Resource-Aware Compiler Prefetching for Many-Cores

George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, Uzi Vishkin
University of Maryland, College Park
{*gcaragea,tzannes,keceli,barua,vishkin*}*@umd.edu*

***Abstract*—Super-scalar, out-of-order processors that can have tens of read and write requests in the execution window place significant demands on Memory Level Parallelism (MLP). Multi- and many-cores with shared parallel caches further increase MLP demand. Current cache hierarchies however have been unable to keep up with this trend, with modern designs allowing only 4-16 concurrent cache misses. This disconnect is exacerbated by recent highly parallel architectures (e.g. GPUs) where power and area per-core budget favor lighter cores with less resources.**

Support for hardware and software prefetch increase MLP pressure since these techniques overlap multiple memory requests with existing computation. In this paper, we propose and evaluate a novel Resource-Aware Prefetching (RAP) compiler algorithm that is aware of the number of simultaneous prefetches supported, and optimized for the same. We show that in situations where not enough resources are available to issue prefetch instructions for all references in a loop, it is more beneficial to decrease the prefetch distance and prefetch for as many references as possible, rather than use a fixed prefetched distance and skip prefetching for some references, as in current approaches.

We implemented our algorithm in a GCC-derived compiler and evaluated its performance using an emerging fine-grained many-core architecture. Our results show that the RAP algorithm outperforms a well-known loop prefetching algorithm by up to 40.15% and the state-of-the art GCC implementation by up to 34.79%. Moreover, we compare the RAP algorithm with a simple hardware prefetching mechanism, and show improvements of up to 24.61%.

***Keywords*-parallel architectures; optimizing compilers**

I. Introduction

Memory systems have been under a lot of pressure to keep up with the increasing demand for parallelism coming from every new generation of microprocessors. Super-scalar, out-of-order processors can have a large number of memory operations in flight in the execution window at one time (up to 48 load and 32 store operations for a Intel Pentium 4 processor, several of which can be cache misses). In simultaneous multi-threading (SMT) architectures, multicores and manycores, the demand on Memory-Level Parallelism (MLP) has further increased. This has put additional pressure for memory systems to support numerous concurrent memory requests.

Current cache hierarchy designs however have been unable to support this high level of demand for parallelism. Existing architectures employ lock-up free caches (e.g. [1]) to avoid stalling the CPU and allow the cache miss to be serviced in the background. Fig. 1 depicts a cache system and its attached Miss Handling Architectures (MHA). This consists of several *Miss Information/Status Holding Register* (MSHR) Files, and is responsible for keeping track of the outstanding concurrent misses. To meet the demand for high bandwidth and low latency, each MSHR has its own comparator, and the MSHR file can be described as a small *fully associative cache.* The maximum number of outstanding cache misses the system supports is limited by the number of MSHRs.

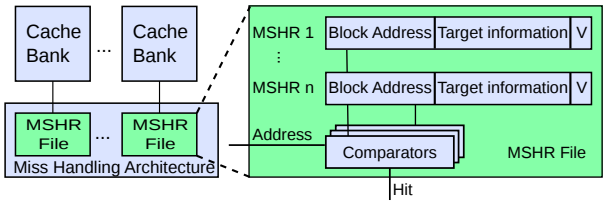


Figure 1. Miss Handling Architecture (MHA) for a banked cache system and the Miss Information/Status Holding Register (MSHR) file

Fully associative caches (and hence MHAs) are very costly in terms of chip area, but most importantly power usage: when a new request is received, all the comparators must be activated in parallel in order to retrieve the corresponding entry in one clock cycle. This severely limits the size of the MSHR file that can be included, even for today's large transistor budgets. For example, the L1 cache of an Intel Pentium 4 processor supports only 8 outstanding misses. For more recent AMD Opteron and Intel Core i7 architectures, an empirical study showed that single thread performance does not improve past 7 concurrent memory requests, proving the same limitation holds [2].

Currently, the two predominant paradigms of how single-chip processors are built are: (i) Limited-scale multi-cores that replicate the single-processor model on one die and strive to maintain backwards compatibility. They generally target applications with low degrees of parallelism, programmed to take advantage of local caches and limit expensive inter-core communication. Presently such systems have 2-12 cores, each supporting coarse grained threads and they are not expected to exceed a few tens of cores in the foreseeable future. And (ii) Many-cores that are not typically confined to traditional architectures and programming models, and use tens to hundreds of lightweight cores

in order to provide stronger speedups. The best known representatives of this class are GPU architectures from NVIDIA and AMD, which are now increasingly used for general-purpose computation. Other examples include the Sun Niagara 2 and Intel Larabee. Research machines such as UT Austin's TRIPS, MIT's RAW and UMD's XMT are also examples of single-chip many-cores.

