An Event-Triggered Programmable Prefetcher for Irregular Workloads

Sam Ainsworth
University of Cambridge, UK
sam.ainsworth@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

Abstract

Many modern workloads compute on large amounts of data, often with irregular memory accesses. Current architectures perform poorly for these workloads, as existing prefetching techniques cannot capture the memory access patterns; these applications end up heavily memory-bound as a result. Although a number of techniques exist to explicitly configure a prefetcher with traversal patterns, gaining significant speedups, they do not generalise beyond their target data structures. Instead, we propose an event-triggered programmable prefetcher combining the flexibility of a general-purpose computational unit with an event-based programming model, along with compiler techniques to automatically generate events from the original source code with annotations. This allows more complex fetching decisions to be made, without needing to stall when intermediate results are required. Using our programmable prefetching system, combined with small prefetch kernels extracted from applications, we achieve an average 3.0× speedup in simulation for a variety of graph, database and HPC workloads.

CCS Concepts • **Computer systems organization** → *Architectures*; • **Software and its engineering** → *Compilers*;

Keywords Prefetching

ACM Reference Format:
Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/https://doi.org/10.1145/3173162.3173189

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/https://doi.org/10.1145/3173162.3173189

1 Introduction

Many modern and emerging workloads execute on large volumes of data, which cannot fit in current systems' caches. Often these accesses are irregular and difficult to predict in advance, resulting in heavily memory-bound execution with frequent stalls from high DRAM latency [31, 34, 49].

There are several techniques available to address these challenges. One option is to utilise thread-level parallelism within an application, to cope with latency using aggressive multi-threading [38], effectively parallelising loads by having many threads stalled at once. This is typical of workloads running on graphics cards, for instance [42]. However, this technique only works if the application exhibits a great deal of thread-level parallelism. This is often not the case in big data workloads, due to complex and unpredictable reads and writes to the same data [41], and the difficulty of creating effective partitions for the parallel cores to work on.

Another option is prefetching, either through hardware prefetch units or software instructions. However, traditional address-based (stride) prefetchers [14, 29, 52, 54] only work for very regular computations, typically involving either dense matrices or entirely sequential memory accesses. History-based prefetchers [25, 26] only work for highly repeated computation. Neither of these apply to many big-data applications, such as database, graph and many high performance computing (HPC) workloads, which exhibit much more complicated, irregular traversals of data, involving pointer chasing and indirect array lookups [38]. Techniques have been proposed specifically for irregular accesses, such as pointer fetching prefetchers [17], which fetch plausible pointers from observed memory loads. However, these lack the ability to look ahead in arrays, cannot fetch commonly-used index-based data structures (as the loaded memory doesn't include pointers), and suffer from severe over-fetching from memory, due to a lack of ability to have fine-grained control over prefetches. Software prefetching [2, 12, 40], on the other hand, fills the main CPU's pipeline with many extra instructions, and is unable to deal with accesses involving multiple loads without stalling.

Nevertheless, despite the lack of success for traditional, implicit prefetching techniques on these workloads, it is still possible to mitigate the cost of latency associated with

memory accesses. Techniques to extract memory-level parallelism for a variety of memory-bound applications exist [1, 30, 31, 34, 49] through explicit configuration of traversal patterns, gaining significant performance improvements for the targeted workloads. However, currently such architectural techniques are highly specialised to the target computation, so adding them to general-purpose systems may be infeasible due to the lack of wide applicability. Further, they are unable to deal with the rapid evolution of algorithms within the field due to their fixed-function nature.

To this end, we have designed an event-based programmable prefetching system for general-purpose workloads in a variety of domains that feature sparse memory accesses [8] including graphs, databases and HPC. We couple a conventional high-performance out-of-order computation core with a specialised prefetching structure for the L1 cache, attached to several in-order programmable prefetch units. The event-based programming model allows each pre-fetch unit to issue and react to multiple loads at once without stalling. This enables the system to prefetch based on the results of earlier prefetches, in addition to prefetching from multiple data structures concurrently.

We further provide compiler techniques to generate event programs for these cores based on the original source code and thus alleviate manual effort for simpler access patterns, using annotations to specify what needs to be prefetched.

On a wide set of memory-bound benchmarks, we achieve a 3.0× average speedup, with high utilisation of prefetches brought into the cache, and negligible additional memory accesses for most workloads.

2 Existing Work

There is an abundance of work in the literature concerning prefetching, and we describe the most relevant works here, highlighting the elements that are beneficial for workloads with irregular memory accesses. Summaries include the works of Mittal [43] and Falsafi and Wenisch [19].

Fetcher Units Much of the research into efficient execution of irregular workloads has focused on highly specialised fetcher units. These take control of memory accesses for a particular access pattern, extracting performance through parallel loads of data, often with large performance improvements. SQRL [34] and DASX [35] are fetcher systems designed for iterative accesses of B-tree, vector and hash table structures. Similarly, Kocberber et al. [30, 31] focus on the optimisation of database inner joins by parallel hash table walking. In a later paper, they emulate a similar technique in software [32]. Ho et al. [24] generalise the concept of fetcher units to cover more accesses by encoding memory accesses as a set of rules, to allow loads and stores to be mapped to a dataflow architecture. Fetcher units can realise energy savings through removal of the original load instructions.

```
1  for(x = 0; x < in.size; x++) {
2  SWPF(htab[hash(in.key[x+dist])]);//Software prefetch
3  Key k = in.key[x];
4  Hash h = hash(k);
5  Bucket b = htab[h];
6  ListElement l = b.listStart;
7  while(l != NULL) {
8  if(l->key == k) {
9  wait_til_oldest(); //Multithreading
10 out.match[out.size] = k;
11 out.size++;
12 }
13 l = l->next;
14 }
15 signal_iter_done(x); //Multithreading
16 }
```

Figure 1. Hash join kernel with two latency-hiding methods.

However, the original application has to be modified, yielding code incompatible with devices not featuring fetcher units, and stores to the fetched data are typically disallowed.

Configurable Prefetchers This paper develops a configurable prefetcher exposed at the architectural level, and ideas showing the benefits of this have been proposed in the past. Al-Sukhni et al. [3] use explicit Harbinger instructions at the program level to control linked-list pointer fetching. Yang and Lebeck [57] develop a programmable prefetching scheme for linked data structures. The programmable fetchers are allowed to stall, and so cannot deal with patterns which require overlapping of memory accesses to achieve high performance. Ainsworth and Jones [1] design a configurable prefetcher specifically for graph workloads, gaining large speedups, but only targeting specific traversals for a particular graph format. Kohout et al. [15, 33] design a configurable prefetcher to fetch lists of lists.

Implicit Irregular Prefetchers Many attempts have been made at prefetching irregular structures using more traditional, implicit schemes without configuration. This is desirable, as it reduces manual effort, and does not require recompilation. However, though progress has been made, none have been implemented in commercial systems [19].

Pointer-fetching prefetchers [17], which fetch all plausible virtual addresses from cache lines read by a core, have been proposed in several schemes. The main downside to these approaches is the large over-fetch rate. In addition, these schemes are unable to deal with the array-indirect patterns seen in many workloads.

