Compiler-based Data Prefetching and Streaming Non-temporal Store Generation for the Intel® Xeon Phi™ Coprocessor

Rakesh Krishnaiyer†, Emre Kültürsay††, Pankaj Chawla†, Serguei Preis†, Anatoly Zvezdin†, and Hideki Saito†
† Intel Corporation
† The Pennsylvania State University

Abstract—The Intel® Xeon Phi™ coprocessor has software prefetching instructions to hide memory latencies and special store instructions to save bandwidth on streaming non-temporal store operations. In this work, we provide details on compiler-based generation of these instructions and evaluate their impact on the performance of the Intel® Xeon Phi™ coprocessor using a wide range of parallel applications with different characteristics. Our results show that the Intel® Composer XE 2013 compiler can make effective use of these mechanisms to achieve significant performance improvements.

I. INTRODUCTION

The Intel® Xeon Phi™ coprocessor based on Intel® Many Integrated Core Architecture (Intel® MIC Architecture) is a many-core processor with long vector (SIMD) units targeted for highly parallel workloads in the High Performance Computing (HPC) segment. It is designed to provide immense throughput: a single chip can deliver a peak performance of well above one double precision TeraFLOPS. However, achieving performance close to this peak value requires the cores and the vector units in the coprocessor to be kept busy at all times, which in turn requires data to be delivered at a very high rate without stalls. Providing sustained low memory latency is more challenging with the Intel® MIC architecture than the Intel® Xeon® processors. This is mainly because (i) as opposed to traditional large, power-hungry, out-of-order cores which are known to be good at hiding the latency of cache/memory accesses by exploiting instruction level parallelism, the coprocessor uses small, power efficient, in-order cores, (ii) the last-level cache capacity per core (and per thread) is smaller on the Intel® Xeon Phi™ coprocessor than the latest Intel® Xeon® processors, and (iii) the Intel® Xeon Phi™ coprocessor has a higher memory access latency.

The Intel® Xeon Phi™ coprocessor relies on software and hardware data prefetching techniques to bring data into local caches ahead of need to hide the latency of accessing caches and memory. It also employs special vector instructions that can be used for streaming non-temporal stores in order to save memory bandwidth. While those prefetch and streaming non-temporal store instructions can be inserted by expert programmers using intrinsics, in practice, it is a hard and error-prone task to identify when these instructions will be useful and with which parameters they should be used to bring benefits. It is much more practical to reduce the role of the programmer and delegate the compiler to be responsible for performing the bulk of the work: analyzing the source code, identifying the cases where these instructions can be used, and generating them. In this paper, we:

• Present how the Intel® Xeon Phi™ coprocessor software prefetch and non-temporal streaming store instructions are generated by the Intel® Composer XE 2013 compiler.
• Evaluate the impact of these mechanisms on the overall performance of the coprocessor using a variety of parallel applications with different characteristics.

Our experimental results demonstrate that (i) a large number of applications benefit significantly from software prefetching instructions (on top of hardware prefetching) that are generated automatically by the compiler for the Intel® Xeon Phi™ coprocessor, (ii) some benchmarks can further improve when compiler options that control prefetching behavior are used (e.g., to enable indirect prefetching), and (iii) many applications benefit from compiler generated streaming non-temporal store instructions.

The remainder of this paper is organized as follows. In Section 2, we provide the details of the Intel® Xeon Phi™ coprocessor based on the Intel® MIC architecture. In Sections 3 and 4, we present the software/hardware data prefetching mechanisms and streaming non-temporal store instructions in this architecture together with how they are generated automatically by our compiler to improve application performance. Section 5 gives the details of our experimental framework and presents an in-depth analysis of the impact of data prefetching and streaming non-temporal stores on our target benchmarks. Section 6 discusses related work and we conclude the paper with Section 7.

II. BACKGROUND

The Intel® Xeon Phi™ coprocessor is designed to serve the needs of applications in the HPC segment that are highly parallel, make extensive use of vector operations, and are occasionally memory bandwidth bound. We start this section with a presentation of the salient architectural characteristics of this coprocessor, contrasting it with the Intel® Xeon® processor E5-2600 series product family. We then approach this new architecture from a programmer’s point of view. Before moving on to the details of compiler-based data prefetching and streaming store instructions in the next sections, we also give an overview of our compiler.

A. Architectural Overview of the Intel® Xeon Phi™ Coprocessor

1) Cores and Parallelism: Targeting highly parallel applications, the Intel® Xeon Phi™ coprocessor [1] employs 60 (or more) cores, each core supporting four hardware
thread contexts which gives a total of 240 (or more) threads that can run simultaneously. The cores are in-order and can issue two instructions per cycle. This dual instruction issue is coupled with two pipelines (u and v-pipes), which improves the utilization of the units inside the core. The two pipes are not symmetrical and there are restrictions on which instructions can be scheduled on which pipe (e.g., vector operations can only be scheduled on the u-pipe). At each cycle, the instruction decoder picks instructions from the threads scheduled on the core in a round-robin order with the restriction that at least one cycle is needed between decoding instructions from the same thread. Hence, if only one thread is scheduled on a core, its instructions can be issued at every other cycle. This restriction was a design choice to balance the latency of the pipeline stages in the processor and prevent the decode stage from reducing the clock frequency.

Overall, in contrast to the Intel® Xeon® processors, the Intel® Xeon Phi™ coprocessor trades off core complexity for coarse grain parallelism. While the former family employs up to eight complex out-of-order, four-issue cores, the latter makes use of a larger number of simpler in-order cores running at a lower frequency. These cores are instances of the Intel® P54C architecture that is extended based on the needs of target applications (e.g., 64-bit addressing to satisfy the need for using large memory, long vector units to improve performance of data-parallel workloads). The reduction in the complexity of the cores enables significant reduction in the core silicon area and power consumption, which in turn enables the integration of a larger number of cores on chip. Despite its sacrifice from single-thread performance and instruction level parallelism, Intel® Xeon Phi™ products can realize significantly higher thread level parallelism.

