas bt involves streaming as well as strided accesses. Results show that on
both hardware platforms, 2-stage coordinated prefetching as proposed for the respective platform, either outperforms or performs close to the best performing prefetching strategy for all benchmarks. It is interesting to note that 2-stage coordinated hw-sw prefetching which is the best performing strategy on SandyBridge, is the worst performing on Xeon Phi owing to the difference in interaction between hardware and software prefetching on the two platforms. On Xeon Phi, the software prefetches cannot train the hardware prefetcher and both cannot co-exist. On SandyBridge, on the other hand, we employ software-based prefetching to train the hardware prefetcher (because such an opportunity is provided by hardware) to overcome its limitations, and thus outperform other strategies. It is similarly interesting that the 2-stage sw prefetching which is the best performing strategy on Xeon Phi performs poorly on SandyBridge. This is because of a very small LFB at the L2 cache in SandyBridge. The L2 sw prefetching performs poorly on SandyBridge for the same reason. The L1 sw prefetching performs worse than the L2 sw prefetching on Xeon Phi due to fewer MSHRs at L1 cache (and thus more contention), whereas it performs slightly better than the L2 sw prefetching on SandyBridge due to prefetching the data to the L1 cache (that has same-sized LFB as L2 cache). The performance results for each benchmark are discussed in more detail below.

swim, bwaves and matmul. These benchmarks involve streaming access patterns with long loops. Although matmul has temporal locality, it behaves like other streaming benchmarks when it is not tiled since the data set size is much larger than L2 cache size. Here, multi-stage prefetching performs considerably better than the other prefetching strategies. It wins over the hardware and L2 sw prefetching for hiding the L2-to-L1 latency by prefetching the data to the L1 cache. The improvement is significant since the L1 misses are significant given the streaming nature and large working sets in these benchmarks. On SandyBridge, the 2-stage coordinated hw-sw prefetching improves considerably over the hardware-based prefetching in case of matmul even though the L1 hardware prefetcher proves sufficient for the small number of streams in matmul. This is because L1 software prefetches with large prefetch distances help in increasing the prefetch distance of the hardware prefetcher that is essential for the small loops in matmul. The L1 hardware prefetcher on SandyBridge prefetches only the next cache line, and thus cannot sufficiently increase the prefetch distance of the L2 hardware prefetcher. It is for this reason that we observe (using the Vtune performance monitoring tool [2]) that a significantly more number of loads that miss the L1 cache hit the LFB, in case of matmul that uses the hardware prefetcher against the one which uses our 2-stage coordinated software prefetching. The higher number for the former suggests pending prefetch requests at the L2 cache because of insufficient prefetch distance. On Xeon Phi, the Intel compiler performs worse than even the hardware prefetcher in matmul, because of inserting redundant prefetch instructions for the array reference (reference C[i][j] in Figure 2) that has temporal locality in an inner loop. In swim, the Intel compiler performs poorly as it does not prefetch data for all references. In addition, the data is prefetched only to the L2 cache and not to the L1 cache. In bwaves, the Intel compiler performs worse than coordinated prefetching because of issuing redundant prefetches for references that have temporal locality in the innermost loop, while leaving out references that have spatial locality in the innermost loop. We believe that it makes a wrong decision because of performing loop interchange optimization. It also does not prefetch data to the L1 cache in this case.

cactus. In cactus, bulk of the computation and memory references happen in a very large loop nest. Each iteration of the loop nest provides sufficient cycles to hide the memory latency. Since the loop nest involves significant computation, even requests to prefetch the data directly to the L1 cache are finished without much stalls. As a result, all software prefetching strategies perform similarly. The multi-stage prefetching achieves slightly better performance as compared to the L1 or L2 prefetching because of achieving better performance in other smaller loop nests that also involve streaming accesses. On Xeon Phi, the Intel compiler performs slightly worse because of inserting all prefetch instructions at the end of the loop instead of interleaving them with the computation. This leads to stalls due to contention at L1 cache that hurts performance. An important observation here is that the baseline hardware prefetcher on Xeon Phi performs significantly worse than the other strategies. This is because, the large loop nest contains streams (81 data streams) that are much larger than that can be handled by the hardware prefetcher (16, in case of Intel Xeon Phi). The performance difference is much less on SandyBridge for 2 reasons, (1) the hardware prefetcher can prefetch data for 32 streams instead of 16, and (2) the out-of-order execution tolerates most of the stalls since there is significant computation interspersing memory requests in the large loop nest in cactus.

