An Event-Triggered Programmable Prefetcher for Irregular Workloads

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 generalize 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 for a variety of graph, database and HPC workloads.

1. Introduction

Many modern and emerging workloads perform computation on large amounts of data, which cannot fit in the caches of current systems. Many of these accesses are irregular and difficult to predict in advance, resulting in heavily memory-bound execution with frequent, long stalls from high DRAM latency [1–3].

There are several techniques available to address these challenges. One option is to utilize thread-level parallelism within an application, to cope with latency using aggressive multi-threading [4], effectively parallelizing loads by having many threads stalled at once. This is typical of workloads running on graphics cards, for instance [5]. 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 [6], 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 stride-based prefetchers [7, 8] only work for very regular computations, typically involving either dense matrices or entirely sequential memory accesses. History-based prefetchers [9] only work for highly repeated computation. Neither of these apply to many big-data applications, such as database operations, graph workloads and many high-performance computing (HPC) kernels, which exhibit much more complicated, irregular traversals of data, involving pointer chasing and indirect array lookups [4]. Techniques have been proposed specifically for irregular accesses, such as pointer fetching prefetchers [10], 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 [11, 12], 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 [1–3, 13, 14] through explicit configuration of traversal patterns, gaining significant performance improvements for the targeted workloads. However, currently such architectural techniques are highly specialized 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 including graphs, databases and HPC. We couple a conventional high-performance out-of-order computation core with a specialized prefetching structure for the L1 cache, attached to several in-order programmable prefetch units. The event-based programming model allows each prefetch 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 utilization of the prefetches brought into the cache, and negligible additional memory accesses for most workloads.
for(x = 0; x < in.size; x++) {
    SWPF(htab[hash(in.key[x+dist])]); // Software prefetch
    Key k = in.key[x];
    Hash h = hash(k);
    Bucket b = htab[h];
    ListElement l = b.listStart;
    while(l != NULL) {
        if(l->key == k) {
            wait_til_oldest(); // Multithreading
            out.match[out.size] = k;
            out.size++;
            l = l->next;
        }
        signal_iter_done[x]; // Multithreading
    }
}

Figure 1: Hash join kernel with two latency-hiding techniques.

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.

Fetcher Units Much of the research into efficient execution of irregular workloads has focused on highly specialized fetcher units. These systems take control of memory accesses for a particular access pattern, extracting performance through parallel loads of data, often with large performance improvements. SQRL [2] and DASX [15] are fetcher systems designed for iterative accesses of B-tree and hash table structures. Similarly, Kocberber et al. [1, 13] focus on the optimization of database inner joins by parallel hash table walking. Fetcher units can realize energy savings through removal of the original load instructions. However, they are highly specialized to specific data accesses, and the original code has to be significantly modified, yielding code incompatible with devices not featuring fetcher units.

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. [16] use explicit Harbinger instructions at the program level to control linked-list pointer fetching. Yang and Lebeck [17] 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 [14] design a configurable prefetcher specifically for graph workloads, gaining large speedups, but only targeting specific traversals for a particular graph format.

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, although significant progress has been made, none have been implemented in any commercial system [18].

Pointer-fetching prefetchers [10], 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 [19–21]. These run dynamically-detected streams on programmable units when the start of a set of loads is identified, to prefetch the data. Mutlu et al. propose a runahead scheme [22], which utilizes idle chip resources on a cache miss to dynamically 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.

Yu et al. [23] pick up stride-indirect patterns using runtime analysis of executed code on the CPU, 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 core, 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 [24] use auto-generated “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 [25]. Further, the lack of a hardware event queue makes synchronization on loads difficult and expensive. Lau et al. [26] 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 synchronization 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.

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 that we develop 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 \texttt{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 2a 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 \cite{11} can be more beneficial than using a hardware prefetcher, since we can encode the hash function inside the prefetch instruction. Figure 1 shows this instruction and its position within the code. We prefetch at a fixed number of for-loop iterations into the future (\texttt{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 2b 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 that 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 synchronization is required to prevent this.

