UNIVERSITY OF CALIFORNIA, SAN DIEGO

Memory Prefetching for the GreenDroid Microprocessor

A Project submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science and Engineering

by

David Herrick Curran

Project Advisor:

Professor Michael B Taylor

2012

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT OF THE PROJECT

Memory Prefetching for the GreenDroid Microprocessor

by

David Herrick Curran

Master of Science in Computer Science and Engineering

University of California, San Diego, 2012

Professor Michael B Taylor

This project explores memory prefetching in the context of UCSD's *GreenDroid tiled microprocessor*. It details the process by which a simple, removable stride-based prefetcher was developed and simulated on GreenDroid's cycle accurate simulator, and it discusses the implementability of such a prefetcher on an actual version of the GreenDroid chip. The simulation results illustrate that this prefetcher can achieve buffer hit ratios of nearly 100% during the execution of certain commonly used algorithms, and that it can be minimally intrusive when its hit ratios are low.

Chapter 1

Introduction

This project sets out to develop and simulate a simple stride-based prefetcher for the GreenDroid microprocessor. This prefetcher is designed to be low in complexity, have a low footprint, and be easily added to the GreenDroid system without major overhauls of its other systems.

Moore's Law illustrates a now well-understood trend in computing: the density with which transistors can be put onto a piece of silicon has been doubling roughly every year and a half for the last fifty years [33]. For some time, as the physical footprint of the transistor had grown exponentially smaller, we had seen an exponential increase in the speed at which microprocessors could be clocked. This trend lasted until about 2005 when the microprocessor industry hit the infamous *power wall* [7, 15, 21, 22, 43]. With the transistor's minimal surface area decreasing exponentially quickly and its power requirements decreasing only quadratically, its power dissipation per surface area requirements had grown exponentially. The microprocessor industry hit the power wall when microprocessors became more limited by their power dissipation requirements than by the speed of their individual transistors.

The power wall gave rise to an age of increasingly power efficient hardware [22, 28], parallel processors [6, 13, 18, 19, 22, 26, 32, 35, 36, 41, 42, 44, 45], and increasing amounts of *dark silicon* [12, 19, 20]. There are reasonable limits to the rate at which a given area on a chip can be cooled; a chip's power budget is determined largely by its overall surface area. Dark silicon comprises the parts of a microprocessor which rarely perform power-consuming operations. If parts of a chip are "darkened" (covered with useful low-power hardware or hardware that is rarely used), performance can be increased without imposing a heavy burden on the chip's power budget. Furthermore, areas of dark silicon can be used to dissipate heat generated by their more active neighboring regions. Most dark silicon comes in the form of large on-chip caches [32, 41]. However, the effect of a cache size increase on overall processor performance tends to diminish as the cache grows in size; thus, as the amount of dark silicon required to keep a chip running within its power budget has increased (as an effect of Moore's Law), novel devices such as *conservation cores* or *c-cores* [46] (see Section 2.1.1 on c-cores) and *unconventional cores* or *U-cores* [9] have begun to spring up as potential utilizers of this expanding dark silicon real estate. These sorts of technologies incorporate specialized processing cores alongside the general-purpose processor which are capable of performing common computational tasks more energy-efficiently than the general-purpose processor. Since these specialized processors are only used during particular parts of a computation, they do not often contribute to the overall power budget of the chip.

Another effect of Moore's Law has been a gap in performance which has sprung up between mathematical processing speed and large memory access speed. As processors have gotten faster and memories (as well as applications' memory requirements) have gotten larger, reducing the effects of memory latency has become an issue of paramount importance in increasing overall processing performance for common com-

putational tasks. In 2005, when the power wall began to slow the rate of serial processing speed increases, parallel execution secured a place at the forefront of microprocessor research. With increasing levels of parallelism in execution, memory contention has continued to push forth the margin between processor and memory performance [38]. Memory latency is one of the most significant contributers to computation time in today's computing systems. While many mathematical and control operations can be performed by a processor within a few cycles, the time required to access the data being processed can be on the order of 100 cycles or more.

Reducing the effects of memory latency on computation time is and has been a very active area of research. For niche computational tasks, a myriad of techniques have emerged; these include streaming memory systems with streaming processors [13, 35, 36, 42] and barrel systems [5]. These systems can be very effective within their problem domains, but are often lacking in general-purpose computation. For general-purpose computing, the primary solution is a combination of caching and prefetching. A passive caching system keeps data close the processor once it has been requested, expecting that it will be requested again shortly. A prefetcher, by contrast, works to request data that the processor will need before the processor itself ever calls for it.