Many-cores are motivated by a variety of factors such as high scalability to a large number of lightweight cores in a single chip, very low inter-core communication latencies, high on-chip communication bandwidth, overcoming wire-length restrictions at high clock speeds, and fast hardware-assisted inter-core synchronization. Many have demonstrated great results, far exceeding the performance of traditional multi-cores for the same silicon area. Such designs are an exciting new frontier in computer architecture, with a bright future as evidenced by commercial and research interest.

Regardless of the type or motivation, in all many-cores each lightweight core has much smaller area than a traditional core. In terms of prefetching resources, this has two repercussions: (i) the size (number of entries) of MSHR files is much more constrained in a many-core due to area constraints; and (ii) the total energy consumption of all MSHRs across cores in a processor is much higher than in a traditional processor given the larger number of cores, which further limits the size of MSHRs. As a consequence it becomes a crucial to carefully manage the use of scarce MSHR resources for these architectures. We present a new prefetching method called Resource-Aware Prefetching (RAP) to manage MSHR resources carefully. It is mainly beneficial in many-cores because of their limited number of MSHR. It can apply to traditional multi-cores as well, but given their larger number of MSHR entries and their much larger area and power budget for other latency tolerating tecniques, the run-time gains from RAP are likely to be very small, so we do not discuss those further.

To evaluate the RAP algorithm, we are using XMT[1] – a general-purpose manycore architecture [3]. A recent study showed that when configured to use the same chip area, XMT can outperform both an Intel Core 2 (speedups up to 13.83x [4]), AMD Opteron (speedups up to 8.56x [5]) and also an NVIDIA GTX280 GPU (speedups of up to 8.10x [6]). The XMT hardware design is simple and scalable, which allowed us to easily understand and tune architectural parameters. An open-source, highly configurable cycle-accurate simulator and compiler are available, allowing us to implement and evaluate our optimizations on different hardware configurations. An overview of the XMT platform is presented in Section IV. An alternate evaluation platform would have been GPUs from NVIDIA or AMD; however they are not used because in the current generation there is no support for software prefetching in these platforms. Moreover, the vendors do not disclose architecture or compiler details, making it difficult to implement and evaluate hardware or software improvements.

[1]This refers to the XMT architecture developed at University of Maryland and not the Cray XMT system.

II. Existing Software Prefetching Methods

Mowry et. al [7] introduced a compiler algorithm to insert prefetch instructions into scientific applications that operate on dense matrices. Consider the code in Fig. 2 as our running example. Fig. 2(a) shows the original program code. In the figure, assume that the matrices A, B and C contain double precision floating point elements (64 bits) and our hypothetical system has a cache line of 16 bytes; thus two doubles fit per cache line. Also, assume that a cache miss latency is $MissLatency = 50$ clock cycles. Note that this simplified model, assuming only one level of cache and a fixed cache miss latency, is widely used in prefetching literature; accurately modeling the cache memory hierarchy in the compiler is often too complex to be viable. The algorithm proceeds as follows:

Algorithm 1 *Mowry's Loop Prefetching*

I. *For each static affine array reference, use locality analysis to determine which dynamic accesses are likely to suffer cache misses and therefore should be prefetched. For the code in Fig. 2(a), one cache line can hold two array elements, and thus every second dynamic access for the* `A[i]`, `B[i]` *and* `C[i]` *references will be a cache miss and requires a prefetch instruction.*

II. *Isolate the predicted dynamic miss instances using loop-splitting techniques such as peeling, unrolling, and strip-mining. This avoids the overhead of adding conditional statements for prefetching to the loop bodies. This yields the code in Fig. 2(b), where the loop has been unrolled two-fold and the last 6 iterations have been pulled out in a separate loop.*

III. *Schedule prefetches the proper amount of time in advance using software pipelining (by using the computed necessary prefetch distance), where the computation of one or more iterations is overlapped with prefetches for a future iteration. The prefetch distance is computed so that all latency can be hidden completely, using the formula:*

$$PrefDistance = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil \quad (1)$$

IterationTime is the estimated running time of the shortest path through the loop when software prefetching is enabled. Assume for example that $IterationTime = 20$ *clock cycles (after unrolling), and thus* $PrefDistance = \lceil 50/20 \rceil = 3$ *iterations. The code in Fig. 2(c) contains the transformed code, where prefetches for the references to the A, B and C arrays have been inserted three iterations in advance.*

Mowry's algorithm as presented successfully filters out most unnecessary prefetch instructions and significantly

(a)
```
for (i=0;i<1000;i++)
  A[i] = B[i] + C[i];
```

(b)
```
for (i=0;i<994;i += 2 ) { /* Unrolled */
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
/* Last three iterations peeled */
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(c)
```
for (i=0;i<994;i += 2 ) {
  /* prefetch 3 iterations in advance */
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  prefetch(C[i+6]);
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(d)
```
for (i=0;i<994;i += 2 ) {
  prefetch(A[i+6]);
  prefetch(B[i+6]);
  /* Does not prefetch C */
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
for (i=994;i<1000;i++)
  A[i] = B[i] + C[i];
```

