Technical Report

UCAM-CL-TR-923
ISSN 1476-2986

Number 923

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Prefetching for complex memory access patterns

Sam Ainsworth

July 2018

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 2018 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Modern-day workloads, particularly those in big data, are heavily memory-latency bound. This is because of both irregular memory accesses, which have no discernible pattern in their memory addresses, and large data sets that cannot fit in any cache. However, this need not be a barrier to high performance. With some data structure knowledge it is typically possible to bring data into the fast on-chip memory caches early, so that it is already available by the time it needs to be accessed.

This thesis makes three contributions. I first contribute an *automated software prefetching* compiler technique to insert high-performance prefetches into program code to bring data into the cache early, achieving 1.3× geometric mean speedup on the most complex processors, and 2.7× on the simplest. I also provide an analysis of when and why this is likely to be successful, which data structures to target, and how to schedule software prefetches well.

Then I introduce a hardware solution, the *configurable graph prefetcher*. This uses the example of breadth-first search on graph workloads to motivate how a hardware prefetcher armed with data-structure knowledge can avoid the instruction overheads, inflexibility and limited latency tolerance of software prefetching. The configurable graph prefetcher sits at the L1 cache and observes memory accesses, which can be configured by a programmer to be aware of a limited number of different data access patterns, achieving 2.3× geometric mean speedup on graph workloads on an out-of-order core.

My final contribution extends the hardware used for the configurable graph prefetcher to make an *event-triggered programmable prefetcher*, using a set of a set of very small micro-controller-sized *programmable prefetch units (PPUs)* to cover a wide set of workloads. I do this by developing a highly parallel programming model that can be used to issue prefetches, thus allowing high-throughput prefetching with low power and area overheads of only around 3%, and a 3× geometric mean speedup for a variety of memory-bound applications. To facilitate its use, I then develop compiler techniques to help automate the process of targeting the programmable prefetcher. These provide a variety of tradeoffs from easiest to use to best performance.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Sam Ainsworth
February 2018

Acknowledgements

I'd like to start by thanking Tim Jones, who has been an incredibly encouraging and active supervisor throughout this entire experience. Other people at the Computer Lab I'd like to thank are my secondary advisor Rob Mullins, for always being around for an interesting chat, Eiko Yoneki, for giving me the original idea that sparked the hardware graph prefetcher, and Ross Anderson, for being an outstanding mentor via the Churchill College SCR-MCR scheme. I'd also like to thank my PhD examiners, Alan Mycroft and Erik Hagersten, for their valuable feedback.

This PhD could never have been as successful as it was without mentorship from Arm, for which I am incredibly grateful. Tom Grocutt has played an absolutely invaluable role throughout: our high frequency of meetings meant that I was always making progress. I'd also like to thank Andreas Sandberg, for his often ingenious suggestions on how to make gem5 work for even the most challenging of use cases, and anyone who suggested benchmarks for me to test throughout the work in these following chapters.

I'd like to thank my undergraduate director of studies, John Fawcett, for his world-class teaching throughout my first degree, and for his advice that I should do a PhD, without which I would never be here now. And my friends from undergraduate who later became colleagues at the Computer Lab, Ian Orton and Colin Rothwell, for in Colin's words, their "entertainingly singular take on life in the Computer Lab."

Finally, I'd like to thank all of my family and friends for being there throughout the past three years of the PhD, and before.

Thanks, all – it's been a blast.

Contents

Chapter 1

Introduction

The von Neumann bottleneck in computers isn't getting any smaller. Fundamentally, compute speed grows faster than memory speed does, creating starvation in terms of getting data to a core: what is the point in having fast computation if the processor is always stalled waiting for data?

For many workloads, solutions to this exist. A large proportion of silicon-chip space is dedicated to a hierarchy of caches: increasingly smaller and faster memories, to give the appearance of high performance and large capacity. These work well when workloads have temporal locality: reused data can be accessed from the fast caches, and the high latency of main memory is masked. However, applications increasingly have very large data sets that need the amounts of storage only available off-chip. A partial solution to this is exploiting spatial locality: if data is accessed in sequence, it can be brought in multiple values at a time by using cache lines, and hardware stride prefetchers can predict the pattern to overlap memory accesses.

Still, with the advent of big data workloads, increasingly this approach doesn't solve the problem either. Memory accesses are often highly spread out by necessity, due to complex structure within the data. This results in patterns that appear to be unpredictable, and thus performance is limited: computation becomes bound by the high latencies associated with the off-chip dynamic random access memory (DRAM) typically used as main memory for systems. Such patterns appear all over modern workloads, such as social-media analysis, graph computations, bioinformatics workloads, sparse-matrix multiplication, databases, and neural networks [11, 17, 46, 79, 83].

However, for many such workloads, these are not fundamental limitations. It is possible to work out which memory locations will be accessed many cycles ahead of them actually being used, and thus overlap the memory accesses to hide this latency. Still, since the patterns are difficult to observe and depend on the data itself, it is necessary to have some information about the traversal pattern being used.

The easiest places to get access to this information are either directly from the user, or

within the compiler. This dissertation takes advantage of both, designing and evaluating a body of techniques, in hardware and software, to achieve high-performance prefetching for these complicated access patterns.

1.1 Example

An example of code that has this property is given in figure 1.1. The use of a as an indirect index into b causes memory accesses to be widely spread out, and thus little temporal or spatial locality is observed. We see in figure 1.2 that cache miss rates in both the L1 data cache (labelled 'L1D misses' in the figure), and the last-level cache (labelled 'LLC misses', in this case the L3 cache), are very high, and this results in a high-performance processor exhibiting frontend stalls for 90% of the runtime of the program. Effectively, this means that the processor is starved waiting for data, and unable to do any useful work.

However, future memory accesses are simple to work out if we have access to the code or knowledge of the algorithm. We can look ahead in array a and use that information to generate an address to prefetch into array b, to overlap the accesses to b and bring the data in before it is required. This simplistic view avoids many of the problematic details: that this stride-indirect is one of the simplest irregular memory-access patterns, that the lookup of values in a can themselves cause cache misses, that we need to know the access pattern to work this out, and that we need to know the addresses of arrays a and b to be able to issue these prefetches. But the fact that this is possible shows that such patterns need not be an impediment to high performance.

1.2 Hypothesis

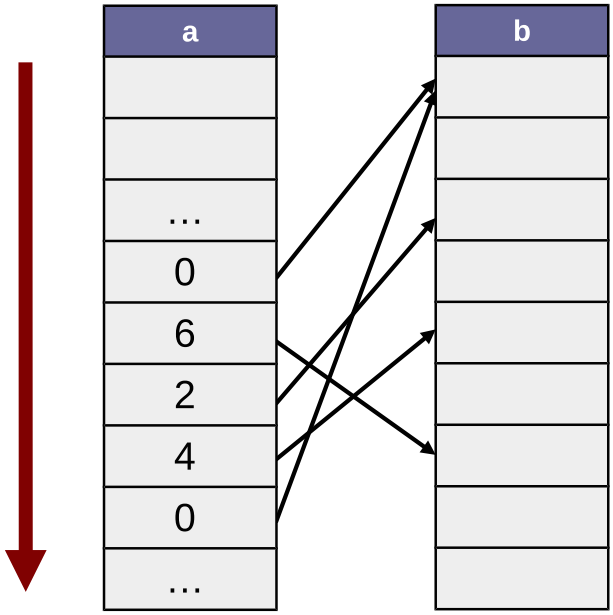
We can use algorithmic memory-access pattern knowledge to accurately identify and prefetch load addresses many cycles before they are required, and use this information in both hardware and software to improve performance for memory-bound workloads by reducing cache miss rates.

If we can use this information as part of our memory accesses, to bring data into the cache well in advance, as we can already do for simpler patterns, then performance should be significantly improved. Software prefetching [22], already implemented on most systems, is a start here, in that we can do address calculation for future memory accesses by hand in the program stream, then issue non-blocking loads for the data we will access. But since the patterns here are readily observable in the code, it is likely that such prefetching is amenable to compiler analysis to automatically generate code to do this.

Similarly, we are likely to gain even greater benefits if we can build this access pattern

```
1 for (i=0; i<a_size; i++) {
2       b[a[i]]++;
3 }
```

(a) Code



(b) Memory-access pattern

Figure 1.1: An example of the memory accesses in the Integer Sort benchmark from the NAS Parallel Benchmark suite [18]. Memory accesses to the array b are spread throughout memory, making the pattern impossible to predict just from looking at memory addresses. This means such code is memory-latency bound, but this isn't a fundamental constraint: we can work out which values will be accessed by looking ahead in a from where computation is currently, and using that information to generate prefetches.

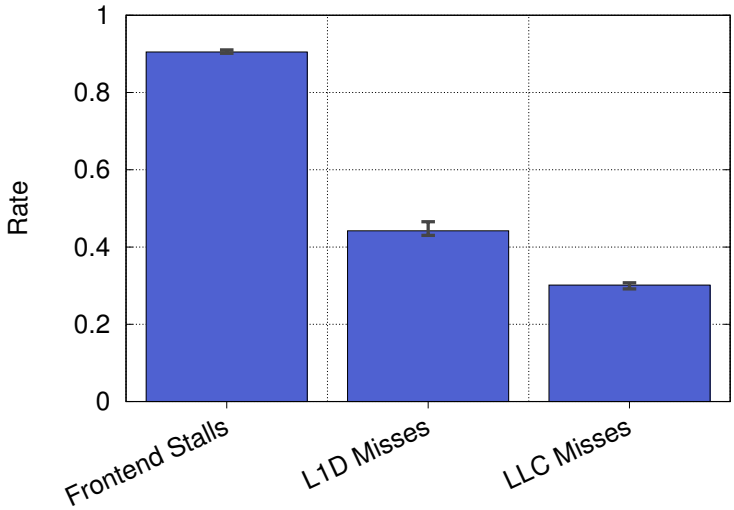


Figure 1.2: Performance characteristics of Integer Sort, on an Intel Haswell machine.

knowledge into hardware. In effect, the enormous performance improvements attainable from hardware prefetchers for easily observed patterns [35] should be exploitable for these more complicated patterns too, provided we can manage the complexity of specifying the memory-access structure, and of executing address calculation for such a wide space of different memory accesses in an efficient way.

1.3 Contributions

This dissertation provides four contributions to research in the fields of computer architecture and compilers.

First, an analysis on where and why software prefetching is useful, across many different microarchitectures, along with a compiler technique to automatically generate good software prefetches based on this. This has resulted in a publicly available LLVM compiler pass and associated paper, the artefact evaluation for which won an award at CGO 2017 [3].

Second, a methodology for designing configurable hardware prefetchers, which can be made aware of the access patterns of a program, to significantly improve memory-system performance. In particular, this is applied to graph workloads, but the concept is more general.

Third, an architecture for a programmable prefetcher. This can run custom prefetching code written in a high-level language and automatically compiled down to a conventional RISC ISA. This code takes in observations from the memory accesses of a main core and uses them to issue timely prefetches to support the main core. The fact that extracting memory-level parallelism is fundamentally a parallel task is exploited with an event-triggered programming model that reacts separately to each memory access observed, allowing highly efficient small cores to be used to calculate and perform prefetching. This research resulted in a joint patent application with Arm [40].

Finally, compiler techniques to ease the development of prefetching programs to be run on the event-triggered programmable prefetcher. These take the form of small, software prefetch-like descriptions inserted within the code to specify which data structures should be prefetched, or pragmas that specify the region of code for which data should be prefetched. This gives a range of techniques, depending on the level of investment the programmer deems necessary, to trade off performance for effort.

1.4 Structure

The remainder of this dissertation is structured as follows.

Chapter 2 covers the background of current prefetching techniques, in both hardware and software, along with a discussion of their limitations.

Chapter 3 analyses the ability of software prefetching to improve performance for workloads featuring indirect memory accesses, which I argue are particularly amenable to the technique. A state-of-the-art automated software prefetching compiler algorithm is designed to generate code to do this automatically, and I analyse which workloads and microarchitectures gain the most from software prefetching and why.

Chapter 4 develops hardware to improve on software prefetching, by still allowing the use of knowledge about the access pattern, but avoiding the limitations and overheads software prefetching exhibits. This configurable graph prefetcher hardware is used to attain performance improvements on out-of-order cores that are significantly larger than when using software prefetching, for both breadth-first searches and more simple indirect access patterns on graph workloads.

Chapter 5 extends the work of Chapter 4 to make an event-triggered programmable prefetcher, to target a greater range of current and future memory-access patterns. I develop a highly parallel programming model that allows the issuing of latency-tolerant prefetches on a set of small micro-controller sized cores, along with fixed-function prefetch-support hardware, so that prefetching can keep up with and improve the performance of the main processor even for complicated access patterns.

Chapter 6 develops compiler techniques to ease the use of the event-triggered programmable prefetcher, by abstracting away its complicated programming model. These allow the programmer to either specify programmable prefetches using software prefetch intructions (software-prefetch conversion), which are then analysed and converted into the highly parallel event model described in Chapter 5, or to simply specify which loops are likely to need prefetching by using pragmas (pragma generation), leaving the compiler to generate prefetch code from analysis of the original loads themselves.

Chapter 7 concludes, and gives examples of where the techniques developed throughout this thesis can be applied elsewhere, to other beneficial system properties.

1.5 Publications

Research from this dissertation has been published at the following conferences:

- Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In the International Conference on Supercomputing, ICS, 2016. [7]

- Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In the International Symposium on Code Generation and Optimization, CGO, 2017. [8]
- Sam Ainsworth and Timothy M. Jones. An event-triggered programmable prefetcher for irregular workloads. In the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2018. [9]

Chapter 2

Background and related work

Prefetching is a broad topic, breaching into both pure hardware and software techniques, along with everything in between. A structural representation of the entire space is given in figure 2.1. A more detailed coverage is given below.

2.1 Hardware prefetching

A hardware prefetcher sits alongside the cache, observing accesses and misses in that cache to bring in new data that the observed program is likely to need in a few cycles' time. I start by looking at the most common types of prefetchers in existence today: those based purely on address observation. I then move on to techniques in the literature for more irregular address patterns, before looking at techniques that, as with my hypothesis, require some user or compiler support. Finally, I look at helper-core techniques, where some amount of programmable hardware is used to assist prefetching.

2.1.1 Address-based prefetching

The most widely implemented [93] type of prefetcher works on regular memory accesses, such as walking arrays or matrices. Such "stride" prefetchers observe patterns in address sequences to predict future accesses, and prefetch accordingly. Chen and Baer [24, 25] introduce the concept of a reference prediction table used to implement this style of prefetching. Other more complicated schemes exist to pick up more complex address patterns [54, 89].

The workloads we are interested in exhibit data-dependent memory accesses. This limits the utility of such address-based techniques, as typically no pattern exists within the address sequence that is able to be picked up.

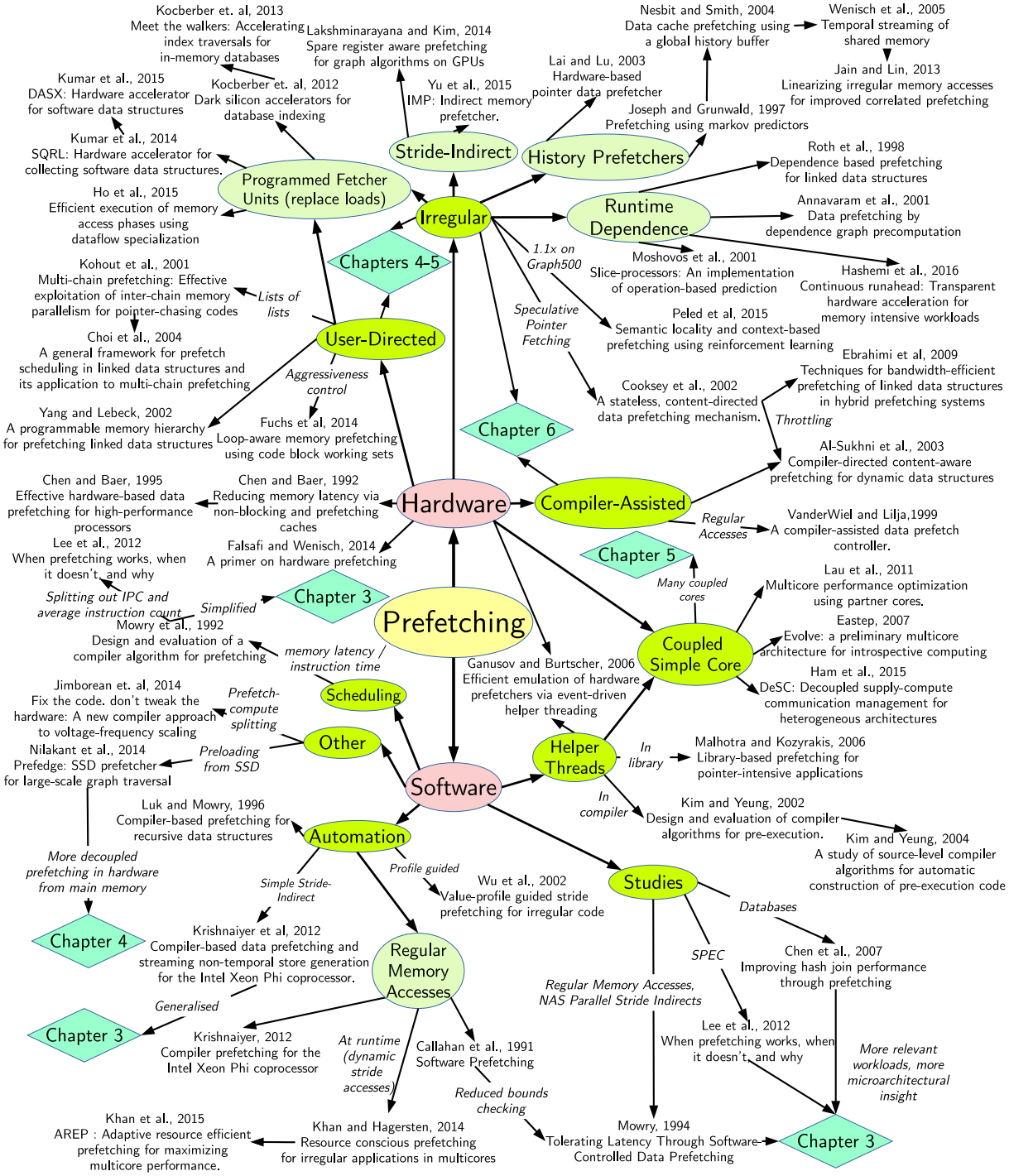


Figure 2.1: An overview of the links between the most relevant papers on prefetching and this dissertation, both in software and in hardware.

2.1.2 Irregular prefetching

When memory accesses are more complicated, stride prefetchers are no longer able to predict them effectively. Prefetchers for so-called "irregular" memory accesses, where the pattern is no longer a simple sequence or a stride pattern, and instead is data-dependent, become necessary. Because the patterns are less simple, current success is also more limited, and such prefetchers haven't yet seen implementation in commercial systems [35]. Techniques from academic literature are considered below.

2.1.2.1 Stride-indirect prefetching

A common memory-access pattern in many workloads, such as sparse matrix operations, is the stride-indirect. This involves looking up values in sequence in one array, then using the data as an index into another array or as a pointer. Techniques exist to try to observe these patterns, infer the data structures, and prefetch them at runtime. Lakshminarayana and Kim [64] do this for GPUs, and Yu et al. [97] develop a CPU scheme. The disadvantages are that the analysis hardware required is significant: because the pattern involves chains of address accesses and data, many comparisons have to be made to actually pick up the stride-indirect pattern. The total number of patterns that can be picked up are also limited: many modern workloads are memory-bound despite not following a simple stride-indirect pattern.

2.1.2.2 History prefetching

Instead of trying to learn a pattern, why not just replay it? This is the technique used by history-prefetching schemes. These typically track traces of past cache misses, and on observation of a load to an entry previously tracked, elements observed shortly after the table was previously filled are prefetched.

Markov prefetchers [49] are an early example of such an approach. Nesbit and Smith extend this by using a better, more general on-chip data structure to store the state: a global history buffer [82]. Lai and Lu [63] use a similar technique to store pointers indexed by memory accesses.

The limitations of such techniques are twofold. The first is that, by having to store data on-chip, a very limited amount of history can be stored – it can be better to instead increase the size of the last level cache, as this is likely to be useful for more workloads, and for many repeated workloads it isn't possible to store every memory access on-chip due to size constraints, and thus patterns are overwritten before they repeat. The solution to this is storing the state for the prefetcher off-chip: this is the approach taken by Wenisch et al. with their temporal streaming technique [94], and Jain and Lin with their irregular stream buffer [47]. However, this results in many costly accesses to main memory, which

are particularly punishing when the prefetcher is unable to help performance.

Indeed, more fundamentally, unless the workload repeats work multiple times, a history prefetcher is useless. Even if the workload does repeat, such a technique cannot help with the first run on a given input, and also cannot prefetch working sets larger than the amount of storage dedicated to the prefetcher. History prefetching is also wasteful: many patterns, as we shall see, can be described and impelemented concisely without storing a history of the entire working set.

2.1.2.3 Runtime dependence

Examples of attempts to extract dependence-graph streams at runtime by detecting dependent loads exist, from around the turn of the 21st century [15, 76, 88]. These run dynamically-detected load sequences on a programmable unit 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 execute the detected streams. The look-ahead is also limited, in that multiple streams cannot be prefetched at once, and the techniques don't attempt to move multiple iterations ahead of the program, limiting memory-level parallelism.

There is a good reason such work typically targets linked data structures. These feature very little memory-level parallelism: there is a sequentialisation from following each successive pointer. This means the workload is fundamentally latency-bound, and thus little prefetch compute performance is needed. When this isn't the case, such as with arrays, or accesses where multiple indirect loads can be performed at once, much more performance is required from the prefetching hardware: this is why I provide a parallel programming model in chapters 5 and 6.

More recent work by Hashemi et al. works in a similar vein to runtime dependence extraction [42]: their "continuous runahead" technique uses runtime analysis hardware with access to all microarchitectural state to generate code fragments featuring critical instructions. These fragments still stall on dependent loads, but are offloaded to simpler hardware to fetch data, and remove some redundant execution. Again, the technique suffers from the inability to extract parallelism, which we will later see is important, and the invasive access of main-core state restricts implementation viability.

2.1.2.4 Other approaches

Mutlu et al. propose a runahead scheme [80], which utilises idle chip resources on a cache miss to dynamically prefetch loads. This is limited by being tightly bound to the instruction stream, thus are unable to exploit significant look-ahead, or prefetch from other prefetched loads for accesses with indirection. Indeed, later work [81] requires dedicated prefetch hardware even to pick up patterns as simple as the stride-indirects of

section 2.1.2.1, as the miss on initial values is too crippling to allow runahead to function well.

Peled et al. [86] try to pick up complicated prefetching patterns using reinforcement learning. However, for workloads with data dependent accesses, their scheme's abilities are limited: for example, on Graph500 search (a benchmark I use to evaluate my approaches and for which I achieve over 2.5× improvement) only a 1.1× speedup is observed: this is worse than a basic stride prefetcher included in any modern system can achieve.

Cooksey et al. [28] fetch any plausible-looking address that is loaded as data from a cache line. However, unless the data structure being observed is a list of pointers, all of which will be referenced, such schemes over-fetch dramatically, causing cache pollution and performance degradation. Other work attempts to control this by selectively enabling the technique through compiler configuration [12, 33], but the benefits are still limited to arrays of pointers and linked lists.

2.1.3 User- and compiler-directed prefetching

If a pattern is too complicated, or too expensive, to learn through runtime observation, why not use a prefetcher that has been given some information about its access pattern? This is a philosophy generally taken throughout this dissertation, and many other works have used a similar idea. This information can be observed during compilation, through direct programming, or from hints by the programmer.

An example of this for regular memory accesses is the work of Fuchs et al. [37], who use explicit compiler annotations to control the aggressiveness of the prefetch. Al-Sukhni et al. [12] use special "Harbinger" instructions to specify the layout of the structures involved, for prefetching linked-list-style dynamic data structures in hardware. Ebrahimi et al. [33] use compiler direction to throttle pointer chasing. Kohout, Choi et al. [26, 57] design a configurable prefetcher to fetch lists of lists. The latter are interesting as an example of memory-level parallelism within linked structures: if many elements can be prefetched at once from many different linked lists, overlapping of accesses can be achieved. I use a similar property later when looking at the Graph500 list benchmark in chapter 5.

2.1.3.1 Fetcher units

Some user-directed techniques commonly labelled as prefetchers aren't really prefetchers at all. Many of the architectural techniques proposed to tackle specific memory-access patterns are in fact "fetcher units": the difference is that, while a prefetcher speculatively brings in data to the cache, which the processor then loads again, a fetcher unit is delegated the actual task of fetching data for the main processor, which then accesses the data from the fetcher unit non-speculatively. Examples of this approach include SQRL and

DASX [60, 61], designed for iterative accesses of B-tree, vector and hash table structures. Similarly, Kocberber et al. [55, 56] focus on the optimisation of database inner joins by parallel hash table walking. Ho et al. [45] generalise the concept of fetcher units to cover a wider variety of access types by encoding memory accesses as a set of rules, allowing loads and stores to be mapped to a dataflow architecture.

When prefetching, the prefetcher performs the address calculation once speculatively, and then the main processor must repeat that for the true execution. Avoiding this second address calculation is a tempting proposition. However, it removes the ability for speculation by the prefetcher. In effect, it means that the program must feature thread-level as well as memory-level parallelism, since intermediate data values used in the fetching of loads must be correct. This means that common patterns, such as scatter-gather read and writes, can be ruled out due to intermediate writes potentially changing the accessed data. It also makes it more difficult to target such hardware using compiler passes with imperfect information: while a prefetching approach can speculate on runtime behaviour, the penalty for incorrect information within a fetcher unit is an incorrect program. Another issue prefetching handily avoids is that of binary compatibility: a processor implemented without a fetcher unit cannot execute code compiled with the assumption of the fetcher unit's existence, as the fetcher unit performs true work instead of speculation, whereas a missing prefetcher simply reduces performance.

2.1.4 Helper cores

Several works place a small programmable unit next to a processor to assist in prefetching. This is related to the helper threading techniques covered later in section 2.2.4, but with hardware support, and typically heterogeneity between the main core and the smaller helper core.

Lau et al. [66] add a single small helper core to a main core to assist with processing tasks including prefetching. The Evolve project [32] uses communication paths between cores in a tiled architecture to implement a prefetcher for JPEG encoding. Ham et al. [41] provide a scheme where a core is split based on separate access and execute threads, which run different code. Ganusov and Burtscher [39] use helper threading to emulate common Markov [49] and stride prefetching schemes in software, by adding in hardware support to forward observed loads to newly spawned threads. Vanderwiel and Lilja [92] automate generation of prefetches for a programmable hardware prefetch controller, but their scheme is limited to very simple, data-independent prefetches as it cannot read any data from a prefetch or load.

There are two problems with all of these approaches. The first is that by using a traditional sequential programming model for the helper it isn't possible to load intermediate values for prefetches without stalling, limiting memory-level parallelism. The second is

that to achieve more memory-level parallelism than an out-of-order superscalar core can by itself, a large amount of compute must be dedicated to prefetching: one small core is not enough when an application is mostly made up of memory accesses.

I take the idea of helper cores in chapter 5 and expand it, to achieve the level of performance necessary without increasing power usage or silicon area significantly. This is achieved by using many small cores attached to a main processor, and by using a highly parallel programming model that can avoid memory stalls.

2.2 Software prefetching

It isn't necessary to have dedicated observation hardware to perform prefetching. A non-blocking load instruction is enough to allow prefetching in software: these instructions can be issued in anticipation of data that is likely to be used later. Here, I consider both the most relevant and the most impactful schemes.

2.2.1 Performance studies

A number of performance studies have shown software prefetching in action on suites of benchmarks. However, they are all lacking in various ways. Lee et al. [67] show speedups for a variety of SPEC [43, 44] benchmarks with both software and hardware prefetching. However, SPEC tends to feature benchmarks with data sets that either consist of regular memory accesses, small datasets that fit in the cache, or that only spend a small amount of time performing irregular accesses. This limits the observability of interesting memory effects, and thus applicability to workloads that perform less well with respect to the memory system. By comparison, Mowry [77] considers both Integer Sort and Conjugate Gradient from the NAS Parallel Benchmarks [18], which feature the kind of indirect array indexing that causes issues for existing hardware prefetchers. Both papers only consider simulated hardware. However, we see in chapter 3 that the microarchitectural impact on the efficacy of software prefetching is important: Integer Sort gains a 7× improvement on an Intel Xeon Phi machine, but a negligible speedup on an Arm Cortex-A57, for example. Chen et al. [23] insert software prefetches for database hash tables, where large speedups can be observed on out-of-order hardware. I explain why this is in chapter 3.

2.2.2 Automation

Ideally, prefetching is entirely transparent and can be applied to all programs running on a machine. This is true with unconfigurable hardware such as a stride prefetcher, for example. However, with software prefetching, actual instructions do need to be inserted to do the prefetching. A nice middle ground, then, is utilising the compiler to perform

prefetching analysis and insert prefetches. Some work on doing this for regular memory accesses already exists: it is enabled by default in Intel's Xeon Phi C compiler [59], for example.

2.2.2.1 Regular memory accesses

Methods for inserting software prefetches into loops for regular memory accesses exist, for example Callahan et al. [22]. Mowry [77, 78] extends this with techniques to reduce branching, removing bounds checks for prefetches inside loops by splitting out the last few iterations of the loop, and also uses reuse analysis to avoid redundant prefetches. Wu et al. [95] use profiles to prefetch for applications that are irregular in software prefetching terminology but happen to exhibit stride-like regular patterns at runtime: i.e. the patterns that could be picked up by a hardware stride prefetcher. Zucker et al. [98] use software prefetching to emulate a hardware stride prefetcher.

Khan et al. [50, 51] choose to instead insert software prefetches dynamically using a runtime framework for code modification. This allows prefetching to be performed for applications where the source code is unavailable, and also gives access to runtime data. On the other hand, it has limited access to static information such as types, and also adds overhead. They use runtime information to dynamically turn off the hardware stride prefetcher when software prefetches are effective, to reduce overfetching.

2.2.2.2 Irregular memory accesses

Examples also exist in the literature for software prefetching of both recursive data structures, for example Luk and Mowry [71] prefetch linked lists, and function arguments, which Lipasti et al. also perform [70]. The benefits of software prefetching are limited here, because typically we can only fetch the next element once we have loaded the previous one, limiting the ability to overlap memory accesses.

Mowry's PhD dissertation [77] discusses prefetching for stride-indirect patterns on high-level C-like code. In contrast, in chapter 3 I give a full algorithm to deal with the complexities of intermediate representations, including fault-avoidance techniques and value tracking. An algorithm for simple stride-indirect patterns is implemented in the Xeon Phi compiler [58], but it is not enabled by default and little information is available on its inner workings. Further, it picks up relatively few access patterns, and is comprehensively outclassed by the technique in chapter 3.

2.2.3 Scheduling

A variety of fine-grained prefetch scheduling techniques, to set the appropriate look-ahead distance, have been considered in the past. Mowry et al. [78] consider estimated instruction

time against an estimated memory-system time. The former is difficult to estimate correctly on a modern system, and the latter is microarchitecture dependent, which makes these numbers difficult to get right. Lee et al. [67] extend this by splitting instructions-per-cycle (IPC) and average instruction-count, which are both determined from application profiling. As these are all small numbers, and errors are multiplicative, accuracy is challenging: the latter multiplies the prefetch distance by 4 to bias the result in favour of data being in the cache too early.

In chapter 3 I take a simpler approach, based on observations across many different microarchitectures. It turns out that the most important thing to get right is the number of loads between a prefetch being issued and being used, and this approach leads to a much more stable number being generated.

2.2.4 Helper threads

Software prefetches can also be moved to different threads, to reduce the impact of the large number of extra instructions added to facilitate prefetching. Kim and Yeung [52, 53] use a profile-guided compiler pass to generate "helper threads", featuring prefetch instructions, to run ahead of the main thread. Malhotra and Kozyrakis [73] create helper threads by adding software prefetch instructions to shared libraries and automatically detecting data-structure traversals. This means more complicated prefetching schemes can be used than can be generated by a compiler pass, however, the benefit is limited to code from within common libraries that have been manually edited.

Such schemes use an additional core or hardware thread, which is expensive in terms of power and resource, and may be wasteful in cases where the main core is stalled most of the time, for example with in-order cores. Further, synchronisation between threads, which may be costly, is required. They still feature the limitation, as with in-line software prefetches, that prefetching data that hasn't already been prefetched causes a stall, limiting the ability to overlap accesses. This limits applicability for complicated memory accesses with dependencies.

The hardware programmable prefetcher, described in chapter 5, is somewhat inspired by the helper-thread approach. However, it features key differences that give it greater abilities at lower cost. The use of many small cores gives high computation ability at low power and area overheads. The hardware prefetcher data channel avoids any synchronisation overhead. And the highly parallel programming model avoids the need to stall on loads used to calculate prefetch addresses.

2.2.5 Other approaches

Rather than directly inserting software prefetches within loops, some works have used them as parts of separate loops to improve performance or power efficiency. Jimborean et al. [48] use compiler analysis to duplicate and simplify code, to separate loads and computation. This is to enable different frequency-voltage scaling properties for different sections of the code. In chapter 3 I instead keep loads and computation together. This allows the non-blocking prefetches to occur at the same time as the computation, allowing compute time to be hidden within load latencies of the prefetches themselves. These latencies exist due to limitations on memory-level parallelism caused by the core and memory system, and we can exploit this to hide computation time.

Another work that is interesting, in that it inspires some of the techniques used in chapter 4, is PrefEdge [83]. This work uses information from input graph structures to bring in data from an SSD into main memory, overlapping multiple accesses to achieve memory-level parallelism. This uses non-blocking IO rather than software prefetching, and doesn't bring data into caches, but is otherwise an interesting use of a similar technique.

2.3 Summary

Hardware stride prefetching is vital for high performance in modern processors, but prefetchers for more irregular accesses haven't had the same impact. This is because current techniques in the literature all suffer from numerous flaws, either because they target too few access patterns, only work for repeated patterns and require large amounts of storage, or have limited performance. This means many workloads remain cripplingly latency-bound even on modern systems.

