Data Prefetching on a Manycore Architecture Case study: The XMT Platform *

George C. Caragea
Dept. of Computer Science
University of Maryland, College Park
george@cs.umd.edu

Abstract

Multicore architectures are becoming ubiquitous in the microprocessor market today. All major vendors are pushing up the number of cores, with plans to roll out as many as 80 cores on a chip by the year 2010. The creation of manycore architectures – hundreds to thousands of cores per processor – is seen by many as a natural evolution of multicore. We present several compiler optimizations targetting fine grained, highly parallel manycore architectures. We are focusing on automatic techniques to unlock the potential of the complex, heterogeneous, deep cache memory hierarchies of these architectures, with an emphasis on software (or compiler-directed) data prefetching. We exemplify using the XMT Platform, a research project under development at the University of Maryland comprising a prototype processor, a compiler and a PRAM-like programming model.

1 Introduction

Most if not all modern architectures spend a significant amount of their execution time waiting for memory requests to complete. Complex cache memory hierarchies along with compiler locality optimizations are effective in reducing the stall time, but cannot completely eliminate it. Mechanisms such as software and hardware prefetching are commonly used to tolerate the continuously increasing (in terms of cycles) memory latencies.

XMT is no exception [NNTV01, NNTV03, VDBN98, BV06, WV07]; on the current prototype, a DRAM access takes over 200 clock cycles, and, even more important, a round-trip to the first level of shared cache is approximately 24 clock cycles. Features such as prefetch buffers at the TCUs and read-only caches at the cluster level have been introduced precisely to alleviate the impact of these high latencies, but they are ineffective without proper compiler support.

2 Related work

Prefetching - Serial Machines. Two approaches to prefetching on serial architectures have been studied and implemented: (i) software, or compiler-assisted prefetching assumes the existence of non-blocking prefetch instructions and is usually enabled by the compiler; and (ii) hardware prefetching, where specialized hardware infers prefetching opportunities by monitoring run-time behavior.

*This scholarly paper is submitted in fulfillment of a partial requirement for the completion of the Masters Degree without Thesis in Computer Science.

Hardware prefetching schemes present in the literature include prefetching for unit and non-unit stride memory accesses [LRB01, Jou90, PK94], and mechanisms to identify and prefetch pointer data structures [RMS98, Coo02]. Hardware prefetching works with all existing code and is typically capable of hiding higher memory latencies, but lacking the benefit of compile-time information, it usually has poorer accuracy and coverage [WBM+03, VL00]. The hardware complexity of these mechanisms makes them unlikely to scale to a many-core architecture such as the proposed 1024 TCU XMT.

Various *software prefetching* techniques, with different degrees of applicability depending on the access patterns and data structures of applications, have been proposed. Extensive research has been done on compiler algorithms to enable prefetching for loop-based array computations, based on affine function analysis [MLG92, Mci98, CKP91, KL91]. To prefetch for applications with irregular data access patterns, which include most pointer-based data structures, techniques such as greedy prefetching and jump pointers [LM99, CM01] have been studied. *Software-hardware hybrid schemes*, where hints inserted by the compiler are used by a specialized hardware prefetcher, have also been proposed [WBM+03, Gor95].

Most prefetching techniques bring data either to the level of cache nearest to the CPU (L1), or to a dedicated prefetch cache. Kim and Veidenbaum [KV97] propose a hybrid software-hardware mechanism to prefetch data from DRAM to L2 on a serial architecture. We plan to implement our own software-only prefetching algorithm to bring data from DRAM into the L2 cache on XMT, which will take advantage of the programming and hardware characteristics of a many-core SPMD architecture.

Another widely used technique for hiding memory latencies is *out-of-order execution*, a mechanism which accounts for a significant area and power budget on modern processors. We note however that for a many-core architecture such as XMT, area restrictions require each processing unit to be kept as simple as possible, trading instruction-level in favor of thread-level parallelism.

Prefetching - Parallel Machines. Prefetching for multiprocessor systems differs from uniprocessors for the following reasons: (i) the programming paradigms are different, and they might provide additional information regarding data layout and access; (ii) the memory hierarchies are more complex, with higher latencies and different targets for prefetching source and destination; and (iii) performance is more sensitive to prefetching, since additional traffic caused by prefetching can quickly deplete the limited bandwidth provided by the interconnection networks.