hmmer. In hmmer, a subroutine called P7Viterbi is most computationally intensive and contains small loops. All the data referenced in the subroutine fits the L2 cache. Thus, although 2-stage prefetching that brings the data to the L1 cache wins over other strategies, the performance difference is small. On Xeon Phi, the Intel compiler also prefetches to both the L1 and L2 cache, but
uses a much larger prefetch distance which proves costly, given the small loops.

bt. In bt, four subroutines, compute rhs, x_solve, y_solve and z_solve contribute almost entirely to the execution time. These subroutines contain both streaming as well as strided accesses, and the working set is large. Thus, although a likely candidate for significant performance gains from 2-stage prefetching as in swim, the performance gains are small. This is because, the loop nests in bt, particularly some of the time-consuming nests in x_solve, y_solve and z_solve, are dominated by computation than by memory references. The performance improvement achieved by coordinated prefetching on SandyBridge is smaller than that achieved on Xeon Phi because higher computation in loop nests allows for tolerance of stalls due to out-of-order execution. Most of the performance improvement achieved by 2-stage prefetching stems from prefetching for strided accesses. These strided accesses are missed by even the strided hardware prefetcher in SandyBridge (that can detect strides up to 2K bytes) since the strides are long, given loops with large trip count. On Xeon Phi, the Intel compiler does not yield much improvement over the baseline because of not adequately prefetching for strided accesses.

void quantum_cnotq ( quantum_reg *reg, ... )
...
  for (i = 0; i < reg->size; i++)
    prefetch([reg=+node][i]+Pd[i], L1, F); // exclusive
  if (reg=+node)[i], L2, F];
    C[i] += values[i] + [C[reg=+node][j]];
...
Figure 6: An example loop nest with prefetching instructions to L1 and L2 cache in the libquantum benchmark.

2. Programs with array of structures (and pointers). In libquantum, the computationally intensive loop-nests contain an array of structures, that is responsible for bulk of the memory accesses. One such loop-nest is shown in Figure 6, and the memory reference reg -> node is an array of structures, that has state as one of its fields. In such cases, our multi-stage prefetching algorithm determines the size of the structure and prefetches the entire structure on the assumption that majority of the fields of the structure will be referenced - this may lead to prefetching more than a single cache line. However, in libquantum, the structure has only 2 fields and a size of 16 bytes, so we prefetch just one cache line. Our multi-stage prefetching wins over the hardware-based prefetching, as the hardware prefetcher cannot run sufficiently ahead of the program counter for timely prefetching, given the small computation in the loop-nests and larger size of the data structure. It is for this reason that on SandyBridge, coordinated hw-sw prefetching where L1 software prefetches train and thus help in increasing the prefetch distance of the hardware prefetcher outperforms all other strategies. The performance improvement of coordinated sw prefetching over the baseline hardware prefetcher on Xeon Phi is more significant than that on SandyBridge because of no hardware prefetcher at L1 in Xeon Phi to prefetch to the L1 cache. On Xeon Phi, the Intel compiler does not prefetch array of structures and thus performs as well as the hardware prefetcher.

3. Programs with indirect array indexes. An example program with indirect array indexes is sparse matrix-vector multiplication as shown in Figure 7, where the array reference B[colIdx[j]] is indirectly indexed. In such cases, we prefetch data for the directly indexed reference, colIdx, that is used to reference the indirectly indexed reference B. We also prefetch the array reference values, but not reference C that has temporal locality in the innermost loop j. On Xeon Phi, we get considerable performance improvement over the baseline hardware prefetcher due to prefetching to L1 cache. On SandyBridge, however, the hardware prefetcher performs slightly better than the 2-stage coordinated hw-sw prefetching as even the hardware prefetcher at L1 cache is successful in bringing the data to the L1 cache given few memory references in the loop nest. The slight better performance of the hardware prefetcher is due to no instruction overhead of prefetching. It is important to note that the loop iteration time in spmv is not as small as in matmul due to an indirectly indexed array reference that hurts efficient vectorization, and thus the benefit from increasing the prefetch distance of the hardware prefetcher through software prefetches is not significant. On Xeon Phi, the Intel compiler performs similar to the coordinated sw prefetching since it also prefetches to both the L1 and L2 cache using different distances. Although it uses a much larger distance that needed for the L1 prefetches, the performance is not hurt due to very long data streams.

for (i=0; i<xM[i]; i++)
  int start = row_start[i], stop = row_start[i+1];
  if (start; i; stop; i++)
    prefetch(&values[i]+Pd[i], L1, F);
    prefetch(&values[i]+Pd[i], L2, F);
    prefetch(&colIdx[i]+Pd[i], L1, F);
    prefetch(&colIdx[i]+Pd[i], L2, F);
C[i] += values[i] + [B[reg=+node][j]];
Figure 7: Sparse matrix-vector multiplication with prefetching instructions to L1 and L2 cache.