Software prefetching can more easily express irregular patterns by being user-controlled, but I argue that performance studies in the literature misinterpret when and why they are useful, and how to generate good prefetches, and so automatic techniques for insertion are limited.

In the following chapters, I address these issues, first advancing the state of the art in software prefetching, then moving on to design hardware to escape the fundamental limitations involved with software-only techniques.

Chapter 3

Prefetching in software

Before considering how we can extend the architecture to make it perform better for workloads with complicated memory accesses, it is worth seeing how much we can achieve just using hardware that already exists today. I present an analysis of how software prefetching can be used to improve the properties of modern workloads. I use this information to automatically generate and schedule good software prefetches within the compiler, and also show the limitations of software prefetching across multiple architectures, and how these manifest. This work resulted in a paper on automated software prefetch generation at CGO 2017 [8].

First, I look at which memory-access patterns software prefetching is useful for, and how to generate good prefetch instructions by hand. I then develop an algorithm for doing this automatically in the compiler, before evaluating how closely this matches what we can achieve with prefetching by hand, followed by a more detailed look at when, where and to what extent prefetching in software is useful.

3.1 Software prefetching

Software prefetching is a widely supported operation across many different architectures. It is typically implemented as an instruction similar to a load, but with two important differences:

- No value is written back to a register. This means we need not wait for the prefetch to finish before retiring the instruction.
- The prefetch need not be successful. This means that prefetches to invalid locations don't cause the processor to fault.

However, it is notoriously difficult to get right. Typically, attempts to insert software prefetching instructions go awry, because people either insert them for the wrong loads, or don't manage to create a successful prefetch for the right ones.

3.1.1 What should we software prefetch?

When designing an algorithm to improve performance for memory accesses using software prefetching, many different types of memory access could be targeted. However, for this to be successful, two properties need to be fulfilled:

- It must be easy and cheap to generate multiple future addresses, so that the loads to them can be overlapped through prefetching.
- The loads themselves must frequently miss without software prefetching.

The first potential access style we might consider would be regular memory accesses. These take the form $A[x]$ for an array A and iteration over a variable x. These are very easy to generate many future addresses for: issuing prefetches to $A[x + N]$ is cheap and allows lots of overlap by concurrent prefetching of several values. However, hardware stride prefetchers are designed to cover these patterns well, and so any extra benefit from software prefetching will be limited.

Another access pattern we might consider is linked data, for example linked lists. Accesses here take the form $A \rightarrow next$ for some element A and some pointer $next$. These succeed on the second criterion, in that the pointer accesses are often hard to predict and typically spread throughout memory, meaning cache misses are very common. But finding addresses to prefetch is problematic. To prefetch an element several ahead of the current one, and thus overlap many memory accesses, we need to have loaded all previous values: the prefetch would look like $A \rightarrow next \rightarrow next \rightarrow next \rightarrow next$. But this makes the prefetch useless, as only one prefetch can be outstanding before the value has to be loaded in to do the next prefetch. These structures are fundamentally sequential, which limits the memory-level parallelism to an amount that an out-of-order core can already trivially extract.

More profitably, then, we may consider indirect memory accesses, an example of which is $A[B[x]]$, for some arrays A and B, and iteration over a variable x. This pattern can also be generalised such that it has multiple arrays, features some computation in the address calculation, or that arrays except the one being iterated over can be replaced with pointer accesses. These are likely to miss in the cache, due to the data-dependent memory access to A, and are also easy to predict: we can look ahead in B to generate the addresses that will miss in A. It is also fortunate that these memory accesses are relatively common in many programs: sparse matrix operations [17, 18], graph analytics [79, 83], databases [56] and any workloads using arrays of pointers [28] all tend to feature a similar pattern.

3.1.2 How do you generate good software prefetches?

Even with an understanding of what is useful to prefetch, generating good software prefetches that actually improve performance can still be a challenge. Code listing 3.1 shows an example of a simple indirect memory access taken from Integer Sort from the NAS Parallel benchmark suite [18]. Here, memory accesses to key_buff1 are unpredictable and likely to miss, but it is possible to prefetch them in software by looking ahead in key_buff2. However, as we see in figure 3.1 this is by no means enough to get optimal performance. Even though the Intel Haswell processor has a hardware stride prefetcher, in this case it is also beneficial to prefetch the regular memory access used in the calculation of the software prefetch, likely because the alternating pattern of accessing key_buff2 at both i and i+offset confuses the prefetcher. Interestingly, this is consistently true on every microarchitecture I have observed, regardless of instruction set architecture.

Including both prefetches is enough to get a 1.3× speedup even on a large out-of-order superscalar core. However, that is assuming an optimal look-ahead distance (offset). With too small an offset, performance can be lower than without any prefetching at all: prefetches are issued to addresses that have already missed and been fetched, thus increasing instruction overhead without any benefit. Likewise, too large an offset results in prefetched values being evicted from the cache, due to the normal replacement policy, before they are used.

Indeed, the appropriate look-ahead value is both microarchitecture- and workload-dependent. This might lead one to conclude that automatically setting this value is intractible: indeed, previous work [67, 78] has tried a variety of complicated and inadequately performing schemes to solve this issue, as discussed in section 2.2.3. However, when evaluating on multiple benchmarks and a wide variety of microarchitectures, it becomes apparent that the problem is much easier to solve than it might first appear. Indeed, using a constant divided by the static number of loads works surprisingly well across every example I have seen, since as long as values aren't evicted from the cache before they are used, and the prefetch is able to account for a sufficient amount of memory latency, a large amount of leeway in setting the look-ahead distance is permissible.

3.2 Automated software prefetching generation

Algorithm 1 shows pseudocode for an algorithm I have developed for automatically generating software prefetches. I implement this as an LLVM intermediate representation (IR) pass [5], though the algorithm itself is more general. First, the IR is analysed to find references within loops that refer to values identified as induction variables. The pass then identifies the subset of these for which no invalid loads will be generated if we prefetch them, and which feature indirection, and so are likely to miss in the cache. It then inserts

```
1 for (i=0; i<NUM_KEYS; i++) {
2   // The intuitive case, but also
3   // required for optimal performance.
4   SWPF(key_buff1[key_buff2[i + offset]]);
5   // Required for optimal performance.
6   SWPF(key_buff2[i + offset*2]);
7   key_buff1[key_buff2[i]]++;
8 }
```

Code listing 3.1: An integer-sort benchmark showing software-prefetch locations. The intuitive prefetch to insert is just the one at line 4, whereas optimal performance also requires that at line 6.

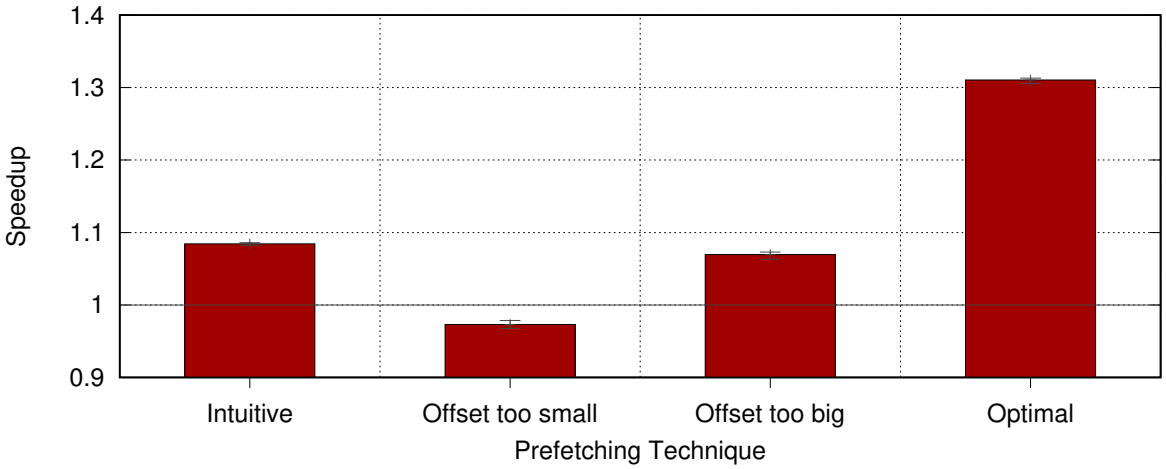


Figure 3.1: Software prefetching performance for code listing 3.1 on an Intel Haswell micro-architecture. Inserting software prefetches for maximal performance is a challenge even in simple cases, because intuitive schemes leave performance on the table, and choosing the correct offset for look-ahead is similarly critical for high performance.

prefetches based on this information, at a look-ahead offset into the array calculated based on the number of loads in the indirect memory access.

3.2.1 Analysis

The overall aim of the analysis pass is to identify loads that can be profitably prefetched and determine the code required to generate prefetch instructions for them. Target loads are those where it is possible to generate a prefetch with look-ahead: the algorithm checks whether it can generate a new load address by increasing the value of a referenced induction variable within the address calculation by a certain offset. The analysis considers one function at a time and does not cross procedure boundaries.

It starts with loads that are part of a loop (line 26 in algorithm 1). The data-dependence graph is walked backwards using a depth-first search from each load to find an induction variable within the transitive closure of the input operands (line 1). The algorithm stops searching along a particular path when we reach an instruction that is not inside any loop.

```
DFS(inst) {
  candidates = {}
  foreach (o: inst.src_operands):
    // Found induction variable, finished this path.
    if (o is an induction variable):
      candidates ∪= {(o, {inst})}
    // Recurse to find an induction variable.
    elif (o is a variable and is defined in a loop):
      if (((iv, set) = DFS(loop_def(o))) != null):
        candidates ∪= {(iv, {inst}∪set)}
  // Simple cases of 0 or 1 induction variable.
  if (candidates.size == 0):
    return null
  elif (candidates.size == 1):
    return candidates[0]
  // There are paths based on multiple induction variables, so choose
  // the induction variable in the closest loop to the load.
  indvar = closest_loop_indvar(candidates)
  // Merge paths which depend on indvar.
  return merge_instructions(indvar, candidates)
}

// Generate initial set of loads to prefetch and their address
// generation instructions.
prefetches = {}
foreach (l: loads within a loop):
  if (((indvar, set) = DFS(l)) != null):
    prefetches ∪= {(l, indvar, set)}

// Function calls only allowed if side-effect free.
remove(prefetches, contains function calls)
// Prefetches should not cause new program faults.
remove(prefetches, contains loads which may fault)
// Non-induction variable phi nodes allowed if pass can cope with
// complex control flow.
remove(prefetches, contains non-induction phi nodes)

// Emit the prefetches and address generation code.
foreach ((ld, iv, set): prefetches):
  off = calc_offset(list, iv, load)
  insts = copy(set)
  foreach (i: insts):
    // Update induction variable uses.
    if (uses_var(i, iv)):
      replace(i, iv, min(iv.val + off, max(iv.val)))
    // Final load becomes the prefetch.
    if (i == copy_of(ld)):
      insts = (insts - {i}) ∪ {prefetch(i)}
  // Place all code just before the original load.
  add_at_position(ld, insts)
```

Algorithm 1: The software prefetch generation algorithm, assuming the intermediate representation is in static single assignment (SSA) form.

When it finds an induction variable, it records all instructions that reference this induction

variable (directly or indirectly) along each path to the load (lines 6 and 10). If multiple paths reference different induction variables, the algorithm only records instructions that reference the innermost ones (line 18). This reflects the fact that these variables are likely to be the most fine-grained form of memory-level parallelism available for that loop.

The recorded set of instructions will become the code to generate the prefetch address in a later stage of the algorithm. However, we must constrain this set further, such that no function calls (line 31) or non-induction-variable phi nodes (line 36) appear within it, because the former may result in side-effects occurring and the latter may indicate complex control flow changes are required. In these cases my implementation throws away the whole set of instructions, and does not generate prefetches for the target load. Nevertheless, both could be allowed with further analysis. For example, side-effect-free function calls could be permitted, allowing the prefetch to call the function and obtain the same value as the target load. Non-induction phi nodes require more complicated control-flow generation than I currently support, along with more complex control-flow analysis. However, without this analysis, the conditions are required to ensure that we can insert a new prefetch instruction next to the old load, without adding further control flow.

3.2.2 Fault avoidance

While software prefetch instructions themselves cannot cause faults, intermediate loads used to calculate addresses can (e.g., the load from key_buff2 to generate a prefetch of key_buff1 at line 4 in code listing 3.1). We therefore need to ensure that any look-ahead values will be valid addresses and, if they are to be used for other intermediate loads, that they contain valid data.

To address this challenge, I follow two strategies. First, I add address bounds checks into the software prefetch code, to limit the range of induction variables to known valid values (line 28 in algorithm 1). For example, checking that i + 2*offset < NUM_KEYS at line 6 in code listing 3.1. Second, I analyse the loop containing the load, and only proceed with prefetching if the algorithm does not find stores to data structures that are used to generated load addresses within the software prefetch code (line 33 in algorithm 1). For example, in the code x[y[z[i]]], if there were stores to z, we would not be able to guarantee the memory safety of prefetches to x. This could be avoided with additional bounds checking instructions, but would add to the complexity of prefetch code. I also disallow any prefetches where loads for the address-generating instructions are conditional on loop-variant values other than the induction variable. Together, these ensure that the addresses generated for intermediate loads leading to prefetches will be exactly the same as when computation reaches the equivalent point, several loop iterations later.

The first strategy requires knowledge of each data structure's size. In some cases, this is directly available as part of the intermediate representation's type analysis. For others,

walking back through the data-dependence graph can identify the memory allocation instruction that allocated the array. However, in general, this is not the case. For example, it is typical in languages such as C for arrays to be passed to functions as a pointer and associated size, in two separate arguments. In these cases, and more complicated ones, we can only continue if the following two conditions hold. First, the loop must have only a single termination condition, since then we can be sure that all iterations of the loop will give valid induction values. Second, accesses to the look-ahead array must use the induction variable which should be monotonically increasing or decreasing.

Given these conditions, the maximum value of the induction variable within the loop will be the final element accessed in the look-ahead array in that loop and we can therefore use this value as a substitute for size information of the array, to ensure correctness. Although these conditions are sufficient alone, to ease analysis in the prototype implementation, I further limit the second constraint such that the look-ahead array must be accessed using the induction variable as a direct index (base_array[i] not base_array[f(i)] for an expression f(i)) and add a constraint that the induction variable must be in canonical form (monotonically increasing by one on each loop iteration).

The software prefetch instructions themselves cannot change correctness, as they are only hints: all microarchitectures ignore any faults that occur as a result of a prefetch, and silently throw the prefetch away. The checks described in this section further ensure that address generation code doesn't create faults if the original code was correct. However, the pass can still change runtime behaviour if the program originally caused memory faults. While no memory-access violations will occur if none were in the original program, if memory-access violations occur within prefetched loops, they may manifest earlier in execution as a result of prefetches, unless size information comes directly from code analysis instead of from the loop size.

3.2.3 Prefetch generation

Having identified all instructions required to generate a software prefetch, and met all conditions to avoid introducing memory faults, the next task is to actually insert new instructions into the code. These come from the set of instructions recorded as software-prefetchable code in section 3.2.1 and augmented in section 3.2.2.

The algorithm inserts an add instruction (line 28 in algorithm 1) to increase the induction variable by a value (line 40) that is the offset for prefetch. Determining this value is described in section 3.2.4. It then generates an instruction (either a select or conditional branch, depending on the architecture) to take the minimum value of the size of the data structure and the offset induction variable (line 28). It creates new copies (line 41) of the software prefetch code instructions, but with any induction-variable affected operands (determined by the earlier depth-first search) replaced by the instruction copies

(line 28). Finally, it generates a software prefetch instruction (line 31) instead of the final load (i.e., the instruction we started with in section 3.2.1).

I only generate software prefetches for stride accesses if they are part of a load for an indirect access. Otherwise, I leave the pattern to be picked up by the hardware stride prefetcher, or a more complicated stride software prefetch generation pass which is able to take into account, for example, reuse analysis [78].

3.2.4 Scheduling

The goal is to schedule prefetches by finding a look-ahead distance that is generous enough to prevent data being fetched too late, yet avoids polluting the cache and extracts sufficient memory-level parallelism to gain performance. Previous work [78] has calculated prefetch distance using a ratio of memory bandwidth against number of instructions. However, code featuring indirect accesses is typically memory bound, so execution time is dominated by load instructions. I therefore generate look-ahead distances using the following, simpler formula.

$$\mathit{offset} = \frac{c(t-l)}{t} \tag{3.1}$$

where t is the total number of loads in a prefetch sequence, l is the position of a given load in its sequence, and c is a microarchitecture-specific constant, which represents the look-ahead required for a simple loop, and is influenced by a combination of the memory latency and throughput (e.g., instructions-per-cycle (IPC)) of the system. High memory latency requires larger look-ahead distances to overcome, and high IPC means the CPU will move through loop iterations quickly, meaning many iterations will occur within one memory latency of time.

As an example of this scheduling, for the loop in code listing 3.1, two prefetches are generated: one for the stride on key_buff2, and one using a previously prefetched look-ahead value to index into key_buff1. This means $t = 2$ for these loads. For the first, $l = 0$, so $\mathit{offset} = c$ by equation 4.3, and we issue a prefetch to key_buff2[i+c]. For the second, $l = 1$, so we issue a prefetch to key_buff[i+c/2].

The formula has the property that it spaces out the look-ahead for dependent loads equally: each is prefetched $\frac{c}{t}$ iterations before it is used, either as part of the next prefetch in a sequence, or as an original load.

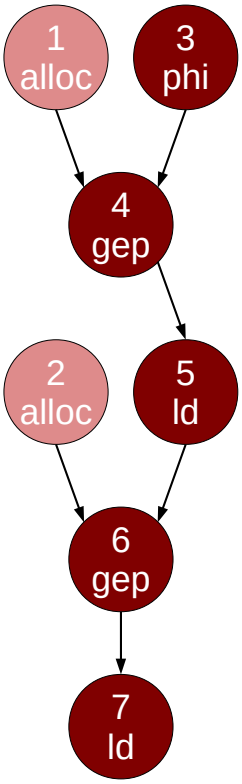
As we shall later see in section 3.4.2, this works surprisingly well: because memory latency invariably dominates performance, this works better than more complicated schemes [67, 78], in that it achieves close to optimal performance while only requiring a simple analysis, and being microarchitecture independent save for the constant c. However, we can go a step further than this: for every microarchitecture I have tested, c can be set

```
1  start:  alloc a, asize
2          alloc b, bsize
3  loop:   phi i, [#0, i.1]
4          gep t1, a, i
5          ld l2, t1
6          gep t3, b, t2
7          ld t4, t3
8          add t5, t4, #1
9          str t3, t5
10         add i.1, i, #1
11         cmp size, i.1
12         bne loop
```

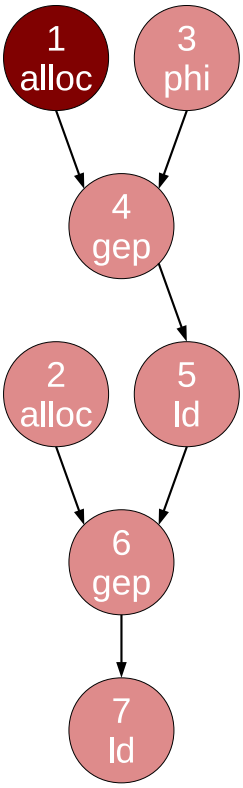
(a) Original compiler LLVM IR, for the code in code listing 3.1

```
1 add p1, i, #32
2 min p2, p1, asize
3 gep p3, a, p2
4 ld p4, p3
5 gep p5, b, p4
6 prefetch p5
7 add p6, i, #64
8 gep p7, a, p6
9 prefetch p7
```

(d) Generated prefetching code



(b) Depth-first search



(c) Fault avoidance

Figure 3.2: My automated software prefetching compiler pass running on an integer sort benchmark [18].

to the constant 64 and still achieve near-optimal performance. This is across a wide range of systems ranging from simple in-order processors to aggressive out-of-order superscalars. The reason is because as long as a large number of prefetches are active at once, and prefetches aren't evicted from the L1 cache before they are used, a large amount of leeway from the true optimal value can be tolerated.

3.2.5 Example

An example of my automated software prefetching pass in operation is given in figure 3.2. From the load in line 7 in figure 3.2(a), we work backwards through the data-dependence graph (DDG) using a depth-first search. The path followed is shown in figure 3.2(b). From the gep in line 6, we find an alloc that is not in a loop (line 2), and so stop searching down this path and follow the next. We subsequently encounter the ld in line 5 and continue working through the DDG until reaching the alloc in line 1, which is also outside a loop, stopping search down this path. These two allocation instructions give the bounds of the a and b arrays.

Continuing along the other path from the gep is the phi in line 3, at which point we

have found an induction variable. We take this set of instructions along the path from the phi node to the original load (dark red in figure 3.2(b)) and note that there are two loads that require prefetching (lines 7 and 5). Therefore we calculate the offset for the original load as 32 and that for the load at line 5 as 64. One of the prefetches itself introduces a new load. We therefore perform fault avoidance analysis (figure 3.2(c)) by searching back from the old load, to find the address bounds (dark red) of array a from its allocation instruction.

From this, we generate the code shown in figure 3.2(d), where all references to i are replaced with min(i+32, asize) for the prefetch at line 6 to avoid creating any faults with the intermediate load (line 4).

3.2.6 Prefetch loop hoisting

It is possible for analysed loads to be within inner loops relative to the induction variable observed. In this case, the inner loop may not feature an induction variable (for example, a linked-list-walking loop), or may be too small to generate look-ahead from. However, if we can guarantee control flow, and remove loop-dependent values for some iterations, it may be beneficial to add prefetches for these outside the inner loop.

This is implemented by generating prefetches for loads inside loops where control flow indicates that any phi nodes used in the calculation reference a value from an outer loop. The phi node in the prefetch is replaced with the value from the outer loop, and then the algorithm attempts to make the prefetch loop invariant by hoisting the instructions upwards. This will fail if there are other loop-invariant values on which the load depends. We must also guarantee the control flow occurs such that the loads generated by the software prefetches won't cause any new faults to occur. We can do this provided we can guarantee execution of any of the original loads we duplicate to generate new prefetches, or that the loads will be valid due to other static analyses.

3.2.7 Summary

I have described a compiler pass to automatically generate software prefetches for indirect memory accesses, which are likely to miss in the cache, cannot be picked up by current hardware prefetchers, and are simple to extract look-ahead from. I have further provided a set of sufficient conditions to ensure the code generated will not cause memory faults, provided the original code did not feature memory faults. I have also described a scheduling technique for these prefetches which is aimed at modern architectures where, despite variation in performance, the critical determiner of look-ahead distance is how many dependent loads are in each loop, rather than the total number of instructions.

System	Specifications
Haswell	Intel Core i5-4570 CPU, 3.20GHz, 4 cores, 32KiB L1D, 256KiB L2, 8MiB L3, 16GiB DDR3
Xeon Phi	Intel Xeon Phi 3120P CPU, 1.10GHz, 57 cores, 32KiB L1D, 512KiB L2, 6GiB GDDR5
A57	Nvidia TX1, Arm Cortex-A57 CPU, 1.9GHz, 4 cores, 32KiB L1D, 2MiB L2, 4GiB LPDDR4
A53	Odroid C2, Arm Cortex-A53 CPU, 2.0GHz, 4 cores, 32KiB L1D, 1MiB L2, 2GiB DDR3

Table 3.1: System setup for each processor evaluated.

3.3 Experimental setup

I implement the algorithm described in section 3.2 as an LLVM IR pass [65], which is used within Clang. Clang cannot generate code for one architecture I evaluate on, the Xeon Phi, so instead I manually insert the same prefetches my pass generates for the other architectures and compile using ICC. For Clang, I always use the O3 setting, as it is optimal for each program; however, for ICC I use whichever of O1, O2 or O3 works best for each program. I set $c = 64$ for all systems to schedule prefetches, as described in section 3.2.4, and evaluate the extent to which this is suitable in section 3.4.2.

3.3.1 Systems

Table 3.1 shows the parameters of the systems I have evaluated. Each is equipped with a hardware prefetcher to deal with regular access patterns; my software prefetches are used to prefetch the irregular, indirect accesses based on arrays. Haswell and A57 are out-of-order superscalar cores; A53 and Xeon Phi are in-order.

3.3.2 Benchmarks

To evaluate software prefetching, I use a variety of benchmarks that include indirect loads from arrays that are accessed sequentially. I run each benchmark to completion, timing everything apart from data generation and initialisation functions, repeating experiments three times.

3.3.2.1 Integer Sort (IS)

Integer Sort is a memory-bound kernel from the NAS Parallel Benchmarks [18], designed to be representative of computational fluid dynamics workloads. It sorts integers using a bucket sort, walking an array of integers and resulting in array-indirect accesses to

increment the bucket of each observed value. I run this on the NAS parallel benchmark size B [18] and insert software prefetches in the loop that increments each bucket, by looking ahead in the outer array, and issuing prefetch instructions based on the index value from the resulting load.

3.3.2.2 Conjugate Gradient (CG)

Conjugate Gradient is another benchmark from the NAS Parallel suite [18]. It performs eigenvalue estimation on sparse matrices, and is designed to be typical of unstructured grid computations. As with IS, I run this on the NAS parallel benchmark size B.

The sparse matrix multiplication computation exhibits an array-indirect pattern, which allows software prefetches to be inserted based on the benchmark's NZ matrix (which stores non-zeros), using the stored indices of the dense vector it points to. The irregular access is on a smaller dataset than IS, meaning it is more likely to fit in the L2 cache, and presents less of a challenge for the TLB system.

3.3.2.3 RandomAccess (RA)

HPCC RandomAccess is from the HPC Challenge Benchmark Suite [72], and is designed to measure memory performance in the context of HPC systems. It generates a stream of pseudo-random values which are used as indices into a large array. The access pattern is more complicated than in CG and IS, in that we must look ahead in the random number array, then perform a hash function on the value to generate the final address for prefetching. Thus, each prefetch involves more computation than in IS or CG.

3.3.2.4 Hash Join 2EPB (HJ-2)

Hash Join [91] is a kernel designed to mimic the behaviour of database systems, in that it hashes the keys of one relation, and uses them as index into a hash table. Each bucket in the hash table is a linked list of items to search within. In HJ-2, I run the benchmark with an input that creates only two elements in each hash bucket, causing the access pattern to involve no linked-list traversals (due to the data structure used). Therefore, the access pattern is prefetched by looking ahead in the first relation's keys, computing the hash function on the value obtained, and finally a prefetch of this hashed value into the hash table. This is similar to the access pattern in RA, but involves more control flow, therefore, more work is done per element.

3.3.2.5 Hash Join 8EPB (HJ-8)

This kernel is the same as HJ-2, but in this instance the input creates eight elements per hash bucket. This means that, as well as an indirect access to the hash-table bucket, there

are also three linked-list elements to be walked per index in the key array we can use for look-ahead. It is unlikely that any of these loads will be in the cache, therefore there are four different addresses we must prefetch per index, each dependent on loading the previous one. This means a direct prefetch of the last linked-list element in the bucket would typically cause three cache misses to calculate the correct address. To avoid this, we can stagger prefetches to each element, making sure the previous one is in the cache by the time the next is prefetched in a future iteration. For example, we can fetch the first bucket element at offset 16, followed by the first linked-list element at offset 12, then offsets 8 and 4 for the second and third respectively.

3.3.2.6 Graph500 Seq-CSR (G500)

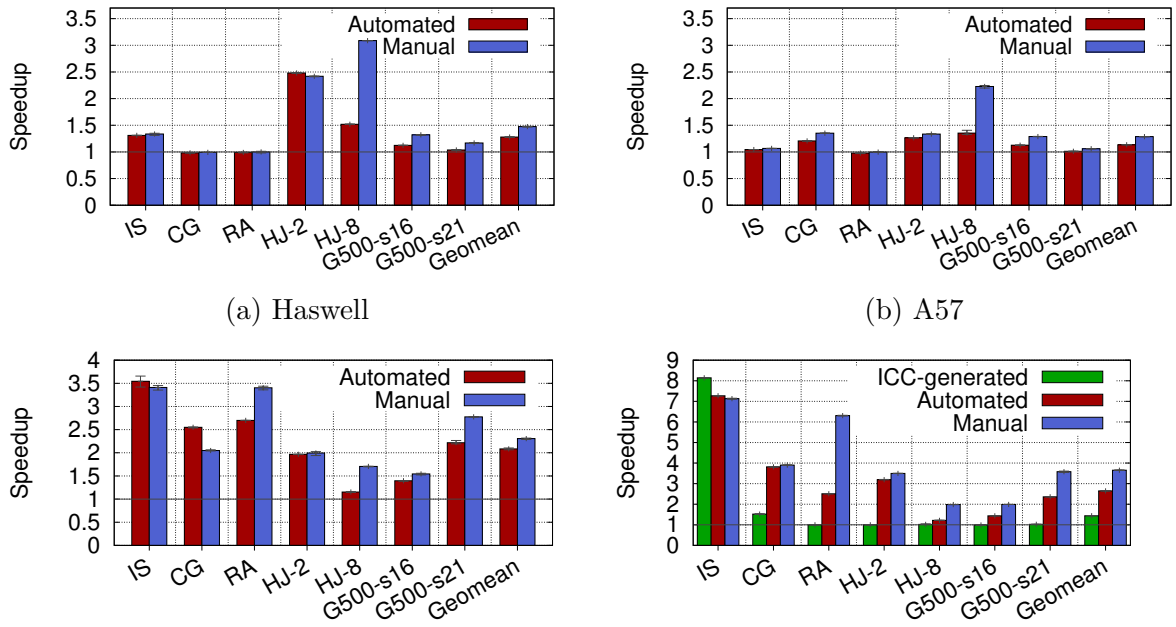
Graph500 [79] is designed to be representative of modern graph workloads, by performing a breadth-first search on a generated Kronecker graph in compressed sparse-row format. This results in four different possible prefetches. We can prefetch each of the vertex, edge and parent lists from the breadth-first search's work list using a staggered approach, as for HJ-8. Further, as there are multiple edges per vertex, we can prefetch parent information based on each edge, provided the look-ahead distance is small enough to be within the same vertex's edges. The efficacy of each prefetch then depends on how many instructions we can afford to execute to mask the misses and, in the latter case, how likely the value is to be used: longer prefetch distances are more likely to successfully hide latency, but are less likely to be in the same vertex, and thus be accessed.

I run this benchmark on both a small, 10MiB Graph, with options -s 16 -e 10 (**G500-s16**), where -s specifies the log of the number of vertices and -e specifies the average number of edges per vertex, and a larger 700MiB graph (**G500-s21**, options -s 21 -e 10), to get performance for a wide set of inputs with different probabilities of the data already being in the cache.

3.3.3 Implementation details

The full source of my automated software prefetching technique is available as an LLVM pass both on Github [5] and in a data repository [4], complete with integration into the Collective Knowledge [38] framework, to make it easy to automate and extend evaluation.

In terms of LLVM details, it is implemented as a FunctionPass. The depth-first search is implemented as given in algorithm 1, though with optimisation to avoid recalculating redundant depth-first search trees, to improve time complexity. Though in pseudocode, the cases of induction over an array, and the increment of a pointer within a range, are treated identically, in practice they result in different LLVM IR, and so both are treated as separate cases. Loops are rerolled before the pass is run, to simplify the analysis, and



(a) Haswell

(b) A57

(c) A53

(d) Xeon Phi

Figure 3.3: Performance of my automated software prefetching pass and the best manual software prefetches found. Also shown for the Xeon Phi is the performance of ICC-generated software prefetches.

the pass is run during the EP_VectorizerStart phase of the compiler: this is just before vectorisation occurs, which limits some of the more complex cases to increase coverage of the prefetch generation, but is otherwise relatively late in the compilation stage, to allow other optimisations to occur first.

3.4 Evaluation

I first present the results of my automated software prefetching pass across benchmarks and systems, showing significant improvements comparable to fine-tuned manual insertion of prefetch instructions. I then evaluate the factors that affect software prefetching in different systems, and consider the reasons for these.

3.4.1 Automated software prefetching performance

Figure 3.3 shows the performance improvement for each system and benchmark using my automated software prefetching compiler pass, along with the performance of the best manual software prefetches I could generate.

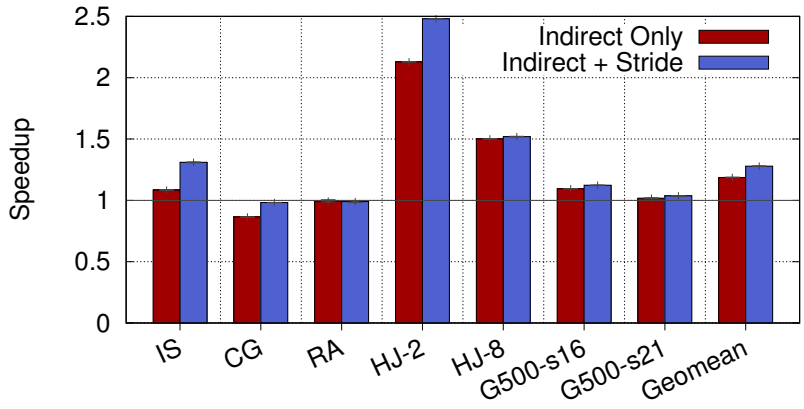


Figure 3.4: Performance of inserting staggered stride software prefetches along with the indirect prefetch, compared to the indirect alone, for Haswell, with my automated scheme.

3.4.1.1 Haswell

Haswell gets close to ideal performance on HJ-2, and IS, as the access patterns are fully picked up by the automated software prefetching compiler pass. This is also true of CG but, as with RA, performance improvement with software prefetches is limited because the cost of executing the additional code masks the improvement in cache hit rates.