2) Vector (SIMD) Operations: To improve the performance of applications that make extensive use of vector operations, the Intel® Xeon Phi™ coprocessor employs 512-bit vector units. Using a 512-bit vector unit, 16 single precision (or 8 double precision) floating point (FP) operations can be performed as a single vector operation; and, with the help of the fused multiply-add (FMA) instruction, up to 32 FP operations can be performed at each core at each cycle.

In comparison to 128-bit Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) and 256-bit Intel® Advanced Vector Extensions (Intel® AVX), this new coprocessor can pack up to 8x and 4x the number of operations into a single instruction, respectively.

3) Caches and Main Memory: Each Intel® Xeon Phi™ coprocessor core has a private 512KB L2 cache that is kept coherent using a distributed tag directory. The caches, directories, memory controllers, and the PCIe interface are connected using two unidirectional ring networks. A miss in a local L2 cache (that can be a hit in a remote L2 cache or trigger an off-chip memory access) must travel through this interconnect to reach its destination. Having more blocks connected to the ring network increases the average number of hops each message has to travel on the network, which in turn increases the total access latency.

While the Intel® Xeon Phi™ coprocessor has a total on-chip cache capacity of 30MB, which is higher than the sum of L2 and L3 caches on Intel® Xeon® processor E5-2600 product family, its per-core cache capacity is actually smaller. Instead of 256KB L2 and 2.5MB L3 effective cache capacity per-core, an Intel® Xeon Phi™ coprocessor core has only 512KB L2 cache. Further, this space is shared by four threads. This reduction in per core cache capacity is a design choice that enables such a high number of cores to be integrated in a single chip. As a result, the on-chip cache in this architecture is more limited and effective management of its capacity is more important to achieve high performance.

The Intel® Xeon Phi™ coprocessor resides on a separate add-in card connected to a host system through PCIe. The memory system on the coprocessor is completely independent from the host memory system. The coprocessor has 16 channels of GDDR5 memory that are controlled by 8 on-chip memory controllers. Operating at a 32-bit interface and 5.5GT/s speed, the memory subsystem has a maximum memory bandwidth of 352GB/s. On the other hand, Intel® Xeon® processor E5-2600 product family members have 4 channels that can deliver up to 51.2GB/s maximum bandwidth per socket at DDR3-1600. With its bandwidth advantage, Intel® Xeon Phi™ products can be expected to achieve higher performance on applications that are memory bandwidth bound.

Although the Intel® Xeon Phi™ coprocessor has a high memory bandwidth, it also has a high off-chip memory access latency. This is mainly because the GDDR5 memory it uses is optimized for bandwidth rather than latency, and therefore has a higher access latency than the DDR3 memory technology [2] used with other Intel® Xeon® processors.

Considering the smaller per-core cache capacity, an increase in off-chip memory access latency, together with its in-order cores that are not able to hide cache/memory access latency as out-of-order cores, optimizing the cache/memory behavior of applications becomes critical in reaching high performance with the Intel® Xeon Phi™ coprocessor. In an unoptimized case, application threads will frequently miss in local L2 caches and stall for extended periods of time until the needed data is retrieved from remote L2 or off-chip memory, which will underutilize the core computational resources and lead to poor performance.

In Sections 3 and 4, we show two important methods to alleviate these issues: data prefetching and streaming store instructions. Our work demonstrates compiler-based generation of these instructions and evaluates their impact on performance.

B. Programming and Optimizing for the Intel® Xeon Phi™ Coprocessor

After providing an architectural overview, we now provide a view of this coprocessor from a programmer’s angle. For
further details on programming for this architecture, we refer the reader to [3], [4].

1) Usage Models: There are two usage models for the Intel® Xeon Phi™ coprocessor: offload and native. In this work, we evaluate both usage models using applications that are parallelized using OpenMP. For our applications that use the offload usage model, execution starts on the host processor and then scalable parallel regions of the application are offloaded to the coprocessor. These codes are structured such that they transfer data from the host processor to the coprocessor upon entry to offload regions and in the other direction upon exit from offload regions. For our applications that use the native usage model, the entire execution of the application happens only on the coprocessor. In this usage, application data is allocated only on the coprocessor and is never transferred between the host processor and the coprocessor. Both models can also be used to structure MPI programs that scale to multiple processors and coprocessors. In the native model, processors and coprocessors both have MPI ranks, whereas in the offload model, only the former has MPI ranks and each processor can use its coprocessors to offload some of its work.

2) Optimizing for the Coprocessor: There are fundamental analyses and optimizations that must be performed to achieve high performance from the Intel® MIC architecture such as scalability/parallelization, vectorization, code transformations, data layout transformations, etc. The most important property an application must possess in order to be suitable for the Intel® MIC architecture is scalability: it must scale well to hundreds of threads to effectively utilize the large number of cores. However, it must be kept in mind that the highest performing configuration is not always the one that uses the highest number of threads (or even cores). For instance, while maximum performance can be obtained by scattering one thread per core for one application, fully utilizing all hardware thread contexts and using a balanced thread mapping can be better for another. The number of threads/cores used and the affinity setting chosen can significantly affect performance. Using more threads per core can better hide cache/memory access latencies through fine-grain hardware multithreading, but it also reduces the effective cache/TLB capacity and bandwidth per thread.

As the Intel® Xeon Phi™ coprocessor also relies on effective utilization of its long vector units to achieve high performance, a significant portion of the application should be vectorizable and the vectorized code should efficiently use all the lanes in the vector units. There can be various reasons for underutilization, the most important one being an inefficient algorithm that does not make use of all vector lanes. Other examples are the use of non-unit-stride or unaligned memory accesses in vector-loops. For instance, using arrays of structures can lead to constant, but non-unit stride accesses. In this case, the programmer can manually perform AoS (Array of Structures) to SoA (Structure of Arrays) data layout transformations to improve performance.