Several multiprocessor *hardware prefetching* mechanisms which extend upon uniprocessor schemes have been proposed [Gor95, DDS95, KK97, WH03]. Most of these techniques extend uniprocessor hardware prefetching schemes to dynamically adapt to run-time parameters such as available bandwidth and volume of coherence traffic. As noted before, the complexity of these hardware units makes it prohibitive to implement for a highly parallel architecture such as XMT.

Multiprocessor *software prefetching* techniques augment uniprocessor algorithms with more sophisticated analyses to filter out harmful or incorrect prefetch instructions, which can happen when shared data is prefetched into local memories or caches [Mow98, Mci98]. This can cause significant additional traffic to maintain coherence, or introduce stale data if no coherence guarantees are made by the architecture. These factors can erode or even overcome the benefits of prefetching [JHL06, TE95]. We plan to use similar analyses to devise cache-coherence aware prefetching algorithms for XMT.

A different approach is examined by Gornish et al. [GGV90] who propose using block transfers to bring data from the shared global memory to the private memories of processors before it is needed. An analysis is performed for every loop to identify memory regions used by each processor. Using asynchronous block transfers reduces the communication overheads, but it can increase congestion in the interconnection network, and it might generate unnecessary prefetches if a loop does not access a contiguous memory region. The cited paper does not take into account the size of the local processor memories, assuming that it is large enough to hold the memory region accessed within a loop. We plan to use a similar analysis to determine the data accessed by all the threads in a spawn block, but enhance the compiler algorithm with scheduling capabilities

for the case when this data exceeds the size of the on-chip cache.

3 Prefetching on XMT

The XMT platform [NNTV01, NNTV03, VDBN98, BV06, WV07] has several particularities which set it apart from other parallel architectures. Novel prefetching algorithms and techniques are needed to take advantage of these features. These differences include, but are not limited to:

- **Multilevel heterogeneous memory hierarchy.** The XMT memory hierarchy is composed of the following levels, each with different properties and characteristics:
  - **Prefetch buffers:** Small (8 or 16 words) fully associative cache located at TCUs, used to store prefetched locations. On the current prototype, the latency of a hit is 2 clock cycles, comparable to a register access.
  - **L0:** Read-only caches situated at cluster level. These are shared by the TCUs in a cluster, but are local to each cluster. It is unrealistic to have one cache for each TCU because of area constraints, but current projections show as feasible to include a 8KB cache for each 16 TCU cluster, adding up to 512KB for a 1024 TCU XMT chip. The latency of an L0 access is approximately 4 clock cycles on the current prototype.
  - **L1:** Shared by all TCUs, 2-way set associative, write-through. This cache is located on the other side of the interconnection network from the TCUs, leading to a latency of approximately 24 cycles for an L1 hit in the current XMT embodiment. Estimates show a 2MB L1 cache divided in 64 modules is feasible for a 1024 TCU configuration.
  - **L2:** Shared by all TCUs, 4-way set associative, write-back. Approximately 4-8 times larger than L1 and operating at a lower speed. Located between L1 and the DRAM ports, with a projected latency of 28-30 clock cycles.
  - **DRAM:** The current XMT prototype uses off-the-shelf DDR2 DRAM; the access latency varies significantly depending on the access pattern, and it is expected to be about one order of magnitude higher than for the on-chip cache.
  - **Master TCU cache:** Located at the Master TCU, this cache is used only while operating in serial mode and its contents are flushed to shared cache when starting a serial section. The current prototype includes 16KB of direct-mapped, write through cache, with a latency of 3 cycles.
- **Two-phase hardware thread scheduling.** On XMT, the programmer can request starting any number of threads using the `spawn` construct, and the threads are dynamically assigned to the available TCUs by a hardware mechanism. This is fundamentally different from existing parallel machines, where the assignment is usually static and known at compilation time. The scheduling takes place in two phases:
  - **Fixed ordering:** The first NUM_TCU threads are assigned to the existing TCUs in increasing order of the thread-id, in a one-to-one mapping.
  - **Greedy scheduling:** The remaining threads are assigned to TCUs as they become available.

  This approach provides load balancing by ensuring that no hardware resources are idle while there are still threads which haven't been executed. On the downside, there is no compile time information on the order in which threads scheduled in the second phase execute on the TCUs.