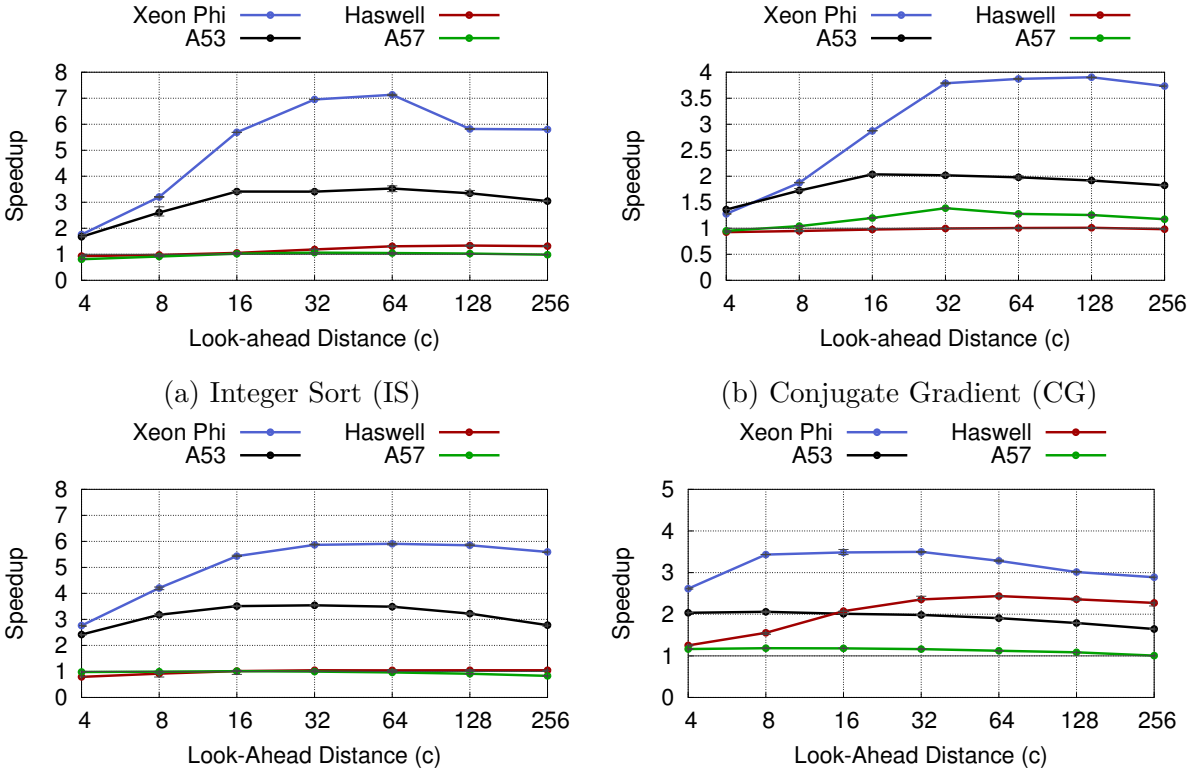
HJ-8 gets a limited improvement. The stride-hash-indirect pattern is picked up by the compiler, but the analysis cannot pick up the fact that we walk a particular number of linked-list elements each time in a loop. This is a runtime property of the input that the compiler could never know, but manual prefetches can take advantage of this additional knowledge.

While G500 shows a performance improvement for both the s16 and s21 setups (described previously in section 3.3.2.6), it isn't close to what we can achieve by manual insertion of prefetch instructions. This is because the automated pass cannot pick up prefetches to the edge list, the largest data structure, due to complicated control flow. In addition, it inserts prefetches within the innermost loop, which are suboptimal on Haswell due to the stride-indirect pattern being short-distance, something only known with runtime knowledge.

3.4.1.2 A57

The performance for the Cortex-A57 follows a similar pattern to Haswell, as both are out-of-order architectures. For IS, CG and HJ-2, differences between the automated pass and manual prefetches are simply down to different code generation. However, the A57 can only support one page-table walk at a time on a TLB miss, limiting improvements for IS and HJ-2. CG's irregular dataset is smaller than for other benchmarks, so fewer page-table walks are required and a lack of parallelism in the TLB system doesn't prevent memory-level parallelism from being extracted via software prefetch instructions. The

newer Cortex-A73 is able to support two page-table walks at once [36], likely improving prefetch performance.



(a) Integer Sort (IS)

(b) Conjugate Gradient (CG)

(c) RandomAccess (RA)

(d) Hash Join 2EPB (HJ-2)

Figure 3.5: Varying look-ahead distance shows that 64 is consistently close to optimal regardless of the system.

3.4.1.3 A53

As the Cortex-A53 is in-order, significant speedups are achieved across the board using my automated software prefetching compiler pass. RA achieves a significant improvement in performance because the core cannot overlap the irregular memory accesses across loop iterations at by itself (because it stalls on load misses). However, automated software prefetching performance for RA is lower than manual, as the inner loop is small (128 iterations). Though this loop is repeated multiple times, the automated-software-prefetching compiler analysis is unable to observe this, and so does not generate prefetches for future iterations of the outside loop, meaning the first few elements of each 128 element iteration miss in the cache.

In the G500 benchmark, the edge-to-visited-list stride-indirect patterns dominate the execution time on in-order systems, because the core does not extract any memory-level parallelism on its own. Therefore, autogenerated performance is much closer to ideal than on the out-of-order systems.

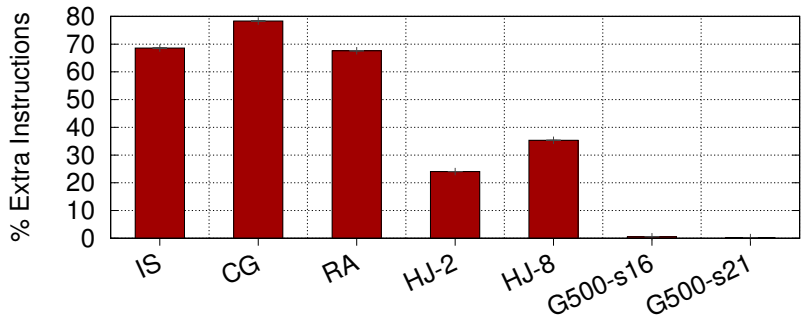


Figure 3.6: Percentage increase in dynamic instruction count for Haswell as a result of adding software prefetches, with the optimal scheme chosen in each case.

3.4.1.4 Xeon Phi

The Xeon Phi is the only system I evaluate for which the ICC compiler can already generate software prefetches for some indirect access patterns, using an optional flag. Therefore, figure 3.3(d) also shows prefetches autogenerated by the Intel compiler's own pass, "ICC-generated".

For the simplest patterns, IS and CG, which are pure stride-indirects, the compiler is already able to generate prefetches successfully. For IS, Intel's compiler is more optimal than mine, and than the prefetches I insert by hand, due to reducing overhead by moving the checks on the prefetch to outer loops.

As the Intel pass only looks for the simplest patterns, their algorithm entirely misses the potential for improvement in RA and HJ-2, as it cannot pick up the necessary hash computation. Its pass also misses out on any performance improvement for G500, despite the two simple stride-indirects present, from both work-to-vertex lists and edge-to-visited lists, likely because it is unable to determine the size of arrays and guarantee the safety of inserting loads to the work list and edge list structures.

We see dramatic performance improvements across the board on this architecture. The in-order Xeon Phi is unable to parallelise memory accesses by itself, so prefetching is necessary for good performance.

3.4.1.5 Stride prefetch generation

As discussed previously in section 3.1.2, performance for prefetching is optimal when, in addition to the prefetch for the indirect access, a staggered prefetch for the initial, sequentially-accessed array is also inserted. Figure 3.4 shows this for each benchmark on Haswell for my automated software prefetching scheme: performance improvements are observed across the board, despite the system featuring a hardware stride prefetcher.

3.4.2 Look-ahead distance

Figure 3.5 gives speedup plotted against look-ahead distance (c from equation 4.3 in section 3.2.4) for IS, CG, RA and HJ-2 for each architecture. Notably, and perhaps surprisingly, the optimal look-ahead distance is relatively consistent, despite the wide disparity in the number of instructions per loop, microarchitectural differences, and varied memory latencies. Setting $c = 64$ is close to optimal for every benchmark and microarchitecture combination. The A53 has an optimal look-ahead slightly lower than this, at 16–32, depending on the benchmark, as does the Xeon Phi on HJ-2, but performance at $c = 64$ is always within 10% of the optimal, and we can set c generously. The trends for other benchmarks are similar, but as there are multiple possible prefetches and thus multiple offsets to choose in HJ-8 and G500, I show only the simpler benchmarks here.

The reasons for this behaviour are twofold. First, the optimal look-ahead distance in general for a prefetch is the memory latency divided by the time for each loop iteration [78]. However, for memory-bound workloads, the time per loop iteration is dominated by memory latency, meaning that high memory latencies (e.g., from GDDR5 DRAM), despite causing a significant overall change in performance, have only a minor effect on look-ahead distance.

Second, it is more detrimental to be too late issuing prefetches than too early. Although the latter results in cache pollution, it has a lower impact on performance than the increased stall time from the prefetches arriving too late. This means we can be generous in setting look-ahead distances in general, with only a minor performance impact.

This lookahead distance is configurable in the technique I have built. However, these results suggest that it is undesirable, for the most part, to expose this complexity to the end user: the extra performance attainable is typically very low.

3.4.3 Summary

My automated software prefetching pass generates code with close-to-optimal performance compared to manual insertion of prefetches across a variety of systems, except where the optimal choice is input dependent (HJ-8), or requires complicated control-flow knowledge (G500, RA).

A compiler pass that is microarchitecture specific would only improve performance slightly: similar prefetch look-ahead distances are optimal for all the architectures I evaluate, despite large differences in performance and memory latency.

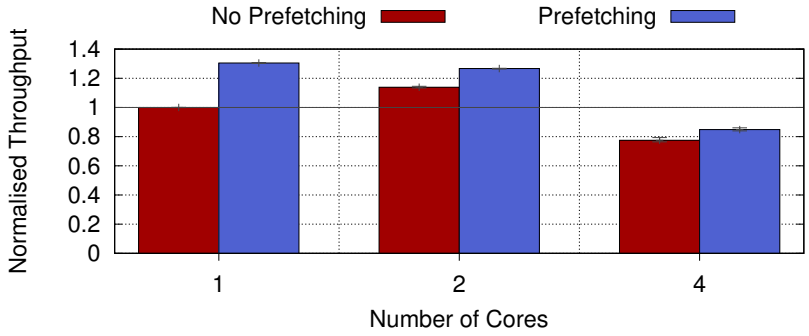


Figure 3.7: Throughput for IS on Haswell, normalised to one task running on one core without prefetching. A value of 1 indicates the same amount of work is being done per time unit as the program running on one core without prefetching.

3.5 What affects the effectiveness of software prefetch?

3.5.1 Costs of prefetching

For some benchmarks, the expense of calculating the prefetches outweighs the benefit from a reduction in cache misses. Figure 3.6 shows the increase in dynamic instruction count for each benchmark on Haswell. For all but the Graph500 benchmarks, dynamic instruction count increases dramatically by adding software prefetching, by almost 70% for IS and RA, and almost 80% for CG. In Graph500 workloads, prefetches reduce performance on Haswell within the innermost loop, and thus are only used on outer loops.

3.5.2 Bandwidth

DRAM bandwidth can become a bottleneck for some systems and benchmarks. Out-of-order cores can saturate the bus by executing multiple loops at the same time. This is demonstrated in figure 3.7. IS running on multiple cores slows down significantly on Haswell, with throughput below 1 for four cores, meaning that running four copies of the benchmark simultaneously on four different cores is slower than running the four in sequence on a single core. This shows that the shared memory system is a bottleneck. However, even with four cores, software prefetching still improves performance.

3.5.3 Compute versus memory

HJ-2 is conspicuous by its very large performance improvement on out-of-order superscalar cores (2.5× on Haswell). This benchmark is notable in that both the prefetching and the load itself requires extra computation when compared with most of the other benchmarks: the memory access involves performing a hash computation as well as an indirect access. Intuitively, one might expect this code to be more compute-bound and less memory-bound as a result. However, this is not the case.

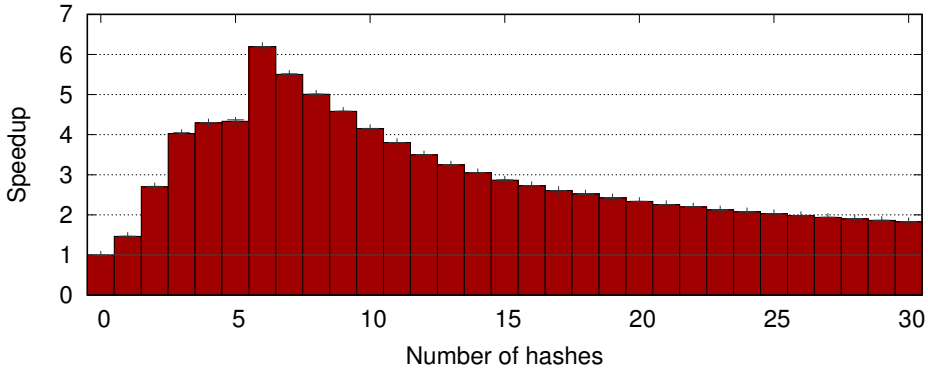
```
1 #define SIZE_OF_DATA 33554432
2 #define c_0 64
3 #define c_1 32
4
5 //initialisation
6 int* array[SIZE_OF_DATA];
7 int array2[SIZE_OF_DATA];
8 for(i=0; i<SIZE_OF_DATA; i++) {
9   array2[i]=i;
10  array[i]=&array2[hash(i)];
11 }
12
13 //timed
14 for (i=0; i<SIZE_OF_DATA; i++) {
15   __builtin_prefetch(&array[i+c_0]);
16   __builtin_prefetch(array[i+c_1]);
17   sum += hash(hash(...hash(*array[i])...));
18 }
```

Code listing 3.2: Pseudocode for Camel.

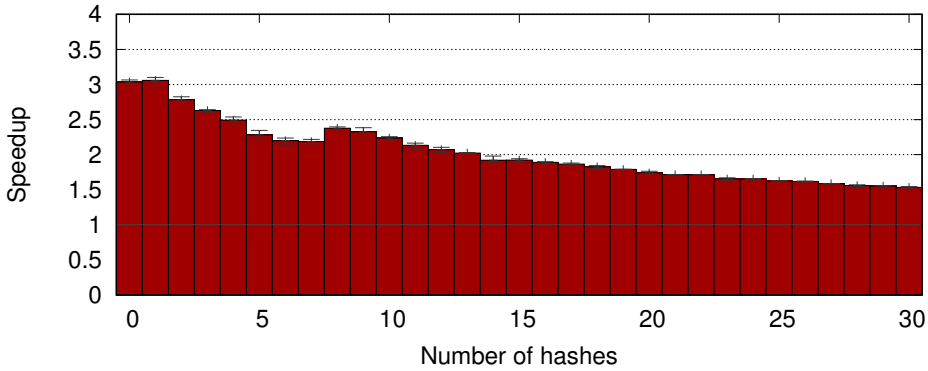
To explore this further, I designed a custom configurable benchmark, called Camel. This involves an array of pointers (the simplest indirect memory access) arranged to point to memory in a pseudorandom order, for which I add the results together of every integer pointed to by a pointer. The benchmark is configured by allowing varying numbers of nested hash computations on the integer before it is added to an accumulator. This allows varying ratios of compute versus memory access. Pseudocode for this is given in code listing 3.2.

Speedups for increasing amounts of computation are given in figure 3.8. The pattern for Haswell (figure 3.8(a)) is surprising. Prefetching doesn't improve performance at all for the basic array-of-pointer access, but as more compute is added, speedups reach a peak of over 6×, and tail off extremely gradually. Even with a huge amount of extra computation, speedups of approximately 2× are observed. This gives us the paradoxical result that making an indirect access more compute bound on an out-of-order core also makes it more memory bound, increasing the potential for prefetching. The reason is that the extra compute code clogs up the reorder buffer of the processor, meaning the ability of the out-of-order processor to exploit memory-level parallelism itself is reduced.

By comparison, the in-order Cortex A53 (figure 3.8(b)) is much less perplexing. Here, prefetching performance peaks when the ratio of memory accesses to compute is least. This is because it cannot reorder memory accesses by itself, and so the software prefetching isn't competing with any reordering hardware. Still, that prefetching performance is high even with large amounts of compute added goes to show how devastating for performance indirect accesses can be.



(a) Haswell



(b) A53

Figure 3.8: Speedup for Camel as a result of prefetching, when varying the number of hash computations performed on each data element, but keeping the memory-access pattern the same.

3.6 Limitations of software prefetching

We have seen that software prefetching can improve performance for memory-bound indirect accesses, and further that it is possible to automate the discovery and generation of these in the compiler. However, software prefetching has a number of fundamental limitations that make hardware schemes more desirable:

- **The extra instruction overhead lowers performance.** Prefetching can often double the dynamic instruction count, in that if a loop is mostly memory accesses, all of those will have to be duplicated. Indeed, the extra code can result in overheads even higher than this, by reducing the amount of loop unrolling the compiler is willing to do, and because dependent loads need duplicating multiple times.

- **Inability to react to prefetches as they arrive.** This means that if we need to load the result of another prefetch (necessary even in the simplest stride-indirect) we have to issue both a prefetch and a load to that data, and schedule them accordingly. More generally, this leads to $O(n^2)$ prefetch code for $O(n)$ code, with an n-deep

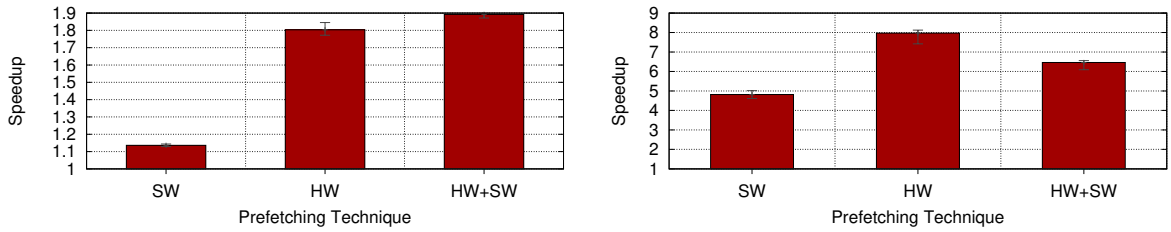
indirection in the memory access.

- **Rigid scheduling.** Though variances in memory latency and compute speed can be overcome for many cases with the simple scheduling scheme described in this chapter, it still isn't possible to always achieve the true optimum regardless of microarchitecture, and it isn't possible to alter distance or throttle based on dynamic behaviour of the system, as the prefetching schedule is fixed at compile time. Matters are made worse when we don't know how many elements we need to access, for example when fetching from multiple short linked lists as can occur with hash tables. This means that creating a fixed schedule to look up elements at well-spaced offsets becomes impossible, as we cannot set those fixed offsets without knowing the number of elements we should set in the first place. Even for simple cases, a look-ahead of 64 isn't always optimal. If we could work out the true value with runtime analysis, this would further improve performance.

- **Difficulty with loops and ranges.** Describing a software prefetch for simple patterns is possible, but when there is > 1 fan-out from one value to the next, for example because we need to fetch a range of values, prefetching loses steam. This is because we need to introduce complicated data-dependent control flow, which requires looping behaviour to fetch a variable number of cache lines.

- **Non-speculative loads.** Prefetches are speculative, which means that incorrect prefetches should not cause incorrect behaviour. However, loads used in the generation of addresses to issue a software prefetch can cause faults, which stop execution. This means we have to be overly conservative with the prefetches we generate in software: a scheme that could silently throw away a load request for a prefetch when an error has occurred would ease both automatic deployment by simplifying analysis, and allow more patterns to be prefetched.

- **Finding delinquent loads.** The term "delinquent load" [85] refers to a situation where a small number of load instructions account for most of a program's cache misses. From the source alone, it can be difficult to work out whether a load is likely to miss in the cache. This is made easier by targeting indirect patterns, as there will be some data-dependent behaviour, but the dynamic execution may still show a regular address-pattern, and thus the overhead of the software prefetching will not be mitigated by the benefits. If we could offload this cost, or turn the prefetching on or off based on dynamic system behaviour, then incorrect speculation of a load requiring prefetching at compile time would not cause performance issues.



(a) Stride (b) Iter

Figure 3.9: Performance improvement on Haswell from software prefetching for the same indirect memory-access pattern as Integer Sort, only with dynamic access patterns predictable by the stride prefetcher.

3.6.1 How close to a hardware prefetcher can software get?

Here I show that, even for the simplest indirect patterns, software prefetching is inadequate compared to a hardware implementation. We can do this by using the same pattern of indirection featured in Integer Sort, if we make sure the memory-access pattern is regular at runtime by choosing the order of data items. This means it can be picked up by a stride prefetcher. I test two patterns:

- **Stride**: One element is accessed per cache line, to avoid any within-line locality.
- **Iter**: Words are accessed successively, with 16 integers per cache line, to give optimal performance improvement from the stride prefetcher.

The performance improvement for software prefetching, and the hardware stride prefetcher, for Haswell is shown in figure 3.9. Even though we prefetch every memory access done in the timed section in hardware, and software prefetching does improve performance, it cannot achieve the same degree of performance improvement that the hardware prefetcher can. Indeed, for Iter, enabling software prefetching in addition to the hardware prefetcher significantly reduces performance. This is because of the extra overheads that we cannot avoid with software prefetching, even in these cases where software prefetching is relatively simple to achieve: to do better, we need to introduce some hardware support.

3.7 Conclusion

For patterns that current hardware stride prefetchers cannot pick up, it is possible to use software prefetching instructions to improve performance. Here I have expanded the state of the art by designing and implementing an automated compiler pass to discover and generate prefetches, gaining performance improvements close to those possible by hand.

However, it is clear that there are fundamental limitations with software prefetching that limit its potential. The following chapters explore how we can get around these for specific memory-access patterns, then generalise this into a programmable hardware scheme that is designed to exploit maximal memory-level parallelism at low overheads.

Chapter 4

Configurable prefetching: a better prefetcher for graphs

Software prefetching can bring about significant performance improvements in many cases, but it is limited. A huge bottleneck in graph analytics workloads is the irregular memory accesses necessary due to the nature of calculation using complicated, highly irregular data structures. And yet, for out-of-order cores (Haswell and A57) the performance improvement for Graph500 in the previous chapter was highly limited. The question I ask here, then, is how can we do better if we have hardware support?

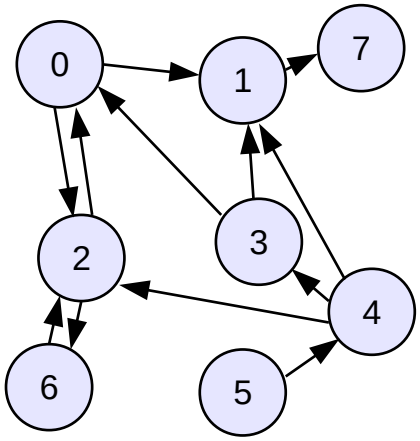
I use the example of breadth-first search to motivate the design of specialised, configurable prefetchers, which can be made aware of complicated memory-access patterns, to optimally prefetch the data in advance. This allows us to gain significantly larger performance improvements, geometric mean 2.3×, with a 3.3× maximum, on an out-of-order superscalar core, for Graph500[79] and code taken from the Boost Graph Library [90], with a configurable graph prefetcher. This work was presented in a paper at ICS 2016 [7].

4.1 A worked example: breadth-first search

Breadth-first search is a common access pattern in graph workloads: it can be used as a basic computation kernel to perform unweighted-distance calculations, connected components [83], maximum flows via the Edmonds-Karp algorithm [34], optimal decision tree walking in AI [29], betweenness centrality [21], and many other algorithms. More recently, the concept has been applied to very large graphs as a kernel within many sub-linear algorithms, that only operate on a small fraction of the input data [87]. Graph workloads operate on very large amounts of data, and thus the data does not fit in the cache, and memory accesses are highly irregular. This leaves this class of workload particularly memory-latency bound, and hence it is a good target for a specialised prefetcher.

```
Queue workList = {startNode}
Array visited[startNode] = true
while worklist ≠ ∅ do
    Vertex N = workList.dequeue()
    foreach Edge E ∈ N do
        if visited[E.to] is false then
            workList.enqueue(E.to)
            visited[E.to] = true
        end
    end
end
```

(a) Breadth-first search



(b) Graph

Figure 4.1: Pseudocode for a breadth-first search, along with an example graph.

4.1.1 Algorithm

Figure 4.1(a) shows pseudocode for a breadth-first search. From the starting vertex, computation moves through the graph adding vertices to a FIFO queue in the order observed via the edges out of each node. For example, starting at vertex 5 in figure 4.1(b), nodes are visited for breadth-first search in the order 5, 4, 1, 2, 3, 7, 0, 6.

4.1.2 Memory-access pattern

One of the most efficient, and therefore most common [11, 83], data structures for representing graphs is the compressed sparse row (CSR) format. An example of using such a data structure for breadth-first search on figure 4.1(b) from node 5 is given in figure 4.2.

In a compressed-sparse-row graph, vertices and edges are stored using an array each. The vertex list stores indices into the edge list, indicating the first edge of a vertex. The element after a given index represents the start of the edges for the next vertex, and thus we can infer the edge range from these two values for a given vertex. Each element in the edge list itself represents the index of a vertex at the other end of the edge.

In this specific example, the FIFO work list is used to work out the order in which to visit vertices, and we thus access it sequentially and use it to index into the vertex list. We then access the values indicating the start and end of the vertex, and use that to access a range of values from the edge list. We then use each edge in that range to index into the visited list. If a vertex on the end of an edge hasn't been visited yet, we then add it to the work list, and continue.

There are a number of reasons this access pattern is fundamentally complicated for the memory system. First, we have a four-deep indirection pattern: we index from the work list, to the vertex list, to the edge list, to the visited list, each encoded as an array. Even

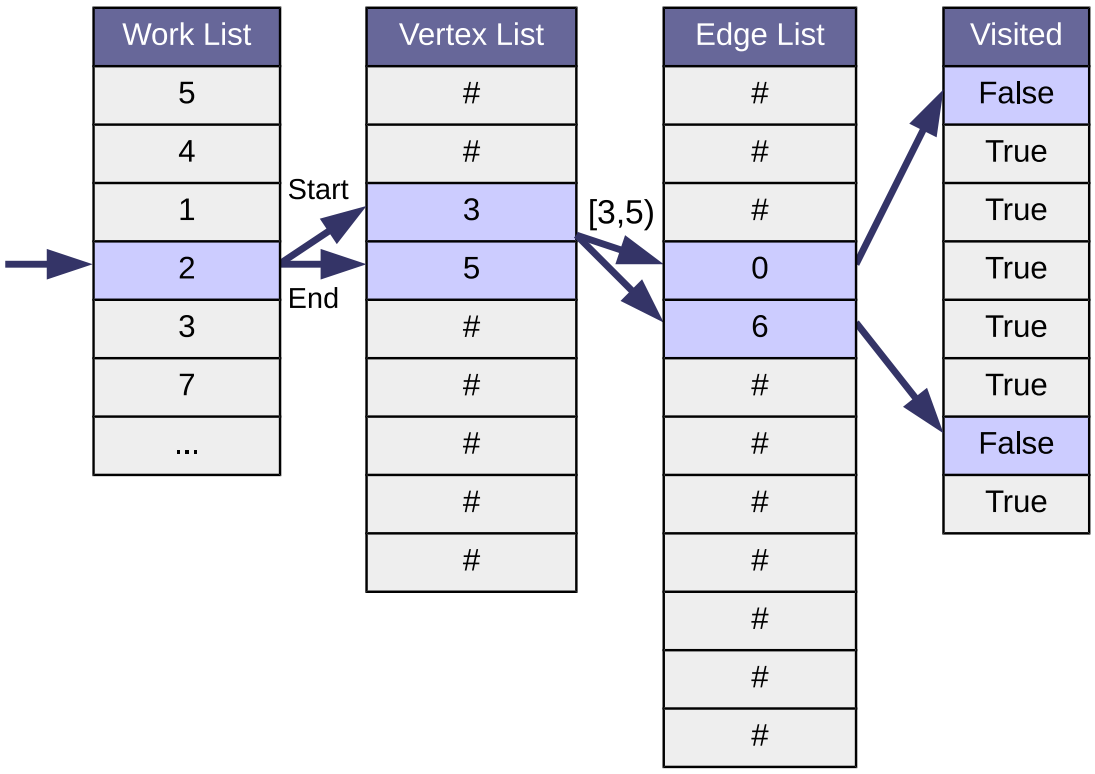


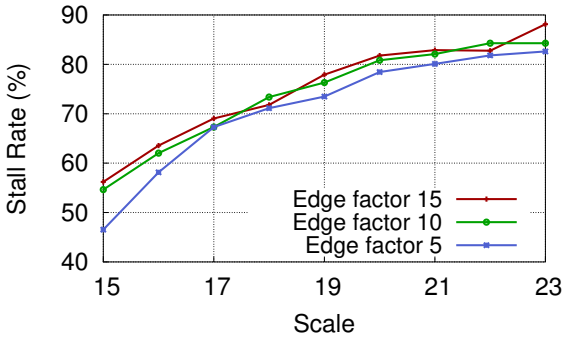
Figure 4.2: A breadth-first search on a compressed-sparse-row data structure.

more complicated is that we get a fan-out pattern: for each work list element, we must access two vertex list elements, a sequential range of edge list elements, then a variety of visited elements scattered around memory. This means that stalls are likely, as memory accesses are both data- and control-flow dependent.

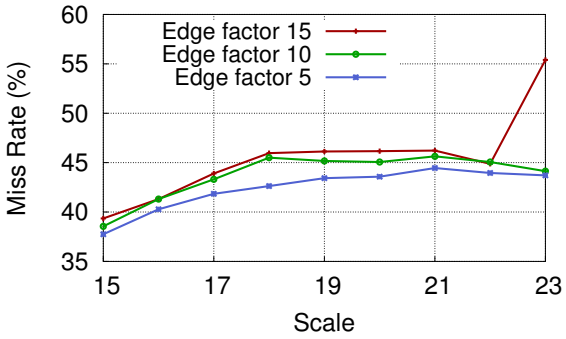
4.1.3 Profiling results

As a benchmark of breadth-first search on a compressed sparse row graph, I use Graph500 search [79]. Results based on running this on Haswell are shown in figure 4.3[1]. We see that Graph500 search experiences stall rates approaching 90%, increasing with graph size. This is due to L1 cache misses approaching 50%, as can be seen in figure 4.3(b). Figure 4.3(c) shows the breakdown of the extra time spent dealing with misses for different types of data, using gem5 [19] for the same benchmark with graph scale 16 and edge factor 10 (previously used in chapter 3). The majority of additional time is due to edge list misses (69%), because the edge list is twenty times larger than the vertex list. In addition, the array that records whether each vertex has been visited or not is also a significant source of miss time (25%). Although this is the same size as the vertex list[2], it is accessed frequently (once for each edge into a vertex) in a seemingly random order.

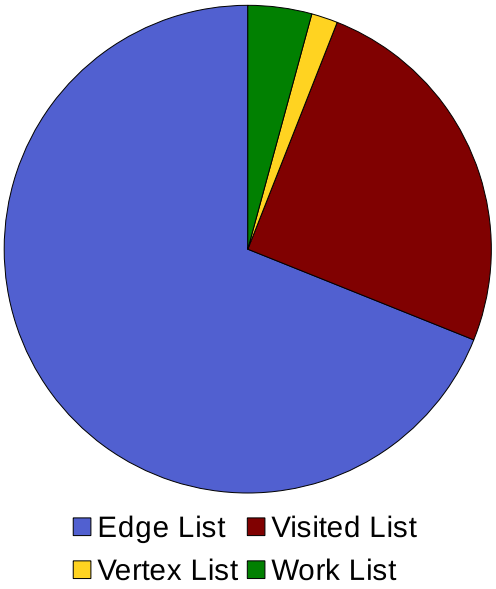
[1]Section 4.4 gives more detail on benchmarks, graphs and experimental setup.
[2]Graph500 search stores the integer parent of a node, rather than a boolean visited value.



(a) Stall rate



(b) L1 miss rate



(c) Source of misses

Figure 4.3: Core stall rate and L1 cache read miss rate for Graph 500 search. Loads from the edge and visited lists account for 94% of the misses.

4.1.4 Opportunity

These cache misses are not fundamental. We can predict the memory accesses that the program will perform many cycles into the future: as we see in figure 4.4, there are two different look-ahead patterns we can exploit. We can look ahead in the work list to find future vertices we will visit, for a coarse-grained look at what we will soon be accessing from memory. Likewise, when lots of edges exist for a given vertex, we can look ahead within the edge list to bring in elements from the visited list in parallel, for a more fine-grained form of memory-level parallelism.

4.1.5 How well can current software and stride prefetching do?

Figure 4.5 shows the ability to improve performance using current prefetching techniques for Graph500 search, on the out-of-order superscalar Haswell. We have already seen in chapter 3 that software prefetching is only mildly effective for Graph500, and it should come as little surprise, given the many data-dependent memory accesses, that the stride prefetchers available on the system also provide little benefit. The L1 DCU IP [93] prefetcher gives the most benefit, and this is likely picking up short-range sequential memory accesses in the edge list, along with the work list.