4. Programs with loop sizes that are fractions of the problem size. This category of programs include both the tiled codes and various scientific codes that compute in parts such as the gemsfd and mgrid benchmarks from SPEC suite. In such codes, our multi-stage prefetching algorithm prefetches data in the following array rows instead of prefetching a few cache lines ahead as shown earlier in Figure 3(c). This helps to not only avoid prefetching useless data but also allows to timely prefetch useful data. On Xeon Phi, coordinated software prefetching wins over the hardware prefetcher and other software prefetching strategies primarily due to prefetching the data to L1 cache and also eliminating redundant prefetches. On SandyBridge, hardware-based prefetching and coordinated hw-sw prefetching win over others due to employing an efficient hardware prefetcher (with no resource contention) to prefetch data to the L2 cache. Other details regarding performance results for the 3 benchmarks in this category are discussed as follows.

gemsfd. On SandyBridge, the coordinated hardware-software prefetching performs slightly worse than the hardware prefetcher because although this benchmark computes in 2 parts, most of the execution belongs to one of those 2 parts. As a result, aggressive prefetching by hardware enables timely prefetching for the 2 parts although at the cost of some useless prefetches. The overhead due to both the useless prefetches and not prefetching the data to the L1 cache are tolerated by the out-of-order execution since the loops nests are not highly memory-intensive and involve considerable computation. On Xeon Phi, the Intel compiler does not achieve improvement over the hardware prefetcher due to insertion of useless prefetches by prefetching using large prefetch distances.

mgrid. On SandyBridge, the improvement achieved by coordinated hw-sw prefetching over the hardware prefetcher is small because this benchmark is highly memory intensive, and thus we observe significant pipeline stalls due to filled up load buffer, when prefetch requests are inserted. Using Vtune, we observe that the stalls due to filled up load buffer in a code with L1 software prefetch instructions increase by a factor of 3 over a code that contains no prefetch instructions. On Xeon Phi, however, there are no addi-
5.3 Results for Multi-stage Prefetching in multi-threaded environment

Figure 8 shows the performance results of our coordinated prefetching strategies for SandyBridge and Xeon Phi on 8 and 32 cores, respectively, in CMP environment. On SandyBridge, as discussed in Section 3.4, the performance improvements achieved by coordinated hw-sw prefetching using single thread diminish for benchmarks that are highly memory-intensive due to increased stalls from resource contention. The benchmarks in this list are swim, gemsfldt, bwaves and mgrid. Other less memory intensive benchmarks such as bt, swim, and mgrid continue to show improvement. Matmul, although memory intensive, has little significant improvement owing to the increase in hardware prefetch distance achieved through software prefetching. On Xeon Phi, the benchmarks behave similar in both the CMP and single thread environment since prefetching does not cause additional stalls in the CMP environment as in SandyBridge.

As discussed in Section 3.4, memory intensive benchmarks with many references in the loop nest benefit from loop distribution optimization on Xeon Phi. Figure 9 shows the performance gains of loop distribution (followed by multi-stage prefetching) in 3 memory-intensive parallel benchmarks, swim, gemsfldt, and cactus. Using the recommended 2-way SMT on Xeon Phi, loop distribution achieves another 13% average speedup over multi-stage prefetching on the 3 benchmarks.

6. RELATED WORK

In the past, there has been significant research on tuning the prefetch distance and reducing overhead due to software prefetching [3, 4, 9, 13]. Mowry et al. [13] defined prefetch distance in terms of the prefetch latency and loop iteration time. Mowry et al. also proposed prefetch predicates to eliminate redundant prefetches. Badawy et al. in [3] proposed to insert prefetch instructions in the epilogue and prologue. Recently, Lee et al. [12] have proposed to calculate the loop iteration time using the average IPC of the application obtained through program profiling. Such an adjustment to prefetch distance is necessary for modern multi-cores that execute out-of-order. In our work, however, since we only use software to prefetch to the L1 cache in multi-cores, we show that using a large prefetch distance such that it does not cause evictions in the L1 cache is a better choice for several reasons. This has the advantage of being loop-based and not program-based, and is calculated statically.