Code for this option is shown in figure 1, and its execution on two threads is shown in figure 2c. 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 \texttt{wait_til_oldest()}; the companion \texttt{signal_iter_done()} signals at the end of each iteration to keep track of the oldest iteration currently executing.

In the example (figure 2c), 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

\(^1\)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.
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 2d. 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.

**Desired Behavior** 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 parallelize 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 stallings 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 2e where, after a warm-up period, computation can proceed without stalls, since data is immediately available in the first level cache. To realize 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

This section develops a programmable prefetcher, suitable for a wide variety of applications, but especially targeted towards workloads with irregular, yet calculable memory accesses. Figure 3 shows the overall structure. We add programmable prefetch units and supporting hardware to generate prefetches based on an application’s current and future working set. The prefetcher is event-based, to avoid stalling, yet enable further fetches to be made from the results of an earlier prefetch.

All snooped reads from the main core, and prefetched data reaching the L1 cache, initially go into an address filter. Filtered addresses move into the observation queue, to be re-

![Figure 3: Structure of the programmable prefetcher.](image)

![Figure 4: An example address filter table entry.](image)
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.

Each prefetcher unit is paused by default. When there is data 
in the observation queue, and there is 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 in-
struction cache. PPUs share an instruction cache between 
theselves, 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 extremely small, so the instruction 
cache size requirements are minor: in the benchmarks de-
scribed in section 7 a maximum of 1KB is fetched from main 
memory by the PPUs for the entirety of each application.

The PPUs do not have a load or store unit, and therefore 
have no need for a data cache. This means 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 addi-
tional 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 ad-
ress. 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.

4.4. EWMA Calculators

For some applications, the lookahead distance for issuing 
prefetches cannot be set using a fixed value. It may be input 
dependent, and may vary depending 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.

Prior research has dealt with this challenge by considering 
the ratio between computation and memory access times. For 
example, Mowry et al. [27] 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 generalize this idea and perform the calculation dy-
namically in hardware using exponentially weighted moving 
average (EWMA) calculators to generate times for a variety of 
observed events. EWMA can be implemented very efficiently 
in hardware with minor amounts of state [28], and means that 
PPUs do not need to perform timing calculations.

The computed EWMA are specified in the address filter 
table, as shown in figure 4. When an observed read occurs 
to a particular data structure, if Obs EWMA is set, 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, when PF EWMA Start is set, 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 PF EWMA End set, when 
we use the time between the two events as input into a load 
time EWMA.

4.5. 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, older requests can be dropped if the queue 
becomes full, without impacting application correctness.

4.6. 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 4.1, 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’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.

4.7. Hardware Requirements

Though the prefetcher features many programmable units, 
each one of these is intended to be a very small, 
microcontroller-style unit, such as the ARM Cortex M0, which 
contains fewer than 12,000 gates [29] (approximately 50,000 
transistors). Compared to a typical out-of-order superscalar 
CPU, for example a first generation Intel i7 processor at over 
700 million transistors [30] (or 66 million per core, excluding 
caches), associated silicon area for our prefetcher is minimal.

4.8. Summary

We have developed a programmable prefetcher that responds 
to filtered load and prefetch observation events. These feed 
into a set of programmable prefetch units, which run kernels 
based on the events to issue prefetches into the main core’s 
cache. The following sections describes how these units 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 their 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 a few limitations. There are no data loads from main memory, stores or stack storage, because the 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 a hash mask).

We add special prefetch instructions, 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 5(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 5(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 5(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 by virtue of the PPUs issuing load requests in advance of the core program reaching them.

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

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 regular cores, avoiding
interactions with the OS simplifies their design (for example, it does not require privileged instructions). As a result, while the prefetcher initiates page table walks, it cannot handle page faults, and such a 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 from an end-user point of view.

Software prefetching [11] 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 software prefetches 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.

6.1. Analysis

Our analysis pass over the compiler’s IR starts from the software prefetch instructions and works backwards as a depth-first analysis of the data-dependence graph for each input. We terminate upon reaching a constant, loop-invariant value, non-loop-invariant load, or phi node. The goal is to split the prefetch address generation into sequences of nodes ending in a single load, which will be turned into the 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 this load.