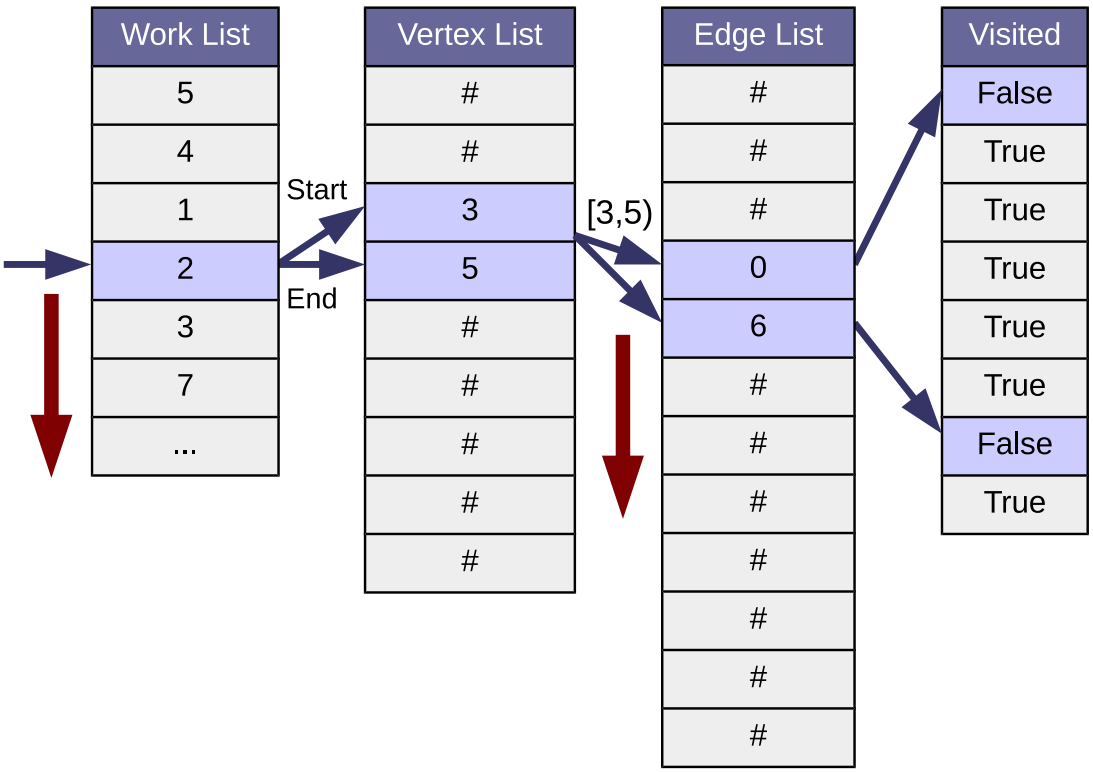
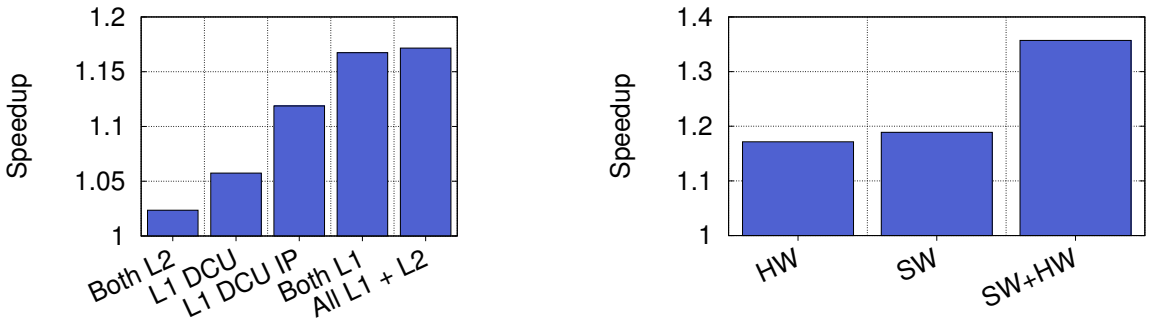


Figure 4.4: We can exploit look-ahead on both the work list and edge list to discover future memory accesses.

The workload is data-dependent, and the memory-access pattern is well defined and can be predicted well into the future using the work list. So why is software prefetching so limited in this case? Figure 4.6 shows the elements we can prefetch with any observable benefit. Prefetching from the vertex indices in the work list into the vertex list is a simple stride-indirect pattern, which is easy enough in software but as misses to the vertex list only cover a small fraction of the total misses, performance impact is negligible. We can also prefetch from the edge list, but having to prefetch a range of values causes issues: using a control-flow loop to prefetch the entire range of edges for a vertex is too complicated and thus causes slowdowns. The best software scheme is thus limited to loading an already prefetched edge index from the vertex list, and using that to prefetch two cache lines' worth of data from the edge list. Prefetching the visited list from these two cache lines in turn reduces performance, as the levels of indirection become too high, not enough of the visited list is covered, many false prefetches are generated as we may not access two cache lines of data, and since each requires its own prefetch, the fan-out generates too many instructions.

We could prefetch directly from the edge list into the visited list, using a stride-indirect pattern. But since this stride-indirect pattern is typically very short (we won't typically access the next contiguous vertex element's edge lists after the current vertex's edges), this is too unreliable to achieve any performance benefit compared to the out-of-order



(a) Hardware prefetchers

(b) Hardware and software prefetchers

Figure 4.5: Hardware and software prefetching on Graph 500 search with scale 21, edge factor 10.

core's own reordering ability.

All of this means that the prefetches we can do in software to improve Graph500 search on CSR graphs are limited to one prefetch to the vertex list, and two prefetches to two cache lines of the edge list. This covers a very small fraction of total misses, and so the benefit is limited compared to the ideal.

4.2 How can we do better than software prefetching?

Though there are limitations that cause us only to be able to profitably prefetch a small number of memory accesses in software, these are not fundamental. We can look ahead in the work list to fetch all elements of the graph that we will access shortly. However, to do this, we need support for several features unavailable from software prefetching.

- **React to prefetches, rather than re-load**: To prefetch chains of dependent loads in software, we need to first prefetch the first value at one offset, then later load that value in to issue the prefetch at a different offset, then later load the first and second at yet another offset to prefetch the third structure. This creates an $O(n^2)$ code growth along with a more rigid schedule than is strictly necessary. What we really want to do is prefetch a value, wait until it is brought into the cache, then use that to trigger a prefetch for the next value in the sequence.

- **No instruction overhead**: By implementing the traversal operations in hardware we can remove the instruction overhead of prefetching from the main instruction stream, thus allowing the processor to spend more of its time and energy on compute.

- **Prefetch ranges**: To profitably prefetch the edge list, we need to be able to prefetch a range of edge values for each vertex, and later trigger prefetches based on all of

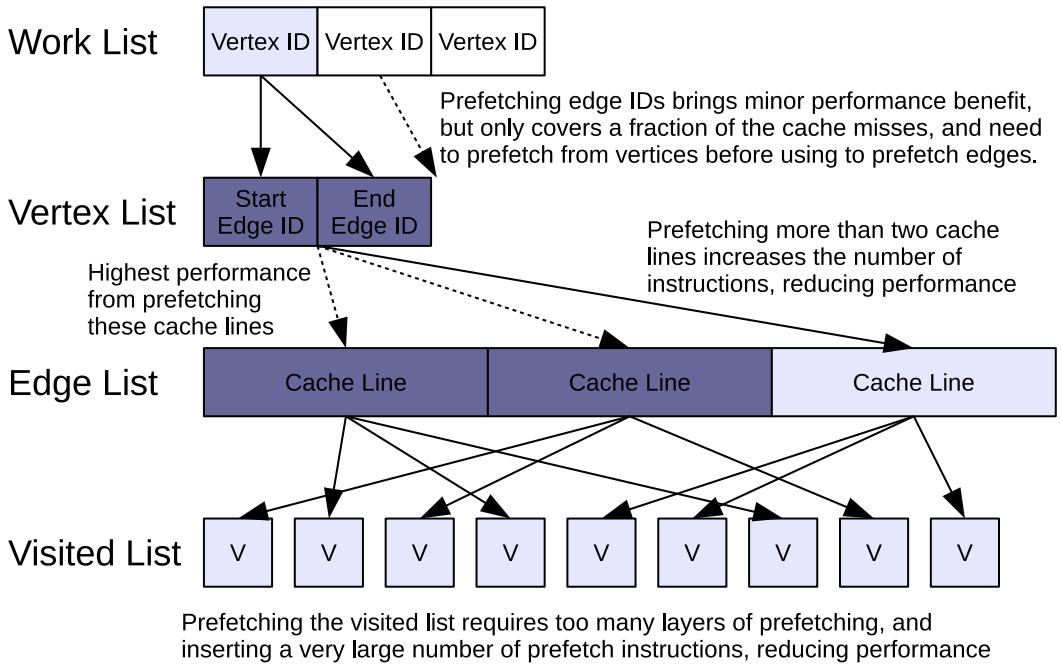


Figure 4.6: Loads as a result of visiting a node in a breadth-first search. Those which can be prefetched in software with any observable benefit are shown with dark shading.

them. Doing this in a software prefetch would result in costly loop control flow, whereas offloading it to a configurable unit would allow more direct implementation.

- **Dynamic scheduling**: With hardware support, there is no longer any need to choose a fixed look-ahead distance: we can dynamically choose it in hardware, and adapt to the specific microarchitecture and dynamic load the system is under. This is more important for a breadth-first search on graphs than with other patterns, as the work list isn't an inner loop: how far ahead we need to look depends on how many edges per vertex there are likely to be.

- **Selective enabling**: There are two possible prefetch positions on breadth-first search: from the work list to the other three data structures, and from the edge list to the visited list. Whether the latter is profitable or not depends on the number of edges per vertex: if this is low, then elements we prefetch won't actually be used. We therefore wish to selectively disable the latter depending on runtime characteristics of the data input, which may change throughout execution.

4.3 A configurable graph prefetcher

I present a configurable hardware prefetcher for traversals of graphs in CSR format, first starting with breadth-first search then moving on to more general patterns. It works by

snooping loads to the cache made by both the CPU and the prefetcher itself to trigger new prefetch requests. Figure 4.7 gives an overview of the system, which sits alongside the L1. Although it is more common to target the L2 cache, prefetching into the L1 provides the best opportunity for miss-latency reduction, and modern cores include prefetchers at both the L1 and L2 levels [93]. It also allows direct access to the data TLB, since the prefetcher works on virtual addresses, and a fine-grained view of all memory accesses, increasing observation granularity. Virtual address prefetchers have been proposed previously [96] and implemented both on Arm CPUs [1], and in the Itanium 2 on the instruction side [74].

As described in section 4.1.3, the majority of the benefits come from prefetching the edge and visited lists. However, these are accessed using the work list and vertex list. Therefore, the prefetcher is configured with the address bounds of all four of these structures (needed so that it can calculate addresses from indices), and prefetches issued for each, so a side effect of bringing in the data we care most about is that we also prefetch data work list and vertex list.

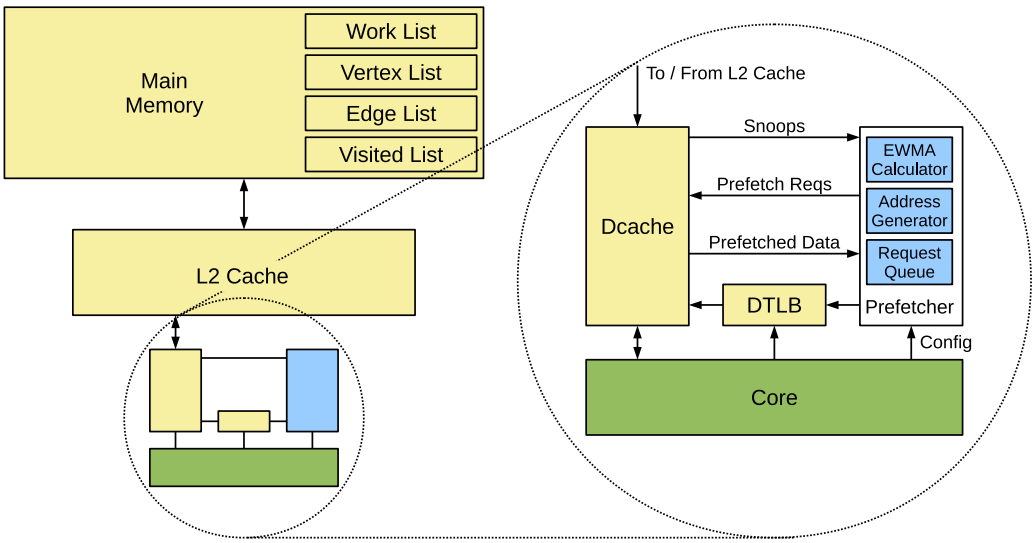
4.3.1 Basic operation

When the application thread is processing a vertex at index n in the work list, we need to prefetch data for vertex $n + o$, where o is a dynamically calculated offset representing the distance ahead that we wish to fetch, based on our expected ratio of fetch versus traversal latencies. Section 4.3.2 gives more information about the calculation of o. To prefetch all information related to the search, the prefetcher needs to perform a fetch of

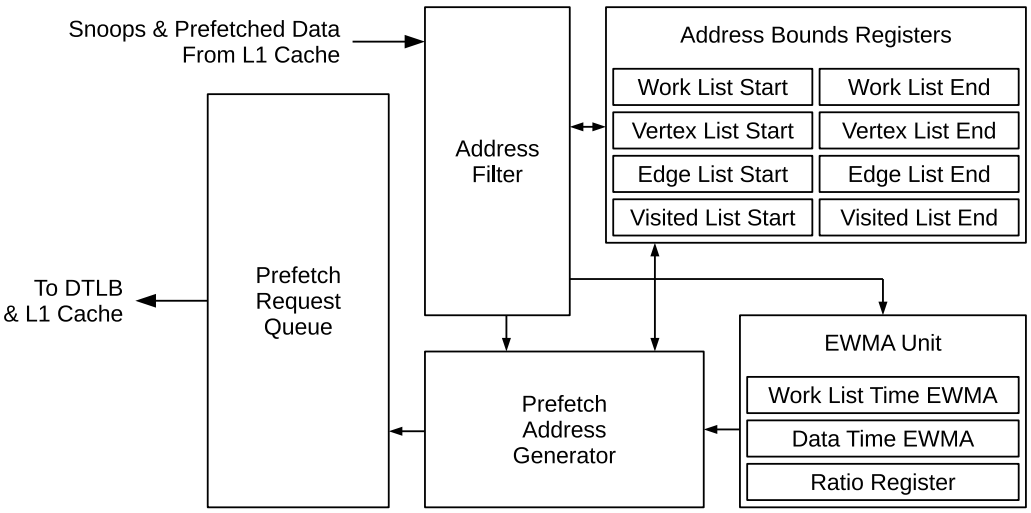
```
visited[edgeList[vertexList[workList[n+o]]]]
```

for all edges out of this node. Prefetching the first element, workList[n+o], gives the vertex ID, v, of the node and vertexList[v] brings in the start edge index. The end edge index (vertexList[v+1]) is usually in the same cache line; if not then I estimate that there will be two cache lines of edge data for the vertex. For each edge, e, prefetching edgeList[e] gives the node ID of a neighbouring vertex to v, which is also the index into the visited list.

The prefetcher snoops L1 accesses by the core. Observation of an access to workList[n] triggers a chain of dependent prefetches for node v, starting with the generation of a prefetch to workList[n+o], which the L1 issues when a miss status holding register (MSHR) is available. The prefetcher snoops the memory bus and detects the return of the data, which it copies. It can then calculate the address in the vertex list to access, and issue a prefetch for that. Similar actions are performed to generate prefetches for the edge and visited lists.



(a) System overview



(b) Graph prefetcher microarchitecture detail

Figure 4.7: A configurable graph prefetcher, configured with in-memory data structures, to which it snoops accesses.

4.3.2 Scheduling prefetches

The key questions with any prefetcher are what to prefetch and when. In the ideal case, we can prefetch all the information for the node at offset o from the current node on the work list using equation 4.1, where *work_list_time* is the average time between processing nodes on the work list and *data_time* is the average time to fetch in data required by a single vertex. In other words, all the data for node $n + o$ on the work list will arrive in the cache just in time for it to be required. Unlike in the software prefetching scheme in chapter 3, there is no need to use a statically calculated offset: we can instead use a

Vertex-offset mode

Observation	*Action*
Load from workList[n]	Prefetch workList[n+o]
Prefetch vid = workList[n]	Prefetch vertexList[vid]
Prefetch from vertexList[vid]	Prefetch edgeList[vertexList[vid]] to edgeList[vertexList[vid+1]] (12 lines max)
Prefetch vid = edgeList[eid]	Prefetch visited[vid]

Large-vertex mode

Observation	*Action*
Prefetch vid = workList[n]	Prefetch vertexList[vid]
Prefetch eid = vertexList[vid]	Prefetch edgeList[eid] to edgeList[eid + 8*lineSize - 1]
Load from edgeList[eid] where (eid % (4*lineSize)) == 0	Prefetch edgeList[eid + 4*lineSize] to edgeList[eid + 8*lineSize - 1]
Prefetch vid = edgeList[eid]	Prefetch visited[vid]
Prefetch edgeList[vertexList[vid+1]]	Prefetch workList[n+1]

Table 4.1: Actions taken by the prefetcher in response to observations on L1 activity.

dynamic implementation that makes use of runtime data.

$$o * work\_list\_time = data\_time \tag{4.1}$$

Since *work_list_time* and *data_time* can vary wildly both between and within applications, depending on the number of edges out of each node in the graph, I use exponentially weighted moving averages (EWMAs) to estimate their values for any given point in time. Equation 4.2 gives the generalised EWMA form. I use $\alpha = 8$, where α is a smoothing factor and a higher α indicates more smoothing, to estimate *work_list_time* and $\alpha = 16$ to estimate *data_time*, the latter of which is more heavily damped to avoid chance edges in the L2 from reducing the estimate too dramatically. I evaluate the impact of altering α in section 4.5.

$$avg\_time_{new} = \frac{new\_time + (\alpha - 1) avg\_time_{old}}{\alpha} \tag{4.2}$$

The EWMA approach works well for graphs of different sizes, as well as those with a highly-variable number of edges per vertex. Due to the bias of breadth-first search [62], a search is more likely to visit vertices with more edges first and smaller ones towards the end, and thus the search proceeds in phases.

4.3.2.1 Vertex-offset mode

When *data_time* > *work_list_time*, I use equation 4.3 to prefetch at an offset from the current node on the work list, where k is a multiplicative constant to mitigate the fact that an average always underestimates the maximum time to fetch (2 in my simulations), and also to bias the timeliness of the prefetcher to make it more conservative, ensuring data arrives in the cache before it is requested.

$$o = 1 + \frac{k * data\_time}{work\_list\_time} \tag{4.3}$$

The vertex-offset mode is used when prefetching all information for a node on the work list takes more time than the application takes to process each node. In this situation we need to start prefetches for several vertices in advance, in order to ensure the data is in the cache when the program wants to use it.

4.3.2.2 Large-vertex mode

On the other hand, when *data_time* < *work_list_time*, then each vertex takes longer to process than the time to load in all data for the next. Prefetching at a simple offset of 1 from the work list runs the risk of bringing data into the L1 that gets evicted before it is used. In this case the prefetcher enters *large-vertex mode*, where prefetches are based on the progress of computation through the current vertex's edges. As we know the range of edge list indices required, it prefetches 21 cache lines' worth of data (determined experimentally), followed by prefetches of stride size 14 upon read observation. In other words, it continually prefetches

```
firstLine = edgeList[idx + 14*lineSize]
```

where *idx* is the current edge list index being processed, and *lineSize* is the size of a cache line. This means we have a constant, small, fetch distance in these situations.

I schedule a fetch for the next vertex in the work list when we are four cache lines away from the end of the current vertex's edge list. Although we could use a variable distance based on past history, this access pattern involves comparatively few cache lines at once, so we can afford to be conservative, targeting the case where little work is done between edges, and all other cases will be adequately accommodated as a result.

4.3.3 Implementation

Given the two modes of operation described in section 4.3.2, the prefetcher can be implemented as several finite state machines that react to activity in the L1 cache that it snoops on. Table 4.1 shows the events that the prefetcher observes, along with the actions it takes in response.

4.3.3.1 Configuration

It is too complex for the prefetcher to learn the address bounds of each list in memory, therefore the application must explicitly specify these as a configuration step prior to traversing the graph. Although this requires a recompilation to make use of the prefetcher, functionality can be hidden in a library call and for high performance applications this is unlikely to be a major hurdle.

4.3.3.2 Operation

Whenever an address from a load or prefetch is observed, it is compared to each of the ranges to determine whether it is providing data from one of the lists. If so, then an appropriate prefetch can be issued to bring in more data that will be used in the future. For example, when in vertex-offset mode, a load from the work list kicks off prefetching data for a later vertex on the work list using the offset calculated in section 4.3.2. On the other hand, observation of a prefetch from the work list means that the prefetcher can read the data and proceed to prefetch from the vertex list.

The prefetcher assumes that consecutive values in the vertex list are available in the same cache line, which greatly reduces the complexity of the state machine as it never needs to calculate on data from multiple cache lines at the same time. The downside is that it reduces the capability of the prefetcher in cases where the start and end index of a vertex actually are in different cache lines. In these cases I assume all edge list information will be contained in two cache lines and, if we're in large-vertex mode, then this information is corrected once the true value has been loaded in by the application itself.

4.3.3.3 Hardware requirements

The configurable graph prefetcher consists of five structures, as shown in figure 4.7(b). Snooped addresses and prefetched data from the L1 cache are processed by the address filter. This uses the address bounds registers to determine which data structure the access belongs to, or to avoid prefetching based on L1 accesses to memory outside these structures. We require 8 64-bit registers to store the bounds of the 4 lists when traversing a CSR graph.

Accesses that pass the address filter move into the prefetch address generator. This contains two adders to generate up to two new addresses to prefetch, based on the rules shown in table 4.1. In addition, for prefetches to the work list, it reads the values of the three registers from within the EWMA unit. The output is up to two prefetch addresses which are written into the prefetch request queue.

Alongside the three registers (two EMWAs and one ratio), the EMWA unit contains logic for updating them. The EWMAs are efficient to implement [31], requiring an adder

and a multiplier each, and can sample accesses to the lists to estimate their latencies. The ratio register requires a divider, but as the ratio is updated infrequently it need not be high performance.

In total, the prefetcher requires just over 1.6KiB of storage (200 × 64-bit prefetch request queue entries and 11 × 64-bit registers), 4 adders, 2 multipliers and a divider. This compares favorably to stride prefetchers (typically around 1KiB storage) and history-based prefetchers, such as Markov [49], which require large stores (32KiB to MiBs[35]) of past information to predict the future.

4.3.4 Generalised configurable prefetching

While my configurable graph prefetcher is designed to accelerate sequential breadth-first search, it can also be used for a parallel search or other traversals on CSR graphs. Indeed, specialised configurable prefetchers could be designed for many different memory-access patterns, and a user or compiler could choose which to use for their application, and configure it with the correct address bounds accordingly.

4.3.4.1 Parallel breadth-first search

For graphs with many edges and low diameters, it may be beneficial to parallelise the whole breadth-first search on multiple cores [68]. This exchanges the FIFO queue assumed above for a bag, where multiple threads can access disjoint areas of the structure, which are conceptually smaller queues. My configurable graph prefetcher works without modification because each individual thread still reads from the bag with a sequential pattern. Therefore we can have multiple cores with multiple prefetchers accessing the same data structure. With multiple threads on a single core, I simply use separate EWMAs to predict per-thread queue access times.

4.3.4.2 Indirect prefetching

Another common access pattern for graphs is iteration through the vertex and edge data, typically for iterative calculations such as PageRank [84]. For the actual access of the edge and vertex information, a traditional stride prefetcher will work well, however such workloads typically read from a data structure indexed by the vertex value of each edge, which is a frequent, data-dependent, indirect access where a stride prefetcher cannot improve performance. My configurable graph prefetcher views this as the same problem as fetching the visited list for breadth-first searches. By reacting to edge list reads instead of work list reads, we can load in the vertex-indexed "visited-like" data at a given offset. This results in the essentially the same strategy described for the large-vertex mode.

4.3.4.3 Other access patterns

More generally, a similar technique can be used for other data formats and access patterns. The prefetcher relies on inferring progress through a computation by snooping accesses to preconfigured data structures, a technique that can be easily applied to other traversals. For example, a best-first search could be prefetched by observing loads to a binary heap array, and prefetching the first N elements of the heap on each access.

For different access patterns (e.g., array lookups based on hashes of the accessed data [56]), hardware such as the prefetch address queue, which isn't traversal specific, could be shared between similar prefetchers, with only the address generation logic differing. This means that many access patterns could be prefetched with a small amount of hardware.

4.3.5 Summary

I have presented a prefetcher for traversals of graphs in CSR format. The configurable graph prefetcher is configured with the address bounds for the graph data structures and operates in two modes (*vertex-offset* and *large-vertex*) to prefetch information in reaction to L1 accesses by the core. The prefetcher is designed to avoid explicitly stating which vertex traversal starts from. This information is inferred from reads of the lists: we assume that at any point we read the work list, we are likely to read successive elements. This makes the prefetcher both more resilient against temporary changes in access pattern, and also increases its generality: it can also accelerate algorithms that aren't pure breadth-first searches, such as ST connectivity.

The concept of a configurable prefetcher is more general. This work is intended to be a blueprint of how one might design configurable prefetchers for many different common access patterns, which can then be chosen between by an application. Indeed, when compared with the programmable prefetcher of the next chapter, configurable schemes trade off an area reduction for a reduction in generality. The decision to implement one over another depends on the complexity of expected workloads, and the chip area available to be devoted to prefetching.

4.4 Experimental setup

To evaluate hardware-configurable prefetchers, we can no longer utilise real systems, unlike in chapter 3, as this would require building a full silicon implementation. I instead model a system using the gem5 simulator [19] in full system mode with the setup given in table 4.2 and the ARMv8 64-bit instruction set.

4.4.1 Benchmarks

The applications evaluated are derived from existing benchmarks and libraries for graph traversal, using a range of graph sizes and characteristics. I simulated the core breadth-first-search-based kernels of each benchmark, skipping the graph construction phase. Benchmarks were compiled using aarch64-linux-gnu-gcc.

The first benchmark is from the Graph 500 community [79]. I used their Kronecker graph generator for both the standard Graph 500 search benchmark and a connected-components calculation. The Graph 500 benchmark is designed to represent data-analytics workloads, such as 3D physics simulation. Standard inputs are too large to simulate, so I used smaller graphs with scales from 16 to 21 (logarithmic number of vertices) and edge factors from 5 to 15 (average number of edges per vertex), to give graphs of between 10MiB and 700MiB in size (for comparison, the Graph 500 "toy" input has scale 26 and edge factor 16). This benchmark is also evaluated in chapter 3, but here I evaluate it in more detail with a greater range of vertex and edge sizes. I use both the standard breadth-first search algorithm, along with an implementation of connected components using a breadth-first search style kernel on the same graphs.

The prefetcher is most easily incorporated into libraries that implement graph traversal for CSR graphs. To this end, I also evaluate on the Boost Graph Library (BGL) [90], a C++ templated library supporting many graph-based algorithms and graph data structures. To support the configurable graph prefetcher, I added configuration instructions on constructors for CSR data structures, circular buffer queues (serving as the work list) and colour vectors (serving as the visited list). This means that any algorithm incorporating breadth-first searches on CSR graphs gains the benefits of the prefetcher without further modification. I evaluate breadth-first search, betweenness centrality and ST connectivity which all traverse graphs in this manner. To evaluate the extensions for indirect prefetching (section 4.3.4) I use PageRank and sequential colouring.

Inputs to the BGL algorithms are a set of real world graphs obtained from the SNAP dataset [69] chosen to represent a variety of sizes and disciplines, as shown in table 4.4. All are smaller than that we might expect to be processed in a real system, to enable complete simulation in a realistic time-frame, but as figure 4.3(a) shows, since stall rates increase for larger data structures, the improvements attained in simulation are likely to be conservative when compared with real-world use cases.

4.4.2 Implementation details

I implemented the configurable graph prefetcher in the gem5 simulator [19]. Bearing in mind that the workloads involved here are memory bound, and thus will usually miss in the TLB, it was necessary to simulate the page table system in full. With the AArch64

instruction set architecture, this is only possible in full system (FS) mode, where the full operating system is run, rather than just syscalls being emulated (SE mode). Indeed, as the TLB model was fairly primitive, I extended it in two ways to make it more realistic: I added an L2 TLB within the gem5 code, which was set-associative and had a multi-cycle hit latency (unlike the standard fully associative, zero-cycle L1 TLB as implemented by default), and I altered the page table walker to support more than one page table walk at a time, to better simulate a system more like both x86 and more modern Arm systems [36].

The prefetcher itself was implemented as a standard gem5 queued prefetcher, much like the stride prefetcher implemented in the original gem5 source. The graph prefetcher is essentially just a state machine, with a connection to the TLB added at runtime by modification of the processor. Prefetches to virtual addresses are sent to the TLB, and when they return these are added into a queue within the C++ code in the simulator: this is necessary because prefetches cannot be issued in gem5 except from when a load, store or prefetch observation is made, and so data in this queue is sent for issue once the next observation occurs.

To support the triggering of prefetches based on prefetches, I also modified the cache slightly. Extra prefetch notification points were added on the receipt of a prefetch, and the cache line from these prefetches was also added to the packet sent to the prefetcher for observation.

The configuration of the prefetcher was implemented by adding new pseudoinstructions to gem5: a set_addr_bounds instruction was added, taking in an address ID for each of the four data structures, start and end bounds, and a word size for each element. The source code for the simulator changes, along with benchmarks, is available in the University of Cambridge data repository [2].

4.5 Evaluation

I first evaluate the configurable graph prefetcher on breadth-first-search-based applications and analyse the results. Then I move on to algorithms that perform sequential access through data structures, and parallel breadth-first search.

4.5.1 Performance

My configurable graph prefetcher brings average (geometric mean) speedups of 2.8× on Graph 500 and 1.8× on BGL algorithms. Figure 4.8 shows the performance of the breadth-first search (BFS) hardware prefetcher against a stride prefetcher under simulation, and a stride-indirect scheme as suggested by Yu et al. [97], which performs sequential access on the edge list into the visited list, essentially treating the pattern as a simple indirect, rather than a full breadth-first search. Stride prefetching performs poorly, obtaining an average of

Main Core	
Core	3-Wide, out-of-order, 3.2GHz
Pipeline	40-entry ROB, 32-entry IQ, 16-entry LQ, 16-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 [27]
Memory & OS	
L1 Cache	32KiB, 2-way, 2-cycle hit lat, 12 MSHRs
L2 Cache	1MiB, 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
Prefetcher	200-entry queue, BFS prefetcher
OS	Ubuntu 14.04 LTS

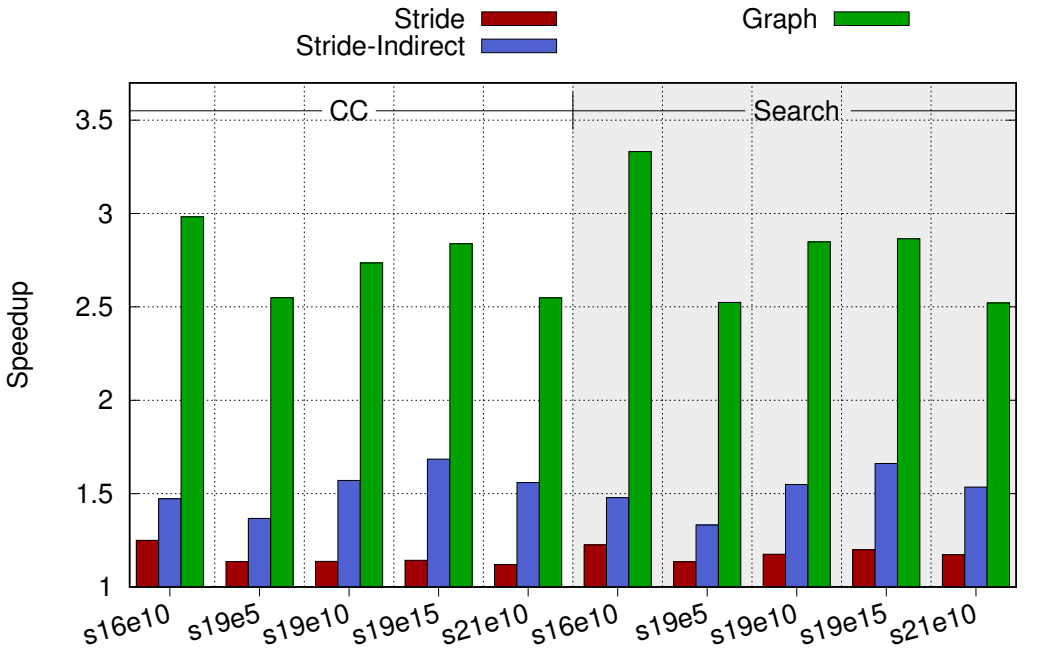
Table 4.2: Simulated CPU core and memory experimental setup.

Benchmark	**Source**	**Name**
Connected components	Graph 500	CC
Search	Graph 500	Search
Breadth-first search	Boost graph library	BFS
Betweenness centrality	Boost graph library	BC
ST connectivity	Boost graph library	ST
PageRank	Boost graph library	PR
Sequential colouring	Boost graph library	SC

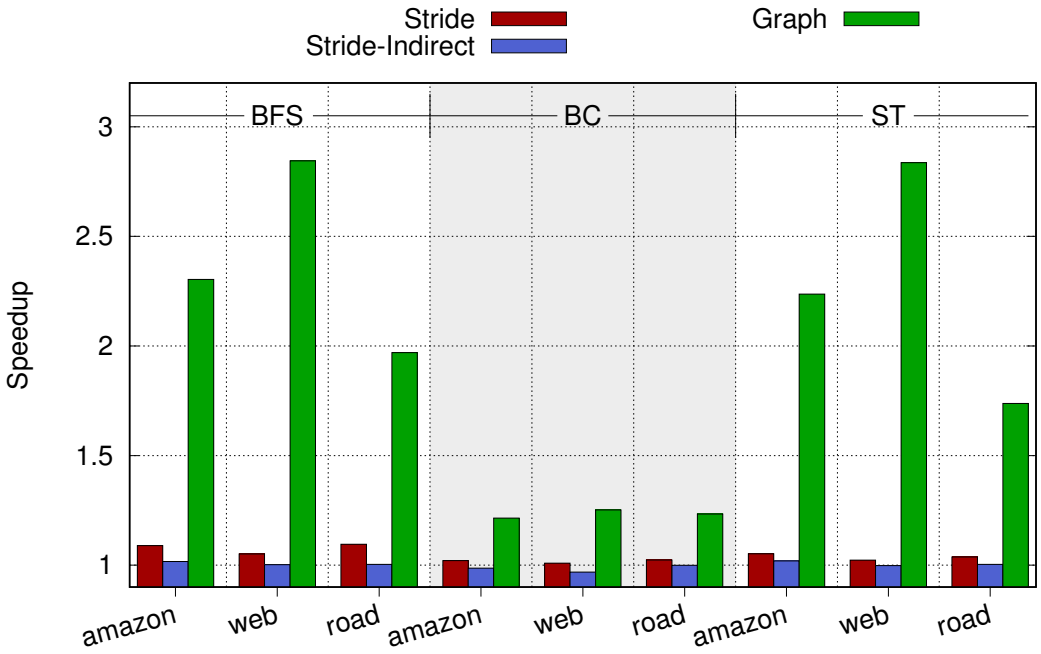
Table 4.3: Benchmarks.

