Resource-Aware Compiler Prefetching for Fine-Grained Many-Cores

George C. Caragea[1], Alexandros Tzannes[1], Fuat Keceli[2], Rajeev Barua[2,3], and Uzi Vishkin[2,4]

[1] Department of Computer Science, University of Maryland
[2] Department of Electrical and Computer Engineering, University of Maryland
[3] Institute for Systems Research, University of Maryland
[4] University of Maryland Institute for Advanced Computer Studies

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 numerous lighter cores with less resources, further reducing support for MLP on a per-core basis.
Support for hardware and software prefetch increases 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 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% in run-time on average across benchmarks and the state-of-the art GCC implementation by up to 34.79%, depending upon hardware configuration. Moreover, we compare the RAP algorithm with a simple hardware prefetching mechanism, and show run-time improvements of up to 24.61%.
To demonstrate the robustness of our approach, we conduct a design-space exploration (DSE) for the considered target architecture by varying *(i)* the amount of chip resources designated for per-core prefetch storage and *(ii)* off-chip bandwidth. We show that the RAP algorithm is robust in that it improves performance across all design points considered. We also identify the Pareto-optimal hardware-software configuration which delivers 53.66% run-time improvement on average while using only 5.47% more chip area than the bare-bones design.

1 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. In simultaneous multithreading (SMT) architectures, as well as multicores and manycores, the demand for 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 support only limited amounts of MLP due to the high cost in terms of chip area and energy use. Existing architectures employ lock-up free caches (e.g. [30]) 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 is a small *fully associative cache.* The maximum number of outstanding cache misses the system supports is limited by the number of MSHR entries.

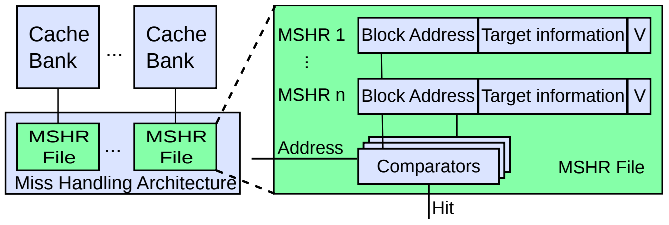


Fig. 1. Miss Handling Architecture (MHA) for a banked cache system and the Miss Information/Status Holding Register (MSHR) file

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, leading to high power consumption and area requirements. 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 [4]. 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, suggesting that the same limitation holds [25].

Currently, the two predominant paradigms of how single-chip processors are built are:

1. 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.

2. 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 that offer abundant parallelism and can be effectively mapped to these GPUs. Other examples include the Sun Niagara 2 [24] and the upcoming Intel Many Integrated Core (MIC) architecture. Research machines such as UT Austin's TRIPS [9], MIT's RAW [28] (available commercially as the TILE architecture from Tilera [2]), and UMD's XMT [33, 36] are also examples of single-chip many-cores.

Traditional multi-cores in the first category use several methods to hide memory latency: out-of-order execution, advanced hardware prefetching mechanisms, large per-core coherent caches, simultaneous multi-threading (SMT) etc. These often mitigate the need for advanced software prefetching. Hence we do not target traditional multicores. In contrast, in many-core architectures, these techniques *do not scale* to the hundreds (and possibly thousands) of cores because of the limited area and power resources available per core, leading to the urgent need for alternative latency-hiding tools. In our work, we are focusing on this challenging, unsolved problem for a proposed high-risk, high-payoff research many-core machine.

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. Some many-core designs 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, 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 due to the larger number of cores, which further limits the size of MSHRs. As a consequence it becomes 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 MSHR file capacity. 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 mentioned above, the run-time gains from RAP are likely to be very small, hence we do not discuss those further.

To evaluate the RAP algorithm, we are using XMT[5] – a general-purpose manycore architecture [23]. 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 [6]), AMD Opteron (speedups up to 8.56x [36]) and also an NVIDIA GTX280 GPU (speedups of up to 8.10x [5] on irregular workloads). The XMT hardware design is simple and scalable, allowing researchers to easily understand and tune architectural parameters. A highly configurable cycle-accurate simulator and an open-source compiler are available, enabling us to implement and evaluate our optimizations on different hardware configurations. An overview of the XMT platform is presented in Section 4. 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 true 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.

As an extension of our evaluation, we investigate the ability of the RAP algorithm to adapt to the available amount of prefetching hardware resources for a particular target platform. To accomplish that, we conducted a design-space exploration (DSE) of the XMT architecture in which we varied the size of the MSHR file in order to accommodate more simultaneous prefetch instructions. For a more complete DSE, we also varied the resources allocated for off-chip bandwidth (i.e. memory controllers and DRAM channels), which is the other resource closely related to memory-level parallelism. We present our extensive case study in Section 6 and show that the RAP algorithm robustly improves performance on all evaluated design space points and across a heterogeneous set of benchmarks. In addition, we show that a non-obvious choice for allocating the available chip area between the two resources provides the best performance, and we make recommendations to the hardware designers for future improvements.