Attempts to extract dependence-graph streams at runtime, by detecting dependent loads, have been made [6, 44, 51]. These run dynamically detected load streams on programmable units on identification of the start of a set of loads, to prefetch the data. These require a large amount of analysis hardware to be added to the commit stage of the pipeline, and a large amount of processing power to run the detected streams. Mutlu et al. propose a runahead scheme [47], which utilises idle chip resources on a cache miss to dynamically

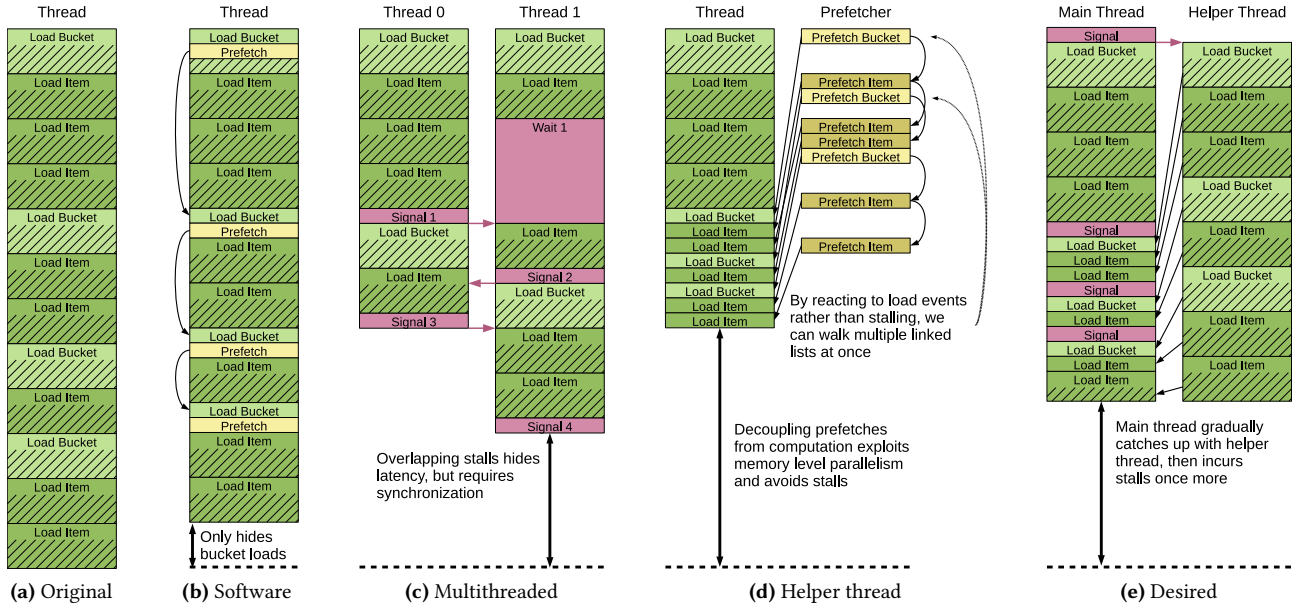


Figure 2. Execution of hash join codes. Software prefetch can only reduce stalls to the hash table buckets. Multithreading overlaps parallel sections, but must synchronise on dependences. Ideally we would prefetch hash table buckets and list items separately from the main computation and allow the prefetcher to issue further prefetches based on values of earlier prefetches.

prefetch loads. These are limited by being tightly bound to the instruction stream, thus are unable to exploit significant lookahead, or prefetch from other prefetched loads. Hashemi et al. extend this [23] by using runtime analysis hardware with access to all microarchitectural state to generate code fragments of critical instructions. These fragments still stall on dependent loads, but are offloaded to simpler hardware to fetch data, and remove some redundant execution.

Yu et al. [58] pick up stride-indirect patterns using runtime analysis of executed code to find the base array and size of each data element. This achieves prefetching of this single pattern, at the expense of complicated analysis hardware in the cache, which may affect the critical path of execution.

Helper Threads One solution for prefetching irregular applications has been to use separate CPU threads to prefetch data in software. Kim and Yeung [27] automatically generate "pre-execution threads" from compiler analysis. These have the desirable property that no extra hardware is required. However, they use an additional thread on a high-performance core, which could consume significant amounts of energy. They are further unable to deal with prefetches based on prefetches without stalling [28]. Further, the lack of a hardware event queue makes synchronisation on loads difficult and expensive. Lau et al. [37] propose a similar scheme, but with architectural support: a single small helper core is attached to a main core to assist with processing tasks. This tight coupling somewhat helps alleviate the synchronisation problem, but still exhibits the same stalls as above. Given this, a single core is rarely able to meet the processing needs of complex access patterns. Ham et al. [22] provide a scheme where a core is split based on separate access and execute threads, which run different code. This again provides closer synchronisation, and as each thread is specialised it can be run on more efficient hardware, but requires high performance from both the load and compute units, and is unable to deal with complicated address generation without stalling on intermediate loads.

Ganusov and Burtscher [20] use helper threading to emulate common Markov [26] and stride prefetching schemes in software, by adding in hardware support to forward observed loads to newly spawned threads. We forward similar data in hardware, but create fully programmable prefetch events which can react to other prefetches, allowing real data structure traversal, and exploit efficient parallelism by using many specialised units with closer coupling and a light abstraction for running custom prefetch code.

Summary While there are elements of techniques from the literature that can help with efficient and timely prefetch of data into the cache for irregular workloads, there is currently no complete solution. We next consider how several existing schemes perform on a complex benchmark kernel, motivating the need for event-based and decoupled programmable prefetching hardware, developed in section 4.

3 Motivation

Figure 1 gives an example of a typical hash join kernel, as used in databases. We have an indirect access to a hash

table array via a hash on a sequential access to a key array, followed by linked-list traversals.

There are several challenges here for existing prefetchers. First, as a result of the hash function, accesses to the hash table array are unpredictable and scattered throughout memory, with no spatial or temporal locality among them. Without knowing the hash function, there is no chance of being able to accurately prefetch entries. Second, the linked-list traversal does not perform a significant amount of work on each element. Although pointer prefetchers could identify `l->next` as the address for the next element to process, the lack of work performed on each iteration of the while loop means that a prefetcher cannot hide the memory access latency of bringing in the next list item.

Figure 2(a) shows how this unmodified code would execute.[1] Light green boxes denote the calculation of the hash and load of the hash-table bucket. Darker green boxes show a load of a linked-list item. Diagonal lines in the boxes show a stall, waiting for the data to arrive from a lower level cache or main memory. As can be seen, each load causes a stall due to the lack of temporal and spatial locality in the code.

Software Prefetching In this example, software prefetching [12] can be more beneficial than using a hardware prefetcher, since we can encode the hash function inside the prefetch. Figure 1 shows this instruction and its position within the code. We prefetch at a fixed number of for-loop iterations into the future (`dist`) to bring hash-table elements into the cache in advance of them being used. However, we cannot help with the linked-list traversal because the software does not get notified about the results of this hash-table item prefetch. We are restricted to prefetching the linked list for the current hash-table item, which suffers the same memory latency hiding challenges as in hardware.

Figure 2(b) shows how the software prefetch improves performance. Yellow boxes denote the calculation of the prefetch address and corresponding prefetch instruction. We assume a prefetch distance of 1 iteration in this example, meaning the first iteration prefetches the hash-table bucket for the second iteration, and so on. As can be seen, for the second and subsequent iterations, there is no stall for loading the bucket (although the prefetch instruction itself incurs an overhead). After four iterations, execution finishes slightly earlier than in the original code, but the inability to prefetch the linked-list items limits the performance increase.

Multithreading A third option is to exploit thread-level parallelism. Each of the for-loop iterations can be executed as a separate thread to hide the memory latencies. However, the algorithm is not embarrassingly parallel, and the order of the output keys could change by executing iterations out of order, so synchronisation is required to prevent this.

Code for this option is shown in figure 1, and its execution on two threads is shown in figure 2(c). When a matching key is found, the thread waits until it is executing the oldest iteration before writing to the output array, to preserve ordering. This is performed by calling `wait_til_oldest();` the companion `signal_iter_done()` signals at the end of each iteration to keep track of the oldest currently executing.

In the example (figure 2(c)), there is a match on the key in the first list item in the second iteration. However, since the first iteration on core 0 is still running, this second iteration must wait until that is finished before writing to the output array. Despite this idle time, the multithreaded version in this example completes faster than with software prefetching by overlapping execution and stalls where possible.