Graph	**Nodes**	**Edges**	**Size**	**Field**
s16e10	65,536	1,310,720	10MB	Synthetic
s19e5	524,288	5,242,880	44MB	Synthetic
s19e10	524,288	10,485,760	84MB	Synthetic
s19e15	524,288	15,728,640	124MB	Synthetic
s21e10	4,194,304	83,886,080	672MB	Synthetic
amazon0302	262,111	1,234,877	11MB	Co-purchase
web-Google	875,713	5,105,039	46MB	Web graphs
roadNet-CA	1,965,206	5,533,214	57MB	Roads

Table 4.4: Synthetic and real-world input graphs.



(a) Graph 500



(b) BGL

Figure 4.8: Speedups for my configurable graph prefetcher against stride and stride-indirect schemes.

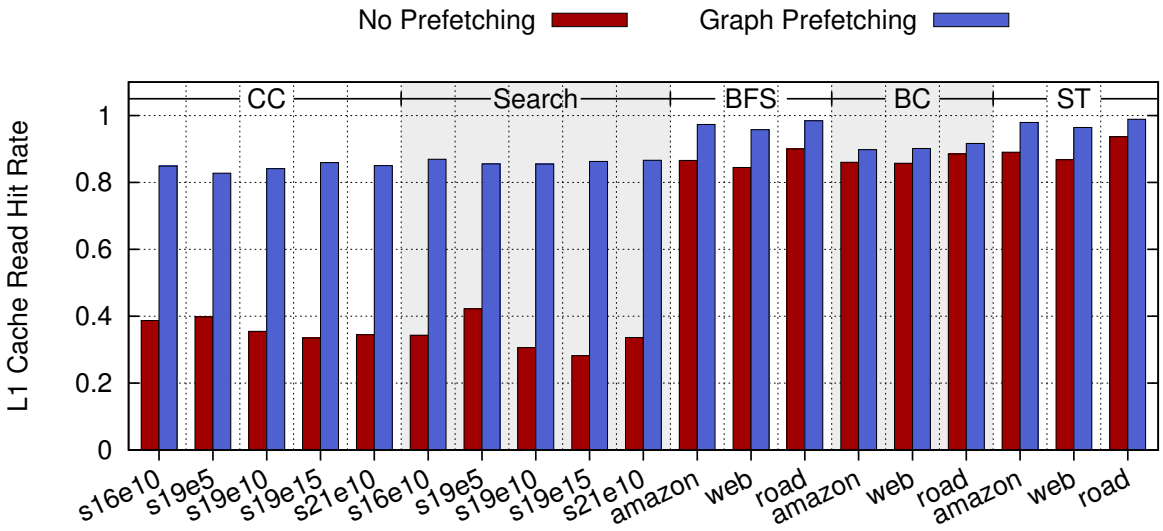


Figure 4.9: Hit rates in the L1 cache with and without prefetching.

1.1×. Stride-indirect performs only slightly better with an average of 1.2×, as breadth-first searches do not exhibit this pattern significantly, causing a large number of unused memory accesses. For comparison, under the same simulation conditions, augmenting binaries with software prefetching gave speedups of no more than 1.1×.

My configurable graph prefetcher increases performance by over 2.5× across the board for Graph 500. In the BGL algorithms, basic breadth-first searches perform comparably to Graph 500's search, but betweenness centrality achieves a much smaller performance increase, averaging 20%, due to significantly more calculation and non-breadth-first-search data accesses. In fact, the Boost betweenness-centrality code involves data-dependent accesses to various queue structures and dependency metrics, which are only accessed on some edge visits and are not possible to prefetch accurately with the same hardware. This algorithm also accesses two data structures indexed by the edge value: the visited list, and also a distance vector. For evaluation, I implemented an extension for the prefetcher to fetch from two "visited" lists, allowing both to be prefetched, improving on average by an extra 5%.

The final algorithm, ST connectivity, is interesting in that it isn't actually a pure breadth-first search: it is instead two searches from the start and end node placed into the same breadth-first search queue. Due to the setup of the prefetcher, which only has the information of nodes being placed into a FIFO queue and doesn't rely on any particular search strategy or algorithm, this doesn't affect its ability to fetch for this workload, getting comparable results to a pure breadth-first search. This reflects the benefits of my approach compared to a more stateful prefetcher: by limiting the information the prefetcher receives we can support more use cases than just breadth-first search alone.

Around 20% of the benefit comes from prefetching TLB entries; due to the heavily irregular data accesses observed, and the large data size, many pages are in active use at

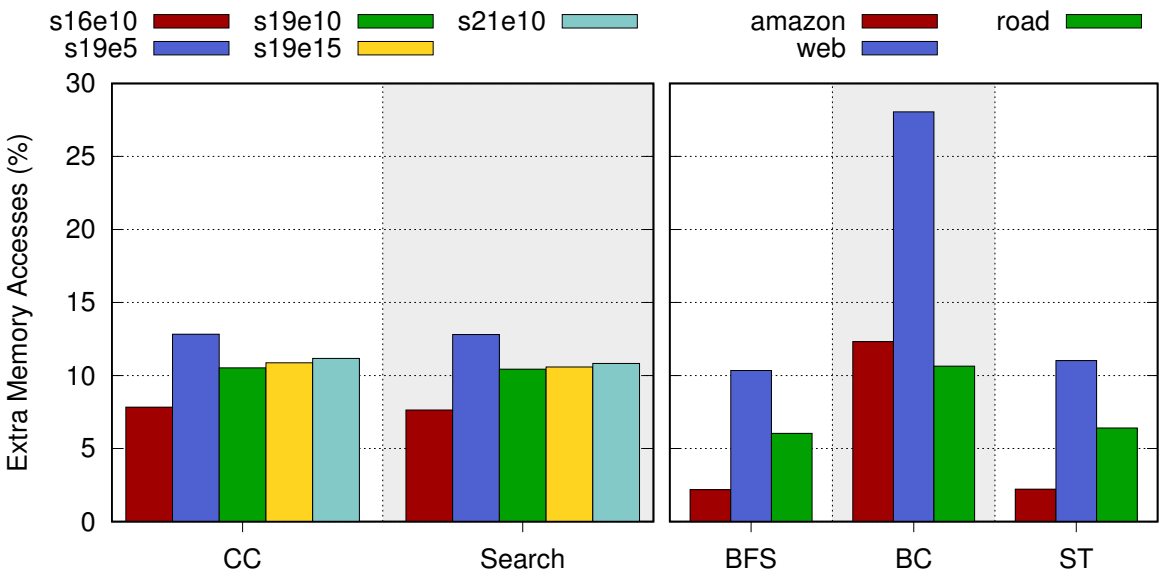


Figure 4.10: Percentage of additional memory accesses as a result of using my configurable graph prefetcher.

once. However, by virtue of prefetching these entries when performing prefetching of the data itself, these entries should be in the L2 TLB when the main thread reaches a given load, avoiding stalls on table walks.

4.5.2 Analysis

I now analyse the effect of the configurable graph prefetcher on the system, considering the changes in L1 hit rates, memory accesses and utilisation of prefetched data, shown in figures 4.9 to 4.11.

4.5.2.1 L1 cache read hit rates

The hardware graph prefetcher boosts L1 hit rates, and even small increases can result in large performance gains. In Graph 500 benchmarks, the baseline hit rates are mostly under 40% and these increase to over 80%. However, in BGL algorithms, baseline hit rates are already high at 80% or more, due to a large number of register spills. These result in loads to addresses that are still in the L1 cache, which are relatively unimportant in the overall run time of the program. The prefetcher increases the hit rate marginally, but crucially these additional hits are to time-sensitive data from the edge and visited lists, resulting in significant speedups.

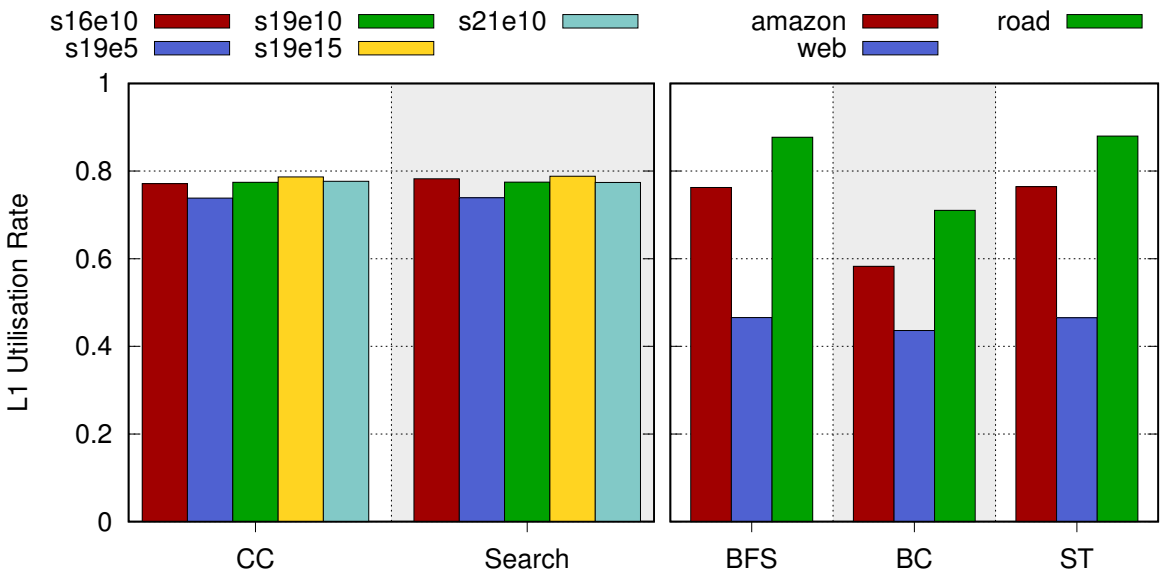


Figure 4.11: Rates of prefetched cache lines that are used before leaving the L1 cache.

4.5.2.2 Memory accesses

If a prefetcher fetches too much incorrect data from main memory, then a potentially severe inefficiency comes about in terms of power usage. To this end, any prefetcher targeting the reduction of energy usage by reducing stalls needs to keep such fetches to a minimum. Figure 4.10 shows the percentage increase in memory bytes accessed from main memory for each of the benchmarks and graphs I tested. The average is 9%, which translates into 150MB/s (or approximately 3 cache lines every 4,000 cycles) extra data being fetched. Betweenness centrality on the web input suffers from the most extra accesses: as it has very low locality and a small number of edges per vertex, the assumption that we will access every edge in a cache line shortly after loading is incorrect. Indeed, this input receives only minor benefit from prefetching visited information, as can be seen in figure 4.12; without visited prefetching we gain 1.24× for 2% extra memory accesses. A prefetcher that only prefetched from individual fetched words would reduce this over-fetch: the scheme evaluated here assumes every element in a cache line is part of a set of edges. The tradeoff there would be that more prefetches for edges would need to be generated, to trigger the correct visited-list prefetches.

4.5.2.3 L1 prefetch utilisation

Figure 4.11 shows the proportion of prefetches that are used before eviction from the L1 cache. These values are more dependent on timeliness than the number of extra memory accesses: for a prefetched cache line to be read from the L1 it needs to be fetched a short time beforehand. However, even when prefetched data is evicted from the L1 before being

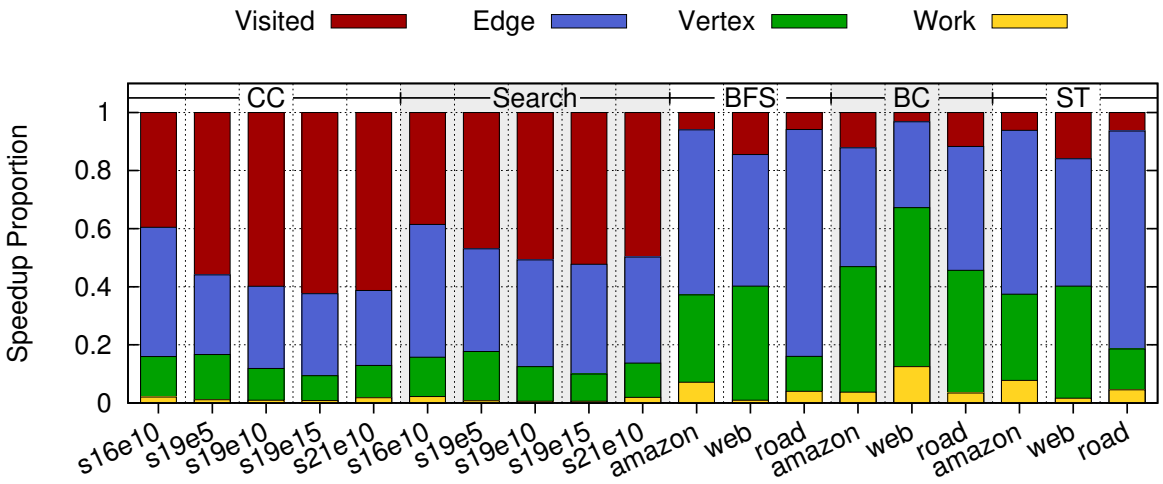


Figure 4.12: The proportion of speedup from prefetching each data structure within the breadth-first search.

used, we still gain the benefits of having it in the L2 cache instead of main memory.

The vast majority of prefetched cache lines are read at least once from the L1 for most test cases. A notable exception is the *web* input for BGL algorithms, where around half of prefetches aren't used before being evicted from the L1. Part of this is likely down to the overly aggressive visited-list prefetch, which assumes every element in a cache line of edges should be prefetched, which is true for graphs other than *web*. Still, significant performance improvement is observed even for *web*; the prefetcher's fetches stay in the L2 and improve performance through avoiding main memory accesses.

4.5.2.4 Breakdown of speedup

Figure 4.12 characterises where performance improvement is being observed from within each benchmark. The Graph-500-based benchmarks gain significantly more speedup from visited information than the BGL based algorithms do: this is because Graph 500 stores 64-bit information per vertex (the parent vertex and the component, for search and connected components respectively), whereas the Boost Graph Library code stores a 2-bit colour value for visited information. This means that the Boost code's visited information is more likely to fit in the last level cache. However, as the data size increases, this will not be the case, so for larger graphs a more significant speedup from visited information prefetching will be observed.

4.5.3 Generalised prefetching

I now show how the prefetcher can be used to accelerate other traversals on CSR graphs, as described in section 4.3.4.

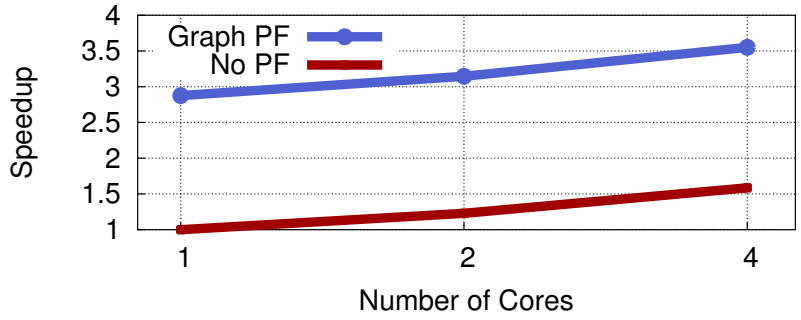


Figure 4.13: Speedup relative to 1 core with a parallel implementation of Graph500 search with scale 21, edge factor 10 using OpenMP.

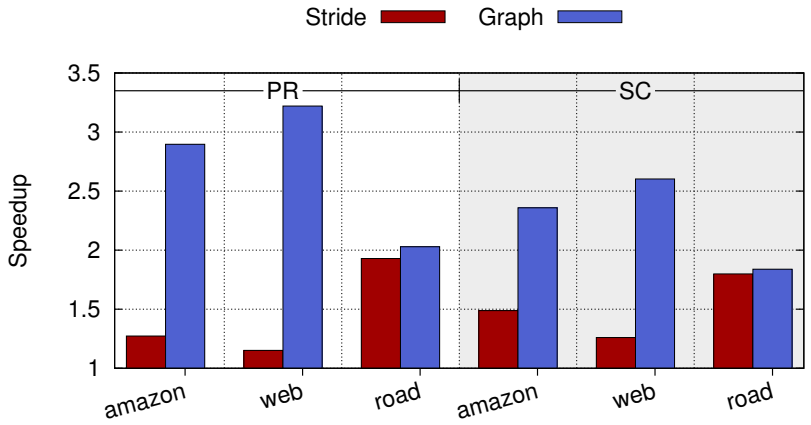


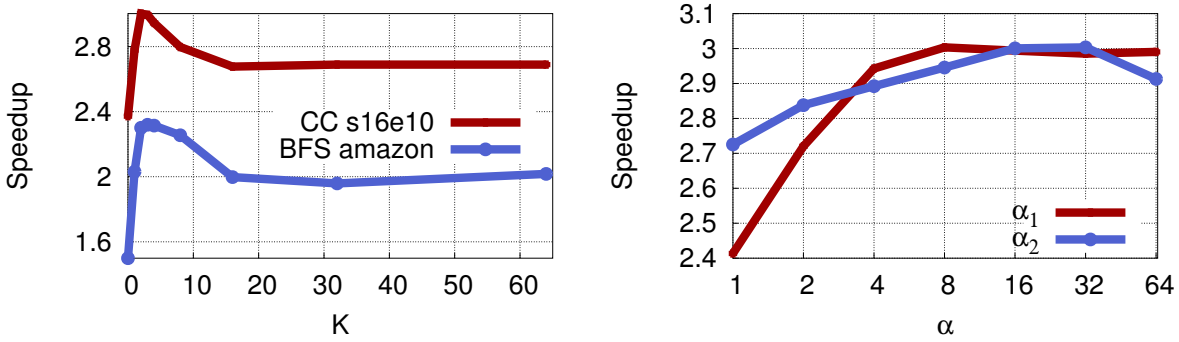
Figure 4.14: Speedup for different types of prefetching when running PageRank and Sequential Colouring.

4.5.3.1 Parallel breadth-first search

Figure 4.13 shows the performance of my configurable graph prefetcher on a parallel implementation of Graph 500 search using OpenMP, with a separate prefetcher per core. Each prefetcher works independently, but all access the same data structures. We attain similar speedup to using the sequential algorithm, showing that the configurable prefetcher can aid both single-threaded and multithreaded applications. In addition, the speedups scale at the same rate both with and without prefetching, but prefetching is significantly more beneficial than parallelising the algorithm: 4 cores with no prefetching brings a speedup of 1.6× whereas a single core with my configurable graph prefetcher achieves 2.9×.

4.5.3.2 Indirect prefetching

Figure 4.14 shows the performance of my extension for sequential-indirect access patterns, along with the same stride baseline setup from section 4.5.1. As this pattern is very predictable, few prefetches are wasted: all of my simulations resulted in under 0.1% extra



(a) Varying k

(b) Varying α

Figure 4.15: Performance as parameters are varied.

memory accesses, with an average utilisation rate of 97% for prefetches in the L1 cache.

Notably, though the performance differential between stride and my configurable graph prefetcher for the web and amazon graphs is very large, it is much smaller for the road-based graph. This reflects the latter's domain: roads tend to have very localised structure, so stride fetching for the data-dependent rank indexing still works relatively well.

4.5.4 Parameter impact

I next evaluate the impact of changing the parameters of the graph prefetcher, showing that there is a sweet spot in the distance-weighting factor, and that the EWMA weights and prefetch queue size must be chosen carefully.

4.5.4.1 Distance weighting factor

Figure 4.15(a) shows the performance for two different graphs and benchmarks with varying values for k, the weighting factor from equation 4.3 in section 4.3.2. Other applications follow a similar pattern. Both benchmarks see peaks at low values of k (2 and 3 respectively), although there is high performance even with large values. This is because a) we always transition into large-vertex mode at the same time, so only vertex-offset mode is affected, and b) when in vertex-offset mode, even though data is likely to be evicted before it is used, it is likely in the L2 instead of main memory when we need it.

4.5.4.2 EWMA weights

For the weighted moving averages, we need to strike a balance between a useful amount of history for high performance and ease of computation in setting α. For the latter, we need to set α to a power of two, so that the divide operation is just a shift. Figure 4.15(b) shows the performance impact for the choice of α for each. For the former, performance is maximal at $\alpha_1 = 8$, and the latter at $\alpha_2 = 32$.

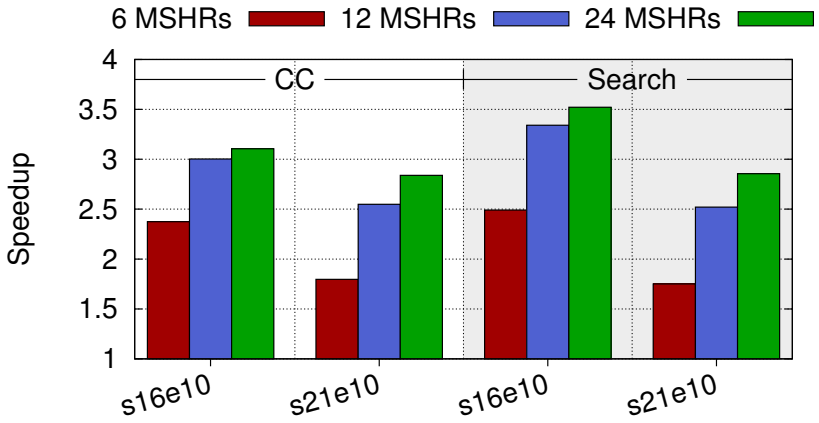


Figure 4.16: Speedup from the prefetcher with varying numbers of MSHRs for the L1 cache.

4.5.4.3 Number of MSHRs

The baseline core cannot use more than six miss status handling registers (MSHRs) due to the size of its load/store queues. However, as the prefetcher can issue its own loads, this no longer becomes the case and lack of available MSHRs is a significant constraint on the number of outstanding prefetches that can be maintained. Figure 4.16 shows the performance gained with various setups for the L1 cache, showing that 12 MSHRs achieves most of the performance gains, with a little more available at 24, particularly for larger graphs.

4.5.4.4 Queue size

As prefetches to main memory take a long time to complete, a large queue of addresses is beneficial to deal with the large number of requests created. Figure 4.17 shows the effect of queue size for Graph 500 on the s16e10 graph and Boost BFS on amazon. Although performance for the former improves with larger queues, more conservatively sized address queues have minor performance impact. Therefore, the storage requirements of the prefetcher could be reduced with only minor performance degradation.

4.5.5 Summary

I have evaluated the configurable graph prefetcher on a variety of applications and input graphs, showing that it brings geometric mean speedups of 2.2× for breadth-first-search-based algorithms and 2.4× for sequential iteration access patterns. For breadth-first search applications, the prefetcher incurs only 9% additional memory accesses and 70% of the prefetched cache lines are read directly from the L1.

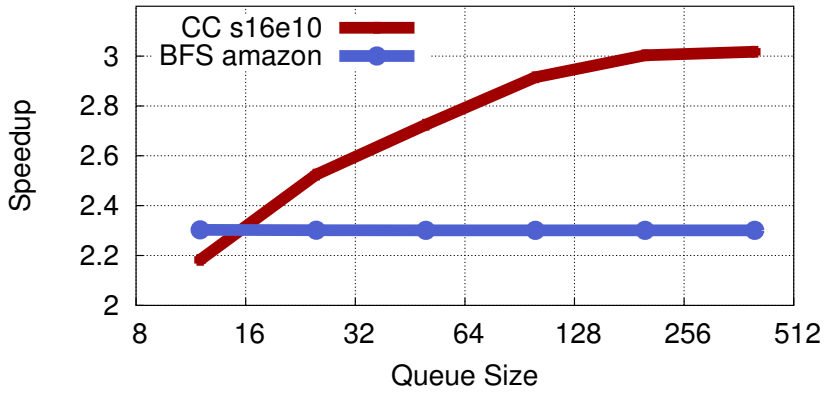


Figure 4.17: Performance for different queue sizes for two benchmarks and graphs.

4.6 Comparison with PrefEdge

Though I was the first to use the concept for a hardware prefetcher, the idea of looking ahead in the work list of a breadth-first search to improve performance is not new. PrefEdge [83] uses this concept to bring edge data into DRAM from an SSD in parallel, using vertex data kept in DRAM. One major difference is the granularity this is performed at. For bringing data into caches, the access times are lower than from SSD, and we therefore have less time to devote software compute resources to the problem. However, the differences caused by us being able to dedicate hardware to bring data into the cache are more illuminating in highlighting the strength of the configurable graph prefetcher presented in this chapter.

Because PrefEdge can only issue non-blocking IO requests to the SSD controller, without being able to directly read the data when it arrives, only edge information can be stored on an SSD: vertex information must be kept in DRAM to achieve performance. This means that the work list, vertex list and visited list must be in fast memory. The configurable graph prefetcher has no such requirements: all data structures can be kept in slow memory. This is because there is no need to stall on any load: we can react to the prefetching events themselves, and so every memory access can be slow but parallel.

This idea of "event triggering" I use, that triggers prefetches on the receipt of other prefetches is useful more generally. I explore it further in chapter 5 to develop a fully programmable, latency-tolerant prefetcher.

4.7 Conclusion

This chapter considered the development of a configurable graph prefetcher implemented in hardware for breadth-first search and sequential indirection, using configuration instructions to specify the memory-access pattern and the positions of the data structures involved. I looked at the design of an explicitly controlled prefetcher for breadth-first searches and

sequential accesses on compressed sparse row graphs, based on snooping reads of a search queue, achieving geometric mean speedups of 2.3×, and up to 3.3×, across a range of applications and graph sizes.

However, the concept can be more generally applied to other memory-bound workloads. We can imagine a small set of configurable prefetchers, each built using a small amount of logic, and custom-designed to implement different common access patterns.

This design has many benefits over software prefetching. By removing prefetch instructions from the main program, we can reduce instruction count and increase performance. We can design better schemes that react to prefetched data as it arrives, instead of prefetching then reloading data. And we can generally implement more complex prefetching behaviour without overloading the processor with compute. However, it is unlikely to be the case that we could implement all desired prefetchers for every possible memory-bound workload that could benefit from the technique. Many memory accesses don't fit into a small number of fixed patterns, or include some highly custom code for generating addresses. We might not even wish to devote silicon area to some relatively uncommon cases that would still gain speedup. In these cases, software prefetching, as a more flexible programming model, would be more beneficial.

What we need is something flexible enough to implement complicated prefetching strategies, but with enough fixed-function hardware to be efficient. The next chapter devotes itself to designing such a system.

Chapter 5

Generalised programmable prefetching

No matter how many configurable hardware prefetchers we include in a system, not all access patterns will be covered. Indeed, when new workloads come along there will inevitably be a latency between the workload becoming performance critical, and it being supported in hardware, and many memory accesses are sufficiently custom that they don't fit into a small set of basic patterns that can be implemented purely in logic, for example database workloads [56], where the hash function used to index into a hash table may take on any number of different forms.

This chapter concerns itself with extending the ideas presented in the previous chapter into something more general. In essence, we need to replace the fixed-function address generation logic with something programmable. One option would be to use a general purpose processor. However, given that in many memory-bound workloads, most of the instructions are used for address generation and loading, this core would need to be as powerful as the main processor itself. A doubling of energy and area to allow prefetching is far too large for a realistic implementation. So, instead, we can exploit the fact that achieving memory-level parallelism is itself a thread-level parallel operation. Utilising this, the programmable prefetcher can instead use a large number of extremely small, parallel units, along with a highly parallel latency-tolerant model to program them, based on non-blocking events to allow us to overlap the computation of multiple chains of dependent loads. This allows high prefetch performance at extremely low overhead.

This and the following chapter contain work on developing architecture and compiler techniques for an event-triggered programmable prefetcher, published at ASPLOS 2018 [9].

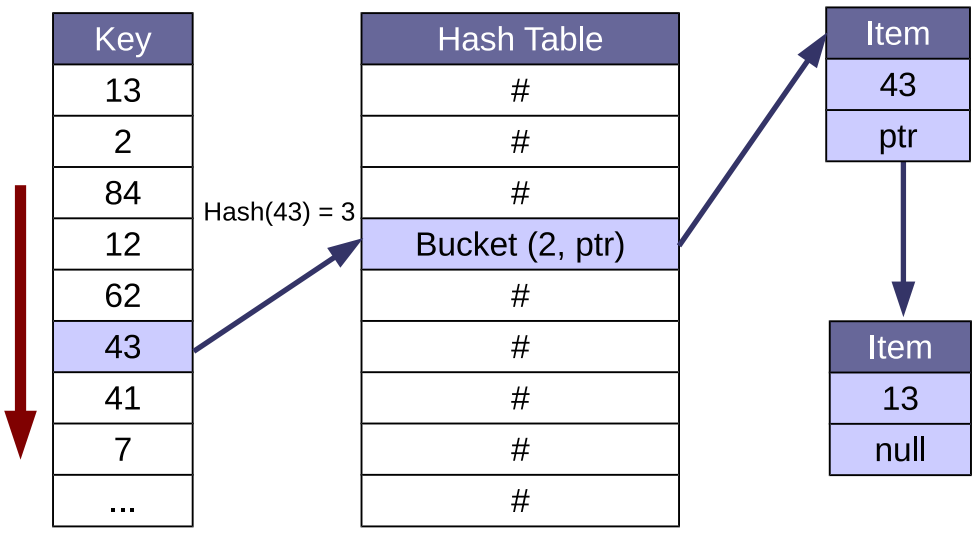


Figure 5.1: We can exploit look-ahead to make database workloads less memory bound, but this is complicated by both the computation of a hash function, and the linked-list walk for each bucket.

5.1 A worked example: database hash joins

Figure 5.2 gives an example of a typical hash-join kernel, as used in databases. We looked at some examples of these earlier, with the HJ-2 and HJ-8 kernels in chapter 3. A pictorial example of the memory-access pattern is given in figure 5.1. 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.

Though this is memory-bound, we again have a situation where we can work out future memory accesses by looking ahead in the array of keys. However, there are two factors that make this more problematic than the examples I considered in chapter 4. The first is that to index the hash table we have to perform some computation, which is likely to vary between different hash-table implementations. The second is that a variable number of linked list items hang off of each hash-table bucket, all of which are also likely to miss in the cache.

Figure 5.3(a) shows how this unmodified code would execute. 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.