Our main contributions are:

1. *Resource-aware compiler loop prefetching algorithm.* We propose an enhanced compiler loop prefetching algorithm and compare it with existing solutions. We show that by providing the compiler with accurate information about the hardware, we can improve the prefetch performance on average by up to 36.7% over a hardware-oblivious one, depending on the hardware configuration.
2. *Empirical design space exploration* We scale two parameters of the architecture: (i) off-chip DRAM bandwidth and (ii) capacity of MSHR files, and combine performance results with area information to determine the *Pareto-optimal* configurations. These are design points such that no other configurations have better performance for a lower area. Such configurations represent meaningful trade-offs for the hardware designer; all others can be eliminated. We maintain our focus on the memory sub-system and pay special attention to both software and hardware support for data prefetching. For example,

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

our findings show that a 64-processor configuration with 2 DRAM channels and 8-word per-TCU prefetch buffer unit capacity outperforms a similar configuration with 8 DRAM channels and 4-word prefetch buffer units by 4% on average, while actually using 4.3% less chip area.

The remainder of this paper is organized as follows. In Section 2 we review existing compiler prefetching approaches and identify their shortcomings, and in Section 3 we present our enhanced Resource-Aware Prefetching (RAP) algorithm. Section 4 introduces XMT, our experimental platform, which is then used in Section 5 to evaluate the performance improvements of our RAP implementation. In Section 6 we present the findings of the Design-Space Exploration study, followed by a related work survey in Section 7 and our conclusions in Section 8.

2 Existing Software Prefetching Methods

Mowry et. al [21] 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 the 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. Moreover, since the cache is usually a system-wide shared resource, it is impossible to model interference from external sources such as other running processes. Mowry's 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:*

$$PrefDistance = \left\lceil \frac{MissLatency}{IterationTime} \right\rceil \tag{1}$$

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 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:

$$\mathit{MaxRequests} = \mathit{NumRefs} \times \mathit{PrefDistance} \tag{2}$$

where *NumRefs* represents the number of static references that require prefetching. Going back to the code in Fig. 2(c), *NumRefs* $= 3$ since the references to `A[i]`, `B[i]` and `C[i]` will cause a cache miss at each iteration and need prefetching, leading to *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:

1. 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.
2. The MHA does not accept the prefetch request, stalling the issuing CPU until one MSHR becomes available (which happens when one request returns from DRAM or lower cache level). Stalling the CPU was exactly what prefetching was aiming to avoid, and thus the benefits of prefetching are again lost, leaving only the overheads.

To summarize, overflowing the MSHR file is detrimental in all cases, and needs to be addressed by all prefetching approaches. The code in Fig. 2(c), which is the outcome of Mowry's algorithm, fails to address this issue. We discuss proposed improvements next.

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 it 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