The vector units in the coprocessor have a vector length of 512-bits (64B). A vector load/store that is aligned to 64B boundary can be performed as a single instruction, whereas an unaligned vector load/store requires two separate memory accesses (vloadunpackedo/vpackstorelo and vloadunpackedi/vpackstorehi) and a merge/split operation (performed implicitly) on them to write/read the data to/from a vector register. To obtain the benefits of aligned memory instructions, programmers must (i) instruct the compiler and runtime system to allocate data items from aligned memory locations and (ii) inform the compiler that it is safe for it to generate aligned memory instructions when accessing those data items.

There are also various other code transformations that can significantly impact the performance of applications on Intel® MIC architecture. For instance, loop tiling cache locality optimization can make a bigger impact due to its higher off-chip access latency, and loop unrolling can lead to better core utilization due to having simpler in-order cores.

While we briefly mentioned these optimizations in this section for completeness, these topics are not the main subject of this paper. We refer the interested reader to various articles published in the literature for multi-core and vector architectures. In this paper, we evaluate applications that are highly threaded, efficiently vectorized, and optimized by the programmer and/or the compiler for the Intel® MIC architecture.

C. Intel® Composer XE 2013

Intel® Composer XE 2013 includes an optimizing C/C++/Fortran compiler that delivers high performance for applications running on Intel® Core™ or Intel® Xeon® processors, Intel® Xeon Phi™ coprocessors, and IA-compatible processors. It includes state-of-the-art compiler optimizations including but not limited to code restructuring, inter-procedural optimizations, memory optimizations, partial redundancy elimination, and partial dead store elimination. The compiler performs automatic parallelization and automatic vectorization. It also supports various directive-based methods to express parallelism and vectorization. Parallel execution can be obtained using OpenMP pragmas or Intel® Cilk™ Plus task/data parallelism constructs. The programmer can manually annotate loops that needs to be vectorized using the #pragma simd pragma, or use the Intel® Cilk™ Plus array notation that helps the compiler in generating vector code. For complex loops with function calls that cannot be vectorized even after function inlining, elemental functions can help in obtaining vector code. Further details about the compiler can be found in [5].

III. PREFETCHING ON THE INTEL® XEON PHI™ COPROCESSOR

The Intel® Xeon Phi™ coprocessor relies heavily on prefetching for hiding cache and memory access latency. It has two prefetching mechanisms. The first is software
prefetching that uses explicit prefetch instructions. The second is hardware prefetching that is transparent to software, which dynamically identifies cache miss patterns and generates prefetch requests. Both methods aim to retrieve data ahead of need to prevent blocking of application threads due to cache misses.

A. Software Prefetching

Software prefetching involves static identification of memory accesses in a program and insertion of prefetch instructions for these memory accesses such that the prefetched data will be readily available in the on-chip caches when the memory access is executed. While prefetch instructions can be inserted by expert programmers using low-level intrinsics, this is a tedious and error-prone task. The compiler incorporates advanced techniques to do efficient software prefetching for memory accesses.

1) Architectural Support: The Intel® Xeon Phi™ Co-processor contains different types of software prefetching instructions. The vprefetch instruction brings a 64B line of data from memory to L2 cache, and vprefetch0 further pulls it to the L1 cache. The cache line allocated for the prefetched data is put into shared (S) coherence state in the MESI protocol. There is also another variant of both prefetch instructions that mark the cache line as exclusive (E) in the tag directory (vprefetch1 and vprefetch0) [1]. Prefetching loads data into the cache speculatively, which may lead to an attempt to access memory locations that are not allocated to the current process. In this case, the hardware does not generate page faults and silently ignores the prefetch instruction. This eliminates the need to use guarding bound-check statements around prefetch instructions.

2) Prefetch Distance Computation: Loops constitute almost all of the execution time of applications and typically access large ranges of memory regions, and therefore, they are the best targets to apply data prefetching. As temporal locality in L1 cache and register promotion can simply hide the latency of scalar memory accesses, the main concern for a prefetcher is aggregate variables. In a loop, hiding the latency of an access to an array location that will be used in one loop iteration typically requires it to be prefetched a few iterations ahead (of that particular access). In other words, these prefetches target array locations that will be accessed in future iterations of the loop. Prefetch distance is defined as how many loop iterations into the future the generated prefetch instructions target. Since the compiler performs prefetching after the compilation-phase that initiates vector code-generation, prefetch distances are expressed in terms of vector loop iterations (in case the loop is vectorized). The compiler can also generate prefetches for vector load/store intrinsics inserted by the programmer, and pointer accesses that are similar to array accesses, where target address can be predicted in advance. It supports prefetch address calculations that involve affine functions of surrounding loop indices and functions of those indices expressed using additional instructions in the loop (that can be identified by the compiler using induction variable substitution).

Given a loop with a memory access to associate a prefetch with, there are two measures that determine the optimum distance for the prefetch. The first one is the expected latency of the prefetch operation, i.e., how long it will take for the prefetch to complete. The second one is the rate of execution of the loop, i.e., how rapidly will the program execution progress and reach the actual memory access. Ideally, the prefetch operation should complete just before the memory access is encountered, otherwise either the latency will not be hidden completely or a useful block may be evicted from the cache.

The factors that determine the latency of a prefetch operation are (i) the latency of accessing the target level in the memory hierarchy (more than ten cycles to access local L2 cache, hundreds of cycles to access main memory), and (ii) the amount of load in the system that adds additional queuing delay on top of these base latency values.

Calculating the rate of execution of a loop involves finding the expected total latency of executing one iteration of the loop. This can especially be complicated when (i) there are inner loops, and (ii) there are multiple threads sharing the instruction issue cycles. In the presence of inner loops, an estimate of the inner loop trip-count is used to identify the latency of executing one iteration of the outer loop.