- **Compiler-maintained cache coherence.** XMT offers to programmers the abstraction of a shared-memory, uniform memory access (UMA) architecture, where all memory locations are equally distanced from all processing units. However, in addition to the shared on-chip cache, local cluster and TCU caches have been added to achieve better performance. In the presence of private caches, the contents of one memory location can be stored in more than one place at a time, raising the issue of coherence. Enforcing cache coherence in hardware requires complex protocols and additional logic, and the costs of scaling such mechanisms to the envisioned 1024 cores of an XMT chip seems prohibitive, even with directory-based protocols such as [LLG+00, KOH+98]. Exposing this issue to the programmer is equally problematic and it would significantly hinder her productivity. Therefore, the approach taken by XMT is for the compiler to be responsible for maintaining cache coherence for the TCU prefetch buffers and the cluster caches.

We plan to address each of the above listed XMT aspects by designing compiler algorithms tailored for XMT's unique architecture. Thereafter, we will implement each of these methods as part of the proposed infrastructure development effort in the XMT compiler. *Such a mature infrastructure with state-of-the-art prefetching techniques implemented in a robust and highly tested platform is necessary for the larger community of XMT users to avail of the full potential of the XMT architecture*. The prefetching methods we will design and implement on XMT include:

- **DRAM prefetching:** Determining regions of memory from DRAM used by a spawn block and prefetching them to the shared cache. This operation can potentially be scheduled over several rounds if the memory footprint of a spawn block exceeds the capacity of the on-chip cache.
- **Prefetch to TCU buffers:** Inserting prefetch instructions in the parallel sections to bring data from the shared cache to the TCU prefetch buffers. We plan to investigate algorithms for computing prefetch distance given all the XMT parameters and to optimize for loop-based and irregular access patterns.
- **Collaborative prefetching algorithms.** We plan to design prefetching algorithms to take advantage of all the levels of the XMT memory hierarchy, different techniques being applicable depending on the prefetching source and target and their characteristics. We plan to determine the right *set* of prefetching techniques which will cooperate and interact to achieve the best performance on a wide range of applications. For example, computing the prefetch distance for an algorithm using the TCU prefetch buffers should take into account if the datum has been prefetched into L2 or needs to be fetched from DRAM.

The predictability of the thread-to-TCU assignment in the first scheduling phase presents different prefetching opportunities than the threads scheduled in the second phase. We were unable to find any applicable literature for the latter case, especially for fine-grained parallelism, when the threads are usually short. To optimize hardware usage and execution time in the presence of the **two-phase hardware thread scheduling**, we plan to investigate the following compiler algorithms:

- **Prefetching using `early spawn`:** The amount of code in a spawn block before the data is needed may not be enough to hide the latency of the first memory accesses. To avoid having all the treads stalling for data, we can usually overlap the memory accesses with the serial code executing before the parallel section. This requires a slight shift from the XMT execution model, which states that the serial execution is suspended while in parallel mode and viceversa. **Early spawn** provides a way to prefetch for the threads scheduled using the fixed ordering phase. More precisely, since it is known a priori on which TCU will the first NUM_TCU threads execute, we can insert an "early" spawn block in the serial code before the original spawn to start prefetching data to the TCU prefetch buffers while the serial code still executes. Data will be brought to the TCUs and will ideally be available when the "real" spawn block starts.

- **Early get-id** is a mechanism through which the id of the next thread to run on a TCU can be determined ahead of time, and prefetch instructions for its data can be issued while the current thread is still executing. This will effectively prefetch for threads running under the second ("greedy") phase of scheduling, without affecting the load balancing mechanism in a significant way.

- **Thread clustering** is a compiler transformation which groups several SPMD threads into one longer thread, usually by executing them within a loop and replacing the thread-id with the loop variable. The main advantage of clustering is that it increases the predictability of the thread allocation for the second scheduling phase, enabling the compiler to insert prefetching instructions for future iterations of the clustering loop. We note here that **thread clustering for load balance** is an extension of this mechanism which leaves a number of unclustered threads to be executed towards the end of a spawn block. This will allow for better hardware utilization in the presence of the coarser grained parallelism created by clustering.

- **Linear prefetching** is an optimization which inserts prefetch instructions in the code as soon as the address for read operations are computed. This will partially hide the latency of each memory access, and is applicable in both scheduling phases.