(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];
```

Fig. 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.

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. Although the GCC algorithm in Fig. 2(d) addressed the MSHR file overflowing issue encountered by Mowry's original approach in Fig. 2(c), it comes short of the main goal of hiding the memory latency of all the memory references.

In the next section we discuss an enchanced prefetching algorithm, which aims at addressing the limitations of previous approaches and obtain better runtime on a series of benchmarks.

3 New Resource-Aware 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. As will be discussed below, 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.*

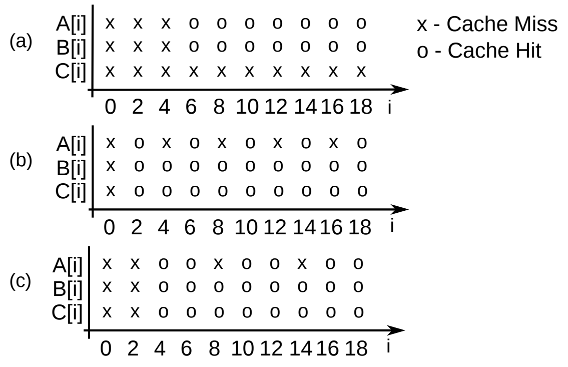


Fig. 3. Dynamic cache trace for the code in Figure 2. (a) GCC loop prefetching with *PrefDistance* = 3 and (b) RAP algorithm with *PrefDistance* = 1 (c) RAP algorithm with *PrefDistance* = 2.

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 $\mathit{IterTime} = 20 + 49 = 69$. Using Equation (1), we need $\mathit{PrefDistance} = \lceil 50/69 \rceil = 1$ iteration in advance. However, GCC schedules prefetches using $\mathit{PrefDistance} = 3$ iterations in advance, according to the original calculation. Moreover, because of this inconsistency, no prefetch instruction is inserted for the `C[i]` reference, causing a miss at every iteration as ilustrated in Fig.3(a).

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 $\mathit{PrefDistance} = 1$ iteration instead of 3, we can now issue prefetches for all three references, using a total of $\mathit{MaxRequests} = 3 \times 1 = 3$ 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). For any prefetch distance* $PD_{RAP} < PD_{Mowry}$ *and*

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

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 $PD_{RAP} < PD_{Mowry}$:

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

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 value:

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

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

- ***III.2*** *If* $PD_{Mowry} \times NumRefs > PrefetchSlots$ *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*.

3.1 Additional Optimizations

During the design and evaluation of the RAP compiler algorithm, we implemented a few other compiler transformations, which contributed to the performance benefits reported in our experimental results.

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 work units, high degree of parallelism – prefetching can be enabled for some benchmarks by using a simple compiler transformation. The compiler can insert several short independent work units (or tasks) in a loop within a coarser task, effectively enabling the use of loop prefetching, at the possible cost of a less load-balanced execution. This compiler technique, called *thread clustering* [23], allowed us to evaluate the loop prefetching algorithm on all our benchmarks.

Prefetching after reference. In most modern architectures, a MSHR entry is allocated for a prefetch request as soon as it is issued, affecting the value of the *MaxRequest* value. Consider a memory reference `A[i]` and its associated prefetch instruction `prefetch(A[i+PD])`, with PD the prefetch distance. Right before `A[i]` is accessed we will have pending (unconsumed) prefetches for references `A[i], A[i+1],...A[i+PD-1]`, using up a total of PD MSHR entries. At this point we can chose to issue the prefetch for `A[i+PD]` before or after the reference to `A[i]`. If the prefetch is inserted textually in the code before the reference to `A[i]` (i.e. as in Fig. 2), an additional MSHR will be needed (for a total of $PD + 1$). Alternatively, the prefetch instruction for `A[i+PD]` can be inserted in the code right *after* the reference to `A[i]`. Hence one prefetched value will be consumed before issuing the new one, thus require reserving only PD MSHR entries. This leads to a saving of one MSHR entry per reference prefetched, at the cost of hiding slightly less latency, since the prefetch is now issued closer to the use. In severely resource-constrained environment, this minor optimization can have a non-trivial effect on performance.

We implemented the "prefetch-after reference" optimization in the RAP algorithm, but we left Mowry's and the GCC implementations unchanged to use the default "prefetch-before" variant.

4 The XMT Framework

In Section 1 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 forward-looking lightweight-core platform. Recent benchmarking efforts have shown that XMT can achieve consistent performance improvements when compared to modern architectures [6, 36, 5], 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. [23]) is improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available with new fabrication technologies. 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 [34–36].

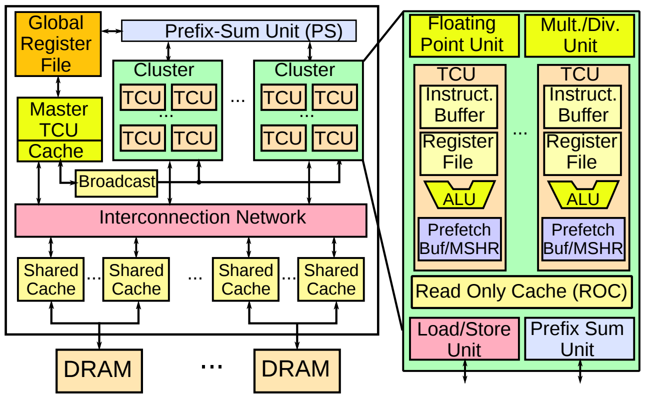
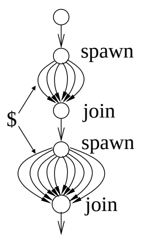


Fig. 4. XMT architecture overview.

The XMT architecture, depicted in Fig. 4, includes an array of lightweight cores called 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 [3]; 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 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.

```
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)

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

The underlying programming model of the XMT framework is PRAM-like, where the lock-step execution of PRAM is relaxed, and threads proceed independently within a spawn block. 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. As in SPMD models, threads access data based on their unique id (specified by the `$` identifier). An algorithm programmed for 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 fine-grained 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 virtual threads (tasks) as the number of available processors. Tasks 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.

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 configurable number of MSHR entries. The XMT unit of work consists of a virtual thread (the code executed with one unique thread id), or a loop consisting of several thread ids if clustering is used. The RAP algorithm is applied at the work unit level, and to the innermost loop.

Prefetching as a tool towards ease of programming. Ease of programming is a necessary condition for the success of a general-purpose many-core platform, and it is one of the main objectives of XMT. Relying on compiler prefetching to optimize for the length of sequence of roundtrips to memory (LSRTM) [32] significantly lowers the programmer's effort, allowing them to focus on designining efficient parallel algorithms instead of on low-level details. Indications that XMT is an easy-to-program efficient parallel architecture, include:

- 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 [29], and shown to be superior to alternative parallel approaches such as MPI, OpenMP and CUDA;
- XMT is *based on a rich algorithmic theory* (PRAM) that provides a solid framework for designing and analyzing algorithms, equivalent to the serial model;
- XMT provides a *programmer's workflow* for deriving efficient programs from PRAM algorithms, and reasoning about their execution time [32] and correctness;
- 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].

5 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 [1].

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 1 describes the benchmarks used for evaluating the compiler algorithm performance. The 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.

Table 1. 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

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 across all benchmarks 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, depending on the number of MSHR entries.

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.