(e)
```
for (i=0;i<996;i += 2 ) {
  /* prefetch 2 iterations in advance */
  prefetch(A[i+4]);
  prefetch(B[i+4]);
  prefetch(C[i+4]);
  A[i] = B[i] + C[i] ;
  A[i+1] = B[i+1] + C[i+1];
}
/* Last two iterations peeled */
for (i=996;i<1000;i++)
  A[i] = B[i] + C[i];
```

Figure 2. (a) Original code before loop prefetching (b) Loop unrolling and peeling to isolate likely cache misses (c) Code after Mowry's prefetching algorithm (*PrefDistance* $= 3$) (d) Code after applying GCC loop prefetching algorithm (prefetch slots=6) (e) Outcome of the RAP algorithm: *PrefDistance* lowered to 2.

reduces the instruction overheads. However, it does not take into consideration the number of in-flight memory requests supported by the hardware. The maximum number of prefetch requests active at any time can be computed using:

$$MaxRequests = NumRefs \times PrefDistance \quad (2)$$

where *NumRefs* represents the number of static references that require prefetching. Going back to the code in Fig. 2(c), we have $MaxRequests = 3 \times 3 = 9$. Suppose that our architecture has 6 registers in the MSHR file. After the first six prefetch requests have been issued, when the next request arrives at the MHA unit, one of the following can happen, depending on the hardware implementation: **(i)** The additional request is silently dropped, and nothing is sent to the lower levels of the memory hierarchy. This causes the program to slow down, since it incurs all the instruction overheads of prefetching, but none of the benefits – the cache miss was not avoided. **(ii)** The MHA does not accept the prefetch request, stalling the issuing CPU until one MSHR becomes available. Stalling the CPU was exactly what prefetching was aiming to avoid, and thus the benefits of prefetching are again lost, leaving only the overheads.

GCC (GNU Compiler Collection), a state-of-the art open source compiler which supports a wide range of architectures and programming languages, includes an implementation of Mowry's algorithm for loop prefetching. The GCC algorithm extends if further by introducing the notion of a platform-specific number of *PrefetchSlots*. This is used to limit the number of prefetches that can be in flight at the same time. As far as we know, GCC's method is the only software prefetching algorithm that attempts to limit the number of in-flight prefetches based on hardware limitations. After performing the same steps 1-2 as above, the GCC algorithm starts scheduling prefetches for all the references in program order. One prefetch instruction issued *PrefDistance* iterations in advance of the reference causes the number of available prefetch slots to be decremented by *PrefDistance*. Once not enough *PrefetchSlots* are left, it stops issuing prefetches for the remaining references.

For our running example, Fig. 2(d) shows the outcome of the GCC algorithm. Since the prefetch instructions for the `A[i]` and `B[i]` references use up all 6 available prefetch slots, no prefetch is issued for the `C[i]` reference. At runtime, this means a cache miss penalty will be encountered every iteration of the unrolled loop, significantly affecting its running time.

III. New RAP Prefetching Method

Intuition. Our main contribution is a new compiler prefetching algorithm – Resource-Aware Prefetching (RAP) – which improves upon Mowry's standard loop prefetching algorithm as well as the GCC implementation by using the very limited MHA resources more efficiently. Our algorithm robustly adapts to constrained resources and uses them to hide as much latency as possible. More concretely, we show that in situations where not enough prefetch slots are available to issue prefetch instructions for all references, it is more beneficial to decrease the prefetch distance and prefetch for as many references as possible. By contrast, the GCC implementation uses a fixed prefetch distance and may prefetch fewer references.

Fig. 2(e) shows the outcome of the RAP algorithm applied to our example code. The prefetch distance has been lowered to two iterations, which allowed prefetches to be issued for all three references. With this transformation, there will be only *one cache miss per three iterations*: once a cache miss is encountered, it gives enough time for all previously issued prefetch requests to complete, including current and next two iterations. By contrast, the GCC implementation encounters one miss per each iteration, which translates to three times more time spent in memory stalls.