In recent past, coordinated prefetching was proposed and implemented in IBM’s Power6 microarchitecture [11]. On Power6 (and its follow-ons), the L2 hardware prefetcher that can hold 32 outstanding prefetch requests, prefetches data from memory using a larger distance (at most 24 lines) and the L1 hardware prefetcher that can hold 8 prefetch requests, prefetches data from L1 using a smaller distance (at most 2 lines). In this work, we use a similar idea to achieve coordination, but through software prefetching. On SandyBridge, software prefetching at L1 coordinates with the
L2 hardware prefetcher that allows us to significantly increase the prefetch distance (without polluting the cache - Section 3.3.2) and overcome the limitations of the hardware prefetcher as mentioned in Section 1. On Xeon Phi, the coordination is achieved entirely through software prefetching. Lee et al. [12] have considered coordination between hardware and software prefetching on existing multi-cores. They manually insert software prefetch instructions and identify benchmarks for their positive, neutral or negative interaction with the existing hardware prefetchers (primarily through simulation). Their work is thus focused on studying the interaction between hardware and software prefetching, but does not propose any particular prefetching strategy. In their tests, they use cooperative hardware-software prefetching where both software and hardware prefetching co-exist; ours is a coordinated hardware-software prefetching technique, where the software prefetches actively train the hardware prefetcher to achieve coordination.

In the past, Santhanam et al. [16] and Caragea et al. [5] have also proposed reduction of prefetch distance due to limited resources in the form of number of outstanding prefetch requests that can be handled by the hardware. However, these works consider HP PA-8000 and a many-core research machine, XMT, respectively, both of which employ single-level cache. Thus, unlike our work, they do not employ the facility to prefetch to multiple levels of cache as in existing architectures, to tackle the problem of resource contention. A recent work from Intel [10] talks about prefetching the data to both the L1 and L2 cache - a technique implemented in the existing version of the Intel compiler for Xeon Phi. However, they do not describe any strategy to prefetch the data to the L1 and L2 cache such that a coordination is achieved. We observe in our results that the Intel compiler for Xeon Phi does not always insert prefetch requests for L1 and L2 cache, and generally uses a larger prefetch distance at L1 than needed. As a result, it cannot match the performance of our coordinated prefetching on Xeon Phi.

Among other works in software-based prefetching are those that employ helper threads to predict future load addresses [7, 8, 17]. The authors in [7, 8] use an idle thread as the helper thread to prefetch data for another compute thread, whereas Son et al. [17] extend the helper-thread prefetching to work with multiple cores by assigning a customized helper thread to a group of compute threads. These works, however, have not evaluated the impact of resource contention due to aggressive prefetching by the helper-thread. In our experiments, we find the impact of resource contention to be important, particularly on many-cores that have blocking caches, and thus implement a simple and low-overhead software prefetching strategy that is different from helper-thread based prefetching.

7. CONCLUSION

In this work, we study software prefetching in light of various architectural features that support prefetching in existing processors. Based on our study of those features, we implement a coordinated multi-stage prefetching strategy for two widely different state-of-the-art processors, the multi-core SandyBridge and the many-core Xeon Phi. However, the means of achieving this coordination is different on either - in SandyBridge, it is achieved through the MLC hardware prefetcher prefetching data to the L2 cache and L1 software prefetches further bringing it to L1 cache; in Xeon Phi, the coordination is achieved through carefully tuned software prefetching at different levels of cache. Results establish the efficacy of these strategies on respective platforms and also the importance of an awareness of the influence of different architectural features on prefetching as studied in this work. The performance results achieved using the 7 different prefetching strategies are discussed for each benchmark considered. Further, the performance of these strategies in multithreaded environment is also presented and discussed. Since our multi-stage coordinated prefetching is a simple, static prefetching technique, and does not require any special hardware, it can be readily incorporated in production-quality compilers for existing architectures.

8. ACKNOWLEDGEMENTS

We would like to acknowledge NSF grants CNS-0834599 and CCF-0708822 for supporting this work.

9. REFERENCES