The reason that the run-time improvements only show up to 12 words MSHR capacity can be identified by looking at the parameters of the architecture and benchmarks. 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

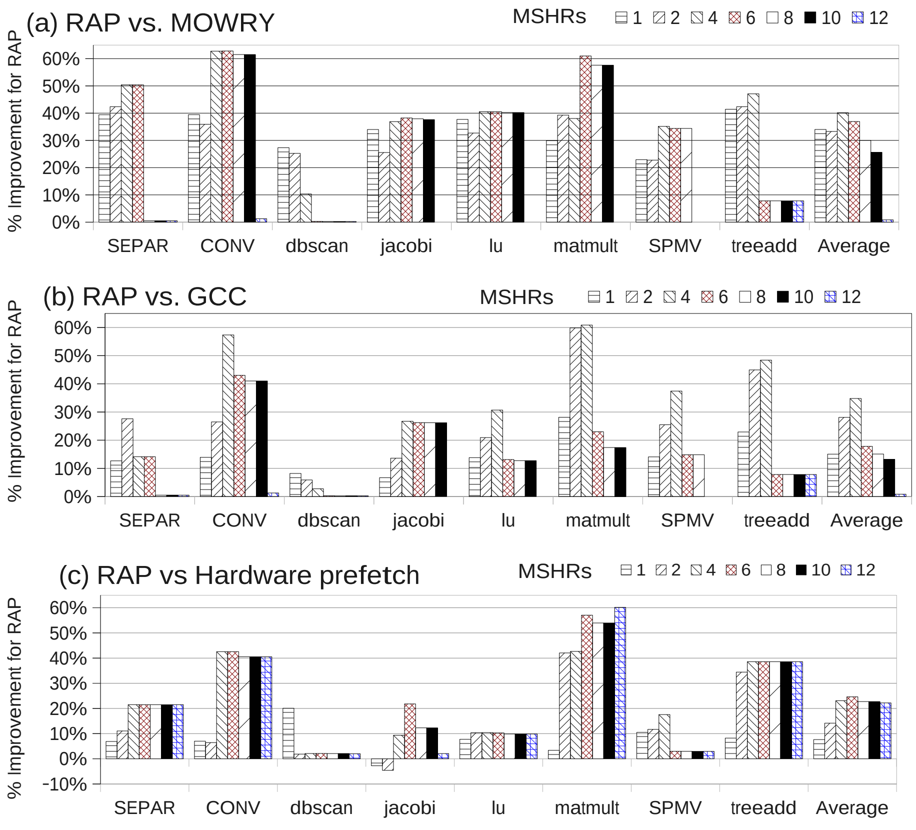


Fig. 6. Performance improvement of RAP compared to (a) Mowry, (b) GCC and (c) one-block-lookahead hardware prefetching

thus the *MaxRequest* value for the benchmarks ranges between 6 and 12 for our benchmarks, as shown in Table 1. 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.

Our results mark a significant improvement over existing approaches, as most of the time will be spent in resource-constrained conditions. For the class of highly parallel architectures we are targeting, a per-core MSHR of 12 entries represents a significant budget of area and power. 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 dis-

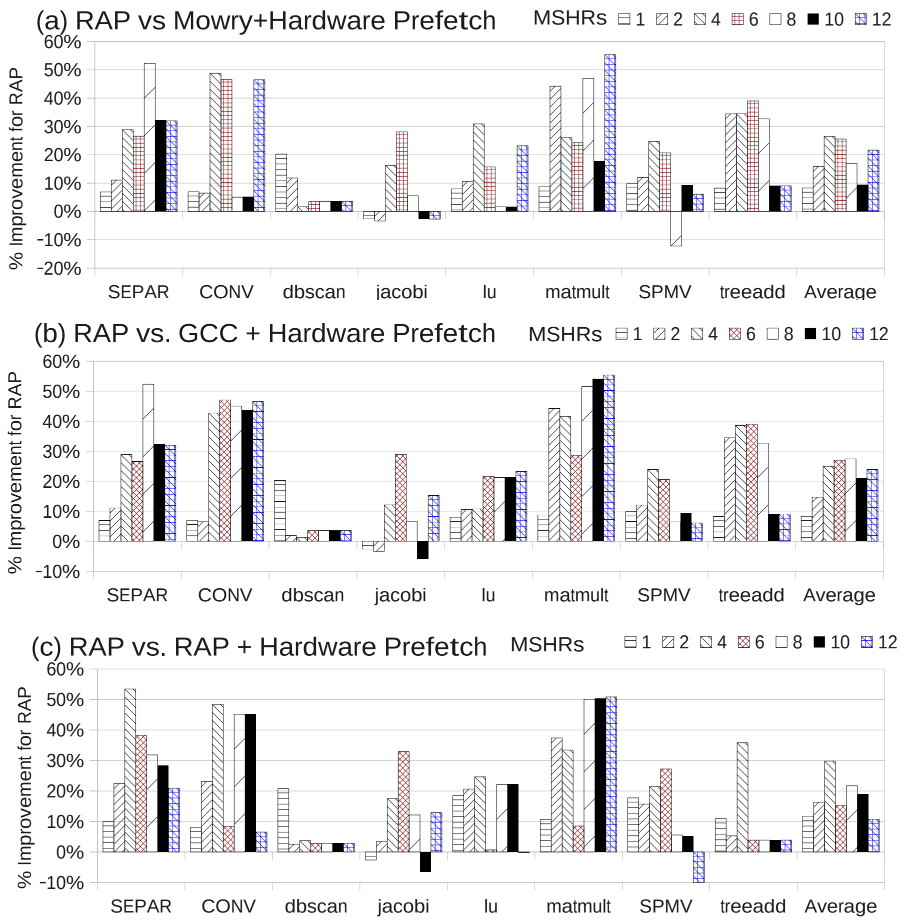


Fig. 7. Interference of software with hardware prefething. Performance improvement of RAP compared to (a) Mowry + OBL hardware prefetching, (b) GCC + OBL hardware prefething and (c) RAP + OBL hardware prefething.

tinguishing 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. [27]), which prefetches the *next cache line* once a particular line is first read.