Implementation. To formulate an algorithm for RAP, it is useful to understand the limitations of GCC's prefetcher. There is a subtle inconsistency in the way GCC schedules prefetching instructions: on one hand, the prefetch distance is computed assuming all memory latencies can be hidden through prefetching; on the other hand, under certain conditions, prefetch instructions for some references are not even issued, causing some references to be cache misses. This affects the iteration time, and therefore the prefetch distance should be adjusted accordingly: if each iteration takes longer, then prefetches can be issued fewer iterations in advance and still be able to hide the latency. However, GCC does not adjust the prefetch distance in these cases, *effectively using a flawed model for scheduling prefetches.*

Figure 2(d) shows an example of the suboptimal scheduling algorithm described above. To help understand the runtime behavior, we show the resulting dynamic cache trace in Figure 3(a). The first three iterations are not prefetched for, hence all references are cache misses. At each iteration from $i = 6$ onward, the read from `C[i]` is going to be a cache miss, which on our hypothetical architecture takes 50 clock cycles. This is 49 cycles more than in the original estimate, and thus *IterTime* $= 20 + 49 = 69$. Using Equation (1), we only need *PrefDistance* $= \lceil 50/69 \rceil = 1$ iteration in advance. However, GCC schedules prefetches using *PrefDistance* $= 3$ iterations in advance, according to the original calculation.

Let us examine an alternative scheduling algorithm in which a smaller *PrefetchDistance* is used. The RAP algorithm discussed in the rest of this paper is based on this scheme. If we use *PrefDistance* $= 1$ iteration instead of 3, we can now issue prefetches for all three references, using a total of *MaxRequests* $= 3 \times 1 = 3$ prefetch slots. The cache trace for this case is shown in Figure 3(b). When $i = 0$, we issue prefetch requests for `A[2]`, `B[2]` and `C[2]`, then we encounter three cache misses for `A[0]`, `B[0]` and `C[0]`. For $i = 2$, we start by issuing prefetches for iteration $i + 2 = 4$, then all references are cache hits, because the prefetch requests issued at the beginning of iteration $i = 0$ overlapped with the previous misses and have had time to complete (see Figure 3(b)). For $i = 4$, we have a cache miss for `A[4]`, but that gives enough time for the prefetches for `B[4]` and `C[4]` to complete, and thus they become cache hits. The cache miss for `A[4]` also gave enough time for all prefetches for iteration $i = 6$ to complete, meaning we have three cache hits in that iteration. The execution enters a *steady state* at this point, with one cache miss every other iteration, until the end of the loop.

Similarly, we can also use *PrefDistance* $= 2$, which yields the code in Fig. 2(e) and the trace in Fig. 3(c). Following a similar reasoning, we observe that in the steady state we encounter one miss every 3 iterations, leading to:

Claim 1 *Let* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *the prefetch distance computed by Mowry's algorithm (and also GCC).*

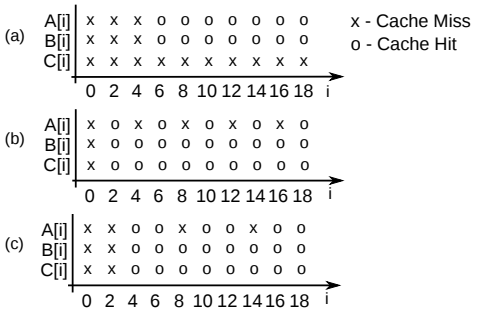


Figure 3. Dynamic cache trace for the code in Figure 2. (a) GCC loop prefetching with *PrefDistance* $= 3$ and (b) RAP algorithm with *PrefDistance* $= 1$ (c) RAP algorithm with *PrefDistance* $= 2$.

For any prefetch distance $PD_{RAP} < PD_{Mowry}$ *and*

$$PD_{RAP} \times \textit{NumRefs} \leq \textit{PrefetchSlots} \tag{3}$$

we can issue prefetch instructions PD_{RAP} *iterations in advance for all references without exceeding the available PrefetchSlots (number of MSHR entries), and this will result in exactly one cache miss per* $PD_{RAP} + 1$ *iterations in the steady state.*

The claim can be easily verified: once a cache miss has been encountered, it allows enough time for all the prefetch requests already issued for the next PD_{RAP} iterations to complete, ensuring they are all hits. However, since PD_{RAP} iterations with all hits do not provide enough time to hide the miss latency, iteration $PD_{RAP} + 1$ encounters a cache miss for the first read. The cycle then repeats.

Using Claim 1, we can compute the average loop iteration time in the steady state when $PD_{RAP} < PD_{Mowry}$:

$$\textit{AvgIterTime} = \textit{IterHit} + \frac{\textit{IterMiss} - \textit{IterHit}}{PD_{RAP} + 1} \tag{4}$$

where *IterHit* is the iteration time when all references are hits (20 cycles in our example) and *IterMiss* is the iteration time with one cache miss (69 for our example).

The average iteration time (4) is a strictly decreasing function of the prefetch distance PD_{RAP}. To minimize the overall execution time, we use the upper bound

$$PD_{RAP} = \left\lfloor \frac{\textit{PrefetchSlots}}{\textit{NumRefs}} \right\rfloor \tag{5}$$

given by (3). In the example in Figure 2(e), we have $PD_{RAP} = \lfloor 6/3 \rfloor = 2$. We can now present our improved compiler algorithm:

Algorithm 2 *Resource-Aware Prefetching*

I-II. *Identical to Steps* ***I-II*** *in Algorithm 1.*

III. *Compute* $PD_{Mowry} = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil$ *and NumRef the number of references. Let* $PD_{RAP} = \left\lfloor \frac{PrefetchSlots}{NumRefs} \right\rfloor$.