While deciding on prefetch distances and generating prefetches, the compiler also takes into account the impact of issuing those prefetches on the TLB (Translation Lookaside Buffer) pressure during execution. If the distance chosen for a prefetch is likely to cause pressure on the TLB, then the compiler reduces the prefetch distance to alleviate any TLB thrashing due to prefetching. The -opt-report compiler option gives detailed information on the software prefetches inserted by the compiler and the distances used for them.

3) Outer Loop Prefetching: For data accesses in inner loops of a loop nest, the compiler decides whether to prefetch for the future iterations of the inner loop or the outer loop. This decision is based on heuristics such as the estimated trip counts of the loops and symbolic contiguity analysis of data accesses in the inner loop. Clearly, it is important for the compiler to have good trip count estimates to make the right decision. The compiler not only easily identifies constant trip count loops, but also uses various other methods to identify maximum trip counts such as analyzing array dimensions. The programmer can also assist the compiler by providing loop trip count hints (e.g., #pragma loop_count(200)).

4) Initial Value Prefetching: Prefetching for future loop iterations does not eliminate cache misses for the first few iterations of loops. To handle these misses, the compiler inserts initial value prefetches before loops. This optimization is especially useful for loops with small trip counts.

5) Exclusive Prefetching: If a memory reference that is being prefetched will immediately be written by the
current thread, then the compiler uses exclusive prefetch instructions to prefetch it. This saves an extra access to the tag directory during the subsequent store instruction to change the state of the cache line from shared to exclusive. While exclusive prefetching can improve performance in this case, if accidentally used too aggressively, it can cause invalidation of useful data in the caches of other cores and can lead to ping-pong of data across cores. Therefore, the compiler generates exclusive prefetches only for the cases where the cache line will be updated shortly by a store instruction.

6) Indirect Access Prefetching: Indirect array accesses do not follow simple strided access patterns and two consecutive indices may access data that are far apart in the target array, residing on different cache lines. A typical example is when an index array is used to access a second array, as given by the A[B[i]] notation. Indirect accesses are frequently observed in scientific applications, such as those working with sparse matrices that are stored and accessed using just the indices of the non-zero elements for space efficiency. When the target array is large, accesses to it become cache misses and the memory latency becomes the main factor that limits performance. Performance can be improved by issuing prefetches to hide the latency of these indirect memory accesses. At loop iteration i, if we want to prefetch for D1 iterations ahead (i.e., prefetching A[B[i+D1]]), we must also make a demand access to B[i+D1]. Therefore, to make sure the access to the index array (i.e., B[i+D1]) will also become a cache hit, we need to make another prefetch for some further D2 iterations (i.e., prefetch B[i+D1+D2]). Note also that the load of the index array (that is used in the address calculation for the indirect prefetch) has the danger of accessing beyond the array boundary and create a page fault. The compiler has to generate a bounds-check to guard against this possibility.

On the Intel® Xeon Phi™ coprocessor, up to 16 iterations of a scalar loop may be packed into one vector iteration. The compiler generates gather/scatter instructions (vgather/vscatter) when it vectorizes loops with indirect array accesses. And it can generate up to 16 prefetch requests to cover all locations that will be accessed by a gather/scatter instruction. Meanwhile, the index array is accessed sequentially, and only one prefetch operation is sufficient to bring 16 index values into the cache.

7) Controlling the Default Compiler Prefetching Behavior: While the compiler can perform all the associated analyses and generate software prefetches fully automatically, it also supports programmer control of various prefetching parameters through compiler options and pragmas for fine-tuning. Compiler options modify the default behavior for an entire compilation unit and pragmas operate at a finer granularity and override the compiler options for their target loops or functions. The compiler supports the following mechanisms to alter the default behavior of the software prefetcher (for prefetching to L2 cache, L1 cache, or both):

- Disable software prefetcher (option and pragma): The programmer may choose to disable generation of software prefetch instructions. For instance, if the hardware is able to prefetch equally well in an application, then one may be able to save some cycles spent for fetching/issuing/scheduling software prefetch instructions by disabling software prefetching. Another example is an advanced programmer who disables compiler-based data prefetching to manually insert prefetch instructions using intrinsics.

- Set prefetch distance (option and pragma): The programmer may choose to control the prefetch distance manually. This could be useful when the compiler is not able to make a good decision on the prefetch distance, for instance when there is an inner loop with an unknown or variable trip count.

- Enable indirect prefetch generation (option): In its default behavior, the compiler does not generate indirect prefetches. Indirect prefetching can be enabled by using a compiler option.

B. Hardware Prefetching

The Intel® Xeon Phi™ coprocessor employs a 16-stream hardware prefetcher [1] that is enabled by default when the system starts. It observes L2 cache misses and, upon detection of a cache miss pattern, it starts issuing prefetch requests to memory. While we believe that there is no reason for typical users to turn off the hardware prefetcher, we have collected data with the hardware prefetcher disabled in order to report the isolated performance of the compiler-based software prefetching and also to analyze the interactions between hardware and software prefetching.

C. Combined Software/Hardware Prefetching

When hardware and software prefetching are used together, a cache miss successfully converted into a cache hit by the software prefetcher will not train the hardware prefetcher. Therefore, for those streams of accesses, the hardware prefetcher will not generate any prefetch requests. On the other hand, if there are some streams that are not handled by the software prefetcher, the hardware prefetcher may still be able to detect them at runtime and issue prefetches for them. One example is when the compiler is unable to analyze an array subscript expression and cannot generate any prefetches.

IV. STREAMING NON-TEMPORAL STORE INSTRUCTIONS ON THE INTEL® XEON PHI™ COPROCESSOR

Streaming non-temporal store instructions are specialized memory store instructions designed to save off-chip memory bandwidth in cases where data with no temporal locality is being streamed into memory. Unlike regular stores, such store instructions do not perform a read for ownership (RFO) for the target cache line before the actual store. The rationale behind this is that any data read from memory for this purpose will anyway not be used and will get overwritten by the data stream.