We implemented this scheme in XMTSim, the XMT cycle-accurate 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.

In Fig. 7 we compared the performance of the software-only RAP algorithm with configurations in which both hardware and software prefetching were enabled. Most modern serial and multi-core architectures have support for both, and they can usually be enabled or disabled through system configuration tools and compiler flags.

Figures 7(a) and 7(b) present the performance improvements of RAP versus configurations where on top of the OBL hardware prefetcher (OBL-HP), the compiler used Mowry's and GCC's software prefetching algorithm respectively. The hardware and software prefetchers interfere and compete for using the MSHR, which leads to less predictable results. In particular, on few configurations, the combined software-hardware outperforms the software-only approach. However, on average across all benchmarks, the RAP algorithm is 8.3-26.5% faster than Mowry plus OBL-HP and 8.3-27% faster than GCC plus OBL-HP.

We also examined the performance of the RAP algorithm when the OBL-HP was enabled, as shown in Fig. 7(c). We observed the same performance degradation caused by the interference of the two mechanisms, with the software-only RAP outperformed RAP plus OBL-HP by 10.8-29.8%.

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.

6 Design Space Exploration for Prefetching and Memory Resources

6.1 Objectives

We conducted a design-space exploration (DSE) of the XMT architecture with respect to the two components that are most relevant to memory-level parallelism: (a) the capacity of the TCU MSHR file, and (b) the off-chip bandwidth to DRAM, as controlled by the number of DRAM channels. More MSHR entries allow more data elements to be simultaneously prefetched, and more loop iterations before they are actually needed, hiding more of the latency. More DRAM channels provide more bandwidth which can be used to prefetch data, in addition to serving the regular, non-prefetched memory accesses.

The goals of our study are: (i) to get a rough estimate of what the overall hardware cost for software-prefetching support is for a lightweight manycore, and whether this cost is feasible; (ii) to understand the tradeoffs between two different hardware-assisted techniques for boosting MLP – software-prefetching and multiple DRAM channels – and determine what combination is a good configuration for a lightweight manycore architecture such as our experimental platform; and (iii) to evaluate the ability of the RAP software prefetching algorithm to achieve good performance for a wide variety of machine configurations in an adaptable manner.

6.2 Design Space

We consider multiple design space points that range from the bare-bones architecture with no MSHR file and one DRAM channel, to a configuration including 20-word MSHR files and 8 DRAM channels. We scale the capacity of the MSHR file linearly in increments of 2 words per core, and use powers of two for the number of DRAM channels (evaluating 1, 2, 4 and 8).

We did not include the hardware required to implement the One-Block-Lookahead hardware prefetching mechanism in the design space. This scheme requires additional per-TCU hardware to compute next block's address and issue the new prefetch request. A MSHR file is still required to keep track of the outstanding memory requests issued but not completed. The results presented in Section 5 show that the OBL scheme is always slower than the RAP software approach while requiring more chip resources, therefore there is no benefit in choosing a configuration with OBL support.

6.3 Targeted ASIC Parameters

Our basic design is based on the HDL description of the XMT Paraleap system [36]. The HDL design was validated by prototyping using both FGPA and ASIC technology. The DSE results in this section are based on the ASIC synthesis process, for which we used the Synopsys Design Compiler and the 90nm IBM CMOS9SF library, part of Artisan's SAGE-X v3.0 Standard Cell Library.

The targeted clock frequency of a 64-TCU XMT ASIC is in the same range as a modern many-core architecture such as GPUs. This claim is based on the results from (i) the ASIC implementation of the Mesh-of-Trees (MoT) interconnection network fabricated in 90nm IBM technology [3], (ii) a complete 64-TCU XMT integer-only chip using the same IBM fabrication process and (iii) a detailed chip area comparison between a 1024-XMT configuration and a NVIDIA GTX 280 [5]. By using the same generation technology as a modern GPU, as well as a comparable amount of engineering and optimization effort, XMT can support clock frequencies in the range of 1-1.3GHz and possibly higher.

6.4 Area Scaling Methodology

The design is highly parametrized, and therefore we can synthesize gate-level descriptions of various configurations. After synthesis, we extract the cell area of units of interest and reference it relative to the total design cell count. A cell is the basic unit of ASIC design, a standard arrangement of transistors that implements a gate or flip-flop. We note that these area numbers are likely to change when the design undergoes the Place-and-Route process. At the present time, our tools did not permit us to obtain direct area numbers for sub-components after the P&R phase, and thus we use the cell areas reported by the synthesis as approximations of the final gate counts.