This project discusses memory prefetchers and their applicability to UCSD's *GreenDroid* [19, 44] *tiled microprocessor* [45]; it details the conception of a prefetcher for the memory system of GreenDroid. The project begins with some background on prefetchers and the GreenDroid processor. It then goes on to discuss the problem of designing a prefetcher for GreenDroid, the goals of such a design, and how such goals relate to the general field of prefetcher design. Next, it discusses the design process of ModeFetch itself, illustrating the design trade-offs by taking the reader through the evolution of the design. Finally, it outlines the simulation of a prefetcher prototype on GreenDroid's cycle-accurate simulator. In simulation, use of the prefetcher lead to

significant improvements in the simulated c-core calculation time as well as the overall calculation time of the simulation. The project concludes by discussing the results of the simulation, and making suggestions for future work.

1.1 GreenDroid

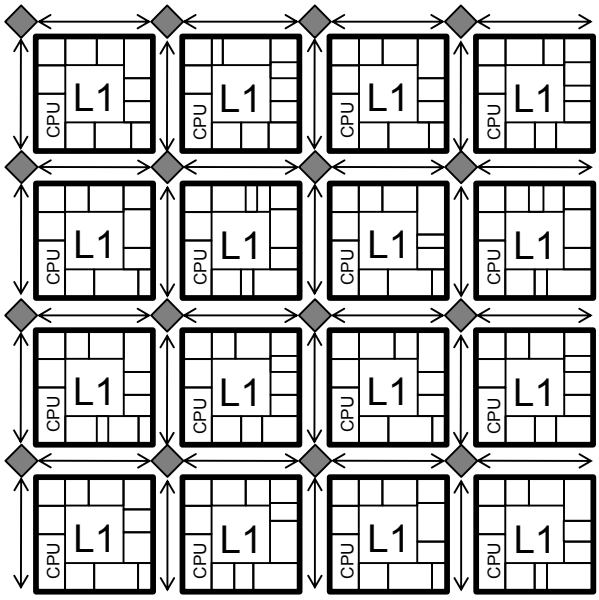

Figure 1.1: An illustration of the layout of the tiles in the GreenDroid microprocessor. [46]

GreenDroid is a high-performance, low-power tiled multiprocessor (see Figure 1.1). Its design is based on the idea that certain computational tasks are performed in many situations. GreenDroid leverages this notion by incorporating a series of c-cores into each of its tiles. Whenever an application seeks to perform a popular task for which there exists a c-core, control is transferred from the general-purpose processor over to that c-core; a special compiler automatically adds the control transfer instructions into the code of GreenDroid's applications at compile-time. Each of GreenDroid's tiles comprises a processing core which has its own cache as well as its own processing

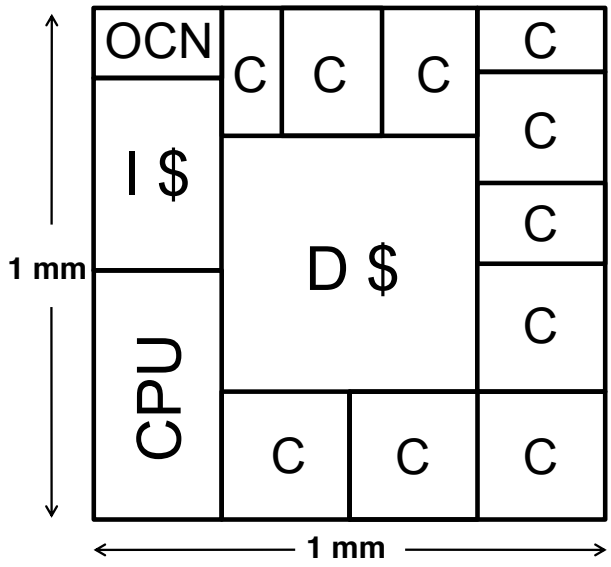

Figure 1.2: An illustration of the layout of an individual tile in the GreenDroid microprocessor. (CPU := general-purpose processor, C := c-core, D$:= d-cache, I$:= i-cache, OCN := on chip network) [46]

elements. Tiles house three primary elements: a general-purpose processor with its own L1 i-cache, a set of c-cores, and an L1 d-cache, which is shared by the general-purpose processor and the c-cores on that tile (see Figure 1.2). C-cores are specialized processing elements which are designed to perform one job very efficiently. Because each c-core is designed to perform just one particular computational task, it can perform that task more efficiently than a general-purpose processor: it does not incur overhead from a register file, an i-cache, or much of the other hardware necessary to a general-purpose processor. GreenDroid saves energy by using c-cores to perform popular computations.