5.1.1 Comparison with configurable prefetching

Configurable prefetching (chapter 4) is a non-starter here, because of the hash function. This is likely sufficiently variable between applications that no fixed function hardware

```
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 5.2: Hash-join kernel with two latency-hiding techniques.

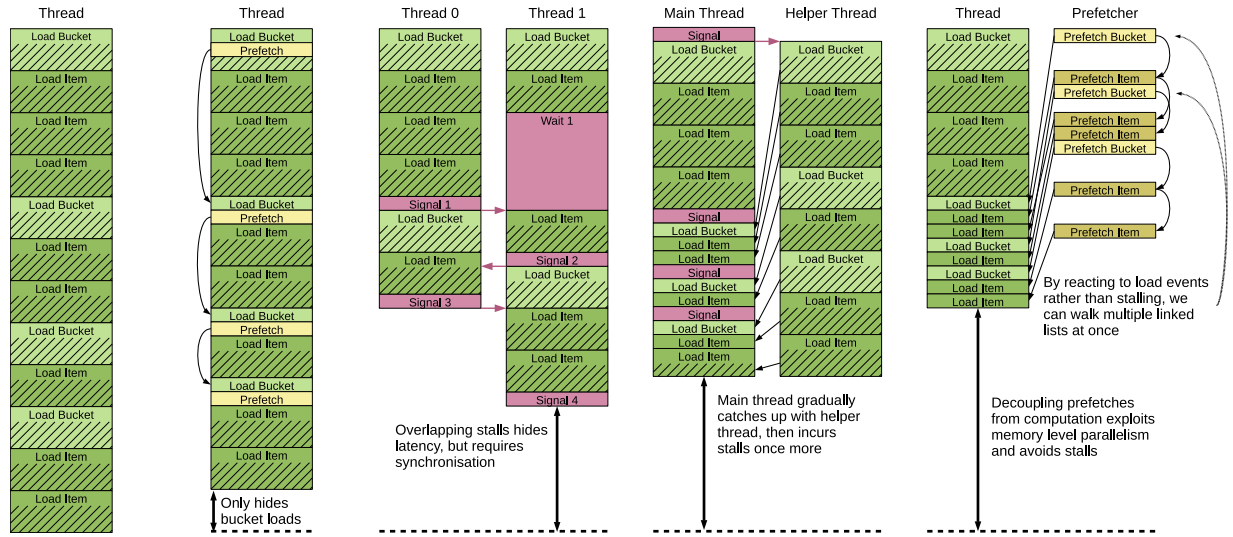
would be able to cover all of the cases, and thus some software support to generate the addresses is necessary. In addition, variances in precise data structure of hash table elements would likely make the amount of configuration necessary extremely complex.

5.1.2 Comparison with software prefetching

Software prefetching is somewhat more promising, as seen by the results of HJ-2 and HJ-8 in chapter 3. Indeed, the amount of compute between each memory access from the hash code means software prefetching is particularly useful even on out-of-order cores, because the compute overhead from the hashing function within the original code limits the core's ability to reorder memory accesses.

However, HJ-2 and HJ-8 have fixed memory accesses: HJ-2 loads no linked list items, and HJ-8 loads three. In a real hash table, the number of buckets will vary between each element of the table, depending on how full the hash table is. This means that a fixed schedule of loads becomes inappropriate, and also that we will frequently issue prefetches for linked list elements that don't exist.

This, in effect, limits prefetching to just the table-array elements, as can be seen in figures 5.2 and 5.3(b). Yellow boxes denote the calculation of the prefetch address and corresponding prefetch instruction. I assume a prefetch distance of 1 iteration in this example, to simplify the 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



(a) Original (b) Software (c) Multithreaded (d) Helper thread (e) Desired

Figure 5.3: 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 the results of earlier prefetches.

original code, but the inability to prefetch the linked-list items limits the performance increase.

5.1.3 Comparison with other techniques

Other techniques for improving memory-bound code do exist, and I briefly consider them here before looking at what we could do with additional hardware support.

5.1.3.1 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.

As with software prefetching, code for this option is shown in figure 5.2. Its execution on two threads is shown in figure 5.3(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 5.3(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, provided we ignore any overheads from cache coherence on the shared output array.

5.1.3.2 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 5.3(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.

5.1.3.3 Other prefetching

The other techniques presented in chapter 2 do not directly apply here, and so are not presented in figure 5.3. Address-based prefetching (section 2.1.1) does not apply due to the data-dependent access patterns exhibited. Stride-indirect prefetching (section 2.1.2.1) cannot pick up the access pattern because the indirection also involves a hash computation, along with linked list walks. History prefetching (section 2.1.2.2) does not apply because the indexing pattern isn't repeated, so no history exists to learn from. And current techniques that can expose this access pattern complexity from the programmer, or by compiler or runtime analysis (sections 2.1.2.3, 2.1.2.4, 2.1.4, 2.2.4 and 2.2.5) do not expose enough parallelism to be able to fetch from a large number of addresses concurrently, so the stall chains on multiple hash table linked list elements prevent memory-level parallelism from being fully exploited.

An exception for this particular access pattern is that fetcher unit research (section 2.1.3.1) has been performed specifically for database memory accesses [56]. But fetching the actual data rather than prefetching the memory locations reorders the operations, thus requiring thread-level parallelism in addition to the memory-level parallelism we are fundamentally trying to exploit. If true ordering is required, then we face the same synchronisation issues as in the multithreading example in figure 5.3(c). This therefore requires program restructuring, and if the compiler is performing this work, automatic thread parallelisation, which requires much stronger constraints than the more speculative prefetching, which does not affect correctness.

5.1.4 Desired behaviour

In the ideal case we would have no stalls at all. The workload 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 5.3(e) where, after a warm-up period, computation can proceed without stalls, since data is immediately available in the first level cache.

The configurable graph prefetcher from chapter 4 can already do this. By issuing prefetches then reacting to them when they arrive, memory accesses can be fetched without the prefetcher stalling. The challenge, then, is to develop a software programming model that can exploit this, along with hardware for the model to run on.

5.2 Requirements

To achieve a hardware prefetcher that can do memory accesses as complicated as this, we require a number of properties.

More general programmability The prefetcher must be able to handle arbitrary computation. This will allow, for example, hash codes to be calculated, and rarer or more complex access patterns to be performed than with a configurable prefetcher (chapter 4).

High performance We need the prefetcher to be able to keep up with the main core, otherwise the prefetching will be useless. Since often a program's memory-access instructions dominate the instruction count, to achieve something fully programmable we'll need as high compute performance as the main core (though not necessarily achieved in the same way).

Low area / power usage If the special purpose logic we add for prefetching dramatically increases the area of a core, it will be infeasible to implement on a real system. Likewise, if we double the energy usage, even if we double performance, the improvements will be debatable. What we want to achieve is much higher performance at a similar budget to an unmodified system.

Micro-controller style cores such as the Cortex M0+ [16] can achieve moderate clock speeds and many orders of magnitude lower energy per instruction and silicon area than a modern out-of-order superscalar processor [13, 14], where much of the energy and area

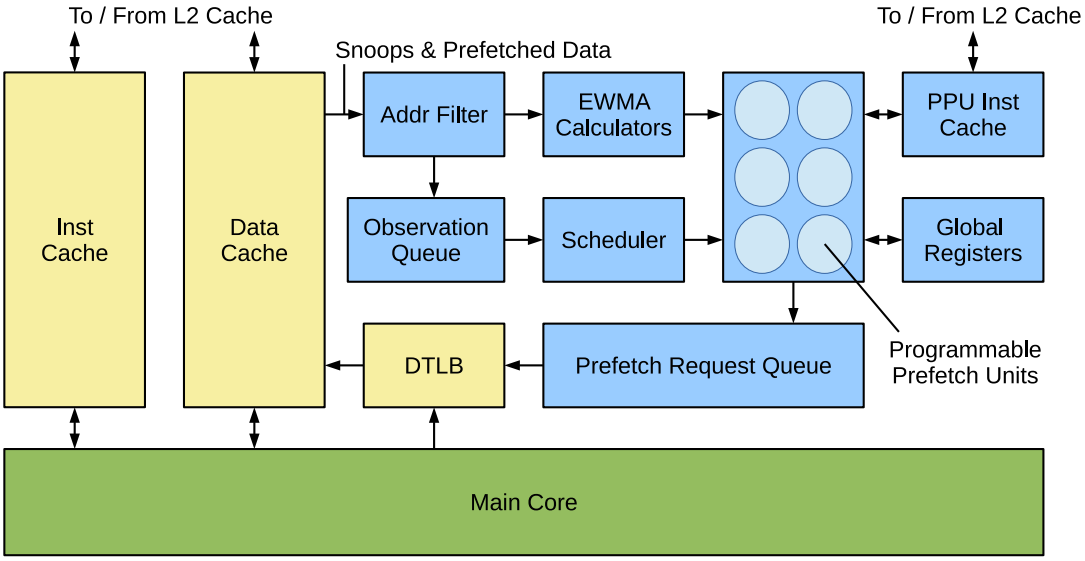


Figure 5.4: Structure of the programmable prefetcher.

budget is dedicated to speculation. Provided we can exploit parallelism for our prefetcher, then an array of small cores is a way of gaining the total compute throughput we need at a low enough area and power budget to be feasible for implementation on a processor.

5.3 Event-triggered programmable prefetcher

Here I develop a novel prefetching architecture based on the abstraction of an event: that is, new prefetches are triggered by read events within the cache, and also by prefetched data reaching the cache, as with common stride prefetchers [24, 93] and the prefetcher of chapter 4.

The difference here is that this scheme is suitable for more diverse and irregular applications. These events have programmable behaviour, triggered by configured address ranges, which cause small, fully programmable event kernels 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 very tiny programmable units. The ability to react to previous prefetches, which cannot be achieved by other schemes with programmability, such as software prefetching [22] or helper threads [52], allows irregular patterns, which typically feature multiple dependent accesses, to be prefetched without stalling.

Effectively, this is the configurable graph prefetcher from chapter 4, but with the address-generation logic replaced by parallel processing units, scheduling and queue logic to allow latency tolerance within the programmable units, and other structures generalised to allow wider use.

Start Addr	End Addr	Load Ptr	PF Ptr	Obs EWMA	PF EWMA Start	PF EWMA End	Batch

Figure 5.5: An address-filter entry.

5.3.1 Overview

Figure 5.4 shows the overall architecture of the design. I add programmable units and supporting hardware to generate prefetches based on a program's current and future working set. The prefetcher is event-triggered, to avoid stalls, yet enable 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 5.3.2). Data that has been filtered to be of interest moves into the observation queue, to be removed by the scheduler (section 5.3.3) when it detects a free programmable prefetch unit (PPU, section 5.3.4). 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 look-ahead distances calculated by the exponentially weighted moving average (EWMA) calculators (section 5.3.5), to generate new prefetches, which are placed into a FIFO prefetch request queue (section 5.3.6). When the L1 cache has available miss status holding registers (MSHRs), it removes a prefetch request and issues it to the L2 cache. The following subsections describe each structure in more detail.

5.3.2 Address filter

Much like in the configurable graph prefetcher of chapter 4, the address filter snoops all loads coming from the main core, and prefetched data brought into the L1 cache. The difference here is that instead of four fixed data structures, this can handle a variable number of structures, with more configurable behaviour.

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 5.2. The address filter is configured through explicit address-bounds configuration instructions running on the main core, as with the previous chapter. 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, an example of which is shown in figure 5.5. It stores virtual-address ranges for each important data structure, but unlike in the previous chapter, it also stores two function pointers to small computation prefetch event kernels: the code run on triggering of particular events to trigger new prefetches. These function pointers are *Load Ptr*, to be run when a load data from that range, observed from a main

processor, arrives in the cache, and *PF Ptr*, to be run when a prefetched to that range arrives in the cache. Some are also used for scheduling purposes (section 5.3.5), 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. There are eight fields in the address filter table, namely:

Start Addr The start of an address range we are interested in, for identifying data structures (64 bits).

End Addr The end of an address range we are interested in (64 bits).

Load Ptr The function pointer for the code to be run on observation of a normal load to this address range (64 bits).

PF Ptr The function pointer for the code to be run on observation of prefetched load data arriving within this address range (64 bits).

Obs EWMA Whether we record a moving average based on observed loads to this address range (1 bit).

PF EWMA Start Whether we start recording data to calculate prefetch load times from this observation (1 bit).

PF EWMA End Whether this observation results in the value stored from a PF EWMA Start event being used to update an EWMA (1 bit).

Batch Whether we are using this observation to feed into a batch prefetcher (section 5.3.9), and thus are only interested in events on an edge of a cache line (1 bit).

5.3.3 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 contents of 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.

Figure 5.6 shows the structure of an observation-queue entry:

Figure 5.6: An observation-queue entry.

Vaddr The virtual address observed (64 bits).

Fun Ptr A pointer to the function to be executed for this observation, derived either from the address filter (section 5.3.2) or memory request tag (section 5.3.8) (64 bits).

Carried Time The timestamp of the original load from the CPU which triggered this observation, either directly or via a prefetch. Used for calculating EWMAs (64 bits).

Data The data for the cache line associated with an observation (512 bits).

5.3.4 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, though then do not need to execute the same ISA as the main core since they run different instructions.

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 prefetch 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 5.5 a maximum of 1KiB is fetched from main memory by the PPUs for the entirety of each application.

The PPUs do not have 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, I 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 I have not found this to be an issue.

5.3.5 Moving average (EWMA) calculators

Again, I implement an EWMA unit for scheduling, as with the configurable graph prefetcher of chapter 4. It dynamically works out the ratio between time to finish a chain of prefetches, and the time each loop iteration takes, inferred from load observations from the main core, and uses 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 signal the start of a timed prefetch EWMA, and attach the current time to the event generated. This is propagated to resulting prefetches until we reach an address range with a flag set, then the time between the events is used as input into a load time EWMA.

The ratios we are interested in are registered with the prefetcher, which then calculates them in hardware. These get updated periodically when new EWMA values are obtained, and the programmable prefetcher units are allowed to read the value to set the prefetch distance. However, to prefetch as much data as possible, the EWMA value seen by the PPUs needs to increase smoothly; otherwise some elements won't be prefetched at all. As a result, the value sent to the PPUs is a smoothed value $S_n = max(EWMA, S_{n-1} + 1)$, where S_n is the previous smoothed value sent to a PPU. Further, if $S_n > S_{n-1}$, this must be signalled to the PPU, which must then issue prefetches at an offset of both S_n and $S_n - 1$ to avoid missing any addresses.

This is just one method of dynamically scheduling prefetches, taken almost directly from what was suitable for graphs in the previous chapter. More generally, analysis hardware that determines whether prefetches are too early or too late might sit here. An example of another scheme that could potentially be used, with some modification to support the pattern of dependent prefetches in our target workloads, is the best-offset technique from Michaud [75].

5.3.6 Prefetch request queue

The prefetch request queue is a FIFO queue containing the virtual addresses that have been calculated by the PPUs for prefetching, but have not yet been processed. Once the

Tag ID	PF Ptr	Obs EWMA	PF EWMA Start	PF EWMA End

Figure 5.7: A tag-filter entry.

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.

5.3.7 Global registers

While the graph prefetcher only required access to the address bounds stored in the address filter, more general purpose prefetchers may require a variety of information, such as array start addresses, array bounds, or hash keys. Such values can be stored in global registers. These can be accessed by the PPUs, and can be set whenever the prefetcher is configured. In practice, this means they are particularly useful for loop-invariant values: we can configure the prefetcher once, then these values can be used for subsequent prefetches.

5.3.8 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 5.3.2, 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 I store a tag in each MSHR containing a prefetch request, generated per prefetch event kernel, identifying an element in the tag filter (figure 5.7). The tag filter is a segment of the address filter that works similarly to address entries, save for having no address bounds. When a prefetch request returns data, and has a registered tag, this tag is looked up in the filter, and the cache line is sent to a PPU loaded with the function pointer associated with that tag. MSHR tags are set by the code that calls a prefetch to a non-contiguous data structure.

In essence, we can see this as a form of very lightweight multithreading: we tell the system which code we expect to be called when a prefetch returns, then allow execution to be started afresh from the code pointer stored in the table. Since no information is assumed to be preserved in registers between events, these do not need to be stored, and so we can achieve this with just a single tag. Meanwhile, another separate prefetch event kernel can be executed, and it too can be reduced to just an address and a tag ID whenever a prefetch request is to be sent, rather than a full set of registers to be stored and reloaded

later.

5.3.9 Batch prefetch

It is common for many workloads to move rapidly through a sequential array, with indirect accesses based on those reads. This means that the data within one cache line for the sequential array is accessed in quick succession. In this case, to reduce traffic to the PPUs, it is preferable to prefetch all indices from the cache line in a single observation. We can then reduce traffic by triggering the array-read event kernel only if the address is at the start of a cache line, as it will still cause all data to be prefetched, and reduce the number of events for the PPUs to respond to.

This is configured by setting a batch bit in the address filter, to limit the number of events triggered. The associated prefetch kernel, which will issue prefetches at some offset in this sequential array, will be set up to issue an entire cache line's worth of prefetches when that data returns, instead of a single word, to compensate. This coarsens the scheduling granularity, and the amount of parallelism in the address calculation, but for most workloads both are still likely to be ample, so the reduction in prefetches is likely to be beneficial.

5.3.10 Summary

I have presented the design of an event-triggered programmable prefetcher that responds to filtered load and prefetch observation events. These feed into a set of programmable units, which run event kernels based on the events to issue prefetches into the cache. The following section describes how these are programmed.

5.4 Event programming model

To target the prefetcher, custom code must be generated for each application. This section describes the event-triggered programming model used for this, which is suited for latency-tolerant fetches on multiple PPUs. It also considers the interaction with the operating system and context switches. In this chapter I assume prefetch code is written by hand in a high-level language. I then consider compiler assistance for automatically generating prefetch kernels in chapter 6.

5.4.1 Event programming model

The PPU programming model is event-triggered, which fits naturally with the characteristics of prefetch instructions that have variable latency before returning data. We require a

programming model that can tolerate this variable latency, proceeding only when data returns to the cache so it can be acted upon. In addition, the PPUs must react to load events observed at the cache from the main core, which may not occur at regular intervals.

The event kernel run on the PPUs is determined from the addresses loaded or prefetched into the cache. During execution, instead of issuing loads that they must stall on, the PPUs may generate one or more further prefetches, which are issued to the memory hierarchy when resources become available, as described in section 5.3. This is naturally latency-tolerant, avoiding PPU stalls while waiting for prefetched data.

If and when prefetches return data, the scheduler can select any PPU to execute the corresponding event kernel, 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.

Each event kernel 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 that 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).

I 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.

A prefetch event kernel can be aborted at any time, since it is 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 event kernels are aborted. In addition, any operations that would usually cause a trap or exception (e.g., divide by zero) immediately cause termination of the prefetch event kernel.

5.4.2 Example

Consider the program in figure 5.8(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 program can be prefetched by loading the PPUs with the code in figure 5.8(b). In this example I 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 configuration instructions in the original code. Similarly, the

```
1 int64_t acc = 0;
2 config_on_load(0,on_A_load);
3 config_on_prefetch(0,on_A_prefetch);
4 config_on_prefetch(1,on_B_prefetch);
5 config_addr_bounds(0,&A[0],&A[N]);
6 config_addr_bounds(1,&B[0],&B[N]);
7 config_addr_bounds(2,&C[0],&C[N]);
8 for(x=0; x<N; x++) {
9   acc += C[B[A[x]]];
10 }
11 return acc;
```

(a) Main program

```
1 void on_A_load() {
2   Addr a = get_vaddr();
3   a += 512;
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 5.8: A loop with irregular memory accesses to arrays B & C, but significant memory-level parallelism for accesses to A. Also shown are the functions executed by the PPUs to exploit this MLP.

addresses of the prefetch kernels in figure 5.8(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 kernel is triggered that fetches the address eight 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 event kernels. 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 because of 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 either register reads or loads from the attached small, shared, prefetcher-state memory, as appropriate.

5.4.3 Operating system visibility

Although they have many of the capabilities of regular cores, PPUs are not visible to the operating system as separate cores, and so the OS cannot schedule processes onto

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 [27]
Memory & OS	
L1 Cache	32KiB, 2-way, 2-cycle hit lat, 12 MSHRs
L2 Cache	1MiB, 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
Prefetchers	
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 [25], degree 8
GHB Prefetcher	Markov GHB Prefetcher [82], depth 16, width 6, index/GHB sizes 2048/2048 (regular) and 67108864/67108864 (large)

Table 5.1: Core and memory experimental setup.

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, they do not require privileged instructions). As a result, while the prefetcher initiates page table walks, it cannot handle page faults, and in such a case it discards 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, and can simply discard them on a context switch. For the same reason, we can also throw away all event kernels in the observation queue and addresses in the fetch queue. Provided context switches are infrequent, this will cause only minor performance losses. 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.

Benchmark	*Source*	*Pattern*	*Input*
G500-CSR	Graph500 [79]	BFS (arrays)	-s 21 -e 10
G500-List	Graph500 [79]	BFS (lists)	-s 16 -e 10
PageRank	BGL [90]	Stride-indirect	web-Google
HJ-2	Hash Join [20]	Stride-hash-indirect	-r 12800000 -s 12800000
HJ-8	Hash Join [20]	Stride-hash-indirect, linked list walks	-r 12800000 -s 12800000
RandAcc	HPCC [72]	Stride-hash-indirect	100000000
IntSort	NAS [18]	Stride-indirect	B
ConjGrad	NAS [18]	Stride-indirect	B

Table 5.2: Summary of the benchmarks evaluated.

5.5 Experimental setup

To evaluate the programmable prefetcher I again model a high performance system using the gem5 simulator [19] in full system mode running Linux with the ARMv8 64-bit instruction set and configuration given in table 5.1: this is almost the same system as in chapter 4, but with the programmable prefetcher attached instead. Another change is that the store queue is increased from the default but small 16 to 32: this was because it was artificially limiting performance for one benchmark, Integer Sort, both with and without prefetching, so was increased for improved external validity, though this did not affect any other benchmark. I compiled the benchmarks using Clang with the O3 setting. This was necessary to give a good comparison for the compiler techniques presented in chapter 6 which were built for Clang, as opposed to GCC which was used in chapter 4, though this did not change the results significantly. I chose a variety of memory-bound benchmarks to demonstrate the scheme, representing a wide range of workloads from different fields: graphs, databases and HPC, described in table 5.2. I skipped initialisation, then ran each benchmark to completion using detailed, cycle-accurate simulation. In addition, I compare against a Markov global history buffer [82] 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 [47, 94] which keep state in main memory. To isolate the improvements from this technique, I provide unlimited bandwidth and zero latency accesses to the Markov baseline's state.

5.5.1 Benchmarks

Table 5.2 shows the benchmarks I evaluate on. The majority are taken from chapter 3, but I add two new benchmarks:

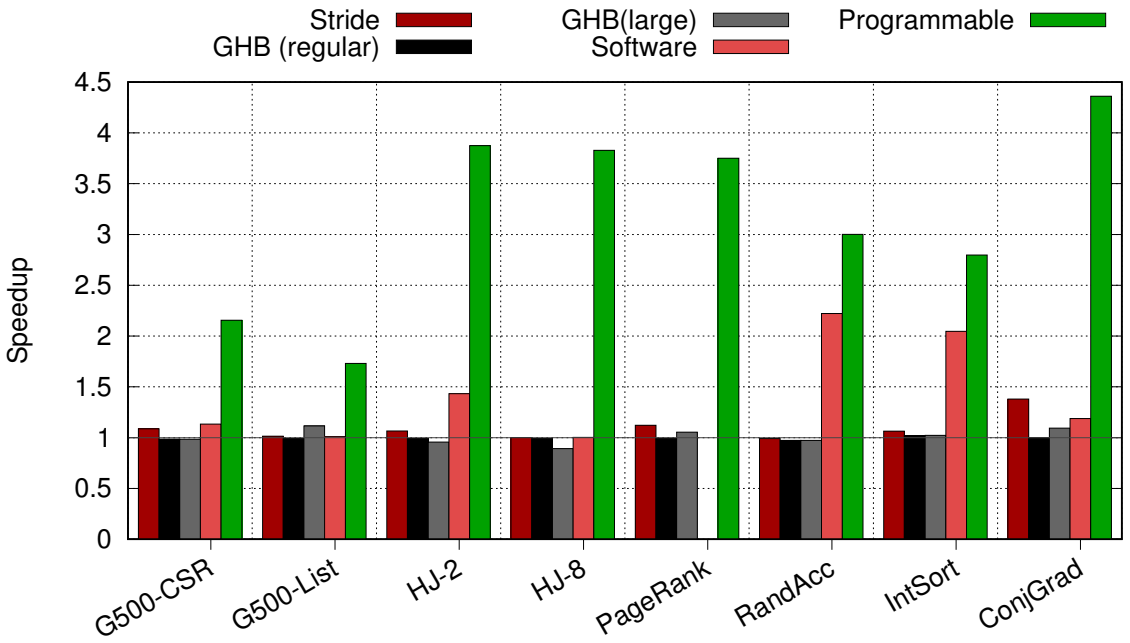


Figure 5.9: The programmable prefetcher realises speedups of up to 4.4×. Stride and software prefetchers cannot effectively prefetch highly irregular memory accesses.

5.5.1.1 PageRank

This benchmark is taken from chapter 4: built from the Boost Graph Library [90], it is unsuitable for manual software prefetching, as it is based on iterators which prevent direct memory access, and automated techniques are similarly limited by the fact we can't easily work out address bounds for the stride of the stride-indirect pattern as a result. This makes the prefetch overly conservative if implemented in software, however this is no problem for the hardware prefetcher, as we can work out the memory bounds by hand before using the iterators, and as we shall see in chapter 6, the speculation afforded to us in hardware also makes it easy to target automatically with the hardware programmable prefetcher.

5.5.1.2 G500-List

This is from the Graph500 set of benchmarks, like G500-CSR which I evaluated on in the past two chapters. Unlike G500-CSR, it stores edge information in a linked list. This means it isn't possible to target in software, as the prefetch pattern requires too long a chain for the $O(n^2)$ algorithm to work effectively, but in hardware we can prefetch from the work list from multiple vertices at once, and let the hardware scheduler deal with fetching from multiple linked lists at once with minimal complexity.

5.5.2 Implementation details

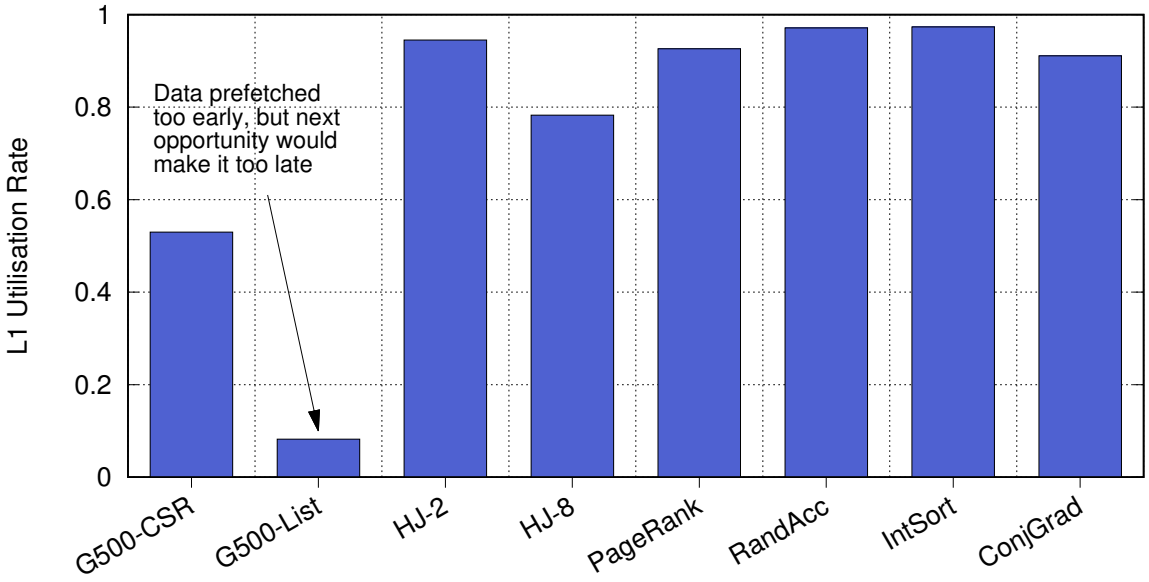
Much like the graph prefetcher in chapter 4, I implemented the programmable prefetcher in the gem5 simulator [19]. PPUs were implemented using the in-order MinorCPU gem5 model, and the main core as an out-of-order O3CPU model, the latter as with the previous chapter. PPUs were quiesced whenever no load or prefetch observations were available to process, and were otherwise awake.

The factor that made this more complex was to get gem5 to support the prefetch event-kernel programs, which needed to run on the PPUs. In syscall emulation mode, this was simple: each core could have its own separate program, and some of those could be event-kernel programs for the PPUs. However, since for full page table support, necessary for accurate prefetch results, full system mode was required, I ran into two problems. The first is that the main program would need to be pinned to the main core and the event kernels pinned to their own PPU, which by default the operating system would see as identical, and since the PPUs would be forcibly quiesced with no prefetching work, the operating system itself would need to run its code only on the main core. More problematically, for AArch64 gem5 could not support more than 4 cores at once in full system mode, limiting the potential numbers of PPUs to far lower than needed.

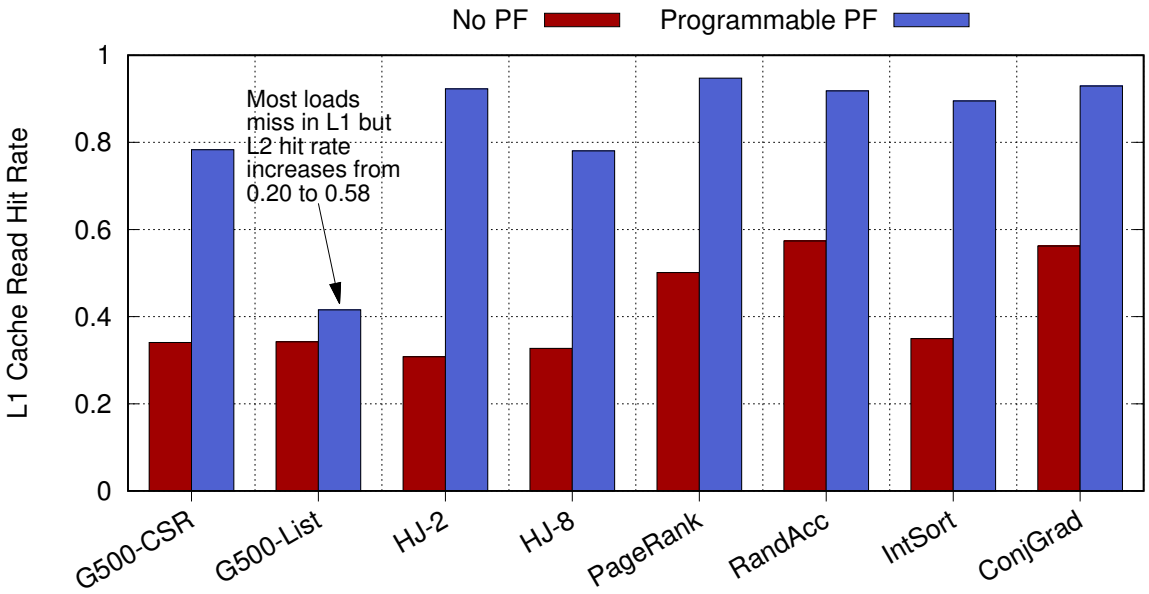
The solution to this problem of needing a main core with a full operating system, and PPUs running event kernels not controlled directly by that unmodified operating system, involved hacking the bootloader. I customised this in such a way that for a core ID of 0, the regular operating system code was booted. Otherwise, the prefetch event kernel initialisation code was jumped to, with the event kernels themselves built into the bootloader to be run as a bare-metal program. This is not how a real-world implementation would work (where the operating system would be modified), but is close enough for simulation purposes.

I reused the set_addr_bounds pseudoinstruction described in chapter 4, both for address bounds setting and for global register configuration, and added new pseudoinstructions to initalise the prefetcher, to trigger prefetch event kernels, to configure event-kernel function pointers, to configure the address filter with prefetch-triggered function pointers to conditionally quiesce the PPUs depending on whether any events are currently available to process, and to access EWMA state.

The source code for the simulator changes, along with benchmarks, is available in a data repository [6].



(a) Proportion of prefetches utilised before eviction from L1



(b) L1 read hit rate

Figure 5.10: 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.

5.6 Evaluation

5.6.1 Performance

Figure 5.9 shows that my event-triggered programmable prefetcher achieves speedups of up to 4.3× (geometric mean 3.0×) with the **programmable** prefetcher, compared to no prefetching, for the memory-bound workloads described in section 5.5, whereas **stride** and **software** prefetchers speed up by no more than 1.4× and 2.2× respectively.

The Markov global history buffer [82] gains no speedup with **regular** settings, since the applications I evaluate access far too much data to be predicted with such a small amount of state. When we increase the amount of prefetcher state (**large**) to 1GiB of data, we still only gain performance for benchmarks that 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.

5.6.1.1 Speedup

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 access based on a single strided load. The spatial locality simplifies 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 my 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, my event-triggered programmable 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 the PPUs attain 3.8×. Here I assume that we can only software prefetch the hash table in HJ-8: dynamically, it is always true that additionally three linked-list elements can be walked, but that is only true due to data input. By comparison, the programmable prefetcher can dynamically deal with any number of linked-list walks, since no fixed schedule is required.

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

There is no entry for software prefetching for PageRank in figure 5.9; the Boost Graph

Library code uses templated iterators which only give access to edge pairs, meaning it isn't possible to directly modify the C++ to get the addresses of individual elements to issue software prefetches to them.

5.6.1.2 Cache impact

Figure 5.10 explores reasons for performance gains in more detail. Figure 5.10(a) shows that while L1 cache utilisation is high for most benchmarks when using my event-triggered programmable 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 5.3(d)). This is fundamental for any prefetcher that works by bringing data into the cache in parallel by traversing the data structure: the coarse-grained nature of the parallelism forces us to bring more data in than would be ideal.

The L1 cache-read hit rate does increase for G500-List, as shown in figure 5.10(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.

5.6.1.3 Comparison with the configurable graph prefetcher

Figure 5.11 shows comparisons of the two benchmark-workload combinations evaluated in both this and the previous chapter. For the simpler workload (PageRank), no deficit in performance is observed from using programmable prefetching: in fact, performance is slightly increased compared to the graph prefetcher from directly targeting the workload. The stride-indirect pattern maps very well onto the programmable prefetcher model, and so performance is very high.

By comparison, for G500-CSR, while the programmable prefetcher is successful, it has lower performance than for the configurable prefetcher. Part of this, as we see in section 5.6.2.1, is because the PPUs can't quite keep up with the main core. However, another issue is that the parallel programming model is slightly limiting compared to the flexibility of what we can do with full control over the hardware. Because prefetches aren't stateful, we can't fully implement the large-vertex mode from section 4.3.2.2: instead of

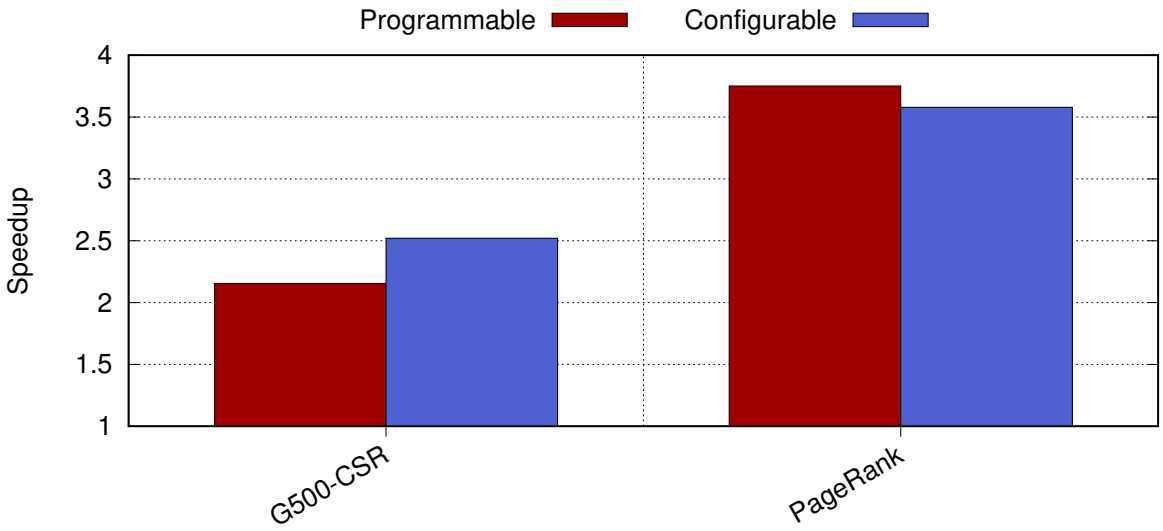


Figure 5.11: Performance of the programmable prefetcher in comparison to the configurable graph prefetcher from chapter 4.

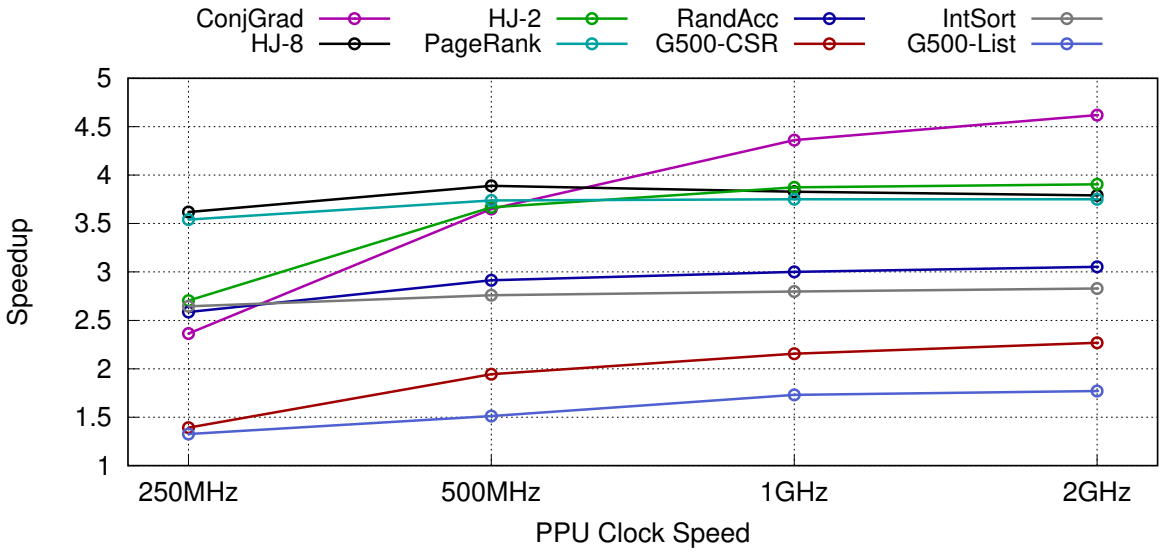
switching between the two modes dynamically, we instead need to manually configure large-vertex mode off and on in software, and work out and store the next vertex in advance, so that the stateless prefetcher can prefetch it when it nears the end of an edge. This lack of state, a sacrifice to achieve the parallel programming model, limits performance slightly.