To evaluate the change in area for each of our design points, we first identify the hardware modules we are going to scale in the HDL description, determine the area they use in a representative subset of configurations from the synthesis output, then extrapolate to the the rest of the points.

6.5 DSE Results

We evaluate the chosen design points from the point of view of average runtime across all the benchmarks. For each hardware design configuration described in Section 6.2, we initialize the XMT cycle-accurate simulator with the respective set of parameters and execute all the benchmarks in the test suite. We use the RAP compiler algorithm described in Section 3, as well as the thread clustering transformation to enable prefetching for all the benchmarks in our study.

As reference point, we use a bare-bones configuration that includes 64 parallel TCUs, no MSHR file and one DRAM channel. For each design point, we evaluate the relative increase in area due to the change in parameters (MSHR file capacity or additional DRAM channels). Fig. 8 shows the observed increase in number of cells used for the DRAM channels and for the MSHR File when scaling the number and capacity respectively. For example, we identified that the area used for increasing the capacity of the MSHR file from 4 to 8 words over all 64 TCUs is 263,872 cells, representing 0.59% increase from the base configuration total chip area of 44,797,832 cells. An additional DRAM channel added 357,819 cells, representing a 0.8% increase in area relative to the same base configuration. Except for the non-linear increase observed between zero and four MSHR entries, due to fixed costs of adding the prefetch datapath, the growth is mostly linear. We extrapolate from our set of synthesized configurations to the rest of the design space points to estimate area increases.

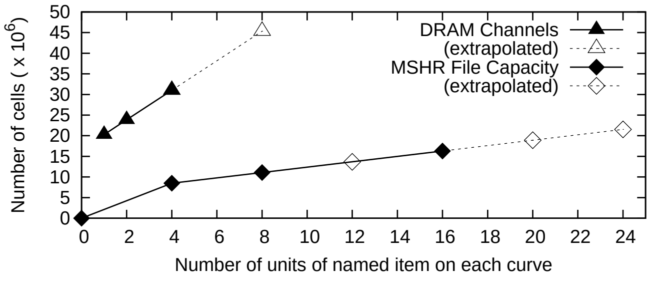


Fig. 8. Number of area cells used for the DRAM channels and MSHR File when scaling the number and capacity respectively.

We record the average execution time across all the benchmarks, normalized to the bare-bones configuration. Fig. 9 represents a Performance-Area scatter plot of all the design points. For each point, we include a 2-tuple representing the number of DRAM channels and the MSHR file capacity. The point labeled (1,0) is the reference configuration used, normalized to a runtime of 100 and area increase of 0%. A first insight is the significant increase in performance resulting from the inclusion of even a small MSHR file. This is illustrated by the difference in performance between the configurations with no MSHR file (1,0), (2,0), (4,0),(8,0) and the rest.

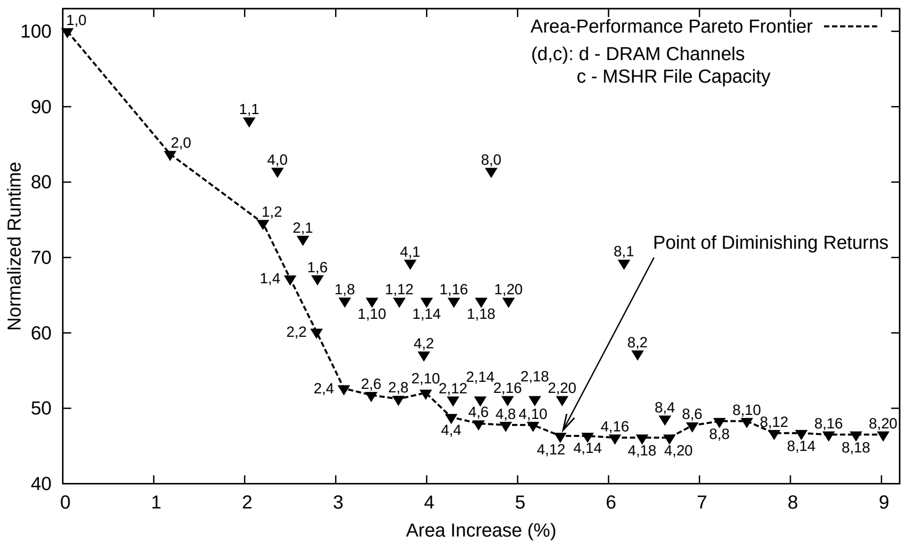


Fig. 9. Area and performance change when varying the number of DRAM channels and the capacity of the MSHR file. Each point includes a 2-tuple (d,c) representing (No. DRAM channels, MSHR file capacity).

We combine collected average performance numbers with area information and determine the *Pareto-optimal* design space points. These represent the only viable choices to the hardware designer, since no other configuration has better performance for a lower area. Note that this curve is computed using a heterogenous set of benchmarks. For a special-purpose architecture, the same methodology can be used, but using a workload that is characteristic of the target application domain, and a different curve will be obtained.