To address the issue of **compiler maintained cache coherence**, we plan to augment the prefetching techniques using the following algorithms and analyses:

- **Inter-thread dependence analysis:** Determine if bringing a memory location into a local cache may cause inconsistencies with the original program semantics in the XMT memory model, for example if that location was used to communicate between two threads.

- **Cache-coherence aware prefetch algorithms:** We plan to use the results of the dependence analysis to devise cache-coherence aware prefetching algorithms, which will restrain from prefetching problematic memory locations, or use special cache-invalidating operations around inter-thread communication points to preserve correctness.

- **L0 caching:** The compiler is responsible for marking data to be stored in the L0 local cluster caches. We will use the results of the dependence analysis to identify data which can be brought into the L0 cache without creating inconsistencies, and then cache only data which is reused, from the same TCU or TCUs in the same cluster.

- **Using non-blocking stores.** In the current XMTC compiler, all writes from the threads are done using non-blocking stores, and a special "store-flush" instruction is inserted before each inter-thread communication operation to ensure correctness according to the XMT memory consistency model. We plan to use the results of the dependence analysis to prevent inserting the store-flush instruction when it is not needed, e.g. when there is no shared data between threads.

4 Implementation Details

Consider the code in Figure 1a to be provided by the programmer. We will discuss several code transformations that can be applied to it by the compiler to implement prefetching. The transformations will be shown using the intermediate representation of the code used internally by the compiler. In this representation the outlined parallel code is separated from the serial parts and `join` instructions are explicitly stated. An example is shown in Figure 1b.

For the rest of this document, we will assume for simplicity that there are 1024 parallel TCUs on an XMT chip. The same considerations apply when this number is different.

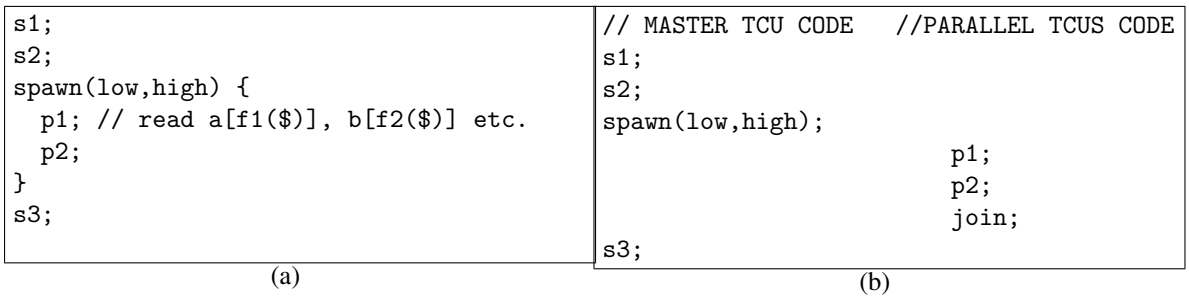

Figure 1: (a) XMTC Code provided by the programmer. `s1-s3` and `p1,p2` represent any (sequences of) XMTC statements. (b) Intermediate format used by the compiler, with outlined spawn block code and explicit `join` instructions added.

We introduce a spawn instruction with a slightly different semantics in the next section, then proceed to disccuss (i) prefetching for threads with $low \leq TID < low + 1024$ and (ii) prefetching for threads with $TID \geq low + 1024$ separately. The compiler should apply both optimizations to obtain best performance.

4.1 Non-blocking spawn `nb_spawn` and `Master_join`

When outlining is done, the end of the spawn block translates into two different types of "join": (i) a "parallel join" (which we will call from now on `p_join`) in the parallel code which tells a TCU that it has reached the end of a virtual thread and (ii) a "Master TCU join" (or `Master_join`) in the serial code which means "wait for all the parallel TCUs to be idle".

In the current implementation of the toolchain, the `Master_join` command is implicit, in that the semantic of the `spawn` command always implies a `Master_join` following immediately after. As described below, including an explicit `Master_join` in the serial code increases the clarity of the intermediate compiler representation and allows changing the semantics of the `spawn` command to enable prefetching.

In the current implementation, when the Master TCU encounters a `spawn` instruction, it passes control over to the special "Spawn" unit, and then blocks until this unit signals that the parallel section is over. We propose a slightly different spawn command, where after the Master TCU activates the "Spawn" unit, it continues executing instructions in serial mode instead of waiting. We call this new type of instruction `nb_spawn`, for *non-blocking spawn*. The compiler inserts a `Master_join` instruction at the point in the serial code where the Master TCU should stop and wait for the parallel TCUs to finish executing the parallel code.