Helper Thread A fourth type of prefetching is to duplicate the memory accessing part of the loop into a separate, helper thread. This thread can run in a different context on the same core as the main thread, if simultaneous multithreading support is available, to prefetch into the main L1 cache. Execution for this technique is shown in figure 2(d). The fundamental limitation of this approach is that the helper thread cannot load data in fast enough to stay ahead of the main thread. The helper thread cannot use prefetches but must stall on each load to be able to use results from it. Though it is possible to use multiple helper threads to alleviate this problem to an extent, this requires a very large amount of system resource, as we need enough helper threads to hide all memory stalls.

Desired Behaviour However, in the ideal case we would have no stalls at all. The workload actually contains a significant amount of memory-level parallelism that existing techniques are unable to exploit: we can parallelise over the array `in.key`, allowing us to prefetch multiple linked lists at once, by overlapping the sequential linked-list fetches. If we could decouple the calculation of prefetch addresses from the main execution in a way that prevents stalling on each load, we would be able to take advantage of this parallelism and bring data into the cache shortly before it is used. This would lead to an execution similar to that in figure 2(e) where, after a warm-up period, computation can proceed without stalls, since data is immediately available in the first level cache. To realise this we must allow the prefetcher to react to data coming back from its own prefetches, and give it knowledge of the computation being performed, so that it can calculate the next set of prefetches based on the data structures being traversed.

4 Programmable Prefetcher

We develop a novel prefetching scheme, along with compiler and hardware support, based on the abstraction of

[1] The nature of this code, where the linked list is dependent on the hash-table bucket load, means that an out-of-order core would not be able to exploit memory-level parallelism through multiple outstanding loads.

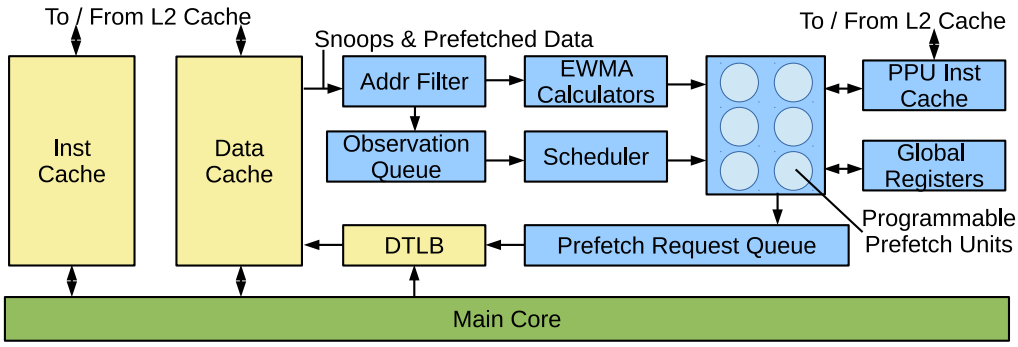


Figure 3. Structure of the programmable prefetcher.

an event. As with common stride prefetchers [14, 54], new prefetches are triggered by read events within the cache, and also by prefetched data reaching the cache. What makes our scheme suitable for more diverse and irregular applications is that these events have programmable behaviour, triggered by configured address ranges, which cause small, fully programmable event code sequences to be run that can generate new prefetches. As each event is separate from the previous one, these are extremely parallel, enabling highly efficient and performant execution on a number of tiny programmable units. The ability to react to previous prefetches, which cannot be achieved by other schemes with programmability, such as software prefetching [12] or helper threads [27], allows irregular patterns, which typically feature multiple dependent accesses, to be prefetched without stalling.

4.1 Overview

Figure 3 shows the overall architecture of our design. We add programmable units and supporting hardware to generate prefetches based on a program's current and future working set. The prefetcher is event-based to avoid stalls while still enabling further fetches from the results of earlier prefetches.

All snooped reads from the main core, and prefetched data reaching the L1 cache, initially go into an address filter (section 4.2). Data which has been filtered to be of interest moves into the observation queue, to be removed by the scheduler (section 4.3) when it detects a free programmable prefetch unit (PPU, section 4.4). These programmable units are low frequency, in-order cores that execute a small, custom computation for each address received from the scheduler, and generate zero or more prefetches as a result. They use data from load events, along with configured state in global registers, and lookahead distances calculated by the EWMA calculators (section 4.5), to generate new prefetches, which are placed into a FIFO prefetch request queue (section 4.6). When the L1 cache has available MSHRs, it removes a prefetch request and issues it to the L2 cache. In cases where the prefetcher is unused, it can be powered off to avoid impacting performance. The following subsections describe each structure in more detail.

4.2 Address Filter

The address filter snoops all loads coming from the main core, and prefetched data brought into the L1 cache from the L2. This filter holds multiple address ranges that we wish to monitor and use to create new prefetches, for example the hash table (`htab`) in the kernel from figure 1. The address filter is configured through explicit address bounds configuration instructions running on the main core. These instructions are generated by the compiler or programmer when creating the code that executes on the PPUs.

The configuration is stored in the filter table. It stores virtual address ranges for each important data structure, along with two function pointers to small computation kernels: *Load Ptr*, to be run when a load is observed in that range, and *PF Ptr*, to be run when a prefetch to that range is completed. Some are also used for scheduling purposes (section 4.5), and these are marked in the table.

Filtered addresses (observations) are placed in the observation queue along with their function pointers and, in the case of a prefetch observation, the prefetched cache line. Address ranges can overlap; an address in multiple ranges stores an entry for each in the queue.

4.3 Observation Queue and Scheduler

Filtered addresses are placed in a small observation queue before being assigned to a core. The queue is simply a FIFO buffer to hold observations until a PPU becomes free. As prefetches are only performance enhancements, in the event of this queue filling up, old observations can be safely dropped with no impact on correctness of the main program.

Once a PPU becomes free, the scheduler writes the cache line and virtual address of the data into the PPU's registers, then sets the PPU's program counter to the registered prefetch kernel for that observation, starting the core. The scheduler's job is simply to monitor the PPUs and assign them work from the FIFO observation queue when required.

4.4 Programmable Prefetch Units (PPUs)

The PPUs are a set of in-order, low power, programmable RISC cores attached to the scheduler of the prefetcher, and

are responsible for generating new prefetch requests. The PPUs operate on the same word size as the main core so that they can perform address arithmetic in one instruction.

Prefetcher units are paused by default. When there is data in the observation queue, and a free PPU, the scheduler sends the oldest observation to that PPU for execution. The PPU runs until completion of the kernel, which is typically only a few lines of code. During execution it generates a number of prefetches, which are placed in the prefetch request queue, then sleeps until being reawakened by the scheduler.

Attached to the PPUs is a single, shared, multi-ported instruction cache. PPUs share an instruction cache between themselves, but not with the main core; PPU code is distinct from the main application, but any observation can be run on any PPU. The amount of programmable prefetch code required for most applications is minuscule, so instruction cache size requirements are minor: in the benchmarks described in section 7 a maximum of 1KB is fetched from main memory by the PPUs for the entirety of each application.

The PPUs have no load or store units, and therefore have no need for a data cache. They are limited to reading individual cache lines that have been forwarded to them, local register storage, and global prefetcher registers. Removing the ability to access any other memory reduces both the complexity of the PPUs and the need for them to stall. Although this limits the data that can be used in prefetch calculations, we have not found a scenario where any additional data is required. Typically the prefetch code will simply take some data from the cache line, perform simple arithmetic operations, then combine it with global prefetcher state, such as the base address of an array, to create a new prefetch address. Having no additional memory also means that each PPU has no stack space for intermediate values, but registers are available and provide ample storage for temporary values. In practice we have not found this to be an issue.

4.5 Moving Average (EWMA) Calculators

For some applications, the lookahead distance for prefetches cannot be set using a fixed value. It may be input dependent, and may vary based on the timing statistics of the particular system. In some workloads, notably breadth-first searches on graphs, the prefetch distance may vary within the computation as phases access differently-sized elements [1].