In this section, we provide details of the streaming non-temporal store instructions, the conditions under which
the compiler generates those instructions, and software-controlled cache line eviction which is closely associated with non-temporal memory accesses.

A. Types of Streaming Non-temporal Store Instructions

The non-temporal store instructions are vector instructions that operate on data whose length is equal to the cache line size. There are only unmasked, aligned versions of these instructions. Therefore, they can only be used when the target store address is aligned to 64B cache line size and the store operation is unmasked (i.e., the entire vector will be written, not a part of it). If the store is unaligned or masked, regular store instructions (vpackstore or vmov) must be used.

Non-temporal streaming store instructions simply allocate a cache line in L2 and then set its entire contents. There are two types of streaming non-temporal store instructions on the Intel® Xeon Phi™ coprocessor: NR and NR.NGO stores. Both of them can be used to save bandwidth, but their performance and applicability are different.

NR Stores. The NR (No-Read) store instruction (vmovnr) is a variant of the standard vector store instruction that can always be used safely. An NR store instruction that misses in the local cache causes all potential copies of the cache line in remote caches to be invalidated, the cache line to be allocated (but not initialized) at the local cache in exclusive state, and the write-data in the instruction to be written to the cache line. There is no data transfer from main memory which is what saves memory bandwidth. An NR store instruction and other load and/or store instructions from the same thread are globally ordered, which means that all observers of this sequence of instructions always see the same fixed execution order.

NR.NGO Stores. The NR.NGO (Non-Globally Ordered with No-Read hint) store instruction (vmovnrng0) relaxes the global ordering constraint of the NR store instruction. This relaxation makes the NR.NGO instruction have a lower latency than the NR instruction, which can be used to achieve higher performance in streaming non-temporal store intensive applications. However, removing this restriction means that any stores following an NR.NGO store can be globally observed (by a different thread) before the NR.NGO store itself. The compiler can generate NR.NGO store instructions for store instructions that it identifies to have non-temporal behavior. For instance, a parallel loop that is detected to be non-temporal by the compiler can make use of NR.NGO instructions. To ensure all outstanding non-globally ordered stores are completed and the threads have a consistent view of memory, the compiler generates a fence (a lock instruction) after such a loop. This fence instruction ensures that all threads have exactly the same view of memory before execution continues with the subsequent code fragment.

It is important to make a comparison of the streaming non-temporal store instructions on the Intel® Xeon Phi™ coprocessor and other Intel processors. Intel® Streaming SIMD Extensions first introduced the movntpd instruction that stores a full 16B vector register to memory. There is only a packed variant of this instruction, which means that the target address must be 16B aligned in memory. Note that while the main memory interface operates at a 64B cache line granularity, the size of data being written using this instruction is only one fourth of a cache line. A typical implementation uses an on-chip buffer to merge the streaming stores that fall into the same cache line and write them together to off-chip memory as a single operation. As the Intel® Xeon Phi™ coprocessor vector length is four times that of Intel® SSE and is equal to the cache line size, it does not need a separate buffer to combine multiple stores. It is also important to note that streaming non-temporal stores on the coprocessor are cached in L2, whereas the implementation on Intel® Xeon processors makes them bypass the on-chip cache hierarchy and get combined on a separate buffer. On the coprocessor, non-temporal stores bypass the L1 cache and directly operate on the L2 cache in order to eliminate the overheads of accessing the L1 cache for data that do not possess temporal locality (i.e., writing to L1 cache and then evicting to L2 cache without accessing the same data again).

B. Cache Line Eviction Instructions on Intel® Xeon Phi™ Coprocessor

Any memory load or store operation that targets a memory location that will not be reused in the immediate future is non-temporal. These non-temporal memory accesses do not benefit from caching and letting them occupy space in the cache until they become the least recently used entry can be a waste of those cache lines. Evicting those cache lines can free up cache space that can be used for caching other data lines. The Intel® Xeon Phi™ coprocessor has special instructions for evicting cache lines from L1 cache and from L2 cache (clevict0 and clevict1). The compiler generates these instructions on loops that are identified to show non-temporal behavior. Note that generating cache line eviction instructions for loops is rather easy on Intel® Xeon Phi™ coprocessor, because its vector length is equal to the cache line size, which means that one cache line can be evicted per vector loop iteration.

C. Controlling the Default Compiler Streaming Non-temporal Store and Cache Line Eviction Behavior

For a non-temporal loop, the default behavior of the compiler is as follows:

- For aligned stores: Generate NR.NGO store instructions, do not generate any prefetch instructions, and generate cache line eviction instructions after NR.NGO stores.
- For unaligned stores: Generate packed store instruction pairs (i.e., two instructions for each store that write the high and low halves of the vector register to cache with mask), generate prefetch instructions, generate cache line eviction instructions to evict the cache lines that are fully written.
- For loads: Generate appropriate load instructions based on alignment (i.e., vloadunpcklo/hi or vmov), generate...
prefetch instructions, do not generate cache line eviction instructions.