GreenDroid represents a dynamic class of processor designs in the sense that its instances are built around the applications they are expected to run. GreenDroid utilizes a special compiler to generate programs that can take advantage of c-cores; that same compiler system is responsible for generating the c-cores that will be included in a particular GreenDroid release. This means that a set of GreenDroid releases, while working within the same framework, may have significant variation with regards to their

c-cores, based on each release's intended use.

GreenDroid's tiles communicate through a set of mesh networks. These networks are decentralized in an effort to support scalability with regards to tile numbers on the chip; while resource contention does not scale well (including contention for memory resources), direct communication between nearby tiles does benefit from the decentralization of the on-chip networks. Each tile is connected directly to its neighbors through three networks: the *Static Network*, the *General Dynamic Network*, or *GDN*, and the *Memory Dynamic Network*, or *MDN*. The first two of these networks are used for tile-tile and tile-device communication, and the third is used only for processor-memory communication. The MDN is required to ensure provably deadlock-free access to the DRAM, which can be used as an overflow buffer for the other networks, alleviating any deadlock that may occur in them. The GDN provides general routed communication between any two processors on the chip. The static network provides efficient direct communication between neighboring tiles. The dynamic networks allow for the routing of packets between non-neighboring tiles and between tiles and the sides of the chip. The memory controller(s) lie on the sides of the chip. Memory communication is bottlenecked, since the area of the processor (and associated number of tiles) grows more quickly than the perimeter (O(<side length>^{2}) vs. O(<side length>)). This means that packets sent by multiple tiles over the dynamic on-chip networks will begin to pile up as they make their way toward the limited number of processor-side DRAM ports. This effect worsens as the number of tiles on the chip is increased.

1.2 Memory System Scaling

As computers have grown faster and more capable, they have required larger memories. Most computing systems incorporate a memory hierarchy stretching from

small, fast, expensive memory through several levels to large, slow, cheap memory. Modern memory systems often use caching to store local copies of data for which there is believed to be an eminent need closer to the processor; this allows that data to be accessed more quickly when needed. Generally the processor must request a piece of data a first time before it is cached for subsequent accesses. Prefetching is a process which involves the anticipation of the processor's needs. By properly modeling the processor's memory request patterns, a prefetcher may use observed or otherwise understood trends to speculate about future memory requests that the processor will make. It may then prefetch the expected data before it is ever explicitly requested by the processor. The data that is prefetched may be stored in the prefetcher's own limited size buffer near the processor, or it may be sent directly to some level of the processor's cache.

Other systems for dealing with the high data fetching latencies associated with large memories include streaming processors with streaming memories [35], which are used for computational work that can be performed on vector data, and barrel processors [5], which mask memory latency by interlacing the execution of many threads together, efficiently rotating between them so that each thread is only active in the processor while its data are available. Both of these latency minimizing/masking methods impose requirements on the particular type of processing being performed: although barrel processors can be well leveraged to effect efficient throughput computing [30] on applications with many independent threads of execution, they are not designed to reduce latency in computing systems which require real-time operation of each thread. Streaming and vector systems are similarly limited to applications involving the standardized processing of series of related data such as SIMD (single-instruction-multiple-data) tasks [14]; although streaming systems can be highly efficient when executing programs from within their domain, they lose their ability to hide large memory latency in applications with irregular memory access patterns.

Caching and prefetching together comprise an often imperfect, but particularly versatile system; this system can be used to reduce latency on some level in nearly any computing task. Caching alone can be very effective at retaining small sets of often-used data in fast memory. As a particular application begins to request data from larger sets, each piece with less frequency, it becomes more difficult for a small cache to keep an effective subset of that data near the processor, and the benefits of having much of that rarely accessed data near the processor diminish. In these scenarios, the prefetcher becomes a more and more important part of the memory system.

Prefetching for complex memory accesses patterns is an open area of research [24]. Most prefetchers base their predictions on observations of the processor's request stream. As the ordering of addresses in this stream becomes more complex, their ability to make predictions tends to wane. This project focuses on a low-overhead, general-purpose prefetcher, intending to make improvements in compulsory cache miss times for the general case without imposing a significant burden on the processor's real estate or power budgets.