We note that the original type of spawn can be easily emulated by using a `nb_spawn` immediately followed by a `Master_join` instruction. This is illustrated in Figure 2. The decision to keep one or both types of spawn instructions can be made by the compiler and hardware designers.

4.2 Prefetching for the first round of running threads

In this section we address prefetching issues which apply only to the first $min(high - low + 1, 1024)$ running threads in a parallel section. These threads share the property that their allocation to the TCUs is known at compile time.

It is very common that the first operations in a thread read data from one or more locations in array(s) located in the shared memory. To avoid having the threads stall for data, we can issue prefetch instructions ahead of time, in the serial code before the spawn command.

```
s1;
s2;
spawn(low,high) {
  p1; // read a[f1($)], b[f2($)] etc.
  p2;
}
s3;
```

(a)

```
// MASTER TCU CODE   //PARALLEL TCUS CODE
s1;
s2;
nb_spawn(low,high);
Master_join;              p1;
                          p2;
                          p_join;
s3;
```

(b)

Figure 2: (a) XMTC Code provided by the programmer. (b) Using `nb_spawn` and `Master_join` to emulate a regular spawn instruction.

A first approach is to create an additional parallel section and place it before the original spawn command. The parallel TCUs will execute this parallel section (called an "early spawn block") while the Master TCU continues executing serial code. An early spawn block will contain one or more prefetch instructions. These will be issued by all the TCUs before the code in the original spawn block is started. Figures 3a,b show two possible code transformations using early spawn which illustrate two different approaches for placing the prefetch code relative to the original spawn block code.

Figure 3c presents an alternative approach: using a "vector prefetch" instruction. Such an instruction would take as arguments a base pointer and a function to compute the offset of the memory location to be prefetched to each TCU.

A comparison of these three solutions is included below.

Combined early spawn (Figure 3a). This solution uses two new primitives, `signal_MTCU_done_serial` and `wait_MTCU_done_serial` which are used to make the parallel TCUs wait for master TCU to finish executing serial code before proceeding with actual thread executions. This is done to preserve the semantics of the original code.

Advantages: can add arbitrary code to prefetch block; can cache evaluated expressions to re-use in thread body (e.g. $f1(\$), f2(\$)$); implicitly does code prefetching by starting broadcasting of parallel code earlier.

Disadvantages: parallel section code has more instructions; all threads execute check and branch instruction.

Hardware support: non-blocking spawn, signal_all_TCUs / wait_for_MTCU

Separate early spawn. (Figure 3b). In this approach, first the early spawn block is broadcasted and starts executing, while the Master TCU continues running the serial code `s2`. When the Master TCU reaches the second spawn block, it starts broadcasting its code; the parallel TCUs start executing this code regardless if they have finished issuing all the early spawn block instructions. As an alternative implementation, we could insert a `Master_join` before the second spawn, but in this case we would delay executing the actual code until the slowest prefetch thread ends.

Advantages: doesn't add to parallel section code.

Disadvantages: two separate parallel sections to broadcast; without `Master_join`, cannot guarantee all instructions in prefetch block are issued before actual thread starts; with `Master_join`, threads have to wait for the slowest prefetch thread before starting; cannot do code prefetching because of the "early spawn" block.

Hardware support: non-blocking spawn, restarting threads when overwriting currently running parallel code.

Vector prefetch. (Figure 3c). A software implementation of this would be similar to the "separate early spawn" above. We only consider a hardware vector prefetch implementation here. This is only applicable if $f1$ and $f2$ are affine functions. In this case, the hardware computes the elements of A and B to prefetch, which are characterized by a start index, an end index and a constant stride.

Advantages: no change to parallel sections.

Disadvantages: complex hardware implementation.

Hardware support: hardware support for evaluating affine functions and initiating prefetch to all TCUs.

I believe that overall the "Combined early spawn" method is the most efficient solution and can be implemented using the least additional hardware. The "Separate early spawn" will suffer the cost of two broadcasting and join operations, which will likely be too expensive. The "Vector prefetch" is restricted to affine functions and has a complex hardware implementation.