III.1 *If* $PD_{Mowry} \times \textit{NumRefs} \leq \textit{PrefetchSlots}$*, schedule prefetch instructions for all NumRef references* PD_{Mowry} *iterations in advance.*

III.2 *If* $PD_{Mowry} \times \textit{NumRefs} > \textit{PrefetchSlots}$ *and* $PD_{RAP} \geq 1$*, schedule prefetch instructions for all*

NumRef references PD_{RAP} iterations in advance.

III.3 *If $PD_{Mowry} \times$ NumRefs $>$ PrefetchSlots and $PD_{RAP} = 0$, schedule prefetch instructions for the first PrefetchSlots references in program order exactly **one** iteration in advance.*

Case III.1 corresponds to the non-resource restricted situation, where we fall back on the same scheduling algorithm as Mowry's (and GCC) algorithm. Case III.2 occurs in situations when there are not enough *PrefetchSlots* to completely hide all cache misses; the algorithm issues one prefetch for each reference using a smaller prefetch distance, resulting in one cache miss every PD_{RAP+1} iterations. Case III.3 occurs in severely resource-constrained cases, where we have more static references than *PrefetchSlots*. The algorithm issues prefetch instructions one iteration ahead to as many references as possible, without exceeding *PrefetchSlots*.

Thread Clustering. Loop prefetching does not naturally apply to all types of workloads and data structures. However, given the nature of fine-grained parallel code – short threads, high degree of parallelism – prefetching can be enabled for some benchmarks by inserting several short thread bodies in a loop within a coarser thread. This compiler technique, called *thread clustering* [3] effectively enabled the use of prefetching for all of our benchmarks.

IV. The XMT Framework

In Section I we argue that the constraint on the amount of Memory-Level Parallelism is a major limitation for many-core architectures. For evaluation purposes, we chose the XMT architecture as a representative lightweight-core platform. Recent benchmarking efforts have shown that XMT can achieve consistent performance improvements when compared to modern architectures [4], [5], [6], while using a straightforward scalable design and an easy-to-program interface.

The primary goal of the eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture (e.g. [3]) is improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available. It is meant to leverage the vast body of knowledge, known as Parallel Random Access Model (PRAM) algorithmics, and the latent, though not widespread, familiarity with it. A 64-core FPGA prototype was reported and evaluated in [5].

The XMT architecture, depicted in Fig. 4, includes an array of lightweight cores, Thread Control Units (TCUs) and a serial core with its own cache (Master TCU). The processor includes several clusters of TCUs connected by a high-throughput mesh-of-trees (MOT) interconnection network [8]; an instruction and data broadcast mechanism; a global register file (GRF); a prefix-sum unit (PS). The first level of cache is shared and partitioned into mutually-exclusive cache modules sharing several off-chip DDR2

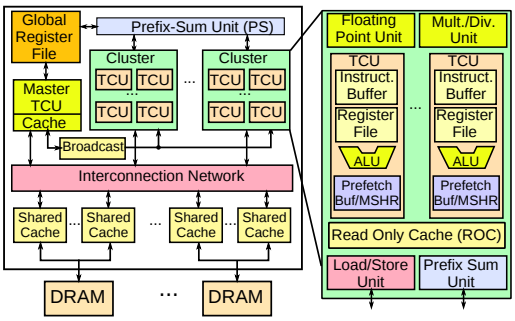
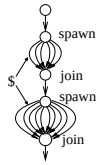


Figure 4. XMT architecture overview.

```
int A[N],B[N],base=0;
spawn(0,N-1) {
  int inc=1;
  if (A[$]!=0) {
    ps(inc,base);
    B[inc]=A[$];
} }
```

(a)



(b)

Figure 5. (a) XMTC program example: Array Compaction. (b) Execution of a sequence of `spawn` and `join`.

DRAM memory channels. The TCU Load-Store unit applies a hashing function on each address to avoid memory hotspots. Cache modules handle concurrent requests and provide buffering and request reordering to achieve better DRAM bandwidth utilization. Within a cluster, a compiler-managed Read-Only Cache (ROC) is used to store constant values across all threads. TCUs include lightweight ALUs, but the more expensive units are shared by all TCUs in a cluster.