The dotted line in Fig. 9 depicts the *Area-Performance Pareto frontier*. A system designer can use this diagram to choose the best performing configuration for a given area, by choosing the appropriate Pareto optimal point. In addition, we can observe the knee of the curve (4,12) which is the configuration point with 4 DRAM channels and 12 MSHR file entries, and represents *the point of diminishing returns*, with 5.47% area increase and 53.66% performance improvement; after reaching point, additional increases in area do not translate in significant increases in performance. This knee of the curve (4,12) represents our best recommended configuration for an XMT implemented with today's technology.

Hardware engineers can use the data in Fig. 9 to make informed design decisions. For example, they can determine that by increasing the total chip area by 3.09%, the resulting system will be 47.36% faster (for the 2 DRAM channels, 4 entry MSHR file configuration), observation that can affect the decision to allocate additional resources or use more aggressive optimizations in the design and synthesis tools.

6.6 Additional Considerations

We are using a number of simplifying assumptions in this study by not including considerations such as the I/O pin count and power/thermal optimizations. Such topics are currently under study as part of the XMT project and we plan to incorporate them as additional dimensions in future work. This study is still useful, however; since if, for example, there is an upper bound on the allowed pin count, the designer can choose among a subset of all the Pareto-optimal designs we derive; namely the subset that also satisfies the pin count constraint.

7 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 [21, 19, 7] relies on the existence of non-blocking prefetch instructions and is usually enabled by the compiler. In hardware prefetching (e.g. [27, 14, 18]) a specialized hardware unit infers prefetching opportunities by monitoring run-time behavior. Prefetching schemes for parallel architectures in both software [19, 31, 20] and hardware (e.g. [8]) 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 [22] explores the effect on execution time of varying the number of outstanding prefetch requests that can be handled simultaneously by the hardware. The author also compares two versions of the prefetch issue buffer hardware - one in which the processor stalls when the buffer is full, and one where additional requests are simply dropped. Their results were mixed, with the former performing slightly better when using the prefetching algorithm described in Section 2. In follow-up work, Mowry [20], as well as McIntosh [19], 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 [7, 15] or hardware prefetching [14]. 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 1, 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 [37] 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.

In recent work, Tuck et. al [30] propose an **alternate MHA organization** that would provide increased MSHR capacity at the expense of slightly higher latencies. However, it is not clear how the performance of the scheme would change when scaled to the degree of parallelism required by the emerging many-core architectures. Different schemes for improving the functionality of existing MHAs, by either dynamically adjusting the MHA capacity to reflect the memory bus load [13] or by devising a MHA-aware cache replacement policy to reduce the number of cache misses [26] have been proposed. Our compiler algorithm is orthogonal to these techniques and can function alongside these hardware enhancements.

In the area of **design-space exploration**, recent years have seen a burst of studies targeted at understanding the interactions between performance, energy, thermal efficiency and area in the design of chip multi-processors (CMPs), or multi-cores. Huh et. al [12] explore the design space of CMPs as the available transistor budget grows with the fabrication technology. They consider in-order vs. out-of-order cores, amount of cache per processor and availability of off-chip bandwidth, while varying the area constraints. Li et. al [16] extend the study by including pipeline depth and width, operating voltage and frequency and cooling mechanisms as design space dimensions, as well as adding thermal constraints. Methods for automatically or semi-automatically design space exploration for specialized architectures such as System-On-Chip and signal processing have also been proposed [10, 17]. These studies are all focused on share-nothing workloads, optimizing architectures designed for throughput rather than single-task completion time.

The research presented in this paper differs from prior work in that it focuses on MLP resources – MSHR capacity and DRAM channels – for design space exploration. It also does so in conjunction with a resource-aware compiler algorithm that efficiently uses the limited resources at each design point. This study is targeted at the needs of lightweight manycore architectures which aim to optimize single-task completion time, an emerging architectural paradigm.

8 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 compilers targeting fine-grained many-core architectures.

In addition, we conducted a Design-Space Exploration focused on resources directly affecting support for Memory-Level Parallelism on an emerging many-core architecture, the XMT Paraleap. We identified the Pareto-optimal hardware-software configuration which delivered 53.66% performance improvement on average while using only 5.47% more chip area than the bare-bones design. The methodology of evaluating each design point by adapting the RAP algorithms to the prefetch resources available is also of interest.

References

1. Software release of the explicit multi-threading (xmt) programming environment. `http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html` (October 2010), v0.82
2. Tilera corporation, `www.tilera.com`
3. Balkan, A.O., Horak, M.N., Qu, G., Vishkin, U.: Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. hoti pp. 21–28 (2007)
4. Boggs, D., Baktha, A., Hawkins, J., Marr, D.T., Miller, J.A., Roussel, P., Singhal, R., Toll, B., Venkatraman, K.: The microarchitecture of the intel pentium 4 processor on 90nm technology. Intel Technology Journal 8(1), 7–23 (February 2004)
5. Caragea, G.C., Keceli, F., Tzannes, A., Vishkin, U.: 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)
6. Caragea, G.C., Saybasili, A.B., Wen, X., Vishkin, U.: 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. pp. 163–165. ACM, New York, NY, USA (2009)
7. Chen, W.Y., Mahlke, S.A., Chang, P.P., Hwu, W.M.W.: Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In: MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture. pp. 69–73. ACM Press, New York, NY, USA (1991)
8. Dahlgren, F., Dubois, M., Stenström, P.: Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 6(7), 733–746 (1995)