1.3 Prefetching for GreenDroid

Unlike other tiled processors such as *RAW* [45] or Tilera's Tile64 [6], GreenDroid utilizes much of its tile space to house c-cores. It keeps only a small L1 cache on each tile, rather than using its dark silicon to accommodate an extra L2 cache. This allows each c-core to share some L1 cache with the general-purpose processor on its tile.

GreenDroid's 16 tiles are heterogeneous, each housing a different selection of c-cores. This heterogeneity is necessary to keep each c-core close to a general-purpose processor and an L1 cache which is itself close to the same general-purpose processor.

So although GreenDroid has 16 tiles, all of those tiles are generally not expected to be used at the same time; rather, each is expected to be used primarily when the current code being executed can make use of the c-cores on that tile. Thus, while the tiled layout of GreenDroid is based off of the same ideas as some other massively parallel architectures (such as RAW), GreenDroid itself is more of a serial or slightly parallel processor which can execute transport its code execution over a series of physical regions.

GreenDroid is not designed specifically for the processing of streaming data; however, depending on the intended use and selection of c-cores for a specific chip design, GreenDroid may find the ability to stream data from DRAM useful. Streaming can be accomplished by using a set of several memory controllers at the side of the chip and requesting data from them in series. This sort of layout could complicate the design of a prefetcher for GreenDroid. Thus, while this prefetcher is designed primarily for a non-streaming chip, care has been taken to ensure that it could be capable of accommodating a streaming setup as well.

1.4 Project Organization

In Chapter 1, the project exposes the reader to the primary problem being addressed and outlines the solution developed. The GreenDroid processor is discussed in detail in Chapter 2; its strengths and limitations are detailed along with the effects they have on the design of a prefetcher for its memory system. The specific problem of designing a prefetcher for GreenDroid is refined in Chapter 3 with a detailing of the prefetcher's goals and constraints. The project illustrates, in Chapter 4, the design process for the prefetcher, outlining decision points, discussing the options available at each point, and arguing for the selections made. Related work is discussed in Chapter 5 with a primary focus on other prefetcher designs which are based on related concepts. The

simulation environment, decisions made regarding simulation, process of simulator development, and simulation results are then discussed in Chapter 6. Chapter 7 concludes the project, evaluating the simulation results and making suggestions for future work.

Chapter 2

GreenDroid

In this section, we discuss GreenDroid in detail, illustrating the features from which the prefetcher benefits most as well as the constraints that GreenDroid places on the prefetcher.

2.1 Architecture Overview

GreenDroid is a tiled microprocessor in development at UCSD. It is designed to run applications built for Google's Android platform. Using c-cores, GreenDroid reduces power requirements for popular computations made by its applications. This method is particularly effective in reducing the power required to run the Android operating system code and common library code since Android applications make extensive use of such code. Android is a good target for GreenDroid in part because it makes extensive use of virtual machines (VMs) and interpreters. This means that by optimizing the execution of segments of the VM and interpreter code using c-cores, GreenDroid can optimize the execution of a myriad of popular Android programs. Moreover, since

for each Android device, there exists a particular set of pre-installed and otherwise popular applications, c-cores can be used to optimize for the needs of the particular popular application set of each device; the GreenDroid manufacturing system can generate a different set of c-cores for each potential application set. Finally, Android makes a good target for the low-power GreenDroid processor because it tends to be used in portable devices with limited battery capacity.

Although GreenDroid is equipped with 16 processing tiles, it is designed largely as a serial processor. GreenDroid's tiled architecture allows it to keep a general-purpose processor and an L1 d-cache near each of its c-cores. Without the tiled nature of the chip, either not all c-cores would be able to be afforded the same level of proximity to an L1 d-cache or a general-purpose processor. By migrating around the processor from tile to tile, a process can have access to any c-core it requires, and can jump between that c-core and a general-purpose processor as needed, using the shared L1 cache to communicate between the c-cores and the general-purpose processor on the tile.

2.1.1 Dark Silicon and C-Cores

GreenDroid's c-cores are a direct response to the power wall that the processor design industry has been facing since 2005. In order to cool down a chip, the chip must either run slowly or part of it must not run at all. As processing techniques have improved, it has been possible to put more logic in the same physical area of a chip. This logic heats the chip up when used. As logic has gotten exponentially smaller, power output per unit area has gotten exponentially larger. The traditional response to this issue has been to replace parts of chips that could hold processing cores with caches of increasing size, because caches can increase the speed of many tasks without expelling very much heat; caches are a form of dark silicon (an area of a processor

which is not generally active). GreenDroid takes a different approach: c-cores. C-cores are inherently dark during much of the active life of a chip, since they can only be used when their particular task is being performed. At any given moment during a particular thread's execution, a majority of the processing elements on GreenDroid are not using any of the chip's power budget, as control for the thread lies in only one general-purpose processing core or c-core. Because much of GreenDroid's area is covered with c-cores or routinely dormant general-purpose processors, GreenDroid is capable of being run at high speed without overheating.