Prior research has dealt with this challenge by considering the ratio between computation and memory access times. For example, Mowry et al. [45] divide the prefetch latency by the number of instructions in the shortest path through a loop to determine the number of iterations ahead to prefetch.

We generalise this idea and perform the calculation dynamically in hardware using exponentially weighted moving average (EWMA) calculators to generate times for a variety of observed events. EWMAs can be implemented very efficiently in hardware with minor amounts of state [18], and mean that PPUs do not need to perform timing calculations. We dynamically work out the ratio between time to finish a chain of prefetches, and the time each loop iteration takes, and use that to decide how far ahead to look in the base array. This means we attempt to prefetch the element which will be accessed immediately after the prefetch is complete.

When an observed read occurs to a particular data structure, the time between this event and the previous event on the same address bound is recorded. This can give us, for example, the time between FIFO accesses for breadth-first search. To time how long loads take, we signify the start of a timed prefetch EWMA, and attach the current time to the event generated. We propagate this to resulting prefetches until we reach an address range with a flag set, then use the time between the events as input into a load time EWMA.

4.6 Prefetch Request Queue

The prefetch request queue is a FIFO queue containing the virtual addresses that have been calculated by the PPUs for prefetching, that have not yet been processed. Once the L1 data cache has a free MSHR, it takes the oldest item out of this queue, translates it to a physical address using the shared TLB, then issues the prefetch to this address. As with the observation queue, old requests can be dropped if the queue becomes full, without impacting application correctness.

4.7 Memory Request Tags

While array ranges, which can be captured by virtual address bounds, can be identified easily by the configuration steps discussed in section 4.2, these aren't the only structures a prefetcher needs to react to. Linked structures (e.g. trees, graphs, lists) can be allocated element-by-element in non-contiguous memory regions and require identification when their prefetched data arrives into the cache. To deal with these we store a single tag in the MSHR that identifies the data structure that the prefetch targets, such as a hash-table bucket's linked list. When a prefetch request returns data, and has a registered tag, the cache line is sent to a PPU loaded with the function pointer for that structure.

4.8 Hardware Requirements

Though the prefetcher features many programmable units, each one of these is a very small, microcontroller sized unit, such as the ARM Cortex M0+, which contains fewer than 12,000 gates [7] (approximately 50,000 transistors). Using public data [4, 5], on comparable silicon processes we should expect the hardware impact of the twelve cores to be approximately 1.3% of the area of a Cortex A57 without shared caches. In practical implementation terms, it may be desirable to support 64-bit operations on these cores, and thus we could expect area to double to 2.6%. When we add 8.5KiB memory [56] for the instruction cache, global registers, prefetch request queue and observation queue, the total overall area overhead is still only 3%. This is comparable in size to an L1 data cache.

```
1 int64_t acc = 0;
2 for(x=0; x<N; x++) {
3   acc += C[B[A[x]]];
4 }
5 return acc;
```

(a) Main program

```
1 void on_A_load() {
2   Addr a = get_vaddr();
3   a += 128;
4   prefetch(a);
5 }
```

```
1 void on_A_prefetch() {
2   int64_t dat = get_data();
3   Addr fetch = get_base(1)
4           + dat * 8;
5   prefetch(fetch);
6 }
```

```
1 void on_B_prefetch() {
2   int64_t dat = get_data();
3   Addr fetch = get_base(2)
4           + dat * 8;
5   prefetch(fetch);
6 }
```

(b) PPU code

Figure 4. A loop with irregular memory accesses to arrays B & C, but significant memory-level parallelism for accesses to A. Also shown are the macros executed by the PPUs to exploit this MLP.

4.9 Summary

We have developed a programmable prefetcher that responds to filtered load and prefetch observation events. These feed into a set of programmable units, which run kernels based on the events to issue prefetches into the cache. We omit coverage of prefetching for multicores here for brevity, but the implementation and expected results are similar [1, 2]. The following sections describe how these are programmed.

5 OS and Application Support

To target the prefetcher, custom code must be generated for each application. This section describes the event-based programming model used for this, that is suited for latency tolerant fetches on multiple PPUs. It also considers the interaction with the operating system and context switches. In this section we assume prefetch code is written by hand. We then go on to consider compiler assistance in section 6.

5.1 Event Programming Model

The PPU programming model is event-based, which fits naturally with the characteristics of prefetch instructions that have variable latency before returning data. Events generate prefetches rather than loads, which can then be reacted to by new events when they arrive in the core. These are issued to the memory hierarchy when resources become available, as described in section 4. This is naturally latency-tolerant, avoiding PPU stalls while waiting for prefetched data.

Events run on the PPUs are determined from the addresses loaded or prefetched into the cache. If and when prefetches return data, the scheduler can select any PPU to execute the corresponding event, rather than being constrained to the originating unit. This makes the architecture suitable for prefetches requiring loads for intermediate values, which would otherwise stall the prefetcher. A benefit of this style of programming is that the PPUs do not need to keep state between computations on each event.

The code for each event resembles a standard C procedure for a more traditional processor, with some limitations. There are no data loads from main memory, stores or stack storage, because PPUs do not have the ability to access memory (apart from issuing prefetches). The only data available to the PPUs is the address that triggered the event, any cache line which has been observed (stored in local registers), and global prefetcher state (stored in global registers, such as address bounds or configured values such as hash masks).

We add special prefetch instructions to the PPUs, which are different from software prefetches because they trigger subsequent events for the PPUs to handle once they return with data. Function calls cannot be made, since there is no stack, and system calls are unsupported.

The prefetch events can be terminated at any time, since they are not required for correct execution of the application running on the main core. This happens, for example, on a context switch when the current application is taken off the main core. At this time, all PPUs are paused and their prefetch events aborted. In addition, any operation that would usually cause a trap or exception (e.g., divide by zero) immediately causes termination of the prefetch event.

5.2 Example

Consider the program in figure 4(a). Its data accesses are highly irregular, featuring indirect accesses to arrays `B` and `C`. However, the sequential access of array `A` means there is a large amount of memory level parallelism we can exploit to load in each iteration over `x` in parallel.

This can be prefetched by loading the PPUs with the code in figure 4(b). We assume that `A`, `B` and `C` are all arrays of 8-byte values. The address bounds of arrays `A`, `B` and `C` are configured with the prefetcher as address bounds 0, 1 and 2 respectively, by placing instructions in the original code. Similarly, the addresses of the kernels in figure 4(b) are taken, and configured to the relevant load events for the prefetcher. On observation of a main program read to `A`, a prefetch event is triggered which fetches the address two cache lines ahead of the current read. On prefetch of this, the fetched data is used as an index into `B` (`get_base(1)`), then into `C` (`get_base(2)`).

Note that the prefetcher code is a transformation from a set of blocking loads to a set of non-blocking prefetch events. The core code for the main program remains sequential and unchanged save for the configuration instructions, but the majority of cache misses should be avoided from the PPUs issuing load requests in advance of the core program.

The prefetcher functions (`get_vaddr()`, `get_base()` and `get_fetched_data()`) are compiler intrinsics that get converted into register reads, or loads from the attached small, shared, prefetcher-state memory, as appropriate.

```
int64_t acc = 0;
for(x=0; x<N; x++) {
  swpf(&C[B[A[x+n]]]);
  acc += C[B[A[x]]];
}
return acc;
```

(a) Software prefetch

```
int64_t acc = 0;
#pragma prefetch
for(x=0; x<N; x++) {
  acc += C[B[A[x]]];
}
return acc;
```

(b) Pragma

Figure 5. Source for auto-generation of PPU code.

5.3 Operating System Visibility

Although they have many capabilities of regular cores, PPUs are not visible to the operating system as separate cores, and so the OS cannot schedule processes onto them. Instead, the OS can only see the state necessary to be saved across context switches. Although there may be situations where it is useful for the OS to see the PPUs as full cores, avoiding interactions with the OS simplifies their design (for example, it does not require privileged instructions). Therefore, while the prefetcher initiates page table walks, it cannot handle page faults, so in this case we discard the prefetch.