4.3 Prefetching for the later threads

After running the first 1024 virtual threads one per TCU, the automatic thread allocation mechanism of XMT uses the parallel TCUs as they become available to run the rest of the virtual threads. Prefetching is different in this case because, unlike the first 1024 threads, the assignment of threads to TCUs is not compile-time predictable. Nevertheless, prefetching is needed for these "later" threads as well; otherwise, after being allocated to a TCU, each virtual thread would start by stalling for data.

Figure 4a shows a first solution, **Early get_next_id**: instead of waiting to finish executing the current virtual thread, a TCU could request in advance the ID of the next virtual thread to run. Using this TID, prefetch instructions can be inserted in the code towards the thread end.

An alternative solution that involves thread clustering is presented next: when the number of virtual threads is relatively large, the compiler can change the thread structure to create a more coarse-grained version of the spawn block. This can be accomplished by wrapping the original thread body within a loop and running $F = (high - low)/1024$ such threads sequentially within a larger ("clustered") thread. The compiler can now insert prefetch instructions in this clustering loop to prefetch the data for the $F-1$ "later" threads running on this TCU.

An immediate drawback of this coarse-graining method is that it adversely affects the automatic load-balancing mechanism of XMT: one of these "clustered" threads can take a long time to finish while most of the other TCUs are idle, potentially cancelling the benefits of prefetching. To avoid this, a mechanism called "hybrid clustering" has been proposed: use a smaller clustering factor $F < (high - low)/1024$, which will leave a number of shorter "singleton" (unclustered) threads; start by running the longer clustered threads, and towards the end of the parallel section switch to running the singleton threads, so as to fill in the gaps created by the coarse-grained threads. How to apply prefetching for the singleton threads is still an open question. [1]

The code transformation for the **hybrid thread clustering** is presented in Figure 4b. The clustering factor F and the number of clustered and singleton threads can be determined before the parallel section is started by analyzing factors such as the number of threads, instructions in a thread and the latencies of memory accesses.

We compare the two methods for applying prefetching for the "later" threads by briefly listing their advantages and disadvantages:

[1] One idea is to use the early `get_next_id` method, but then part of the motivation for doing clustering is lost.

Early `get_next_id`**.** Advantages: small code changes.

Disadvantages: can affect load balancing because of an early `get_next_id` decision; this effect is likely to be minor.

Hardware support: changes to current `get_id` mechanism. In the current implementation, `get_id` blocks if no threads are left to run, and keeps polling to see if additional threads are started using single-spawn. For `Early get_next_id`, if no threads left to run are found it should return a special value (e.g. zero) and continue execution until the `p_join`.

Thread clustering Advantages: reduces thread starting overheads and duplicate work; creates spatial locality at TCU.

Disadvantages: considerable runtime overhead; increases size of parallel section significantly; doesn't work with single-spawn as is; no prefetching for singleton threads; can create highly unbalanced execution if threads are unequal in length.

Hardware support: not required.

I believe that overall the "Early `get_next_id`" would be simpler to implement and get better performance and lower code size. Thread clustering has other merits and it should be implemented as well in a later version.

References

[BV06] A. O. Balkan and U. Vishkin. Programmer's manual for xmtc language, xmtc compiler and xmt simulator. Technical Report UMIACS-TR 2005-45, University of Maryland Institute for Advanced Computer Studies (UMIACS), February 2006.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, New York, NY, USA, 1991. ACM Press.

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[Coo02] Robert Neale Cooksey. *Content-sensitive data prefetching*. PhD thesis, University of Colorado, Boulder, 2002. Director-Dirk Grunwald.

[DDS95] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(7):733–746, 1995.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 354–368, New York, NY, USA, 1990. ACM Press.

[Gor95] Edward Gornish. *Adaptive and integrated data cache prefetching for shared-memory multiprocessors*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Champaign, IL, USA, 1995.

[JHL06] Natalie D. Enright Jerger, Eric L. Hill, and Mikko H. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177– 188, March 2006.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[KK97] Ando Ki and Alan E. Knowles. Adaptive data prefetching using cache information. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 204–212, New York, NY, USA, 1997. ACM Press.

[KL91] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 43–53, New York, NY, USA, 1991. ACM Press.

[KOH+98] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 485–496, New York, NY, USA, 1998. ACM Press.

[KV97] Sunil Kim and Alexander V. Veidenbaum. Stride-directed prefetching for secondary caches. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[LLG+00] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. pages 583–599, 2000.