GreenDroid is concerned with energy consumption rates not only because of heat issues, but also because of its goal of working as a mobile processor. In mobile applications, battery life and device heat dissipation require more strict power limitations than those imposed upon 125+ Watt desktop and server CPUs. By using c-cores, GreenDroid is able to adhere to these extra-restrictive needs without sacrificing speed. Aside from helping with heat dissipation by lying dormant at most times, c-cores can also help to reduce the total amount of energy required to perform a particular computation. A c-core can execute its particular task more power-efficiently than a general-purpose processor. This efficiency comes from the reduced overhead that c-cores have compared to general-purpose processors for performing the same tasks: C-cores do not require elements such as register files, instructions, and i-caches because their functionality is hardwired into their structure. The lower energy requirements of c-cores can help GreenDroid to be not only power-efficient, but also battery-friendly.

GreenDroid is a flexible architecture, which may incorporate c-cores for a variety of different applications. C-cores are generated dynamically for each GreenDroid release. Predicting the layout of a particular GreenDroid release is difficult, since the c-cores are developed on a per application basis. GreenDroid's c-cores are generated by a specialized compiler system, which is also used to generate programs that take ad-

vantage of the included c-cores. This compiler inspects programs being compiled for GreenDroid to find areas of their computation that could benefit from the use of some c-core. When an appropriate area or set of areas is found, the compiler generates an applicable c-core, and then replaces the appropriate code sections of the application with calls to transfer control to that c-core. During runtime, the execution of a program moves around the processor between general-purpose cores and c-cores, attempting to utilize as much specialized hardware as possible in order to save on energy expenditures. GreenDroid's compiler is well aware of the "types" of computations that each release is expected to perform, and sets up the hardware/software system to make the expected computations occur as power-efficiently and energy-efficiently as it can.

2.1.2 Communication

Communication on GreenDroid is accomplished by using a set of mesh networks: the static network, the GDN, and the MDN. These networks comprise a set of decentralized communication mechanisms for tile-to-tile message passing, as well as tile-to-main-memory communication. The static network is used for extra efficient communication between neighboring tiles, requiring no dynamic packet routing information. In order to communicate between tiles which are not adjacent, the GDN is used. The GDN, while effectively more capable, is slightly less efficient than the static network in that values transported over it are wrapped within packets complete with headers describing their source and destination addresses.

All external devices communicate with GreenDroid via interfaces at the chip's side. Interfaces to main memory work in this way. Memory request messages are passed along the MDN, relayed from tile to tile, beginning at tiles which need to access memory until they reach the side of the processor, at which point they are taken off of the

chip by the appropriate DRAM interface. The response data is delivered via the same mechanism, from the side of the processor, by relay, back to the original requesting tile. While each tile has an independent (but still coherent) caching system, main memory is shard between the tiles of the processor. There may be multiple memory controllers at the side of the processor, but for any particular physical memory address, all of the tiles must use the same memory controller.

2.2 Memory Hierarchy

GreenDroid utilizes a relatively standard memory hierarchy for a multiprocessor. At the top level, each tile's general-purpose processor has its own register file, the contents of which are governed by the program being executed on that processor. Next, each tile has an L1 cache; only data from those addresses fetched by the tile in question is stored in this cache. GreenDroid employs a cache coherence system for its network of 16 L1 caches. Behind these caches is the DRAM which may be controlled by one or more off-chip DRAM controllers. [39]

2.2.1 Caching System

GreenDroid performs memory caching on a per-tile basis. Each tile has an L1 i-cache and an L1 d-cache. The d-cache is shared by the general-purpose processor and the c-cores of the each tile. The i-cache is only accessible to the general-purpose processor. C-cores have no need for an i-cache since their instructions are encoded directly into their hardware and need not be fetched from main memory. This system expedites transfer of control between the c-cores and the general-purpose processor by allowing them to access overlapping data quickly.