The prefetch units are used only to improve performance and cannot affect the correctness of the main program. Therefore, the amount of state that needs to be preserved over context switches is small. For example, we do not need to preserve internal PPU registers, but simply discard them on a context switch. For the same reason, we can also throw away all events in the observation queue and addresses in the fetch queue. Provided context switches are infrequent, this will result in little performance drop. EWMA values aren't necessary over context switches, as they can be recalculated.

As a result, all that is required to be saved on a context switch is the prefetcher configuration: the global registers and the address table.

6 Compiler Assistance

Hand-coding events requires considerable manual effort. A way of generating these events from the original code within the compiler is more desirable for an end-user.

Software prefetching[2, 12] is a commonly supported technique whereby a processor can load into the cache system without waiting for the result. These present a high level abstraction for the end user, but have many disadvantages when executed directly, as discussed in section 3. However, we can use the address generation code for these to generate hardware events by working backwards through the loop in which they appear to generate programmable prefetcher code. This allows us to perform the prefetching without slowing down the main computation thread.

We provide compiler passes, implemented in LLVM [36], to both convert software prefetches to programmable events, and also to generate events from scratch, simply by adding a pragma to loops where the programmer requires prefetching, giving a spectrum of techniques trading off manual effort for performance. Pseudocode is given in algorithm 1.

```
// Collect initial software prefetches and their
// address generation instructions.
prefetches = {}
foreach (p: software prefetches within a loop):
  // Search backwards from a prefetch, to find an
  // induction variable.
  if (((indvar, set) = DFS(p)) != null):
    prefetches U= {(p, indvar, set)}

// Function calls allowed if side-effect free.
remove(prefetches, contains function calls)
// Non-induction variable phi nodes allowed if pass
// can cope with complex control flow.
remove(prefetches, contains non-induction phi nodes)

all_events = {}
// Emit prefetches and address generation code.
foreach ((pf, iv, set): prefetches):
  // Find loop invariant loads, for removal and
  // configuration
  loop_invars = get_loop_invariant_loads(set)
  // Attempt to replace invariant loads in events.
  set.replace(loop invariant loads, global reg)

  // Split into single events with only single load
  // references inside.
  events = split_on_loads(set)
  if(!events) continue

  // Find address bounds, replace induction variable.
  addrbounds = infer_bounds(iv)
  events.replace(iv, (addrbounds.max - addr) / size)
  events.replace(final load, prefetch)
  events.replace(first load, prefetch read)

  if(loads still appear in any event) continue

  // Configure prefetcher in original program.
  add_address_bounds_config(addrbounds)
  add_global_register_config(loop_invars)
  all_events U= events

  // Remove unnecessary software prefetch in
  // original program.
  remove(pf)
  dead_code(events, original code)

add_to_event_list(all_events)
```

Algorithm 1. The software prefetch conversion algorithm, assuming the intermediate representation is in SSA form.

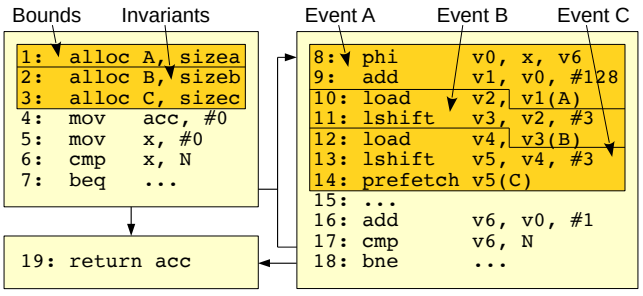


Figure 6. An overview of our software prefetch conversion algorithm on the control flow graph from code in figure 5.

6.1 Analysis

Our analysis pass over the compiler's IR starts from a software prefetch instruction and works backwards using a

depth-first analysis of the data-dependence graph. We terminate upon reaching a constant, loop-invariant value, non-loop-invariant load, or phi node. The goal is to split prefetch address generation into sequences of nodes ending in a single load, to be turned into PPU events in a later pass.

To attain an appropriate level of look-ahead for the PPU code, the software prefetch instruction must be in a loop with an identifiable induction variable. We also need a data structure which is accessed using the induction variable, so that we can infer its value from loads observed in the cache.

Phi nodes identify either the loop's induction variable, or another control-flow dependent value. In the former case, provided no loads have been found in this iteration of the depth-first search, we can replace the induction variable with code to infer it from an address, and use the set of found instructions as the first event for a set of prefetches. The latter case requires more complex analysis, and in practice is rare, so we do not discuss it further.

If multiple different non-loop-invariant loads are found in a search, then more than one loaded value is used to create an address and the event cannot be triggered by the arrival of a single data value. In this case the conversion fails. However, if only one load is found, we package the instructions into an event, and repeat the analysis again starting from the load.

Figure 6 shows the control-flow graph for the code in figure 5(a). Analysis starts from the *prefetch* instruction (line 14), performing a depth-first search on its input, *v5*, and terminating upon reaching the *load* at line 12. Since this is a non-loop-invariant load, the three instructions are packaged together into an event, and analysis restarted with the *load*. This terminates on with the *load* at line 10, and again an event is created. Finally, the third analysis pass terminates with the *phi* node, which is for the loop induction variable, so a new event is created and no further analysis is required.

6.2 Array Bounds Detection

The prefetcher requires the address bounds for each array accessed through an induction variable, storing them in its address filter so as to trigger the correct event when snooping a load or prefetch. For example, in figure 6 code for event A must be executed when observing a load to array A by the main core. Returned prefetches are handled using the memory request tags, described in section 4.7.

The start of each array is trivially obtained from address generation instructions and, in the case of a typed array, the end address is also simple because the size of the array is stated explicitly. However, in languages such as C, where arrays can be represented as pointers, this becomes more challenging. One option is to pattern match for common cases, for example, searching backwards for allocation instructions. Another is to identify the loop termination condition, provided that it is loop invariant.

6.3 Code Generation

The tasks of the code generation pass are to insert prefetcher configuration instructions, generate PPU code and remove the original software prefetch instructions. Using the analysis described in section 6.2, array bounds are known and so configuration instructions for each array are placed immediately before the loop. Configuration instructions are also added for any loop invariant values required by the PPU code, assigning them to unique prefetcher global registers.

To generate prefetcher code, we take sets of instructions identified using the analysis in section 6.1, and turn them into event functions. In the first event, we replace the induction variable *phi* node with the current address observation (accessible from PPU registers) subtracted from the base array address and divided by the size of the array's elements (which is typically converted to a shift). We replace the final instruction in each event, which will either be a load or software prefetch, with a hardware prefetch instruction. For loads, we add a callback so that the next event in the sequence is called once this prefetch returns. We replace all loop invariants with global register accesses to values configured in the main code. The only remaining load must be to the data observed from the current prefetch or load event, so it can be converted into a register access.

Finally, we remove the now-unnecessary software prefetch instructions. Dead-code elimination is then used to remove any code that was only used for a software prefetch, leaving common subexpressions for still-required instructions.

6.4 Pragma Prefetching

While software prefetches are a relatively descriptive mechanism for converting to hardware events, this still involves some manual effort. One option is to let the compiler deal with generating the initial software prefetches [2], which can be converted into events. However, a simple and more direct option is to simply indicate the loop that requires prefetching within it and let the compiler generate the prefetch events from scratch. We support this through a custom prefetch pragma (as in figure 5(b)) using a similar depth-first search approach as in section 6.1. We start the analysis with loads that feature indirection (so are likely to miss), and that have look-ahead based on a discovered induction variable.

Generating code in this manner means we have less information to work on than with the software prefetch pass, which can encode runtime information on what data will miss and be accessed that a simple pragma over a loop can miss (e.g., an array access stride pattern). Further, it isn't possible to decide at compile time, without more information, which loads are likely to access data that is already in the L1 cache, and thus prefetches to that data structure are unnecessary (though these could be disabled at runtime with analysis hardware). However, for simple patterns, this descriptor is equally powerful as software prefetch conversion.