[LM99] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Trans. Comput.*, 48(2):134–141, 1999.

[LRB01] Wi Fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing dram latencies with an integrated memory hierarchy design. *hpca*, 00:0301, 2001.

[Mci98] Nathaniel Mcintosh. *Compiler support for software prefetching*. PhD thesis, Rice University, May 1998. Adviser-Ken Kennedy.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not.*, 27(9):62–73, 1992.

[Mow98] Todd C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Trans. Comput. Syst.*, 16(1):55–92, 1998.

[NNTV01] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Evaluating the xmt programming model. In *Proceedings of the 6th Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 95–108, 2001.

[NNTV03] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *invited Special Issue for ACM-SPAA'01: TOCS 36,5*, pages 521–552, New York, NY, USA, 2003. Springer Verlag.

[PK94] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[RMS98] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, New York, NY, USA, 1998. ACM Press.

[TE95] Dean M. Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Trans. Comput. Syst.*, 13(1):57–88, 1995.

[VDBN98] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 140–151, New York, NY, USA, 1998. ACM Press.

[VL00] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[WBM+03] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 388–398, New York, NY, USA, 2003. ACM Press.

[WH03] Dan Wallin and Erik Hagersten. Miss penalty reduction using bundled capacity prefetching in multiprocessors. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 12.1, Washington, DC, USA, 2003. IEEE Computer Society.

[WV07] X. Wen and U. Vishkin. Pram-on-chip: First commitment to silicon. In preparation, 2007.

A Implementation Plan

In the implementation phase, all of the above prefetching techniques will be fully included in the XMTC compiler. This will allow the broader community of XMT users to achieve the full potential of the XMT architecture for their applications and their research.

The various prefetching-related compiler tasks together with their dependencies are listed in Figure 5.

```
// MASTER TCU CODE                      // PARALLEL TCUS CODE
s1;
nb_spawn(low,high);
s2;                                     if ($ - low < 1024) {
signal_MTCU_done_serial;                  prefetch a[f1($)], b[f2($)];
Master_join;                              wait_MTCU_done_serial;
                                        }
                                        p1;
                                        p2;
                                        p_join;
s3;
```

(a) "Combined" early spawn. Prefetch code is added to the beginning of the original spawn block code.

```
// MASTER TCU CODE                      // PARALLEL TCUS CODE
s1;
nb_spawn(low,low+min(1023,high));
s2;                                     prefetch a[f1($)], b[f2($)]
// Master_join; // optional!            p_join;
spawn(low,high);
                                        p1;
                                        p2;
                                        p_join;
s3;
```

(b) "Separate" early spawn. Prefetch code is placed in its own spawn block and is broadcasted separately.

```
// MASTER TCU CODE                      // PARALLEL TCUS CODE
s1;
vector_prefetch a,f1;
vector_prefetch b,f2;
s2;
spawn(low,high);
                                        p1;
                                        p2;
                                        p_join;
s3;
```

(c) Using vector prefetch commands.

Figure 3: The result of different code transformations implementing prefetching for the first 1024 threads.

```
// MASTER TCU CODE      // PARALLEL TCUS CODE
s1;
s2;
spawn(low,high);
                         p1;
                         temp = get_next_id();
                         if (temp>0)
                           prefetch a[f1(temp)], b[f2(temp)];
                         p2;
                         p_join;
s3;
```

(a) Using **Early** `get_next_id` to prefetch data for next virtual thread.

```
// MASTER TCU CODE                // PARALLEL TCUS CODE
s1;
s2;
compute Nclustered,Nsingle,F;
spawn(0,Nclustered+Nsingle-1);
                                  if ($<=Nclustered) { // clustered thread
                                    clustered = TRUE;
                                    for (temp=$*F;temp<$*F+F;temp++) {
                                  L1:
                                      p1(temp);
                                      if (clustered && temp<$*F+F-1)
                                        // prefetch for next iteration(s)
                                        prefetch a[f1(temp+1)],b[f2(temp+1)];
                                      p2(temp);
                                      if (!clustered)
                                        goto L2;
                                     } // end for
                                   }
                                   else {
                                     clustered = FALSE;
                                     goto L1;
                                   }
                                  L2:
                                  p_join;
s3;
```

(b) Using **hybrid thread clustering** to prefetch data for next virtual thread. To avoid duplicating the code for clustered and singleton threads, we use `goto` statements.