The underlying programming model of the XMT framework is an arbitrary CRCW (concurrent read/write) SPMD (single program, multiple data) with serial and parallel execution modes. The *spawn* and *join* instructions specify the beginning and the end of a parallel section that contains an arbitrary number of virtual threads sharing the same code, as shown in Fig. 5. An algorithm designed in the XMT model usually permits each thread to progress at its own speed from its initiating spawn to the terminating join, without ever having to busy-wait for other threads, methodology called "independence of order semantics (IOS)." XMT also includes a hardware implementation of a powerful prefix-sum primitive similar in function to the NYU Ultracomputer Fetch-and-Add; it provides constant, low overhead inter-thread coordination, a key requirement for implementing efficient intra-task parallelism. Fig. 5(a) illustrates the XMTC programming language, a simple SPMD extension of C. The example shows how it can be used to assign a unique index in array `B` when compacting an array `A`. The non-zero elements of array A are copied into an array B. The order is not necessarily preserved. After the execution of the prefix-sum statement `ps(inc,base)`, the `base` variable is increased by `inc` and the `inc` variable gets the original

value of `base`, as an **atomic** operation.

XMT allows concurrent instantiation of as many threads as the number of available processors. Threads are efficiently started and distributed thanks to the use of prefix-sum for fast dynamic allocation of work and a dedicated instruction broadcast bus. The high-bandwidth interconnection network and the low-overhead creation of many threads facilitate effective support of fine-grained parallelism.

Ease-of-programming is a necessary condition for the success of a many-core platform, and it is one of the main objectives of XMT. Indications that XMT is an easy-to-program efficient parallel architecture, include: (i) XMT is *based on a rich algorithmic theory* (PRAM) that provides a solid framework for designing and analyzing algorithms, equivalent to the serial model; (ii) the *ease of teaching of XMT programming as an adoption benchmark* has been established in repeated instances, from middle-school and up, and by independent education experts [9], and shown to be superior to alternative parallel approaches such as MPI, OpenMP and CUDA; (iii) XMT provides a *programmer's workflow* for deriving efficient programs from PRAM algorithms, and reasoning about their execution time [10] and correctness, and (iv) in a semester-long study supported through the DARPA HPCS program, the *development time* of XMT was, not surprisingly, shown to be about half that of MPI under circumstances favoring MPI [11].

Prefetch support. In the XMT design in Fig. 4, the prefetch buffer unit represents the Miss Handling Architecture (MHA) at the TCU level, consisting of one MSHR file per TCU. Each MSHR file contains a number of MSHR entries. The RAP algorithm is applied at the XMT thread level.

V. Experimental Evaluation

Simulated configuration. We are using XMTSim, a configurable, event driven cycle-accurate simulator of the XMT architecture. XMTSim timing is accurately modeled after the 64-core FPGA implementation, but it can be customized to realistically simulate any configuration, beyond the resource limitations of the FPGA prototype. At this time, off-chip buses and DRAM modules are modeled as fixed latency components in the simulator. The XMT compiler and simulation environment are publicly available [12].

The simulated configuration consists of 64 cores (TCUs) grouped in 8 clusters, with 256KB of shared on-chip cache and 4 DDR2 DRAM channels. The TCU MSHR file size varies between 1 and 12 entries.

Compiler Infrastructure. We used the GNU C compiler (GCC) 4.0.2 as the base compiler for our infrastructure. We adapted and improved upon the loop prefetching optimization pass targeted at the code executed by TCUs while in parallel mode. The loop prefetching algorithm operates using the Tree-SSA framework of GCC 4.0+.

Benchmarks. Table I describes the benchmarks used for evaluating the compiler algorithm performance. The

Table I
Benchmarks used.

Name	Description	Input	*MR* [a]
jacobi	2D PDE solver kernel	1024x1024	12
lu	LU factorization	256x256	12
conv	Image convolution	128x128	12
separ	Separable image filtering	512x256	8
dbscan	SQL Non-indexed Select query	2M records	6
matmult	Dense matrix multiplication	256x256	12
SpMV	Sparse matrix - vector mult.	4M values	9
treeadd	Summation of binary tree nodes	1M values	6

[a]Maximum number of simultaneous prefetch requests required when using loop prefetching