Main Core	
Core	3-Wide, out-of-order, 3.2GHz
Pipeline	40-Entry ROB, 32-entry IQ, 16-entry LQ, 32-entry SQ, 128 Int / 128 FP registers, 3 Int ALUs, 2 FP ALUs, 1 Mult/Div ALU
Tournament Branch Pred.	2048-entry local, 8192-entry global, 2048-entry chooser, 2048-entry BTB, 16-entry RAS
Memory Dep.	Store set predictor [16]
Memory & OS	
L1 Cache	32KB, 2-way, 2-cycle hit lat, 12 MSHRs
L2 Cache	1MB, 16-way, 12-cycle hit lat, 16 MSHRs
L1 TLB	64-Entry, fully associative
L2 TLB	4096-Entry, 8-way assoc, 8-cycle hit lat
Table Walker	3 Active walks
Memory	DDR3-1600 11-11-11-28 800MHz
OS	Ubuntu 14.04 LTS
Prefetcher	
Prefetcher	40-Entry observation queue, 200-entry prefetch queue, 12 PPUs
PPUs	In-order, 4 stage pipeline, 1GHz, shared 4KiB instruction cache (2 ports)
Stride Prefetcher	Reference Prediction Table [13], degree 8
GHB Prefetcher	Markov GHB G/AC Prefetcher [48], depth 16, width 6, index/GHB sizes 2048/2048 (regular) and 67108864/67108864 (large)

Table 1. Core and memory experimental setup.

7 Evaluation

To evaluate our prefetcher we modeled a high performance system using the gem5 simulator [10] in full system mode running Linux with the ARMv8 64-bit instruction set and configuration given in table 1, similar to that validated in previous work[21]. We implemented the compiler techniques presented in section 6 as LLVM passes [36], and compiled our benchmarks using Clang with the O3 setting. We chose a variety of memory-bound benchmarks to demonstrate our scheme, representing a wide range of workloads from different fields: graphs, databases and HPC, described in table 2. We skipped initialisation, then ran each benchmark to completion using detailed, cycle-accurate simulation. In addition, we compare against a Markov global history buffer [48] with regular settings, designed to be realistic for implementing in SRAM, and a version with a large amount of history data (1GiB), to evaluate the maximum potential improvement from more modern history prefetchers [25, 55] which keep state in main memory. To isolate the improvements from this technique, we provide unlimited bandwidth and zero latency accesses to the Markov baseline's state.

7.1 Performance

Figure 7 shows that our programmable prefetcher achieves speedups of up to 4.3× with **manual** programming, compared to no prefetching, for the memory-bound workloads

Benchmark	*Source*	*Pattern*	*Input*
G500-CSR	Graph500 [46]	BFS (arrays)	-s 21 -e 10
G500-List	Graph500 [46]	BFS (lists)	-s 16 -e 10
PageRank	BGL [53]	Stride-indirect	web-Google
HJ-2	Hash Join [11]	Stride-hash-indirect	-r 12800000 -s 12800000
HJ-8	Hash Join [11]	Stride-hash-indirect, linked list walks	-r 12800000 -s 12800000
RandAcc	HPCC [39]	Stride-hash-indirect	100000000
IntSort	NAS [9]	Stride-indirect	B
ConjGrad	NAS [9]	Stride-indirect	B

Table 2. Summary of the benchmarks evaluated.

described in section 7, whereas **stride** and **software** prefetchers speed up by no more than 1.4× and 2.2× respectively.

The Markov global history buffer [48] gains no speedup with **regular** settings, since the applications we evaluate access far too much data to be predicted with such a small amount of state. When we increase the amount of state (**large**) to 1GiB of data, we still only gain performance for benchmarks which access a small amount of data (G500-List,ConjGrad). Other applications either access too much data, even for a very large history buffer, or don't repeat memory accesses, so gain no benefit from the technique.

Our compiler-assisted software prefetch conversion pass (**converted**) achieves similar speedups to manually written events for benchmarks except for on the Graph500 workloads, and our automatic event generation technique based on pragmas (**pragma generated**) is able to speed up simpler access patterns as much as manual, but isn't able to achieve full potential for four of our eight benchmarks.

Speedups Three benchmarks gain significant improvement from software prefetching. These are RandAcc, IntSort and HJ-2, all highly amenable to software prefetching due to their access pattern, which involves an array indirect based on a single strided load. The spatial locality means that they don't incur large numbers of pipeline stalls for the prefetch address calculation. However, in the extreme (IntSort), software prefetching causes a 113% dynamic instruction increase (with 83% extra for RandAcc and 56% for HJ-2).

In contrast, moving the prefetch address calculations to PPUs in our scheme results in larger speedups: from 2.0× with software prefetch up to 2.8× with PPUs for IntSort, from 2.2× to 3.0× for RandAcc and from 1.4× to 3.9× for HJ-2. In other workloads, where stride and software prefetch provide few benefits, our prefetcher is able to unlock more memory-level parallelism and realise substantial speedups. For example, in HJ-8 stride and software prefetching speedups are negligible, yet our PPUs attain 3.8×.

The only significant outlier is G500-List, which, although achieving 1.7×, is the lowest speedup attained by our prefetcher. The reason for this is that there is no fine-grained

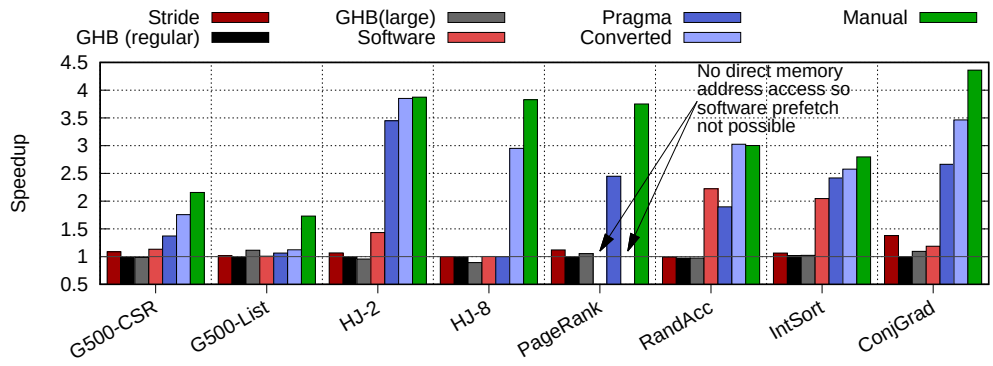
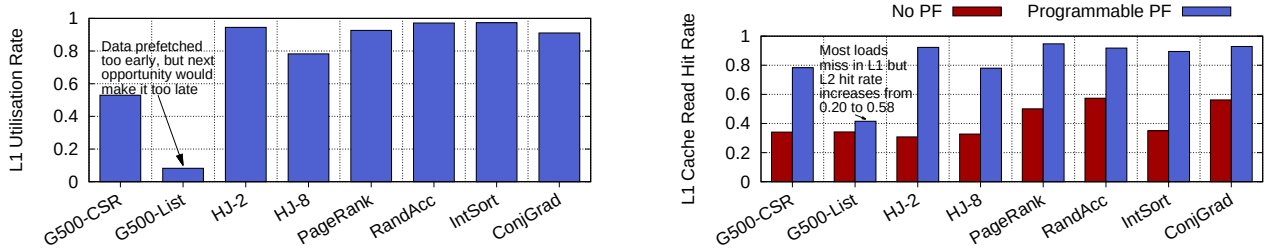


Figure 7. The programmable prefetcher realises speedups of up to 4.4×. Stride, GHB and software prefetchers cannot effectively prefetch highly irregular memory accesses.