2.2.2 Main Memory Access System

When a GreenDroid tile wishes to make a request to main memory, it does so through the on-chip memory dynamic network, or MDN. First, a memory request packet is generated, and then that packet is routed over the mesh network from tile to tile until it arrives at the side of the chip. The packet is then offloaded from the chip and received by a memory controller appropriate to the address being fetched. The memory controller responds to the request with another packet, this one addressed to the requesting tile. That packet is routed from the DRAM controller, back over the chip to the original requesting tile.

While memory controllers are attached to GreenDroid in a similar location and manner as other I/O devices which use the GDN for communication with on-chip tiles, they do require use of the MDN. Unlike the GDN, the MDN is restricted only to memory request traffic; it exists in order to provide provably deadlock-free access to the DRAM; this allows the DRAM to be used as an overflow buffer for the GDN so that it can be rescued from any deadlock conditions which could arise.

Since the number of tiles on the chip is quadratically related to the number of tiles on the perimeter of the chip, off-chip I/O including memory access is the most bottlenecked function of GreenDroid. The maximum number of memory controllers is limited by chip perimeter. Many controllers may be put on the side of the chip with parallel access available due to the nature of the mesh network. However, in a typical case, no more than a few memory controllers on one chip are expected.

2.2.3 Known Potential Modifications

Unused spaces in each tile's L1 cache may be used by adjacent tiles as victim buffers or extended L2 caches. This is a modification that has been considered by other

teams during the time of this project writing, and it will affect the future work section of this project. [39] This modification may be efficiently implemented due to GreenDroid's efficient neighboring tile communication mechanism (namely, the static network).

Chapter 3

Problem Description

This chapter discusses the problem of designing a prefetcher for GreenDroid in detail. It describes the challenges a prefetcher faces and the advantages it sees working within the GreenDroid environment and the expectations of performance from such a prefetcher.

3.1 Memory Prefetchers

Memory prefetchers are designed to help minimize memory latency in computing systems. Memory latency is caused by circuit and physical slowdowns found between processing elements and main system memory. Latency is one of the key factors affecting performance in most general-purpose computer architectures. Since memory latency tends to increase with memory size (and associated complexity), the latency problem is commonly dealt with by imposing a strict memory hierarchy. This hierarchy allows the system to utilize small, low latency memory for storing data that are frequently accessed, while retaining the ability to store large amounts of data which are more rarely used in larger sets of slower memory.

Most prefetchers attempt to monitor the flow of memory requests, seeking to extrapolate patterns. When a pattern is identified, the prefetcher uses it to predict the location of future, near-term memory requests. The prefetcher may then issue a request on behalf of the processing hardware. The goal is that this piece of data will be requested by the processor shortly and may be accessed with lower latency because the initial request to main memory came before the data was needed. This reduces the number of cycles during which the thread in execution will have to wait for the data to arrive, speeding up the overall computation.

Since channels to main memory can be a valued commodity in the memory system, there is a cost to the issuance of a prefetch request. The prefetcher only considers issuing requests that it has determined are likely to be used by the processor soon. The question of cost comes down to the resultant memory contention increase that the prefetch request may cause. Also, in low-power architectures, the energy required for the issuance of extraneous, unnecessary memory requests can be detrimental. For these reasons, the prefetcher must be judicial with its determination of the cost-to-benefit ratio of issuing each request.

3.2 Expected Work Load

In order to determine the work load that a GreenDroid prefetcher is likely to encounter, it is useful to consider the domain of possible applications that may be run on the processor. This is a difficult task, since GreenDroid is a general-purpose processor. However, certain assumptions can be made based on the standard libraries and system code of which Android applications tend make heavy use. One of the motivations for the production of a c-core processor for the Android operating system is that Android applications tend to rely heavily on standard libraries and the operating system itself.

The specific c-core set selected for a GreenDroid release can be used to define a chunk of the application domain for the processor.

3.3 Design Goals

Since the prefetcher of this project is being designed to work with GreenDroid, its primary goals are power efficiency, accuracy, and simplicity.

Accuracy, on the part of the prediction logic, is a key attribute to improving processor performance. It is also key to prefetcher power efficiency. Reducing the number of extraneous fetch requests made to the DRAM can improve power efficiency, since it will in turn reduce the burden of extraneous work laid upon the DRAM. Simplicity is important, both for power efficiency, and for the reduction of the prefetcher's footprint. The prefetcher should not take up more space in GreenDroid's space-limited environment than necessary.

3.4 Expectations of Performance

This section focuses on the theoretical potential for performance increases that any GreenDroid prefetcher could effect. This potential is based on the benefits and limitations placed on a prefetcher by the GreenDroid environment.