The compiler has heuristics to identify loops as non-temporal fully automatically. When these heuristics identify a loop to be non-temporal, the default behavior of the compiler is to generate NR.NGO stores and a fence (a lock instruction) after the loop to ensure safety. However, these fully automatic heuristics rarely get triggered because the compiler needs to know that the trip count of the target loop is large (statically at compile-time). The programmer can assist the compiler in identifying non-temporal loops by manually marking loops as non-temporal using the

```c
#pragma non-temporal pragma or by using the -opt-streaming-stores always compiler option that declares all loops in the target compilation unit as non-temporal. The compiler will then generate NR.NGO instructions for these non-temporal loops and will not generate fences after them. In this case, the programmer is responsible for making sure that this use of NR.NGO instructions is safe.

Alternatively, the programmer can instruct the compiler to generate safe NR stores instead of NR.NGO stores using a compiler option, which case, fences are not needed. The compiler also provides an option for the programmer to disable generation of any cache line eviction instructions.

If an NR or NR.NGO non-temporal streaming instruction is not used for a store operation that shows streaming non-temporal characteristics (e.g., because it is unaligned), the compiler generates prefetches for this instruction. While prefetching will hide the latency of accessing off-chip memory to bring the original cache line contents, it will not have the same memory bandwidth saving benefit of the NR and NR.NGO store instructions. When these special store instructions are used, the compiler does not generate any software prefetch instructions for them (generating prefetches for them will negate their effects), and since these instructions do not generate any L2 cache misses, they also do not train the hardware prefetcher.

If an NR or NR.NGO non-temporal streaming instruction is used for a store operation that does not show non-temporal characteristics, the store will be carried out from the L2 cache. In this case, it will not be possible to exploit temporal locality at the L1 cache level, which may lead to performance degradation. To prevent such a performance degradation, it is advisable to target these instructions only to loops where the non-temporal property holds for the store operations.

V. EVALUATION AND RESULTS

We now present the details of our evaluation of the performance impact of generating prefetch and streaming non-temporal store instructions on applications executing on the Intel® Xeon Phi™ coprocessor. We use an experimental version of the Intel® Composer XE 2013 compiler for all measurements in this paper.

A. Target Applications

In our evaluation, we use applications from various sources. We use eight benchmarks from previous work [6] which provided a detailed evaluation and analysis of these benchmarks from vectorization and parallelization perspectives. We use `swim` and `mgrid` benchmarks from the SPEC OMP 2001 suite with OMPL ref inputs [7]. The measurements for these two benchmarks are not from a formal SPEC run and are not done using the SPEC tools. The source code was also slightly modified to collect detailed timing statistics on a per-loop basis for additional analysis. `swim` is a scalable benchmark, whereas `mgrid` is a moderately scalable benchmark. We report the total (end-to-end) performance of these two benchmarks as well as the isolated performance of their most important three parallel loops.

We use the `Stream` benchmark [8] which is widely used for memory bandwidth testing. `CG` from NAS Parallel Benchmarks [9] is used with its class-B inputs to demonstrate the impact of indirect prefetching. We also use `Leukocyte`, which is a highly parallel benchmark from Rodinia Benchmark Suite [10], and an in-house kernel that performs one dimensional FFT. These applications are executed in native mode on the Intel® Xeon Phi™ coprocessor, which means that the entire application runs on the coprocessor and it makes no data transfer with the host system. We also provide results with two in-house kernels (`3DFD` and `TifajMMV`) that run in offload model whose sequential parts are run on the host machine and parallel kernels are run on the coprocessor. A list of our target applications is provided in Table I. All of our target applications are parallelized using OpenMP pragmas and are vectorized by the compiler. Several of these applications make use of some features available as part of Intel® Cilk™ Plus extensions to express vectorization.

For the applications running in native mode, we used 244 threads. The only exception was for the two bandwidth bound applications Stream and Swim, for which we used 60 and 61 threads (i.e., 1 thread per core), respectively. For the applications running in offload mode, we excluded the last core (which runs the host-target communication processes) from our tests and used 240 threads mapped to the remaining 60 cores.

B. Target System Specifications

Table II provides details of the Intel® Xeon Phi™ coprocessor we used in this work to evaluate the effectiveness of data prefetching and streaming store instructions. Although it is not our primary goal, we also provide some performance comparison with an Intel® Xeon® processor based 2-socket server. The details of this system, which is also used as the host system for applications executed in offload model, are also listed in Table II.

¹When these applications are executed in parallel on the Intel® Xeon® processor, the top three parallel loops constitute 90% of the execution time for `swim` and 83% of the execution time for `mgrid`. 

---

We now present the details of our evaluation of the performance impact of generating prefetch and streaming non-temporal store instructions on applications executing on the Intel® Xeon Phi™ coprocessor. We use an experimental version of the Intel® Composer XE 2013 compiler for all measurements in this paper.
C. Results

In our first set of experiments, we analyze the impact of software and hardware prefetching on the coprocessor when the generation of non-temporal streaming stores is disabled in the compiler. Then, we use the best performing prefetcher configuration as the baseline and evaluate the impact of enabling non-temporal streaming store generation.

1) Prefetching Analysis: To analyze the impact of hardware and software prefetching on the Intel® Xeon Phi™ coprocessor for our target benchmarks, we performed prefetching experiments with the following five configurations: (i) HWP: hardware prefetching only, (ii) NOP: no hardware and no software prefetching, (iii) SWP: default software prefetching only, (iv) HWP+SWP: hardware prefetcher enabled, default software prefetching, (v) HWP+SWP_OPT: hardware prefetcher enabled, manually tuned software prefetching. Based on our results, we categorize the target benchmarks as software prefetching sensitive and insensitive.