Figure 7 shows the control-flow graph for the code in figure 6(a). Analysis starts from the prefetch instruction (line 14), performing a depth-first search on its single 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 7 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 4.6.

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 prefetch 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 that are 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 by later optimizations). We replace the final instruction in each event, which will either be a load or software prefetch, with a hardware prefetch instruction. If a load, 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 that have been configured in the main code. The only remaining load must be to the data observed from the current prefetch or load event, so 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 instructions that were only required for the software prefetch, but leaves those that are common subexpressions for other, still-required instructions.

6.4. Pragma Prefetching

While software prefetches are a relatively descriptive mechanism for converting to hardware events, an easier 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 6(b)) using a similar depth-first search approach as in section 6.1. We start the analysis with loads that feature indirect (so are likely to miss), and that have look-ahead based on a discovered induction variable.

<table>
<thead>
<tr>
<th>Main Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
</tr>
<tr>
<td>Pipeline</td>
</tr>
<tr>
<td>Tournament</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory &amp; OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>L1 TLB</td>
</tr>
<tr>
<td>L2 TLB</td>
</tr>
<tr>
<td>Table Walker</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>OS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefetcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefetcher</td>
</tr>
<tr>
<td>PPUs</td>
</tr>
</tbody>
</table>

Table 1: Core and memory experimental setup.

Generating code in this manner means we have slightly less information to work on than with the software prefetch pass, since software prefetches can encode runtime information on what data will miss and be accessed, which a simple pragma over a loop can miss (e.g., an array access stride pattern). Further, it’s impossible 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.

7. Evaluation

To evaluate our prefetcher we modeled a high performance system using the gem5 simulator [31] in full system mode running Linux with the ARMv8 64-bit instruction set and configuration given in table 1. To evaluate the compiler techniques presented in section 6, we implemented them as LLVM passes [32]. 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 initialization, then ran each benchmark to completion using detailed, cycle-accurate simulation.

7.1. Performance

Figure 8 shows that our programmable prefetcher achieves speedups of up to 4.3× with manual programming, compared to no prefetching, for the memory-bound workloads described in section 7, whereas stride and software prefetchers realize speedups of no more than 1.4× and 2.2× respectively.

Our compiler-assisted software prefetch conversion pass (converted) achieves similar speedups to manual 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 significant numbers of pipeline stalls for the prefetch address calculation. However, in the extreme (IntSort), software prefetching causes a 113% increase in dynamic instructions, incurring significant numbers of pipeline stalls for the prefetch address calculation. Therefore, when prefetching a vertex, each edge can only be identified through a pointer from the previous, essentially sequentializing the processing of edges.

There is no bar for software prefetching or conversion for PageRank in figure 8; 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, both from 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 memory-level parallelism and realize substantial speedups. More generally, we can say that hash tables tend to have few memory-level parallelism and realize substantial speedups.
programmer effort is 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.

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 on performance.

Impact on L1 Cache  Figure 9 explores this in more detail. Figure 9(a) shows that while L1 cache utilization 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 9(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 realizes 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 10 shows how PPU clock speed affects each benchmark and the impact of reducing the number of PPUs. Figure 10(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 realize 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 10(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, as no more is gained by increasing frequency.

PPU Activity Figure 11 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 computa-
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 using a higher 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 utilize 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. As a result, these applications would achieve similar performance with slower PPUs (as shown in figure 10(a)) or fewer of them.

ConjGrad is an outlier in that some PPUs do little work, yet it scales with increasing frequency (figure 10(a)). The reason for this behavior 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 the PPUs only react to events from the main core, and so are not required during phases where no data needs to be prefetched.

**Extra Memory Accesses** For efficient execution, it is desirable to minimize 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 also has variable work per vertex, meaning the prefetch distance must be set conservatively based on the EWMA. This results in 16% extra memory accesses.

**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 utilize the performance benefits of our scheme, with average 1.9× and 2.5× speedup for the two schemes we present.

**References**