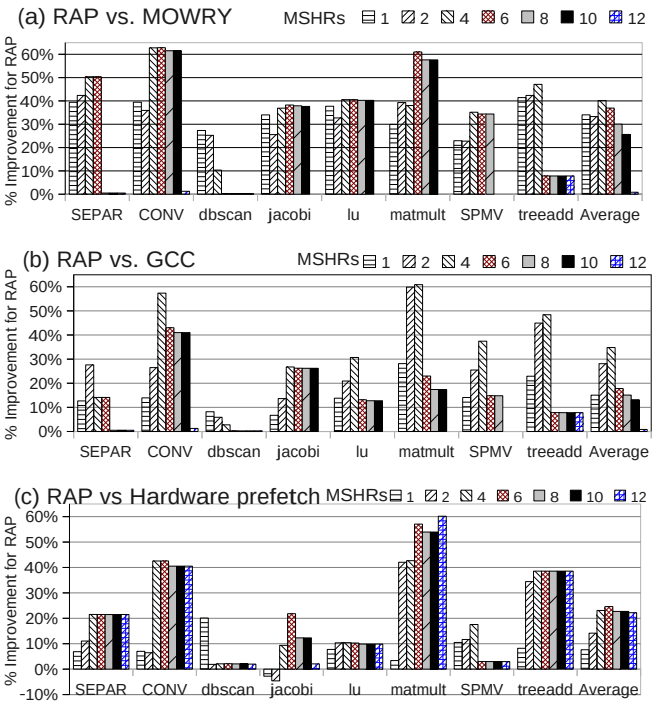


Figure 6. Performance improvement of RAP compared to (a) Mowry, (b) GCC and (c) OBL hardware prefetching

benchmarks are written in XMTC and were chosen from a variety of domains to reflect various access patterns and application types. Our goal in collecting the benchmarks was to sample as many application domains as possible, and as a result we have both integer and floating point kernels from scientific computing, image processing, databases and linear algebra.

Evaluation of Compiler Algorithm. To determine the effectiveness of the RAP algorithm we set out to execute our benchmarks on a series of configurations. For each benchmark, we computed the performance improvement of the RAP algorithm versus both Mowry's original algorithm and the GCC implementation when using configurations with 1, 2, 4, 6, 8, 10 and 12 MSHR file capacity.

The improvements of RAP over Mowry's algorithm and

the GCC implementation are shown in Figs. 6(a) and 6(b). As we see from the two figures, the average run-time improvement from our compiler algorithm ranges from 25.63% to 40.15% when compared to Mowry's algorithm, and from 13.18% to 34.79% when compared to GCC.

Fig. 6(b) shows the comparison of the RAP algorithm with the GCC implementation of loop prefetching. For each configuration, we provide GCC with the exact size of the MSHR file as the number of *PrefetchSlots*. When not enough *PrefetchSlots* are available to hide all latencies, the GCC algorithm does not issue prefetch instructions for some of the references. By contrast, the RAP algorithm decreases the prefetch distance, and issues as many prefetch instructions as the MSHR has capacity. This allows it to hide more of the memory latencies, and to outperform GCC.

Note that on XMT, we are prefetching from the shared L1 cache to the TCUs. The latency for an L1 access is ≈ 24 cycle in the current configuration. Given this latency, the prefetch distance is usually small (1-3 iterations), and thus the *MaxRequest* value for the benchmarks ranges between 6 and 12 for our benchmarks, as shown in Table I. Therefore, for MSHR files with 12 entries or larger, we are not in a resource-constrained regime, and there are no advantages for using the RAP algorithm over the alternative algorithms. However, for the class of highly parallel architectures we are targeting, even a per-core MSHR of 12 entries represents a significant amount of area. Future manycore architectures will probably devote even fewer resources for the MHA, making the RAP algorithm highly relevant.

Comparison with Hardware Prefetching. We compare the RAP software prefetching algorithm with an implementation of XMT that includes a hardware prefetching mechanism. Traditional single- and multi-core processors include sophisticated hardware prefetching units, capable of monitoring and distinguishing multiple independent streams of requests and identifying large access strides. However, the hardware complexities of such units make them prohibitively expensive per-core for a many-core architecture. Only a simple hardware prefetcher, that requires minimal hardware additions, could be considered. A well known such technique is One-Block-Lookahead (OBL, e.g. [13]), which prefetches the *next cache line* once a particular line is first read.

We implemented this scheme in the XMT Simulator. Since TCUs have no regular caches (to avoid coherence costs and area constraints), we prefetch at the granularity of one word, instead of one cache line: once a read request for address x is issued, a prefetch request for address $x + 4$ is automatically generated. The results in Fig.6(c) show that the software RAP prefetching algorithm outperforms the OBL hardware scheme by 7.64% to 24.61% on average. This strengthens the case that given the severe per-core limitations present in many-cores, least resource-intensive latency hiding techniques such as software prefetching offer the best performance.

VI. Related Work

Prefetching is a widely studied technique used to hide the increasingly high latencies (in terms of clock cycles) of memory accesses in modern architectures. Software prefetching [7], [14], [15] relies on the existence of non-blocking prefetch instructions and is usually enabled by the compiler. In hardware prefetching (e.g. [13], [16], [17]) a specialized hardware unit infers prefetching opportunities by monitoring run-time behavior. Prefetching schemes for parallel architectures in both software [14], [18], [19] and hardware (e.g. [20]) build upon uni-processor prefetching by taking into consideration issues caused by sharing of data and resources, such as coherence traffic and overheads.