Figure 1 shows our results with benchmarks that benefit from software prefetching by at least 3%. The performance numbers on the y-axis are normalized with respect to using the hardware prefetcher alone (HWP). For these benchmarks, we typically see that the third bar (SWP) is higher than the first bar (HWP), which indicates that software prefetching performs better than hardware prefetching. For most of these benchmarks, there is no difference between SWP and HWP+SWP values, which means that enabling the hardware prefetcher when we already have software prefetching does not improve performance. Comparing the HWP+SWP and HWP+SWP_OPT bars, we see that ti-offload, st-scale, st-add, st-triad, sw-loop1, and cg gained from manually tuning software prefetching. In ti-offload, we disabled MEM-to-L2 software prefetching and let hardware prefetcher carry out this task. In st-variants and sw-loop1, we increased the default prefetch distance to 64 and 8 iterations for MEM-to-L2 and L2-to-L1 prefetching, respectively. In cg, we manually enabled indirect prefetching in the compiler, which improved the performance of this benchmark by about 50%. For these benchmarks, the gains from software prefetching range from 5% (cv) to 81% (cg).

Our results with benchmarks that are insensitive to software prefetching are shown in Figure 2. There can be two reasons for an application to be insensitive to software prefetching. First, the hardware prefetcher may already be very effective in converting many L2 cache misses into hits. Second, the application may have good cache locality (e.g., a good reuse pattern or a cache-resident data set) which translates into an already very low cache miss rate. As a result, we further classify our target benchmarks as hardware prefetching sensitive and insensitive. For the benchmarks on the left hand side of Figure 2, we see that the bars 1 and 5 (HWP and HWP+SWP_OPT) are almost equal, which means that software prefetching does not improve their performance. On the other hand, the differences between bars 1 and 2 (HWP and NOP) are substantial. This means that the hardware prefetcher plays a vital role for these benchmarks. The performance degradation from disabling hardware prefetching in these benchmarks ranges from 11% (ts) to 87% (sw-loop5).

Of the benchmarks on the right hand side of the same figure, bp, lbc, rd are not sensitive to any kind of prefetching because they either use a small amount of memory that fits into the on-chip cache or have their code optimized (e.g., using tiling) for the cache capacity of the Intel® Xeon Phi™ coprocessor. The st-copy benchmark is also not sensitive to any prefetching because the compiler automatically converts the copy loop into a memcpy operation. This memcpy operation is an optimized library function that implicitly uses software prefetch instructions that we cannot control.
using the compiler. As a result, the NOP configuration where we disable the hardware prefetcher and software prefetch generation in the compiler continues to perform well as it still benefits from software prefetching. For this set of benchmarks, disabling the hardware prefetcher leads to less than 2% performance degradation.

2) Streaming Non-temporal Store and Cache Line Eviction Analysis: Our next study is to identify how much performance benefits can be obtained by using streaming store and cache line eviction instructions on top of optimized prefetching. For this purpose, we used the optimized prefetching configuration (HWP+SWP_OPT) from the prefetching analysis above as the baseline and generated four configurations. Each configuration uses either NR or NR.NGO stores (shown in the figures as +NR or +NGO). Two out of four configurations also have cache line eviction enabled (shown as +CLE). For this analysis, we add a compiler option (-opt-streaming-stores always) that treats all loops in the benchmarks as non-temporal loops. As we will see with the results, this not only helps us identify which benchmarks/loops can benefit from streaming stores and cache line eviction, but also shows us how much performance degradation can occur if we incorrectly treat temporal loops as non-temporal.

Note that there are only aligned versions for the streaming non-temporal store instructions. Therefore, the compiler can only generate these instructions if a store in a non-temporal loop is aligned. When there are multiple unaligned array accesses in a loop, the compiler uses loop peeling to align one of these array accesses to the 64B boundary. The compiler further uses multiversioning, where it generates two versions of the loop. In one version, a second array access is also accessed using aligned memory instructions, whereas the other version uses aligned instructions only for the array access that is aligned using loop peeling. The decisions of which array accesses will be aligned using loop peeling and multiversioning are based on heuristics in the compiler. The heuristics prioritize arrays that are accessed more than once in the loop body where aligning one access automatically aligns other accesses as well (e.g., aligning $A[i]$ also aligns $A[i+64]$). When a loop is either identified by the compiler or marked by the programmer to be non-temporal, then the store instructions in the loop are prioritized to be aligned using loop peeling and multiversioning to enable generation of more efficient NR/NR.NGO store instructions.

We classify the target benchmarks into two groups (shown in two figures) based on whether or not they benefit from the use of streaming non-temporal store instructions. Figure 3 shows our results with benchmarks that receive more than 3% performance improvement when using some combination of streaming stores and cache line eviction. Clearly, the biggest improvements are obtained with the tests st-scale, st-add, and st-triad. In these tests, using NR.NGO stores brings 21-45% performance improvement. This is an expected result as Stream is a benchmark that is designed to be bandwidth bound. By eliminating redundant reads from memory using streaming non-temporal store instructions, a higher Stream score is obtained. sw-loop1 and sw-loop2 also show significant 12% performance improvement, which is the primary reason for having 7% performance improvement in sw-total. While the benefit on cg is also high (13%), the remaining three benchmarks show relatively small but steady improvements in the 3-5% range. Enabling cache line eviction while also having some form of streaming stores...
improves the performance in only mg-loop1. For this loop, the +NGO+CLE configuration achieves 4% speedup.

For most of the remaining benchmarks, shown in Figure 4, there is either no impact on performance or a degradation less than 10% is observed. However, there are a couple of benchmarks that show significant performance degradation. For lbc and ti-offload, treating all loops as non-temporal and using cache line eviction for all store instructions causes 3-10X penalty in performance. The reason for this degradation is the elimination of cache locality over the outer loop in a loop nest. When a vectorized inner loop in a loop nest uses a small output array that is defined over an outer scope (e.g., the body of the outer loop), then the small trip count inner loop can reuse these arrays from the cache for consecutive outer loop iterations. However, when the programmer forces the compiler to treat the inner loop as non-temporal, the compiler evicts these arrays from the cache at the inner loop.