3.4.1 DRAM Configuration

DDR2 DRAM is divided into a set of banks, which are memory partitions that are capable of acting independently in parallel. Within a bank is a series of rows, each of which is divided into columns. A cache line's address is defined by a particular bank,

row, and column. In order to perform a read operation for a particular cache line, the appropriate bank must first ensure that the appropriate row is opened. If it is not opened, the bank must first close another row if it is opened, and then open the new row. The row closing, or *precharge*, operation returns the bank to a neutral state. The opening of the new row is called a Row-Address-Select operation (RAS). Once the appropriate row is opened, the bank may accept Column-Address-Select (CAS) commands to read cache lines from the opened row.

The amount of time required to service a particular request is known as the request latency and it depends on several factors. Firstly, the physical makeup of the DRAM defines certain RAS and CAS latency values. Secondly, the location of the requested address and the current state of the DRAM's banks determines whether a bank will have to be precharged and whether a new row will have to be activated. While the precharge operation does take some time, it is usually much less time than an RAS or CAS operation. The primary factors in determining the latency are the RAS and CAS latencies and whether or not an RAS must be performed.

The second concern with regards to memory request speed is throughput. In a *pipelined* DRAM, multiple requests may be issued before a single request has been serviced. While the time taken to service each individual request will not be effected by this pipelining, the total number of requests serviced in a given period of time may be increased. The maximum throughput in a DRAM is limited by the maximum rate at which the DRAM can accept requests. In an SDRAM this is determined by the clock frequency. If a stream of proximal cache line requests is fed to a DRAM controller, each address may be able to be serviced by a different bank. This means that the CAS latencies of the banks will overlap, and the throughput will increase. Furthermore, the RAS latency can be avoided for most operations since the same row will be accessed over and over within each bank until all of the cache lines in that row have been accessed

and a new row must be opened.

GreenDroid's simulator incorporates a DRAM simulation module, which accepts one memory request address at a time. It waits a predefined number of cycles (in order to simulate latency). Once the latency has been simulated it proceeds to accept another request address (and respond with a value if the original request was a read). This differs from the DDR2 DRAM capabilities discussed above in that it cannot service multiple outstanding requests by pipelining through the leveraging of different banks.

A prefetcher's potential effectiveness is dependent on the memory system for which it prefetches. A pipelined DRAM would be able to handle multiple outstanding cache line requests at once. By contrast, GreenDroid's DRAM simulation module is able to handle only one outstanding request at a time. This distinction has an effect on the design and potential performance of the prefetcher. In the simulated module, the memory system has to wait for an outstanding prefetcher request to be serviced before it is capable of servicing a processor request. For this reason, an inaccurate prefetcher being used with such a DRAM can be a significant drain on the system: if the processor's DRAM requests routinely run into frivolous outstanding prefetcher requests, the effective DRAM latency may be increased by as much as a 2:1 ratio.

In the case of a pipelined DRAM, the amount of time required to wait between the servicing of two concurrent requests may be significantly less than the average latency of the DRAM; it will instead depend on the clock frequency of the DRAM. In this case, a processor request may be made even with one or more outstanding prefetcher or processor requests already in the pipeline. For this reason, in the simulation environment, a prefetcher must be more diligent about making only promising requests than a prefetcher being used with a pipelined DRAM.

The prefetcher's performance will be heavily influenced by the constraints on

the memory system. Firstly, the memory bandwidth to latency ratio will bound the maximum performance of any prefetcher on any system. Where the primary limiting factor in memory access speed is latency as opposed to throughput (for instance, in the case of the simulator, where only one piece of data may be fetched at a time, and sequential data may not be streamed out of multiple banks efficiently), the system is faced with a 2:1 maximum possible latency reduction per DRAM controller, as illustrated in the following analysis:

In the best case, a prefetcher can anticipate all memory accesses ahead of time; if this occurs, there are three cases for what could happen to the processing speed per code segment:

1. The processor had been spending most of its time waiting for cache misses. In this case, the processor would quickly catch up to the prefetcher, and end up waiting for cache misses, even if they had been requested by the prefetcher ahead of time. Thus, in short order, the processor would reach steady-state, spending about the same amount of time waiting for cache misses, since the prefetcher would only be able to make requests at about the same rate as the processor.

2. The processor had been spending most of its time performing processing, rarely missing in the cache. In this case, since the processor had been spending less than half of its time waiting for cache misses, even bypassing all of that time would not reduce total running time by more than half. The maximum performance gain in this case would be governed by Amdahl's Law [2], where the portion of time eligible for speedup were less than half.