9. Gebhart, M., Maher, B.A., Coons, K.E., Diamond, J., Gratz, P., Marino, M., Ranganathan, N., Robatmili, B., Smith, A., Burrill, J., Keckler, S.W., Burger, D., McKinley, K.S.: An evaluation of the trips computer system. In: ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems. pp. 1–12. ACM, New York, NY, USA (2009)
10. Givargis, T., Vahid, F., Henkel, J.: System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on pp. 25–30 (2001)
11. Hochstein, L., Basili, V.R., Vishkin, U., Gilbert, J.: A pilot study to compare programming effort for two parallel programming models. Journal of Systems and Software 81(11), 1920 – 1930 (2008)
12. Huh, J., Burger, D., Keckler, S.W.: Exploring the design space of future cmps. Parallel Architectures and Compilation Techniques, International Conference on 0, 0199 (2001)
13. Jahre, M., Natvig, L.: A high performance adaptive miss handling architecture for chip multiprocessors. Transactions on High-Performance Embedded Architectures and Compilers 4(1) (2009)
14. Jouppi, N.P.: 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. pp. 364–373. ACM Press, New York, NY, USA (1990)
15. Klaiber, A.C., Levy, H.M.: An architecture for software-controlled data prefetching. In: ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture. pp. 43–53. ACM Press, New York, NY, USA (1991)
16. Li, Y., Lee, B., Brooks, D., Hu, Z., Skadron, K.: Cmp design space exploration subject to physical constraints. High-Performance Computer Architecture, 2006. The Twelfth International Symposium on pp. 17–28 (Feb 2006)
17. Lieverse, P., van der Wolf, P., Deprettere, E., Vissers, K.: A methodology for architecture exploration of heterogeneous signal processing systems. Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on pp. 181–190 (1999)
18. Lin, W.F., Reinhardt, S.K., Burger, D.: Reducing dram latencies with an integrated memory hierarchy design. hpca 00, 0301 (2001)
19. McIntosh, N.: Compiler support for software prefetching. Ph.D. thesis, Rice University (May 1998), adviser-Ken Kennedy
20. Mowry, T.C.: Tolerating latency in multiprocessors through compiler-inserted prefetching. ACM Trans. Comput. Syst. 16(1), 55–92 (1998)
21. Mowry, T.C., Lam, M.S., Gupta, A.: Design and evaluation of a compiler algorithm for prefetching. SIGPLAN Not. 27(9), 62–73 (1992)
22. Mowry, T.C.: Tolerating latency through software-controlled data prefetching. Ph.D. thesis, Stanford, CA, USA (1995)
23. Naishlos, D., Nuzman, J., Tseng, C.W., Vishkin, U.: 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. pp. 93–102. ACM, New York, NY, USA (2001)
24. Nawathe, U., Hassan, M., Yen, K., Kumar, A., Ramachandran, A., Greenhill, D.: Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. Solid-State Circuits, IEEE Journal of 43(1), 6–20 (Jan 2008)
25. Porterfield, A., Fowler, R., Mandel, A., Lim, M.Y.: Empirical evaluation of multi-socket, multi-core memory concurrency. Tech. Rep. RENCI TR-09-01, Renaissance Computing Institute (January 2009)

26. Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y.N.: A case for mlp-aware cache replacement. In: ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture. pp. 167–178. IEEE Computer Society, Washington, DC, USA (2006)
27. Smith, A.J.: Cache memories. ACM Comput. Surv. 14(3), 473–530 (1982)
28. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A.: The raw microprocessor: A computational fabric for software circuits and general-purpose programs. IEEE Micro 22(2), 25–35 (2002)
29. Torbert, S., Vishkin, U., Tzur, R., Ellison, D.: 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)
30. Tuck, J., Ceze, L., Torrellas, J.: Scalable cache miss handling for high memory-level parallelism. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 409–422. IEEE Computer Society, Washington, DC, USA (2006)
31. Tullsen, D.M., Eggers, S.J.: Effective cache prefetching on bus-based multiprocessors. ACM Trans. Comput. Syst. 13(1), 57–88 (1995)
32. Vishkin, U., Caragea, G.C., Lee, B.C.: Handbook of Parallel Computing: Models, Algorithms and Applications, chap. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. CRC Press (2007)
33. Vishkin, U., Dascal, S., Berkovich, E., Nuzman, J.: 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. pp. 140–151. ACM, New York, NY, USA (1998)
34. Wen, X.: Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm. Ph.D. thesis, University of Maryland (August 2008)
35. Wen, X., Vishkin, U.: Pram-on-chip: first commitment to silicon. In: SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures. pp. 301–302. ACM Press, New York, NY, USA (2007)
36. Wen, X., Vishkin, U.: Fpga-based prototype of a pram-on-chip processor. In: CF '08: Proceedings of the 2008 conference on Computing frontiers. pp. 55–66. ACM, New York, NY, USA (2008)
37. Yang, C., Yang, X., Xue, J.: Advances in Computer Systems Archiecture, vol. 3740/2005, chap. Improving the Performance of GCC by Exploiting IA-64 Architectural Features, pp. 236–251. Springer-Verlag (2005)