Figure 4: The result of different code transformations implementing prefetching for threads with $TID \geq 1024$.

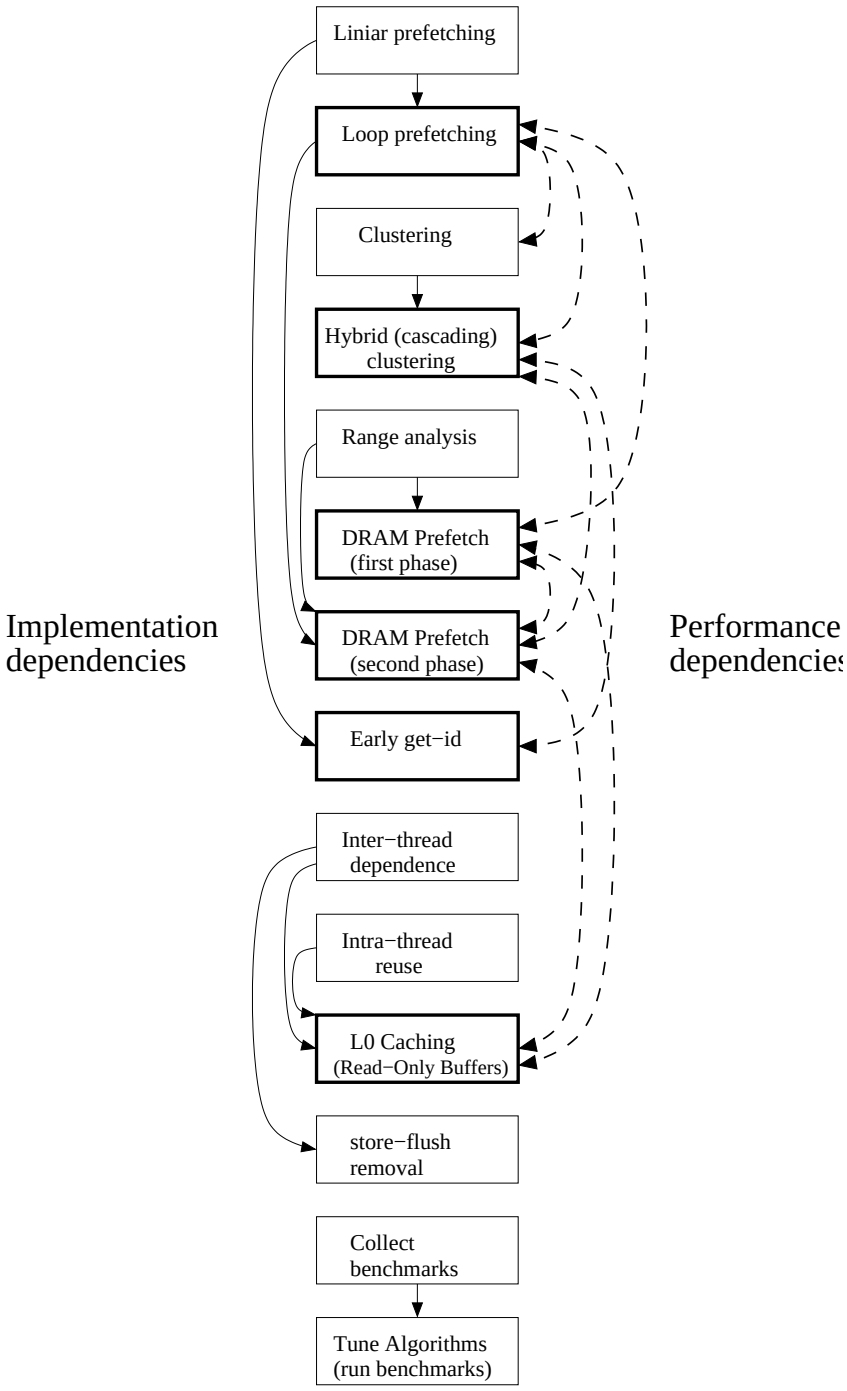

Figure 5: Dependencies between tasks in prefetching implementation. An *implementation dependency* (depicted on the left of the figure) $X \longrightarrow Y$ means X must be implemented before Y. A *performance dependency* (the right side of the figure) $X \longleftrightarrow Y$ means enabling X will affect the performance of Y and viceversa.