Then, in the next iteration of the outer loop, accesses to these arrays become cache misses. These benchmarks show that using cache line eviction instructions for temporal loops can severely degrade performance. Therefore, when evaluating the impact of streaming non-temporal store generation by adding the -opt-streaming-stores always compiler option, it is recommended that programmers also make experiments with cache line eviction turned off. Using the compiler option treats all loops in the target compilation unit as non-temporal and a performance degradation on one loop due to cache line eviction can hide all the benefits obtained on other loops. We recommend the programmers to carefully analyze the impact of streaming non-temporal store and cache line eviction instructions taking the data access pattern of individual loops into account. It is preferable to apply these instructions selectively only for non-temporal loops where there is no benefit from keeping the stored data in any level of the cache hierarchy.

3) Comparison with Intel® Xeon® Processor based System: We also evaluated a subset of our benchmarks on a dual-socket Intel® Xeon® processor based system to compare its performance with the Intel® Xeon Phi™ coprocessor. The results are presented in Figure 5. The 2.2-2.3X difference between the two systems in Stream is the result of the bandwidth on the Intel® Xeon Phi™ coprocessor being much higher. For the two SPEC OMP applications, we see that the Intel® Xeon Phi™ coprocessor performs about 1.7X better for swim, but performs similar to Intel® Xeon® processor based system for mgrid. The reason for this behavior is that swim is a scalable benchmark that is very suitable for this many-core architecture, whereas mgrid is a moderately scalable benchmark. For the rest of the benchmarks in Figure 5, Intel® Xeon Phi™ coprocessor achieved performance improvements in the range of 1.2X (ts) to 4.8X (lbc), with the geometric mean improvement being 2.1X. This is an expected outcome as these benchmarks were specifically written and optimized to efficiently exploit the Intel® MIC architecture.

VI. RELATED WORK

Software and Hardware Prefetching. Starting with the first proposals for compilers generating non-blocking prefetch instructions to improve cache performance [11], [12] and using specialized hardware for data prefetching [13], [14], a significant body of work has been done in both areas. Examples on the software prefetching side include prefetching for recursive data structures [15] and thread-based prefetching using speculative precomputation [16], and on the hardware side include streaming memory controllers to improve throughput by prefetching streams and using hardware prefetchers that utilize idle memory channel cycles by issuing prefetches [17]. A recent work on hardware prefetching focused on using smart stream detection methods that can dynamically modify prefetch settings to adapt to the workload needs [18]. Vanderwiel and Lilja [19] provide a survey on several alternative approaches to hardware and software data prefetching strategies.

Evaluation of Compiler Optimizations and Data Prefetching. Previous studies have examined the effect of compiler optimizations such as automatic vectorization, automatic parallelization, and inter-procedural optimization [20], [21]. These studies have not analyzed the impact of prefetching or non-temporal streaming stores. The prefetching behavior of an Intel® Itanium® 2 processor based SMP system was analyzed by Tian et al. [22]. While we work with a very different single-chip many-core co-processor architecture, we also evaluate the impact of streaming non-temporal store instructions on this system.

Recently, Lee et al. [23] performed a detailed analysis of various software and hardware prefetching methods. Their goal was to provide a deeper understanding of the prefetching concept. They provide a detailed examination of the prefetch behavior of sequential SPEC CPU2006 applications using simulations of various out-of-order processor configurations. Our work differs from theirs in the following aspects. First, we target highly parallel and vectorized applications that put much more pressure on the memory system. Second, we evaluate the Intel® Xeon Phi™ coprocessor with very different characteristics than their target systems. Third, our compiler generates software prefetch instructions automatically while they inserted prefetch instructions manually using intrinsics.

Nontemporal and Streaming Nontemporal Stores. Non-temporal and streaming non-temporal store instructions appeared as a part of SIMD extensions. While SIMD extensions have been in mainframes since the early 80s, they moved to desktop processors in the late 90s with the goal of improving the performance of data-parallel applications. Popular examples of these extensions are Intel® SSE [24], Altivec™ technology [25], and 3DNow!™ [26] from AMD. An early work that evaluates the performance of various

2The peak st-triad value we obtained in our tests (using default 4K pages) on the Intel® Xeon Phi™ coprocessor was 157GB/sec.
applications using Intel® SSE extensions is from Abel et al. [27]. This work uses extensions such as prefetch and cache control instructions to improve performance. While the first extensions used 128 bits, the latest industrial trend in SIMD extensions is to increase SIMD vector length. Recent processors with Intel® AVX use 256 bits vector operations and the Intel® Xeon Phi™ coprocessor vector length is 512 bits. Hence, the importance of effective use of these SIMD extensions is increasing.

**Intel® Xeon Phi™ coprocessor** One of the early papers that presented results with the Intel® Xeon Phi™ coprocessor was by Satish et al. [6]. This work used an Intel® Xeon Phi™ Software Development Vehicle (formerly known as “Knights Ferry”) that was not publicly available, and it focused on generic parallelization, vectorization, and optimization features without any specific emphasis on the coprocessor. It did not include any detailed analysis on prefetching or streaming stores and the associated impact on this new architecture. A scalability study of graph algorithms on the same system was performed by Saule and Catalyurek [28]. Generally, the studies so far have addressed the angle of parallelization and vectorization of a single (or a set of) application(s) and did not disclose any details on the overall prefetching and streaming store behavior. We believe that this is the first effort that analyzes the impact of compiler-based prefetching and generation of streaming store instructions on a wide range of applications on the Intel® Xeon Phi™ product family. With the release of the Intel® Xeon Phi™ coprocessor as a product, more studies that demonstrate the results of porting and optimizing applications to this new coprocessor will appear. These studies can use the software prefetch and streaming non-temporal store generation mechanisms of the compiler that we presented in this work to improve the performance of their applications.

VII. CONCLUSION

We provided details on compiler-based generation of software prefetch and streaming non-temporal store instructions for the Intel® Xeon Phi™ coprocessor. Our experiments showed that by exploiting these mechanisms to better hide memory latencies and save bandwidth, Intel® Composer XE 2013 can achieve significant performance improvements.
We believe that our findings will be useful for users of the Intel® Xeon Phi™ coprocessor as well as users and compiler developers of other many-core vector architectures.

ACKNOWLEDGMENT

The authors would like to thank all members of the Intel compiler team. Emre Kültürsay would also like to thank his academic advisor Mahmut Kandemir.

REFERENCES