5.6.2 Parameters

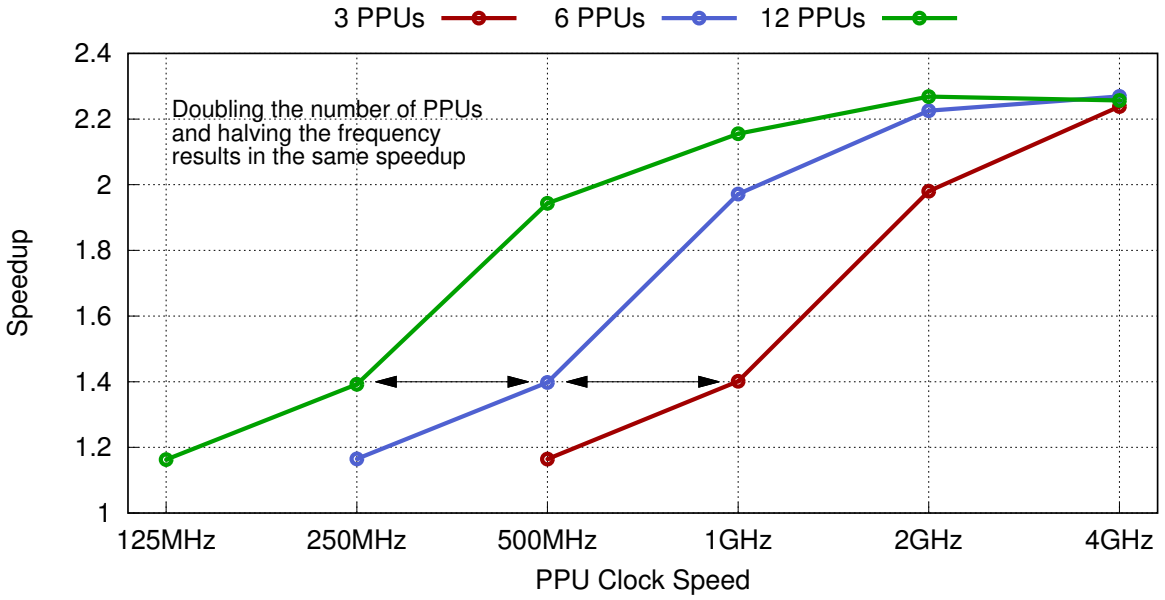
The programmable prefetcher configuration used up until this point contains 12 PPUs, each small, in-order, and running at 1GHz, compared to the 3.2GHz out-of-order superscalar main core. We will now see that this PPU configuration realises most of the benefits when compared with adding more prefetch computation power, and that scaling continues with increasing numbers of PPUs and their frequencies, since the prefetch kernels are embarrassingly parallel.

5.6.2.1 Clock speed

Figure 5.12 shows how PPU clock speed affects each benchmark and the impact of reducing the number of PPUs. Figure 5.12(a) demonstrates that approximately half the workloads gain little benefit from increasing the frequency of the PPUs. On the other hand, for HJ-2 a 500MHz frequency suffices to realise its maximum speedup whereas ConjGrad and G500-CSR achieve speedups that continue scaling with PPU frequencies up to 2GHz. Overall, the majority of the benefits are obtained at 1GHz where the geometric mean of speedups is 3×, increasing to 3.1× at 2GHz.



(a) Clock frequency impact



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

Figure 5.12: 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.

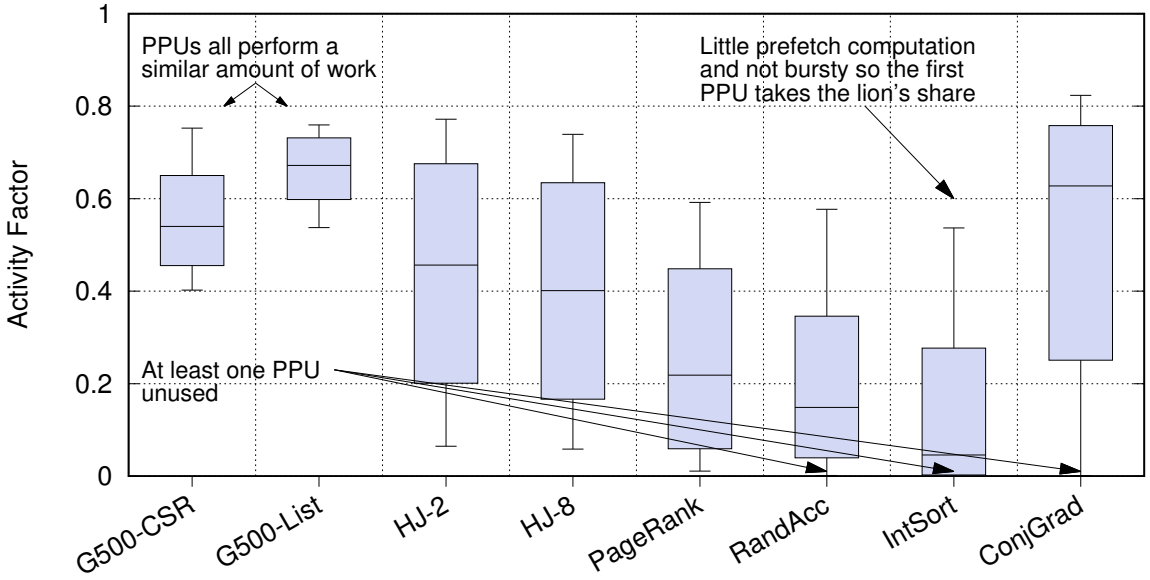


Figure 5.13: Range, quartiles and median for the proportion of time each PPU is awake and calculating prefetches at 1GHz.

5.6.2.2 Number of PPUs

The relationship between PPU frequency and the number of PPUs is explored in figure 5.12(b) for G500-CSR, chosen as an example of an application that continues scaling with frequency increases. PPU frequencies up to 4GHz are shown as a study only, to assess this relationship; we should 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.

5.6.2.3 PPU activity

Figure 5.13 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. My 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 this analysis to be performed.

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 woken. 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 5.12(a)) or fewer of them.

ConjGrad is an outlier in that some PPUs do little work, yet it scales with increasing frequency (figure 5.12(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 the PPUs only react to events from the main core, and so are not required during phases where no data needs to be prefetched.

5.6.2.4 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 5.6.1, but would be reduced with a larger L2 cache to buffer data fetched too early. G500-CSR, with 16% extra memory accesses, has variable work per vertex, meaning prefetch distance must be overestimated relative to the EWMAs. This is slightly higher than in chapter 4, when 11% overhead is observed for the same benchmark, since we can't store state in the programmable prefetcher and thus can't switch as easily between the two modes of operation described in section 4.3.2.2.

5.6.2.5 Event triggering

To examine how much of the performance we attain is through the latency-tolerant event-triggered programming model, I 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 5.14. Where the pattern is a simple stride-indirect, performance is relatively close: we only have to stall on the stride access, and

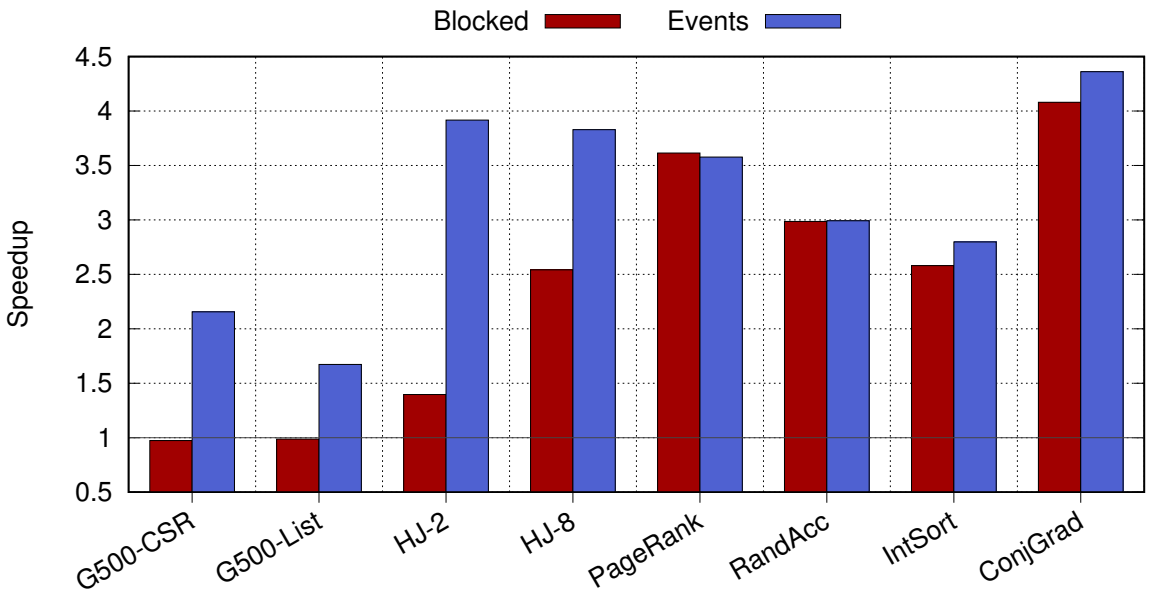


Figure 5.14: Performance with and without blocking on intermediate loads, with 12 fetcher units.

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 event kernels are necessary for the system to work, even with the large amount of parallelism available from twelve PPUs.

5.6.3 Area and power overheads

Though the prefetcher features many programmable units, each one of these is a very small, micro-controller sized unit, such as the Arm Cortex M0+, which contains fewer than 12,000 gates [16] (approximately 50,000 transistors). Using public data [13, 14], on comparable silicon processes we should expect the overall hardware impact, including twelve cores and memory, to be approximately 1.5% of the area of a Cortex A57 without shared caches. This is comparable in size to a 32KiB L1 data cache. In practical implementation terms, it may be desirable to support 64-bit operations on these cores, and thus we could expect the area to double to 3%. This is inevitably bigger than a configurable prefetcher, as in chapter 4, due to the programmable hardware, but is still very minimal compared with the size of an out-of-order superscalar core.

5.7 Conclusion

The event-triggered programmable prefetcher here generalises the configurable schemes presented in chapter 4, by replacing the fixed-function logic with a set of small, highly parallel RISC cores, and an associated parallel programming model based on using events to trigger prefetching. On a selection of graph, database and HPC workloads, the prefetcher achieves a geometric mean 3.0× speedup without significantly increasing the number of memory accesses.

Though the event-triggered programmable prefetcher works well in terms of performance, manually generating programmable prefetch event kernels can be challenging: it requires custom design for each slow memory access. The question, then, is how much of this can be offloaded away from the programmer and instead performed by the compiler. The following chapter explores how we can achieve this, to get both high performance and ease of use.

Chapter 6

Programming the programmable prefetcher

The event-triggered programmable prefetcher developed in the previous chapter is able to issue prefetches with a high throughput, by reacting to events based on filters from the cache load stream, and using a parallelism-tolerant programming model. This results in high performance at the expense of a number of constraints on what can be run in an event kernel, and a complicated programming model.

To ease this complication, we need to provide some sort of assistance for programming it, above and beyond writing event kernels by hand. I propose compiler analysis passes that take these constraints into account to generate good event-kernel code automatically, coming close to the performance of handwritten event kernels for simpler access patterns that the compiler can pick up.

This is made significantly easier by virtue of the event-triggered programmable prefetcher being unable to affect correctness. This means that we can be lenient with the guarantees a compiler analysis technique gives us: for example, if a memory access performed by the prefetcher faults, the event kernel that generated it can be silently discarded, rather than the error propagating to the main program. Similarly, we have no need to guarantee the precise order of memory accesses, or even that an address the prefetcher accesses will be used by the main program at all. This gives us the ability to speculate at compile time, further than we could when we were vulnerable to memory access faults we observed in chapter 3.

This chapter provides two compiler techniques to aid programming hardware prefetchers: a conversion pass that takes in software-prefetch-style non-blocking loads inserted into the main program, and converts them into programmed event kernels, and a higher-level pass that uses pragmas inserted by a programmer or profile-guided compiler to highlight regions of interest to generate prefetches just from existing loads. This allows programmable prefetchers to be targeted by a wide set of programs, with different programming models

trading off ease of use and performance improvement.

6.1 Requirements

To generate event-triggered programmed prefetches, we need to create code that fits within the constraints of the previous chapter: namely, that the prefetches can be successfully run as event kernels. The most salient of these are that:

- A set of prefetches can be triggered by an access to a configured array. This means in practice that we need to be able to infer an induction variable from accessed memory addresses, to deduce computation progress without explicit signalling. It also means that we need to be able to infer address bounds for at least part of a data structure, to trigger the initial prefetches.
- We need to only access one data item in each generated event kernel. This means that we only need to access data from the word prefetched by the previous event kernel in a chain: technically, since the prefetcher can access every element in a cache line on the return of prefetched data, this is conservative, but it's a reasonable approximation that makes generating code easier.
- Any values accessed by the event kernels, other than the address and data from a load or prefetch, must be configured in global registers. To make this easier, and reduce configuration, I ensure that these values must be loop invariant, and so can be set and unset at the start and end of loops.

To make code generation easier, I use the memory request tags to store function pointers to the next event kernel in a chain, rather than using addresses to trigger each new prefetch. This simplifies generation of code. The first constraint could also likely be simplified if we instead allowed triggering on program count instead of just on array bounds, but this isn't explored further here as it is unnecessary for the workloads on which I evaluate.

6.2 Suitable high-level programming models

Fundamentally, we need to provide a way for the prefetcher to be automatically configured by using the compiler to generate event kernels. There are several programming models that could be used to help support this, and I consider several here before implementing two automation schemes at varying levels of programming complexity.

6.2.1 Software prefetch intrinsics

Software prefetches are a relatively expressive way of describing a prefetch. They allow the exact memory-access pattern to be described, along with a look-ahead on an induction variable. Provided the constraints of the previous section hold, then we just need to split each load in a prefetch into an event, and trigger the prefetch based on loads to the base array.

Since programmable prefetches are more latency tolerant than software prefetches, the actual structure of a prefetch for the former differs, in that no staggered prefetching of values used in a later prefetch is required. In addition, no address bounds checks are needed for programmable prefetches as they are speculative in nature. It is therefore safe to say that, though we can convert real software prefetches into events, it is easier, and more expressive, to simply use software-prefetch-style intrinsics to hint to the programmable prefetcher what to prefetch, and convert those into events. I therefore focus on doing the latter, though the techniques I develop equally apply to the former.

6.2.2 Pragmas

Writing software prefetches is complicated, and though the hardware prefetcher removes the need to get right some of the more complex details (scheduling, correctness of non-speculative loads, staggering prefetches in chains) it is still likely that many programs that aren't written by experts in prefetching still require high performance for irregular memory accesses. I therefore give a technique that simply requires an OpenMP-style pragma [30] to be placed around loops with memory accesses that are performance critical and likely to miss. This could either be placed by the programmer or by a profile-guided compiler. This is likely to perform less well than both software prefetch intrinsic conversion and manual event-kernel programming, as the programmer can no longer supply information that would otherwise be available only at runtime, along with domain-specific knowledge, but for simple patterns it should be possible to prefetch a similar amount just from analysing code regions that have been marked as likely to miss.

6.2.3 Automated

Since the technique in chapter 3 can automatically generate software prefetches, it stands to reason that, provided we have a software prefetching conversion pass that also works, then programmable prefetcher events can also be generated automatically. This would result in code similar to the pragma generation technique, only less aggressive: since the pragma indicates regions that particularly require prefetching, I also allow it to prefetch regular memory-access patterns instead of just irregular ones. Similarly, pragmas prevent code being generated in cases where it isn't useful. While both are suitable techniques, for

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

(a) Software prefetch

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

(b) Pragma

Figure 6.1: Source for auto-generation of PPU code.

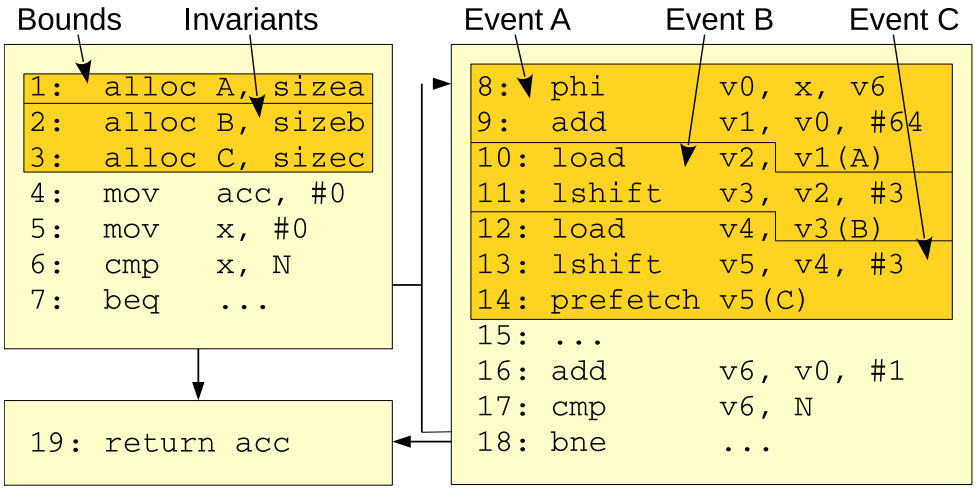


Figure 6.2: An overview of my software-prefetch conversion technique presented in algorithm 2, on the control flow graph from code in figure 6.1.

our benchmarks, automation along with a stride prefetcher would give similar results to pragma generation, and so I consider it no further here.

6.2.4 Other techniques

Software prefetches cannot be used to express loops, whereas programmable events can. This means that there are some prefetch kernels that will be unable to be expressed using software-prefetch conversion. Similarly, since I use my pragma technique to generate intrinsics to feed into the software conversion pass, this will be limited in the same way.

We can expect other access-pattern description techniques, which would be able to express such behaviour, to generate code for more types of access. But since these would likely be more difficult to program, both in the compiler and by the user, I do not consider them here and instead require the programmer to generate manual events in these cases.

```
// Collect initial software prefetches and their address-generation
// instructions.
prefetches = {}
foreach (p: software prefetches within a loop):
  // Same DFS as in chapter 3, but only applied to prefetches instead
  // of loads.
  if (((indvar, set) = DFS(p)) != null):
    prefetches ∪= {(p, indvar, set)}

// Function calls only 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 the 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 the event kernels.
  set.replace(loop invariant loads, global reg)

  // Split into event kernels with only single load references inside.
  event_kernels = split_on_loads(set)
  // Only continue if event splitting was successful.
  if(!event_kernels) continue

  // Find address bounds, and replace induction variable.
  addrbounds = infer_bounds(iv)
  event_kernels.replace(iv, (addrbounds.max - addr) / size)
  event_kernels.replace(final load, prefetch)
  event_kernels.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_event_kernels ∪= event_kernels
  ivs ∪= iv

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

// Optimise the generated events.
inter_event_kernels = common_event_comb(all_event_kernels)
final_event_kernels = batch_prefetch(inter_event_kernels)
add_to_event_list(final_event_kernels)
```

Algorithm 2: The software-prefetch conversion algorithm, assuming the intermediate representation is in SSA form.

6.3 Software-prefetch conversion

To convert software-prefetch-style intrinsics to event-triggered programmed prefetches, we need to ensure the generated code can be altered to fit within the constraints outlined in section 6.1. We then need to convert the address-generation code into event kernels, insert configuration instructions into the original code, and remove the original software prefetch code and any instructions it depends on. Pseudocode is given in algorithm 2.

6.3.1 Analysis

My 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, much like the technique presented in chapter 3, but applied to intrinsics instead of all loads. An example is shown in figure 6.2. It terminates 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 instructions ending in a single load, which will 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 that 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 kernel for a set of prefetches. The latter case requires more complex analysis, and in practice is rare, so I do not discuss it further. Checks in the original code for array bounds can safely be ignored here, and since these checks will not form part of the address calculation itself, do not affect the dataflow analysis, and so prefetches will still be generated even in the face of such branches.

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 kernel cannot be triggered by the arrival of a single data value. In this case the conversion fails. However, if only one load is found, the algorithm packages the instructions into an event kernel, and repeats the analysis again starting from this load.

Figure 6.2 shows the control-flow graph for the code in figure 6.1(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 kernel, and analysis restarted with the *load*. This terminates on with the *load* at line 10, and again an event

kernel is created. Finally, the third analysis pass terminates with the *phi* node, which is for the loop induction variable, so a new event kernel is created and no further analysis is required.

6.3.2 Address bound 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 kernel when snooping a load. For example, in figure 6.2, code for event kernel 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 5.3.8.

This is less strict than the automated software-prefetch generator of chapter 3, as address bounds are only needed for triggering new prefetches, rather than correctness. To be able to pace the prefetch, we need to use this data to infer the induction variable: we can do this by observing loads to an array indexed by the induction variable, subtracting it from the base of the array, and shifting or dividing depending on the element size.

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. I use the same best-effort approach as in section 3.2.2. The difference here is that the loop size as a substitute for array bounds is a closer fit for correctness than in chapter 3, as we are just inferring progress through the loop.

One minor downside of using the loop size to configure the address filter, rather than the array size, is that the configuration must therefore occur while the size of the loop is live, rather than while the size of the array is live, meaning opportunities to push configuration instructions to outer loops may be missed, slightly reducing the potential performance improvements. Still, provided configuration isn't too expensive within the system, this should be minor.

Though in the previous chapter I typically use the address filter to trigger event kernels based on the arrival of prefetches as well as inference of progress based on loads, here I only use the memory request tags (section 5.3.8) for the former, to simplify code generation. This means that, instead of having to infer all memory bounds, we just infer the first in a chain, then use the request tags to effectively continue with a sequence of code once a prefetch has returned.

6.3.3 Address-bounds liveness analysis

As my software-prefetch conversion algorithm uses observed address loads to infer the induction variable, we need to ensure that, as well as the data structure being loaded from on each iteration of the loop, the data structure isn't accessed at any other point while the address-range configuration is active.

Violating the latter doesn't affect correctness of the program, as prefetch programs cannot do so by virtue of only being able to request addresses, and are unable to cause crashes or alter data. Still, having the address table inappropriately configured may cause extra prefetches to be generated which are unnecessary, potentially harming performance. We can afford to be lenient here: correct alias analysis isn't vital as we are after performance, not necessarily correctness, for prefetches.

So, it is desirable to place configuration instructions after any other previous loop that accesses the data structure, and configuration nullifying instructions before any loop that accesses the structure after the target loop. One place that will capture all memory accesses to be prefetched within such configuration instructions is directly before and after the target loop. However, if the loop is nested, this may mean multiple useless configurations, so it may instead be desirable to push the configuration into outer loops provided it is likely to not cause false prefetches to occur.

6.3.4 Generation

The tasks of the code generation pass are to insert prefetcher configuration instructions, generate event kernel code to run on the PPUs and remove the original software prefetch instructions. Using the analysis described in section 6.3.3, 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, the pass takes sets of instructions identified using the analysis in section 6.3.1, and turns them into even kernels. In the first event, it replaces 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). The final instruction in each event, which will either be a load or software prefetch, is replaced with a hardware prefetch instruction. For loads, a callback is added so that the next event kernel in the sequence is called once this prefetched value arrives in the L1 cache. Loop-invariant values are replaced 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 can be converted into a register access.

Finally, the pass removes the now-unnecessary software prefetch intrinsics. 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.

6.3.5 Comparison with true software prefetching

Differences in semantics and performance characteristics of hardware prefetches versus software prefetches mean that different software prefetch intrinsics may be optimal for conversion to run on a programmable prefetcher versus directly running as software prefetches. Indeed, software prefetches that reduce performance as a result of filling the instruction stream with too many extra instructions in software may improve performance significantly when converted to event kernels for an event-triggered programmable prefetcher. Conversely, some software prefetches cannot be converted to the constraints of the programmable prefetcher.

As address generation for a software prefetch can fail and potentially cause exceptions or segmentation faults, whereas prefetch events get safely thrown away in the event of a fault, a converted prefetch can be more speculative and involve fewer checks: we can issue a prefetch to a null pointer without checking it, with no negative consequences, or walk over the end of an array. Further, we can speculatively access the first N elements of a linked list even if not all of them exist. Removing the checks in prefetch code reduces dynamic instruction count, improving performance, and also potentially allowing more prefetches to be computed under the restrictions on memory access allowed by the PPUs.

As the PPUs do not have to stall on loads, higher levels of indirection can be profitably prefetched: this means that software prefetches for conversion to event kernels do not need to stagger prefetches and loads to chained memory locations. We can convert between the two forms by removing the excess code, however, using the common-event combination given in the next section.

6.3.6 Optimisations

The above constraints are enough to generate correct prefetches, that will prefetch all of the data specified in the prefetch intrinsic provided it can be converted into an event. However, it is possible to reduce work from multiple prefetches with some analysis that works out which can be combined to reduce work.

6.3.6.1 Common-event combination

The event generation laid out in section 6.3.4 gives event kernels that generate the correct prefetches. However, identical event kernels may be scheduled to collect the same data for common subexpressions of prefetches as a result of the above scheme.

```
1 uint64_t offset = xoffe[2*vlist[k+8]];
2 __builtin_prefetch(&tree[adj[offset]],0,3);
3 __builtin_prefetch(&tree[adj[offset+1]],0,3);
```

Figure 6.3: Example prefetches from Graph 500 Search.

Consider the prefetch code in figure 6.3. With the algorithm above, on a visit to the vlist array, two separate event-kernel trees would be generated, both generating xoffe[2*vlist[k+8]], before the two prefetches diverge to generate the final addresses for each event. Ideally, we would instead have a single set of events to generate this value, followed by an event kernel that spawns two further event kernels: one for adj[offset] and another for adj[offset+1].

Another example is the staggered prefetches generated by chapter 3. These are useless for a programmable prefetcher: the final prefetch in a sequence, complete with blocking loads, is enough to generate the correct stall-free event-kernel sequence. We should therefore remove all other prefetches to the same data.

To do this, I first combine events that are triggered by the same array in the same loop (events triggered by the same array in different loops may cause different intended prefetches, so should not be combined). Any offsets within a newly merged event kernel are normalised to a single offset, to allow redundant fetches at different offsets, such as used for software prefetching in chapter 3, to be combined. This is then followed by common subexpression elimination within the event, to remove any redundant code. I then merge prefetches to the same data within an event kernel into a single prefetch, combining the event kernel called at the completion of a prefetch for each redundant prefetch into a single event kernel, then recursively apply the same optimisation.

6.3.6.2 Batch prefetch

As an extension of common-event combination, we can note that the prefetcher units can access the entire cache line of data in a prefetched address (section 5.3.9). This allows us to reduce the number of prefetch events generated, as the first prefetch event kernel in a sequence is usually a strided look-ahead in a base array, with every element being accessed. When this is the case, and the number of prefetches generated in a sequence is limited such that merging them won't cause the L1 cache to evict large amounts of useful data, the compiler pass merges events by altering the first event kernel in a sequence, which triggers a stride, to only be triggered on the edges of cache lines, and the second event kernel to generate prefetches based on the whole cache line. This can reduce the computational requirements of the prefetch units.

6.3.7 Extensions

The optimisations above are implemented in the example LLVM [19] passes I evaluate below. However, there are several features, which would improve coverage or performance, that I choose not to implement in this test implementation. These are summarised below.

6.3.7.1 Branches

When a software-prefetch instruction occurs in a different basic block from its induction variable phi node, this can cause two issues. First, the execution of the prefetch may be conditional on particular branches being taken. Typically this is due to bounds checking on arrays used to issue load instructions, which are unneeded in the event kernel programming model, as loads prefetched in an event kernel are allowed to go out of bounds. Thus, while it is possible to add such branch checks to events to prevent the next prefetch in a sequence from being issued, we ignore any such branches, focusing only on the values required to calculate the address itself.

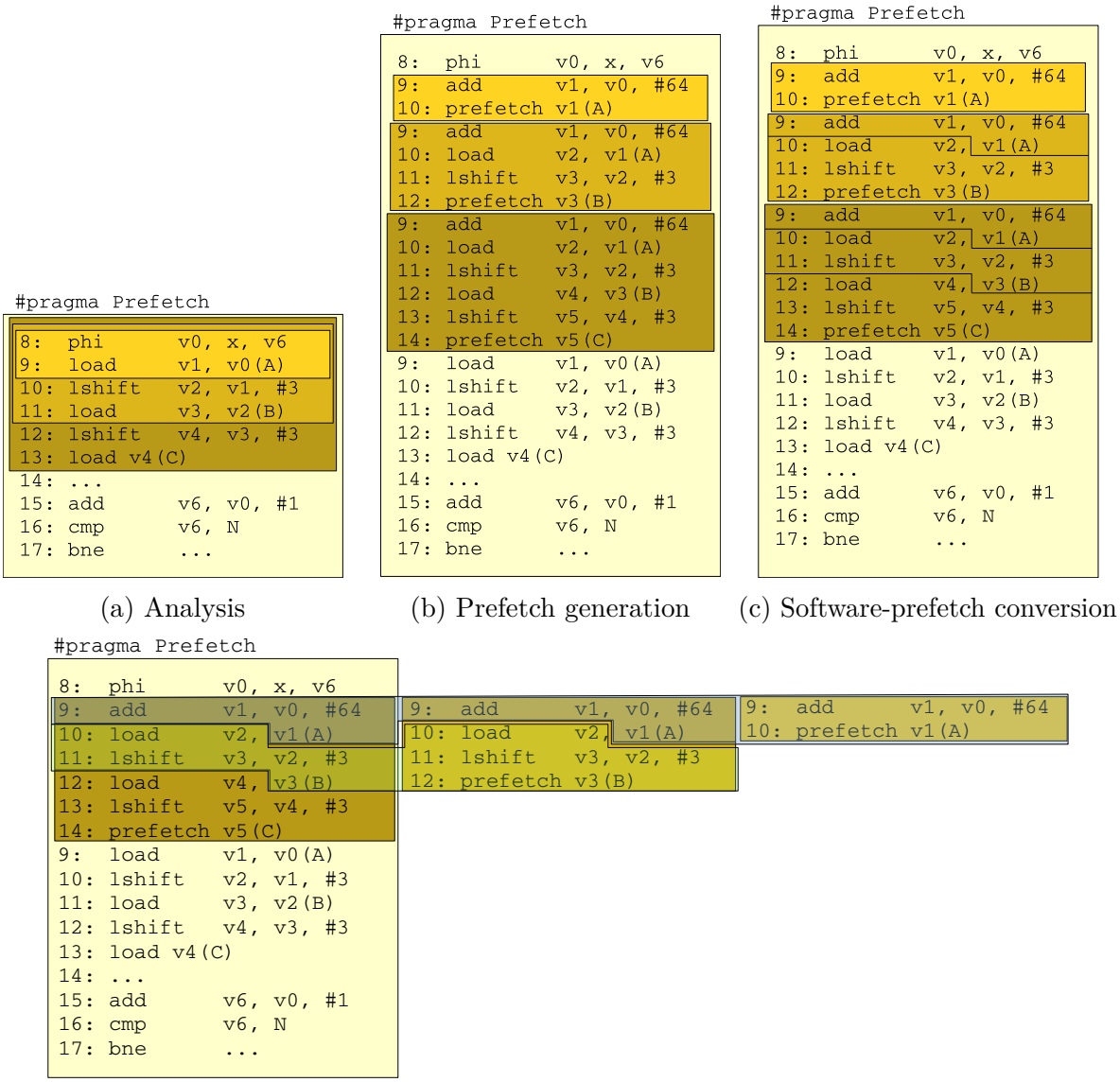
Other branches can cause control-flow effects that change the value of the software-prefetch operand: in these cases, non-induction-variable phi nodes will appear in the set of instructions required to calculate this value. In these cases, we must work backwards and characterise how this changes how many values are loaded: if the load events down both paths are the same, we can include the relevant branches and basic blocks in the same event. Otherwise, we must split the event kernel where the phi node appears in two, and generate separate event-kernel streams based on each. As this complicates code generation and analysis, and appears to be rare in typical software prefetches, I don't implement this feature: for such complex code, events have to be written by hand.

6.3.7.2 Dynamic scheduling

As at compile time we do not know the runtime characteristics of how much time code paths will take, or the memory latency of a system, fixed values are often suboptimal for the look-ahead distance for a software prefetch. As programmable prefetchers feature analysis hardware to take exponentially weighted moving averages (EWMAs) of these properties at runtime, we can replace fixed look-ahead values within a software prefetch with dynamically calculated values set up by the prefetcher configuration. I don't currently implement this, but detection of fixed offsets and replacements is a trivial addition.

6.4 Pragma generation

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



(a) Analysis (b) Prefetch generation (c) Software-prefetch conversion

(d) Common-event combination. Code on the same line indicates redundantly generated event-kernel code, which can be merged with event kernels in the same box, then simplified to discard the redundant instructions.