Several studies have considered the interaction of the architectural parameters with the performance of software prefetching algorithms. In his comprehensive work on software data prefetching, Mowry [21] explores the effect on execution time of varying the number of outstanding prefetch requests that can be handled simultaneously by the hardware. In follow-up work, Mowry [19], as well as McIntosh [14], settle for a fixed-size prefetch issue buffer of 16 locations. Several other papers study the effects of changing the size of the prefetch destination (either cache or dedicated prefetch buffers) for systems with software [15], [22] or hardware prefetching [16]. However, unlike our approach, in all of these existing schemes the prefetch algorithm is unaware of the prefetch hardware configuration, and does not adapt its behavior.

GCC is the only attempt to consider the amount of prefetch resources available as part of the loop prefetching algorithm. As described in Section I, the GCC algorithm limits the number of memory references prefetched to meet a fixed upper bound. However, no guidance is given on how to chose this upper bound, as it is not clear what the underlying hardware limitation is accounted for. Yang et. al [23] empirically set the maximum number of prefetch instructions issued by the GCC compiler for the IA64 platform to 12. However, their study does not address the underlying limitations of the GCC algorithm discussed above. In our approach, we identify the hardware resource dictating the maximum number of prefetch requests allowed (the MSHR file), and provide an original scheduling algorithm which limits the prefetch distance instead of the number of references prefetched, and show that it outperforms both Mowry's and GCC's implementations.

VII. Conclusion

We presented RAP – an improved compiler loop prefetching algorithm targeted at many-core architectures, and showed that under resource constrained scenarios it outperforms Mowry's well known loop prefetching algorithm by up to 40.15%, the GCC improved implementation by up to 34.79% and a simple hardware prefetching scheme by up to

24.61%. The RAP algorithm is robust, providing considerable improvements and never falling behind significantly on any of the hardware configurations tested, making it a timely and necessary addition to any compiler targeting many-core architectures.

REFERENCES

[1] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 409–422.

[2] A. Porterfield, R. Fowler, A. Mandel, and M. Y. Lim, "Empirical evaluation of multi-socket, multi-core memory concurrency," Renaissance Computing Institute, Tech. Rep. RENCI TR-09-01, January 2009. [Online]. Available: http://www.renci.org/publications/techreports/TR-09-01.pdf

[3] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Towards a first vertical prototyping of an extremely fine-grained parallel programming approach," in *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2001, pp. 93–102.

[4] G. C. Caragea, A. B. Saybasili, X. Wen, and U. Vishkin, "Brief announcement: performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2009, pp. 163–165.

[5] X. Wen and U. Vishkin, "Fpga-based prototype of a pram-on-chip processor," in *CF '08: Proceedings of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 55–66.

[6] G. C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin, "General-purpose vs. gpu: Comparison of many-cores on irregular workloads," in *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX, June 2010.

[7] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Not.*, vol. 27, no. 9, pp. 62–73, 1992.

[8] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin, "Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing," *hoti*, pp. 21–28, 2007.

[9] S. Torbert, U. Vishkin, R. Tzur, and D. Ellison, "Is teaching parallel algorithmic thinking to high-school student possible? one teacher's experience," in *Proc. 41st ACM Technical Symposium on Computer Science Education (SIG CSE)*, Milwaukee, WI, March 2010.

[10] U. Vishkin, G. C. Caragea, and B. C. Lee, *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007, ch. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform.

[11] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A pilot study to compare programming effort for two parallel programming models," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1920 – 1930, 2008.

[12] "Software release of the explicit multi-threading (xmt) programming environment," http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html, August 2008.

[13] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[14] N. McIntosh, "Compiler support for software prefetching," Ph.D. dissertation, Rice University, 1998, adviser-Ken Kennedy.

[15] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. M. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*. ACM Press, 1991.

[16] N. 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*, 1990.

[17] W. F. Lin, S. K. Reinhardt, and D. Burger, "Reducing dram latencies with an integrated memory hierarchy design," *hpca*, vol. 00, p. 0301, 2001.

[18] D. M. Tullsen and S. J. Eggers, "Effective cache prefetching on bus-based multiprocessors," *ACM Trans. Comput. Syst.*, vol. 13, no. 1, pp. 57–88, 1995.

[19] T. C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Trans. Comput. Syst.*, vol. 16, no. 1, pp. 55–92, 1998.

[20] F. Dahlgren, M. Dubois, and P. Stenström, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 7, pp. 733–746, 1995.

[21] T. C. Mowry, "Tolerating latency through software-controlled data prefetching," Ph.D. dissertation, Stanford, CA, USA, 1995.

[22] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*. ACM Press, 1991, pp. 43–53.

[23] C. Yang, X. Yang, and J. Xue, *Advances in Computer Systems Archiecture*. Springer-Verlag, 2005, vol. 3740/2005, ch. Improving the Performance of GCC by Exploiting IA-64 Architectural Features, pp. 236–251.