(a) Proportion of prefetches utilised before eviction from L1

(b) L1 read hit rate

Figure 8. While most applications see high prefetch utilisation and L1 hit rates, G500-List has to prefetch data too early to attain memory-level parallelism, so benefits are obtained from having the data in the L2 cache.

parallelism available within the application, since each vertex in the graph contains a linked list of out-going edges. Therefore, when prefetching a vertex, each edge can only be identified through a pointer from the previous, essentially sequentialising the processing of edges. By comparison, Peled et al. [50] achieve no improvement on the same benchmark.

There is no bar for software prefetching or conversion for PageRank in figure 7; the Boost Graph Library code uses templated iterators which only give access to edge pairs, meaning it isn't possible to get the addresses of individual elements to issue software prefetches to them.

Compiler assistance from both pragmas and software prefetch conversion, works well for IntSort, ConjGrad and HJ-2. While PageRank's code doesn't allow software prefetch insertion due to working on high level iterators, this is not a problem for the pragma pass, which works on LLVM IR, and thus can discover the access pattern and generate events automatically. IntSort, ConjGrad and PageRank have slightly reduced performance from pragma generated prefetching, as a result of useless prefetches being generated, as opposed to the patterns not being discoverable.

RandAcc gains less performance from pragma conversion than from manual software prefetching. This is because the benchmark repeatedly iterates over a small 128-entry array, and thus we can encode wrap-around prefetches in a software prefetch. As this is a property of multiple control flow loops, it is difficult to discover in an automated pass, and thus our scheme leaves the first few entries of the array unprefetched. Still, our pragma scheme requires less effort from the programmer than a software prefetch, in that they only need to identify target loops, rather than come up with specific prefetches and look-ahead distances.

HJ-8 gains significant performance improvement from software prefetch conversion, because we can specify to prefetch the first N hash buckets. This differs from software prefetching, where we cannot do this in a latency tolerant manner, as it requires reads of prefetched data, and also from pragma generation, as N cannot easily be discovered from the code. More generally, we can say that hash tables tend to have few elements per hash bucket, so even for the case where there are varying numbers of elements, a conservative "first N" approach should work well. Still, with manual prefetching, we can introduce control flow loops, to walk every bucket until we try to prefetch a null pointer.

G500-CSR gains progressively more performance with increasing programmer effort expended in prefetching. As neither of the compiler passes deal with control flow (as software prefetches fundamentally can't express loops), it isn't possible to prefetch a data-dependent range of edges, and thus we must instead fetch the first N for fixed N. Further, we can't use the knowledge that the start and end value for each vertex in an edge list will be in the same cacheline

in our compiler passes, as they assume access to only one loaded value at a time. The pragma pass is unable to identify the need to fetch edge or visited values from vertex data, due to the complicated control flow involved, so instead only achieves two stride-indirect patterns from FIFO queue to vertices, and edges to visited information, limiting the prefetching achievable. Still, even this is significantly higher than other recent work [50], which achieves less than 10% on the same benchmark.

As G500-List relies heavily on walking long edge lists in a linked list, it requires loop control flow to prefetch effectively. Therefore, we cannot express it as a software prefetch, and our compiler passes have limited impact.

Impact on L1 Cache Figure 8 explores this in more detail. Figure 8(a) shows that while L1 cache utilisation is high for most benchmarks when using our prefetcher, it is comparatively low for G500-List. In this application, for larger vertices, the linked list of edges may be larger than the L1 cache. Traversing this list may result in prefetched data being evicted from the cache before being used due to capacity misses from either a) later prefetches to the same edge list, or b) prefetches or loads to other data. The underlying issue is that the prefetches occur too early, however there is no mechanism to delay them. Instead of starting the edge-list prefetches after a vertex has been prefetched, the only other point that the list prefetches can start is when the actual application thread starts processing the vertex. By this point it is too late because the main thread will need to follow the edges, and so prefetches will execute in lock-step with the main application's loads (much like figure 2(d)).

The L1 cache read hit rate does increase for G500-List, as shown in figure 8(b), but only up to 0.42 from 0.34. However, despite this, the application does gain some benefit from the early edge-list prefetches by virtue of these edges being placed in the L2 cache. In this case, the L2 cache hit rate increases from 0.20 to 0.57.

7.2 Analysis

Our existing programmable prefetcher configuration contains 12 PPUs, each running at 1GHz, compared to 3.2GHz for the main core. We now show that this realises most of the benefits and that scaling continues with increasing numbers of PPUs and their frequencies, since the prefetch kernels are embarrassingly parallel.

Clock Speed Figure 9 shows how PPU clock speed affects each benchmark and the impact of reducing the number of PPUs. Figure 9(a) demonstrates that approximately half the workloads gain little benefit from increasing the frequency of the PPUs. On the other hand, HJ-2 requires a 500MHz frequency to realise its maximum speedup whereas ConjGrad and G500-CSR achieve speedups that continue scaling with the PPU frequency. Overall, the majority of the benefits are obtained at 1GHz where the geometric mean of speedups is 3×, increasing to 3.1× at 2GHz.

Number of PPUs We explore the relationship between PPU frequency and the number of PPUs in figure 9(b) for G500-CSR, chosen as an example of an application that continues scaling with frequency increases. We show PPU frequencies up to 4GHz as a study only, to assess this relationship; we do not expect PPUs to be clocked at this frequency.

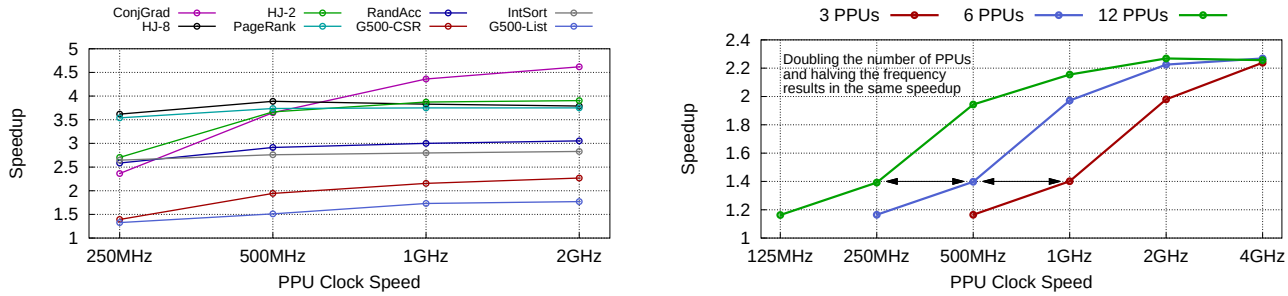
The figure shows that speedups are maintained by doubling the number of PPUs and halving the frequency. Using 3 PPUs at 2GHz, 6 PPUs at 1GHz or 12 PPUs at 500MHz all achieve 1.9×. The prefetch kernels running on the PPUs are embarrassingly parallel, since each invocation is independent of all others, meaning that scaling can be achieved by increasing the number of PPUs or their frequencies. It also shows that performance for this workload saturates with 12 PPUs at 2GHz: no more is gained by increasing frequency.

PPU Activity Figure 10 further explores the amount of work performed by the 12 PPUs at 1GHz. This figure shows the proportions of time that each PPU is awake during computation. Our scheduling policy is to pick the PPU with the lowest ID from those available when assigning prefetch work. This means that the low-ID PPUs are active more of the time than the high-ID PPUs. Other scheduling policies (such as round-robin) would spread the work out more evenly, but would not change the overall performance and would not allow us to perform this analysis.

When the workload is prefetch-compute bound, adding more PPUs or increasing clock speed would improve performance (as in G500-CSR); work is evenly split between PPUs and all are kept busy. In contrast, benchmarks such as PageRank, RandAcc and IntSort cannot fully utilise all PPUs: all of these workloads contain at least one PPU that is never awoken. This is mainly due to them requiring only simple calculations to identify future prefetch targets. These applications would achieve similar performance with slower PPUs (as shown in figure 9(a)) or fewer of them.