Figure 6.4: An example of pragma generation on the code from figure 6.1. First, code in analysed for load patterns with look-ahead. These are then turned into software prefetches, which are then converted to programmable prefetcher events by the algorithm from section 6.3. Finally, common events are combined, to achieve the same events as those generated by the software-prefetch conversion example of figure 6.2.

```
// Generate initial set of loads to prefetch and their address
// generation instructions.
prefetches = {}
foreach (l: loads within a loop marked with prefetch):
  // Same DFS as in chapter 3.
  if (((indvar, set) = DFS(l)) != null):
    prefetch = {(l, indvar, set)}

    // Function calls only allowed if side-effect free.
    if(prefetch contains function calls) continue
    // Prefetches should not cause new program faults.
    if(prefetch contains loads which may fault) continue
    // Non-induction variable phi nodes allowed if pass can cope with
    // complex control flow.
    if (prefetch contains non-induction phi nodes) continue

    // Unlike in chapter 3, we can immediately generate the new
    // prefetches, as offsets are no longer relative to any other
    // prefetches generated, since this is handled by the software
    // prefetch conversion.
    insts = copy(set)
    foreach (i: insts):
      // Update induction variable uses. This time, unlike in chapter
      // 3, I use the same offset regardless of the depth of the
      // prefetch, as this will be cleaned up by the software prefetch
      // conversion based on common events.
      if (uses_var(i, iv)):
      replace(i, iv, min(iv.val + 64, max(iv.val)))
    // Final load becomes the prefetch.
    if (i == copy_of(ld)):
      insts = (insts - {i}) ∪ {prefetch(i)}
    // Place all code just before the original load.
    add_at_position(ld, insts)

// Convert all newly generated prefetch intrinsics into events, and
// clean up duplicates.
run_software_prefetch_conversion()
```

Algorithm 3: The pragma prefetch generation algorithm, assuming the intermediate representation is in SSA form.

generating the initial software prefetches (chapter 3), 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. This is supported through a custom prefetch pragma (as in figure 6.1(b)) using a similar depth-first search approach as in section 6.3.1.

Generating code in this manner means we have less information to work on than with the software-prefetch conversion 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 profile information, which loads are likely to access data that is already in the L1 cache, and

thus which prefetches are unnecessary (though these could generated and executed, then disabled at runtime based on hardware analysis). However, for simple patterns, a pragma descriptor is equally powerful as software-prefetch conversion.

I have developed a pass that attempts to find loads that can be prefetched based on look-ahead within an array, for loops with a prefetch pragma around them,[1] and generates software-prefetch-style intrinsics for those it can successfully target. These can then be fed into the pass for software-prefetch conversion.

Pseudocode for the pragma-generation compiler pass is given in algorithm 3. An overview of this being applied to figure 6.1 is shown in figure 6.4.

6.4.1 Analysis

To analyse whether it is possible to generate a useful software prefetch, the technique performs a similar analysis to that in chapter 3. It looks for loads within pragma-marked loops and checks whether it is possible to generate a prefetch with look-ahead: that is, it checks whether it can generate a new load address by increasing the value of a referenced induction variable within the address calculation. An example is shown in figure 6.4(a). This is again done by searching backwards from a load using a depth-first search. The algorithm attempts to find an induction variable within the transitive closure of each operand. It works backwards until it finds an induction variable down a path, or until it reaches code that isn't in any loop. It then stores any instructions that reference this induction variable, either directly or indirectly, down each path. If two paths reference different induction variables, the algorithm only saves the instructions that reference whichever is innermost: this reflects that this is likely to be the most fine-grained form of memory-level parallelism available for that loop.

Again, I constrain this set of instructions, such that no stores, function calls or non-induction-variable phi nodes appear as instructions we need to change. This is a strict set of conditions that could be relaxed with further analysis, but ensures that we can insert the new prefetch instruction in the same place as the old load, without adding further control flow. We must also ensure all operands for these instructions are available at this point, and that it isn't in an inner loop within the induction variable's loop, so that we only issue the software prefetch once per loop iteration.

6.4.2 Generation

I generate prefetch instructions in the LLVM IR for the main core by inserting instructions based on the values affected by the induction variable from the analysis in section 6.4.1. I

[1] As this is tricky to implement in LLVM, due to new pragmas currently requiring core LLVM data structures to be changed, in my implementation I instead placed the pragma around entire functions. This does not affect the results here, and thus I use my implementation as a substitute for the more ideal case.

work forwards from the first instruction, which will be an induction phi node, adding in an add instruction which operates on the induction variable, increasing it by a fixed value: in the example given in figure 6.4, 64 is chosen, but this can be altered to become dynamic once fed to the software-prefetch conversion pass (section 6.3). I then continue, creating new copies of the instructions the algorithm identified that needed to be changed to create a prefetch, but with any induction-variable-affected operands replaced by the instruction copies. Finally, I replace the final load of the prefetch with a software-prefetch instruction. Example prefetches are shown in figure 6.4(b), and these are then converted into event kernels by software-prefetch conversion, as in section 6.3 (example figure 6.4(c)).

I generate prefetches even if they are a subset of longer prefetches: consider the example in figure 6.4(b). This would generate a prefetch for, A[x+64], B[A[x+64]], and C[B[A[x+64]]]. This is similar to chapter 3, only without the now needless staggering, as it is unnecessary for conversion to events, and this allows us to naïvely generate a prefetch for each observed access, simplifying the complexity of code generation. However, this will be cleaned up by common-event combination (section 6.3.6.1, figure 6.4(d)) when converted to programmable-prefetcher events.

6.4.3 Comparison with automated software prefetch generation

As these software-prefetch intrinsics are used as an intermediate pass to generate prefetch event kernels for an event-triggered programmable prefetcher, which is allowed to speculate and thus doesn't cause exceptions, these prefetches are constrained less than the software prefetches of chapter 3. We no longer need to perform checks on address bounds (section 3.2.2), as the event-triggered prefetcher is designed not to fault on incorrect-address calculations. We also no longer need to make sure that no intermediate stores happen that could affect results of true loads used by the prefetch-generation code, as the validity of addresses is no longer necessary. The loss of these constraints both eases compiler analysis, and increases how speculative the resulting prefetches can be.

Another difference is that the prefetches here aren't staggered. Since the event-triggered prefetcher doesn't stall on intermediate loads, we have no need to generate many prefetches at different offsets for chains of loads. I instead generate prefetches at a single offset, then let the common-event combination pass clean them up later into a set of events (figure 6.4(d)). Once this code is cleaned up by the compiler, it means prefetches can be achieved in $O(n)$ code in the length of a dependent chain, instead of $O(n^2)$.

Generally, we can be much more aggressive with an event-triggered programmable prefetcher than with automated software prefetching, even when generating only based on a pragma. This is because, as well as fewer limitations on what we can safely prefetch, and a smaller growth in number of operations from prefetching, since we ultimately offload the prefetching to a programmable prefetcher, compute resources aren't taken from the

main core, and since they can be potentially turned off at runtime if not useful, slowing down performance is less of a concern. In addition, the use of a special pragma, set by a profile-guided pass, or by the programmer, allows compiler effort to be particularly targeted at important loops, and avoids impact on less common loops, or those that don't require prefetching.

6.5 Experimental setup

Here, I use the same simulated gem5 [19] setup and benchmarks from the previous chapter (table 5.1). As in chapter 3, the two compiler techniques presented here, software-prefetch conversion and pragma generation, are implemented as LLVM passes [65].

6.5.1 Implementation details

Software-prefetch conversion and pragma generation are both implemented at the EP_VectorizerStart stage in LLVM, which is relatively late in the IR pipeline, but before vectorisation and target-specific optimisation. The pragma-generation pass is also implemented as a FunctionPass, but since for common-event combination, and address filter and global register allocation, it is useful for the actual event-kernel generation to have global scope, this is instead done as a ModulePass.

The pragma-generation pass is implemented in a similar way to the software-prefetch-generation pass of chapter 3 (indeed, the latter was based on the former), but with fewer checks, as they aren't necessary when converting to events, and without any need for the scheduling technique used in chapter 3. Since it is easier in implementation than proper loop annotations, I instead look for function annotations to trigger the search for prefetchable loads: I get the llvm.global.annotations at the start of each function, and check if this includes the prefetch attribute.

The software-prefetch-conversion pass analyses the code to isolate and remove the IR instructions used just for prefetching. These instructions are altered as specified in algorithm 2, then this new code is output as individual event kernels to an LLVM IR text file. These then get cleaned up to be placed back in SSA form, have the definitions of the gem5 pseudoinstructions added, and are then converted to a file that can be linked with the custom prefetcher-bootloader described in chapter 5. To remove the old software-prefetch code, an aggressive dead-code eliminator aware of the removal of prefetches themselves is used, to remove any code that isn't used by any real instruction, including if that code could have caused faults in the original execution. Finally, the address bounds and global-register configurations are added into the original code, based on the ID assigned to their code in the combined event-kernels file, and configuration instructions linking those IDs to function pointers are placed at the initialisation of the event-kernels.

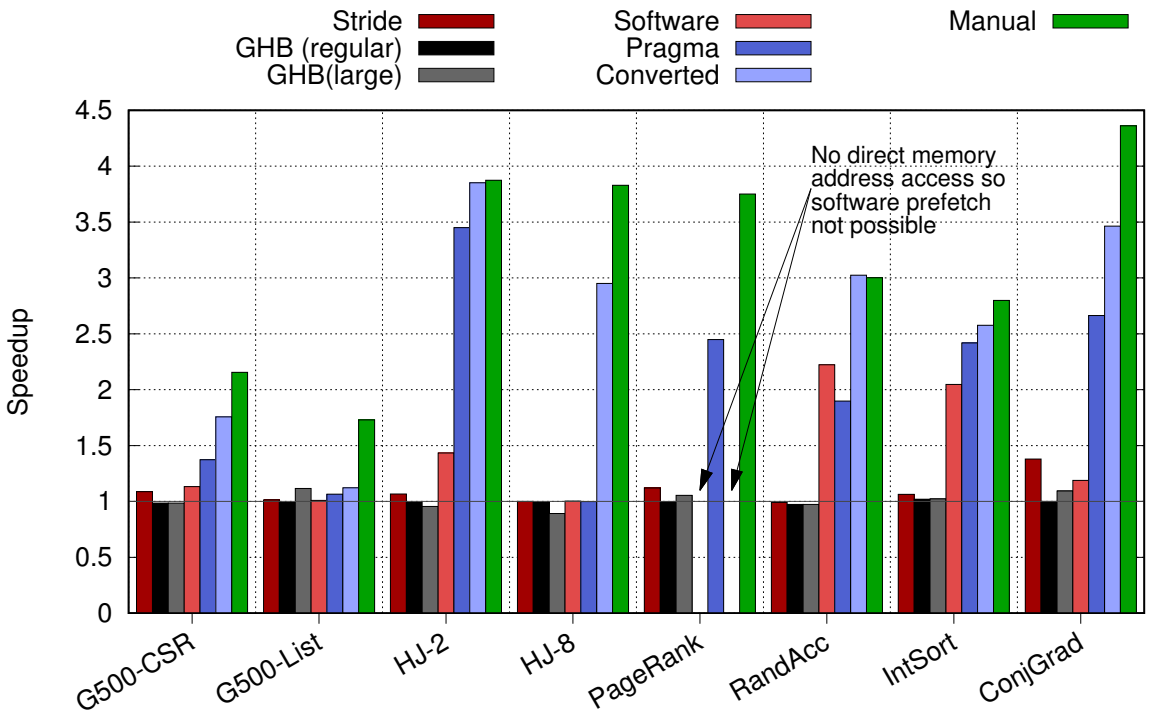


Figure 6.5: Performance for the two compiler assisted schemes, versus the baselines and manual event-kernel generation used in chapter 5.

The source code for the LLVM compiler passes is available in the same data repository as the code from the previous chapter [6].

6.6 Evaluation

Speedups for the two compiler passes, along with every scheme evaluated in the previous chapter, are given in figure 6.5. My compiler-assisted software-prefetch conversion pass (**converted**) achieves similar speedups to manually written events for benchmarks except for on the Graph500 workloads, and my automatic event-generation technique based on pragmas (**pragma**) is able to speed up simpler access patterns as much as manual, but isn't able to achieve full potential for four of the eight benchmarks. There is no bar for conversion on PageRank in figure 6.5, as with software prefetching in figure 5.9; again, the Boost Graph Library code uses templated iterators which only give access to edge pairs, meaning it isn't easily possible to get the addresses of individual elements to issue software prefetches to them in C++ without rewriting the library to avoid the use of iterators.

6.6.1 Performance

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 my scheme leaves the first few entries of the array unprefetched. Still, the 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 for variable numbers of buckets, 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 "prefetch the first N elements" approach should work well. If the prefetcher attempts to prefetch null at any point, that particular chain stops but other prefetches will continue, which is the correct behaviour. Still, with manual prefetching, we can introduce true control-flow loops within and between event kernels, to walk all buckets until we try to prefetch a null pointer, instead of just a finite number of buckets.

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 cache line in either my software-prefetch-conversion or pragma-generation passes, as they assume access to only one loaded value at a time. The pragma-generation 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 both the software-prefetch-conversion and pragma-generation passes have limited impact on performance. A compiler pass able to deal with loops within events may achieve more, but even then an amount of semantic knowledge only accessible to the programmer may be fundamental to working out that, even with the complex looping control flow from accessing edge linked lists, it is still beneficial to prefetch from the outer work list.

6.7 Conclusion

The event-triggered programmable prefetcher can be a challenge to program: manual event kernels have to be tailored for every access pattern, and can be fiddly to get right. To ease this, I have designed two compiler techniques to reduce manual effort: a software-prefetch-style intrinsic converter, to allow events to be specified in a high level way, and a pragma generation scheme, which allows the programmer or a profile-guided optimiser to specify which loops are important, and thus target loads that are accessed using an induction variable as candidates for event-kernel generation.

These two schemes were implemented in LLVM [65], and evaluated on the gem5 simulator [19]. Pragma conversion achieves a geometric mean 1.9× speedup, and software-prefetch conversion achieves 2.5×, compared with 3.0× for manual event-kernel writing.

This gives a spectrum of techniques trading off ease of use for maximal performance. Programmed prefetching can be auto-generated, or hand tweaked, or written fully custom, for maximum performance.

Since the techniques used here are based on software-prefetch abstractions, they are easy to work with, but have a fundamental limitation, in that they cannot deal with control-flow loops of data-dependent size. In particular, the pragma generation is mostly limited in this way because it is used as input to the software-prefetch conversion pass. A more complicated technique may be possible to achieve speedups closer to the optimal, manually-generated events, though since this is just a proof-of-concept, I consider this no further.

Fundamentally, I have demonstrated that the event-triggered programmable prefetcher can be targeted with minimal effort from the programmer, and thus it even has potential uses on commodity systems, where programmers may not be willing to put in the maximal effort for performance.

Chapter 7

Conclusion

Modern workloads are increasingly memory bound, due to unpredictable memory-access patterns. However, for many workloads this limitation isn't fundamental. We can prefetch the memory accesses, which means to bring the data into fast cache memory before it is used, to overlap requests and hide latency. If we have a sufficiently complicated scheme, we can extract arbitrarily complex amounts of memory-level parallelism provided it exists. Current systems support software prefetching, where memory locations can be preloaded in advance so that they are in the cache when requested. I developed a system to automate the generation of good software prefetches for many irregular access patterns within the compiler. However, while this gains some performance improvement, software prefetching cannot extract the full performance potential available. Software-prefetch instructions are costly within the processor, as they significantly increase dynamic instruction count. Further, if the address calculation is complicated, and requires loads to be issued to generate the address, software prefetches become even more suboptimal, as they result in memory stalls or extremely large code growth.

By considering these limitations, I developed a configurable graph prefetcher in hardware, targeted to optimise the performance of graph workloads. Graph workloads tend to fit into a small number of access patterns that can be targeted by special hardware aware of those patterns. This hardware can snoop the loads of the original computation to issue prefetches which are timely and correct, use analysis hardware to achieve good look-ahead distances, issue prefetches to dependent data structures when earlier prefetched data arrives, and change strategy based on dynamic conditions. This gains a significant performance improvement by being able to issue many more prefetches than a pure software scheme. It can both issue prefetches to more-deeply nested data structures, and also is able to issue prefetches to a data range without adding expensive control flow to the main computation, allowing us to remove prefetch instructions from the main core's instruction stream.

While the above is relatively lightweight in terms of hardware overhead, and can target a

set of interesting patterns, in general programmable hardware is necessary: memory-bound applications can include arbitrarily complex computation in their address generation, which cannot map to fixed-function hardware. By considering how the concepts of configurable prefetchers generalise, I developed an event-triggered programmable prefetcher: the ability to react to prefetches as they arrive with fixed behaviour generalises to programmable events, triggered on the arrival of loads and prefetches into the cache. As identified chapter 3 on software prefetching, issuing prefetches for these workloads requires the execution of a significant number of address-generating instructions, and thus a significant amount of compute power. The only way to achieve this in a low area and low energy way is by exploiting parallelism: events are naturally independent, so a good fit is to use a set of tiny micro-controller-sized cores. Still, while events are a fairly natural way of expressing dependent prefetches, it is somewhat fiddly to write the event kernels to run on the prefetch units. To ease the manual effort required to use the event-triggered programmable prefetcher, I developed a series of compiler passes, which progressively trade off performance for the amount of manual effort required.

In conclusion, a significant amount of performance is attainable for a set of memory-bound workloads which are increasingly important on modern systems, if we increase the capabilities of the prefetching system by giving it extra knowledge. In order of increasing hardware complexity, we have compiler-based software prefetching schemes, configurable fixed-function prefetchers targeting specific access patterns, and fully programmable event-triggered prefetchers, though even the latter can be achieved with less than 3% area overhead compared with a small out-of-order superscalar CPU. Which of these is most appropriate for a system is dependent on the amount of available silicon area and power budget, along with the volume of applicable workloads expected to be run on the system.

7.1 Review of the hypothesis

> *We can use algorithmic memory-access pattern knowledge to accurately identify and prefetch load addresses many cycles before they are required, and use this information in both hardware and software to improve performance for memory-bound workloads by reducing cache miss rates.*

From the results of this dissertation, we can see that this is definitively true for a number of common and important workloads [17, 18, 46, 56, 61, 72, 79, 83, 90, 91]. From the simplest in-order to the most complex out-of-order superscalar cores, we can use software prefetching to improve performance for workloads with indirection. Hardware support allows us to extend this further, both by making the same prefetching power cheaper, and by giving us the ability to generate more complex prefetches that are more latency tolerant and can react to arrival of other prefetches. Whether this information is

provided by the programmer or discovered at compile time, highly significant performance improvements can be realised for memory-bound workloads.

7.2 Future work

This dissertation demonstrates the core hypothesis presented: that we can improve performance by using prefetching coupled with application knowledge. However, almost every piece of work found within is capable of extension, either to expand the types of workload covered, improve performance, or achieve contrasting implementations.

7.2.1 Hardware prefetching for other access patterns

The majority of this dissertation has focused on workloads where a base array can be used to look ahead in a computation to find future latency-bound memory accesses. In section 3.1.1 I argue that this is because such workloads feature ample memory-level parallelism and are likely to miss with existing prefetchers. Still, there are many other access patterns that suffer misses in workloads, such as those using linked data structures, that cannot be accessed in this way.

To extend prefetching to these, we would need to both find a way of triggering the prefetcher to issue timely prefetches, and a way to extract memory-level parallelism. The former could be achieved by triggering on the program count of important instructions, instead of on array accesses. The latter is workload dependent, but may involve storing some history information [82] or prefetching more speculatively and less accurately than my current programmable and configurable schemes require.

Another change we might make is to support more complex events than the model described in section 5.4. This means that we can only access data from one cache line per event. This seems to work well in practice, but can't cover all possible memory accesses. We could perhaps sacrifice some degree of latency tolerance by allowing event kernels to prefetch multiple values at once, all associated with the same tag, which are then sent to a single new event kernel once loaded. Another way to achieve this might be to allow stalling on some loads, either relying on just the parallelism of having multiple PPUs instead of within-PPU parallelism.

The configurable prefetchers of chapter 4 could also be extended to cover other workloads. This would depend on which workloads were most important in terms of memory accesses, which may change over time.

7.2.2 Other types of programmable prefetcher

The model I implemented in chapter 5 exploits thread-level parallelism to achieve the required compute throughput for prefetch address calculation. It is likely that this is not the only way of achieving this: it may be equally possible to use data-level parallelism, since we are effectively performing the same operation for each offset from an array. We might, therefore, be able to use fewer PPUs, or perhaps only one, with a very wide SIMD unit, to similar effect.

7.2.3 Scheduling

The exponentially weighted moving average (EWMA) system I used in chapters 4 and 5 is just one method of dynamically setting look-ahead. Variants of, for example, best-offset prefetching [75], modified to support chains of loads rather than simple address-based sequences, may also be possible and may yield more accurate or more stable results.

7.2.4 Compiler-assisted event-kernel generation

As previously discussed in section 6.2.4, a limitation on compiler-assisted event-kernel generation is the mismatch in expressibility between a software prefetch and an event kernel, in that the latter can perform control flow and loops. It would be possible to get around this by providing some other sort of high level specification, easier to use than writing manual event kernels but more expressive than software prefetching.

Alternatively, it isn't a fundamental constraint that pragma generation (section 6.4) must use software prefetch instructions as an intermediate. Without this, a compiler technique may be able to more directly pick up control flow from the original program, and build this into events itself.

7.2.5 Compiler-assisted event-kernel generation without hints

Both the software-prefetch conversion and pragma-generation techniques presented in chapter 6 require some sort of programmer input. Since reconfiguration of the programmable prefetcher introduces overhead in the original program, it is only desirable to use the prefetcher for loops where it will improve performance. Still, in chapter 3, I generate software prefetches for all indirect accesses without any programmer input or significant negative impact, and so clearly it is possible to generate useful programmable prefetches in a similar way. One difference is that the programmable prefetcher should fetch all memory accesses for loops it is used on, for maximum performance, but perhaps this could be accounted for by targeting all memory accesses in any loop where the compiler finds some indirection.

7.2.6 Generation of event-kernels from binaries or at runtime

Needing user specification, or at least recompilation, for use of the techniques in this dissertation is a limitation when compared with a more traditional stride prefetcher [24], which can work on unmodified binaries. It might be possible to mitigate this by moving the analysis, either into a static or dynamic binary translator, or by performing it at runtime, perhaps by allowing the PPU cores to perform more complex analysis. It may even be possible to choose from a selection of common prefetch kernels at runtime using some form of machine learning.

7.2.7 Profile-guided optimisation

The choice of which prefetches are useful, which parts of the program it is particularly useful to prefetch, and what prefetching strategy to use, could be improved by providing some runtime information. This is true for both hardware and software prefetching. It could therefore be useful to allow some of this information to propagate back into the compiler from profiling. This could be used to, for example, set which loops have the prefetch pragma attached, give numbers for constants useful in prefetching but unknown at compile time such as expected number of loop iterations, or to set look-ahead distances.

7.2.8 Fetcher units

Prefetching is not the only way to improve performance for memory-bound workloads. As previously discussed in section 2.1.3.1, instead of speculating on the future memory accesses of the program, then repeating the loads once the program has reached them, we can instead perform the actual loads only once, in a latency tolerant way, then allow the processor to directly access this data. This imposes further constraints on the types of access we can perform, as we are no longer allowed to speculate, we need to be able to place a total ordering on the loads we perform, and we need to avoid hazards in read and write ordering. However, by reducing the need to perform both prefetch and load, we can halve the amount of address calculation and therefore potentially improve performance. It is possible that the programmable prefetching hardware could be extended in such a way, to allow either true prefetching, or one-time fetching, depending on program characteristics.

7.2.9 Further software prefetching

While this dissertation only looks at software prefetching for indirect memory accesses, as they were previously overlooked and yet the most obvious targets for performance improvements, other types of memory access can also miss. While, for example, linked data structure prefetching in software has been studied in the past (section 2.2.2.2), most

of the work in this area is sufficiently dated that the design of techniques for more modern processors are likely to show benefit.

Similarly, the C-like languages I look at in this dissertation are not the only performance-sensitive languages in use today. The access patterns in, for example, functional languages are sufficiently different from simple array-based accesses that specialised techniques are likely to be of use for improving performance.

7.2.10 Micro-controller-sized cores

It turns out the architectural techniques developed for the programmable prefetcher are quite general. Adding to the functionality of a large out-of-order superscalar core using an array of small cores, along with some supporting hardware logic, also allows many other use cases. In separate work, I have used a similar architecture to support hard and soft error detection of a main core [10], and also to create programmable security hardware, able to track and observe the main core's execution to prevent a variety of common attacks.

This is likely to be much more widely applicable: what can we do to improve the properties of code that runs on out-of-order superscalar cores in an era where parallel computation is essentially free? Even if the programs we run don't scale well across many cores, we may be able to use many cores to improve properties of a program. Just one example is prefetching to improve performance, but many others are likely to exist.

Bibliography

[1] ARM Cortex-A72 MPCore processor technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100095_0002_03_en/pat1406314252854.html, 2014. Cited on page 62.

[2] https://www.repository.cam.ac.uk/handle/1810/254642, 2016. Cited on page 70.

[3] Artifact evaluation for CGO 2017. http://ctuning.org/ae/cgo2017.html, 2017. Cited on page 18.

[4] https://www.repository.cam.ac.uk/handle/1810/261180, 2017. Cited on page 43.

[5] https://github.com/SamAinsworth/reproduce-cgo2017-paper, 2017. Cited on pages 33 and 43.

[6] https://doi.org/10.17863/CAM.17392, 2018. Cited on pages 101 and 127.

[7] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS, 2016. Cited on pages 19 and 55.

[8] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO, 2017. Cited on pages 20 and 31.

[9] Sam Ainsworth and Timothy M. Jones. An event-triggered programmable prefetcher for irregular workloads. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2018. Cited on pages 20 and 83.

[10] Sam Ainsworth and Timothy M. Jones. Parallel error detection using heterogeneous cores. In *Proceedings of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2018. Cited on page 136.

[11] D. Ajwani, U. Dementiev, R. Meyer, and V. Osipov. Breadth first search on massive graphs. In *9th DIMACS Implementation Challenge Workshop: Shortest Paths*, 2006. Cited on pages 15 and 56.

[12] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2003. Cited on page 25.

[13] AnandTech. http://www.anandtech.com/show/8542/cortexm7-launches-embedded-iot-and-wearables/2, 2014. Cited on pages 88 and 109.

[14] AnandTech. http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6, 2015. Cited on pages 88 and 109.

[15] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA, 2001. Cited on page 24.

[16] ARM. http://www.arm.com/products/processors/cortex-m/cortex-m0plus.php. Cited on pages 88 and 109.

[17] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006. Cited on pages 15, 32, and 132.

[18] 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 *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC, 1991. Cited on pages 17, 27, 32, 33, 39, 41, 42, 99, and 132.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39 (2), August 2011. Cited on pages 57, 68, 69, 99, 101, 121, 126, and 129.

[20] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2011. Cited on page 99.

[21] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001. Cited on page 55.

[22] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1991. Cited on pages 16, 28, and 89.

[23] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems*, 32(3), August 2007. Cited on page 27.

[24] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1992. Cited on pages 21, 89, and 135.

[25] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995. Cited on pages 21 and 98.

[26] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Transactions on Computer Systems*, 22 (2), May 2004. Cited on page 25.

[27] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA, 1998. Cited on pages 71 and 98.

[28] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2002. Cited on pages 25 and 32.

[29] Ben Coppin. *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., 2004. pp. 76–80. Cited on page 55.

[30] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computing in Science and Engineering*, 5(1), January 1998. Cited on page 113.

[31] P. Demosthenous, N. Nicolaou, and J. Georgiou. A hardware-efficient lowpass filter design for biomedical applications. In *Proceedings of the 2010 Biomedical Circuits and Systems Conference*, BioCAS, Nov 2010. Cited on page 66.

[32] Jonathan Eastep. Evolve: a preliminary multicore architecture for introspective computing. Master's thesis, MIT, 2007. https://dspace.mit.edu/handle/1721.1/40324. Cited on page 26.

[33] E. Ebrahimi, O. Mutlu, and Y.N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, HPCA, 2009. Cited on page 25.

[34] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2), April 1972. Cited on page 55.

[35] Babak Falsafi and Thomas F. Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1), 2014. Cited on pages 18, 23, and 67.

[36] Andrei Frumusanu. The ARM Cortex A73 – Artemis Unveiled. http://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/2, 2016. Cited on pages 46 and 70.

[37] Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. Loop-aware memory prefetching using code block working sets. In *Proceedings of the 47th International Symposium on Microarchitecture*, MICRO, 2014. Cited on page 25.

[38] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective Knowledge: towards R&D sustainability. In *Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition*, DATE, 2016. Cited on page 43.

[39] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2006. Cited on page 26.

[40] T.C. Grocutt, S. Ainsworth, and T.M. Jones. Event triggered programmable prefetcher, May 26 2017. URL http://www.google.com.pg/patents/WO2017085450A1?cl=en. WO Patent App. PCT/GB2016/053,216. Cited on page 18.

[41] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO, 2015. Cited on page 26.

[42] Milad Hashemi, Onur Mutlu, and Yale N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th International Symposium on Microarchitecture*, MICRO, 2016. Cited on page 24.

[43] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7), July 2000. Cited on page 27.

[44] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4), September 2006. Cited on page 27.

[45] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, 2015. Cited on page 26.

[46] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2012. Cited on pages 15 and 132.

[47] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO, 2013. Cited on pages 23 and 99.

[48] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. Fix the code. Don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization*, CGO, 2014. Cited on page 30.

[49] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA, 1997. Cited on pages 23, 26, and 67.

[50] Muneeb Khan and Erik Hagersten. Resource conscious prefetching for irregular applications in multicores. In *Proceedings of the 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS, 2014. Cited on page 28.

[51] Muneeb Khan, Michael A. Laurenzano, Jason Mars, Erik Hagersten, and David Black-Schaffer. AREP : Adaptive resource efficient prefetching for maximizing multicore performance. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, PACT, 2015. Cited on page 28.

[52] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Notices*, 37(10), October 2002. Cited on pages 29 and 89.

[53] Dongkeun Kim and Donald Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems*, 22(3), August 2004. Cited on page 29.

[54] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *Proceedings of the 49th International Symposium on Microarchitecture*, MICRO, 2016. Cited on page 21.

[55] Onur Kocberber, Babak Falsafi, Kevin Lim, Parthasarathy Ranganathan, and Stavros Harizopoulos. Dark silicon accelerators for database indexing. In *Proceedings of the 1st Dark Silicon Workshop*, DaSi, 2012. Cited on page 26.

[56] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO, 2013. Cited on pages 26, 32, 68, 83, 87, and 132.

[57] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2001. Cited on page 25.

[58] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. Compiler-based data prefetching and streaming non-temporal store generation for the Intel(R) Xeon Phi(TM) coprocessor. In *Proceedings of the 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum*, IPDPSW, 2013. Cited on page 28.

[59] Rakesh Krishnaiyer. Compiler prefetching for the Intel Xeon Phi co-processor. https://software.intel.com/sites/default/files/managed/54/77/5.3-prefetching-on-mic-update.pdf, 2012. Cited on page 28.

[60] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. SQRL: Hardware accelerator for collecting software data structures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT, 2014. Cited on page 26.

[61] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. DASX: Hardware accelerator for software data structures. In *Proceedings of the 29th*

ACM on International Conference on Supercomputing, ICS, 2015. Cited on pages 26 and 132.

[62] M. Kurant, A. Markopoulou, and P. Thiran. On the bias of bfs (breadth first search). In *Proceedings of the 22nd International Teletraffic Congress*, ITC, 2010. Cited on page 64.

[63] Shih-Chang Lai and Shih-Lien Lu. Hardware-based pointer data prefetcher. In *Proceedings of the 21st International Conference on Computer Design*, ICCD, 2003. Cited on page 23.

[64] N.B. Lakshminarayana and Hyesoon Kim. Spare register aware prefetching for graph algorithms on GPUs. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA, 2014. Cited on page 23.

[65] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO, 2004. Cited on pages 41, 126, and 129.

[66] Eric Lau, Jason E. Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar, 2011. Cited on page 26.

[67] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization*, 9(1), March 2012. Cited on pages 27, 29, 33, and 38.

[68] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2010. Cited on page 67.

[69] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. Cited on page 69.

[70] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th International Symposium on Microarchitecture*, MICRO, 1995. Cited on page 28.

[71] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS, 1996. Cited on page 28.

[72] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC, 2006. Cited on pages 42, 99, and 132.

[73] V. Malhotra and C. Kozyrakis. Library-based prefetching for pointer-intensive applications. Technical report, Computer Systems Laboratory, Stanford University, 2006. Cited on page 29.

[74] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. In *Proceedings of the 36th International Symposium on Microarchitecture*, MICRO, 2003. Cited on page 62.

[75] P. Michaud. Best-offset hardware prefetching. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, HPCA, 2016. Cited on pages 93 and 134.

[76] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*, ICS, 2001. Cited on page 24.

[77] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Systems Laboratory, 1994. Cited on pages 27 and 28.

[78] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1992. Cited on pages 28, 33, 38, and 48.

[79] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, May 5, 2010. Cited on pages 15, 32, 43, 55, 57, 69, 99, and 132.

[80] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, HPCA, 2003. Cited on page 24.

[81] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th International Symposium on Microarchitecture*, MICRO, 2005. Cited on page 24.

[82] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA, 2004. Cited on pages 23, 98, 99, 103, and 133.

[83] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. Prefedge: SSD prefetcher for large-scale graph traversal. In *Proceedings of International Conference on Systems and Storage*, SYSTOR, 2014. Cited on pages 15, 30, 32, 55, 56, 80, and 132.

[84] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Cited on page 67.

[85] V. M. Panait, Amit Sasturkar, and W. F. Wong. Static identification of delinquent loads. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO, 2004. Cited on page 52.

[86] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, 2015. Cited on page 25.

[87] Dana Ron. Algorithmic and analysis techniques in property testing. *Foundations and Trends ® in Theoretical Computer Science*, 5, 2009. Cited on page 55.

[88] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1998. Cited on page 24.

[89] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO, 2015. Cited on page 21.

[90] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Cited on pages 55, 69, 99, 100, and 132.

[91] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE, 2013. Cited on pages 42 and 132.

[92] S.P. VanderWiel and D.J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, 1999. Cited on page 26.

[93] Vish 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, September 2014. Cited on pages 21, 58, 62, and 89.

[94] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA, 2005. Cited on pages 23 and 99.

[95] Youfeng Wu, Mauricio J. Serrano, Rakesh Krishnaiyer, Wei Li, and Jesse Fang. Value-profile guided stride prefetching for irregular code. In *Proceedings of the 11th International Conference on Compiler Construction*, CC, 2002. Cited on page 28.

[96] Chia-Lin Yang and Alvin R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing*, ICS, 2000. Cited on page 62.

[97] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO, 2015. Cited on pages 23 and 70.

[98] Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn. An automated method for software controlled cache prefetching. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, HICSS, 1998. Cited on page 28.