ConjGrad is an outlier in that some PPUs do little work, yet it scales with increasing frequency (figure 9(a)). The reason for this behaviour is that at 1GHz there is not enough work available for all PPUs to need to be active, but the prefetches are slightly latency-bound. Therefore minor additional benefits are gained when the clock speed increases and the prefetch calculations finish earlier. This is in contrast to G500-CSR, which also scales with the clock speed, where boosting frequency increases the number of prefetches that can be carried out, resulting in higher performance.

No applications have PPUs that run continuously: the maximum activity factor is 0.82. This reflects the fact that PPUs only react to events from the main core, and so are not required during phases where no data needs to be prefetched.



(a) Clock frequency impact

(b) Effect of number of cores on G500-CSR

Figure 9. Some applications see little performance loss with slower PPUs, whereas others continue gaining as clock speeds increase. Doubling the number of PPUs is the same as doubling their frequency.

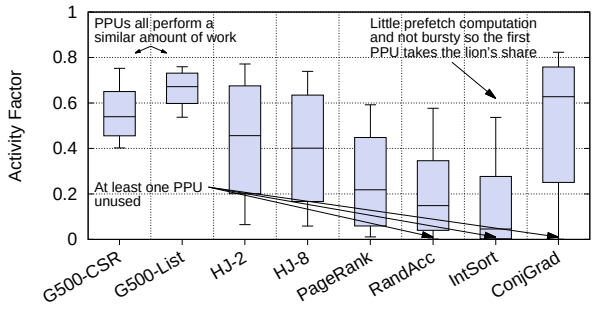


Figure 10. Range, quartiles and median for the fraction of time each PPU is awake and calculating prefetches at 1GHz.

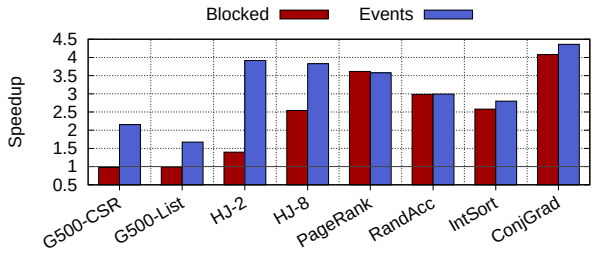


Figure 11. Performance with and without blocking on intermediate loads, with 12 fetcher units.

Extra Memory Accesses For efficient execution, it is desirable to minimise the total extra traffic we add onto the memory bus. In general, a programmable solution should prefetch very efficiently, only targeting addresses that will be required by the computation. For all but the two Graph500 benchmarks, the value is negligible: prefetches are very accurate and timely, and therefore do not fetch unused data. G500-List adds 40% extra accesses due to the lack of fine-grained parallelism available. This is down to a fundamental constraint on the linked list that limits timely prefetching, as discussed in section 7.1. G500-CSR, with 16% extra memory accesses, has variable work per vertex, meaning prefetch distance must be overestimated relative to the EWMAs.

Event Triggering To examine how much of the performance we attain is through the latency-tolerant event-based programming model, we extended the system to support blocking on loads for data used in a further calculation: if a prefetch is the last in a chain, then the core is made available for scheduling, but otherwise must stall, as is necessary without event triggering. The results are shown in figure 11. Where the pattern is a simple stride-indirect, performance is relatively close: we only have to stall on the stride access, and can mitigate the overhead of stalling by prefetching an entire cache line on a single thread, causing a stall for every 8 accesses. This means the memory-level parallelism is still high. However, when this is not the case, performance drops dramatically. In complicated access patterns, stalling limits or even entirely removes the performance gain from prefetching, and latency-tolerant events are necessary for the system to work, even with the large amount of parallelism available from twelve cores.

8 Conclusion

We have presented a programmable prefetcher, which uses an event-based programming model capable of extracting memory-level parallelism and improving performance for a variety of irregular memory-intensive workloads. On a selection of graph, database and HPC workloads, our prefetcher achieves an average 3.0× speedup without significantly increasing the number of memory accesses. We have further provided compiler techniques to reduce the amount of manual effort for the programmer to utilise the performance benefits of our scheme, with average 1.9× and 2.5× speedup for the two schemes we present.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/M506485/1, and ARM Ltd. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.17392.

References

[1] S. Ainsworth and T. M. Jones. Graph prefetching using data structure knowledge. In *ICS*, 2016.

[2] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses. In *CGO*, 2017.

[3] H. Al-Sukhni, I. Bratt, and D. A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *PACT*, 2003.

[4] AnandTech. http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6, .

[5] AnandTech. http://www.anandtech.com/show/8542/cortexm7-launches-embedded-iot-and-wearables/2, .

[6] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *ISCA*, 2001.

[7] ARM. http://www.arm.com/products/processors/cortex-m/cortex-m0plus.php.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10), Oct. 2009.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel benchmarks – summary and preliminary results. In *Supercomputing*, 1991.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), Aug. 2011.

[11] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[12] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, 1991.

[13] T. Chen and J. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[14] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS*, 1992.

[15] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst.*, 22 (2), May 2004.

[16] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA*, 1998.

[17] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.

[18] P. Demosthenous, N. Nicolaou, and J. Georgiou. A hardware-efficient lowpass filter design for biomedical applications. In *BioCAS*, Nov 2010.

[19] B. Falsafi and T. F. Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1), 2014.

[20] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *PACT*, 2006.

[21] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *ISPASS*, 2014.

[22] T. J. Ham, J. L. Aragón, and M. Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *MICRO*, 2015.

[23] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *MICRO*, 2016.

[24] C.-H. Ho, S. J. Kim, and K. Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *ISCA*, 2015.

[25] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[26] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[27] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *ASPLOS*, 2002.

[28] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.*, 22(3), Aug. 2004.

[29] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *MICRO*, 2016.

[30] O. Kocberber, B. Falsafi, K. Lim, P. Ranganathan, and S. Harizopoulos. Dark silicon accelerators for database indexing. In *1st Dark Silicon Workshop (DaSi)*, 2012.

[31] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[32] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. In *VLDB*, 2015.

[33] N. Kohout, S. Choi, D. Kim, and D. Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *PACT*, 2001.

[34] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips. Sqrl: Hardware accelerator for collecting software data structures. In *PACT*, 2014.

[35] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan. Dasx: Hardware accelerator for software data structures. In *ICS*, 2015.

[36] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[37] E. Lau, J. E. Miller, I. Choi, D. Yeung, S. Amarasinghe, and A. Agarwal. Multicore performance optimization using partner cores. In *HotPar*, 2011.

[38] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01), 2007.

[39] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The hpc challenge (hpcc) benchmark suite. In *SC*, 2006.

[40] V. Malhotra and C. Kozyrakis. Library-based prefetching for pointer-intensive applications. Technical report, Online, 2006.

[41] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, 2015.

[42] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *PPoPP*, 2012.

[43] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), Aug. 2016.

[44] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediction. In *ICS*, 2001.

[45] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, 1992.

[46] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, May 5, 2010.

[47] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.

[48] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[49] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. Prefedge: Ssd prefetcher for large-scale graph traversal. In *SYSTOR*, 2014.

[50] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

[51] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS*, 1998.

[52] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[53] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

[54] V. Viswanathan. Disclosure of h/w prefetcher control on some intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, Sept. 2014.

[55] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA '05*, 2005.

[56] M. Yabuuchi, Y. Tsukamoto, M. Morimoto, M. Tanaka, and K. Nii. 20nm high-density single-port and dual-port srams with wordline-voltage-adjustment system for read/write assists. In *ISSCC*, 2014.

[57] C.-L. Yang and A. Lebeck. A programmable memory hierarchy for prefetching linked data structures. In H. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*. 2002. ISBN 978-3-540-43674-4.

[58] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. IMP: Indirect memory prefetcher. In *MICRO*, 2015.