3. The processor had been spending half of its time waiting for cache misses. In this case, the processor would end up finishing its processing time between cache misses just as the next would-be cache miss arrived in the cache or prefetch buffer,

at the prefetcher's request. At this point the processor would be able to use the value requested and continue processing while the next value were prefetched. This, the best case for prefetcher effectiveness, would result in a 2:1 speedup, since the 50% processing time would take 100% of its original time, and the 50% cache miss time would take 0% of its original time.

In any type of DRAM, the ordering of requested addresses can be important. When a set of proximal addresses are fetched in series in ascending order, the DRAM may be able to respond more quickly by using multiple banks in parallel. This can allow the DRAM to avoid the extra RAS latency involved with less contiguous patterns. Because the fastest series of addresses to fetch are often contiguous in memory, *stride* [16], *stream* [23, 25, 37], and *scheduled region* [29] prefetchers can be particularly effective as they tend to issue prefetch request streams pertaining to largely contiguous data.

Only by stacking multiple memory requests on top of each other in a pipeline could a memory system begin to see greater than 2:1 speedups per DRAM controller due to a prefetcher. In this case, a prefetcher could be much more effective at leveraging the advantages of the memory pipeline than a single processing core. A processing core executing a single thread will generally stall while waiting for each cache miss to be serviced in the memory. This means that only one request from the given thread will be put through the pipeline at a time. The reason the processor stalls is that the rest of the thread's execution and the decisions of the processing core will likely depend (at least in part) on the value of the requested memory address. Conversely, a prefetcher's decisions may be largely or completely independent of the value of the data returned from the memory. Thus, the prefetcher may be able to issue multiple requests in short order without any regards for the latency of each request.

The nature of the simulation environment to accept only one outstanding mem-

ory request at a time could have had some bearing on the prefetcher design. In the benchmark situations, the amount of time taken to calculate and request a new address was sufficient to stay ahead of the processor for nearly all of the cases encountered during simulation. The prefetcher seemed able to make its high-value requests as needed with the unpipelined DRAM. Furthermore, the processor spent very little time waiting for the prefetcher, even in the worst cases, indicating that contention was not a big issue. Thus, it seems likely that the prefetcher, as simulated, would have benefited little from a pipelined DRAM. However, perhaps had a pipelined DRAM been used, the prefetcher's aggressiveness could have been higher, and it could have been allowed to make more lower-confidence requests, resulting in higher hit ratios (see 4.4 for a discussion of aggressiveness).

3.5 Prefetchers in Developing DRAM Configurations

If we attempt to look into the future and make some predictions as to how the need for prefetchers will develop by forecasting the development of DRAM, we look at the upcoming DDR4 spec for clues. Similarly to the DDR2 to DDR3 transition, we see an increase in bandwidth (by increasing the clock frequency from 1066MHz to as much as 2133MHz) and a decrease in power consumption by a lowering of the operating voltage (from 1.2-1.65V to 1.05-1.2V). However, if the DDR2 to DDR3 transition is an indicator, we can expect the low-voltage DDR4 DRAM to have higher latencies than the higher-voltage DDR3. With more banks (16 banks per chip for up to eight chips per package allowing for an effective 128 independent banks in DDR4), however, the RAS latency should be encountered less often. The higher bandwidth and the higher number of banks should go to increase the possible throughput for consecutive address streams dramatically.

Since prefetchers are primarily concerned with hiding the effects of latency and can potentially benefit from increased bandwidth, it appears that there will be the same or more need for them in the near future. The DDR3 to DDR4 transition may effect the decisions of a low-power prefetcher such as the one developed in this project: because DDR4 is designed to reduce power consumption of memory operations, low-power prefetchers may be able to be more liberal with their prefetch issuance without having as much of an effect on the overall system power budget.

Chapter 4

Design

This chapter details the design of the prefetcher, walking the reader through decision points in order to illustrate the reasoning behind each decision. It also details the final design of the prefetcher simulated and argues the implementability of the final design.

4.1 Placement

The initial phases of the design were focused on determining the best logical location for the prefetcher and its prefetch buffer. The outcome of this decision would have effects on the size and arrangement of the prefetcher buffer, the maximum rate of prefetcher request issuance, and the physical size and complexity of the prefetcher decision logic itself. Factors considered in determining this placement included the size of the prefetcher buffer, the tile to prefetcher buffer latency, and the prefetcher's effect on the on-chip mesh network.

4.1.1 At the Tile Cache

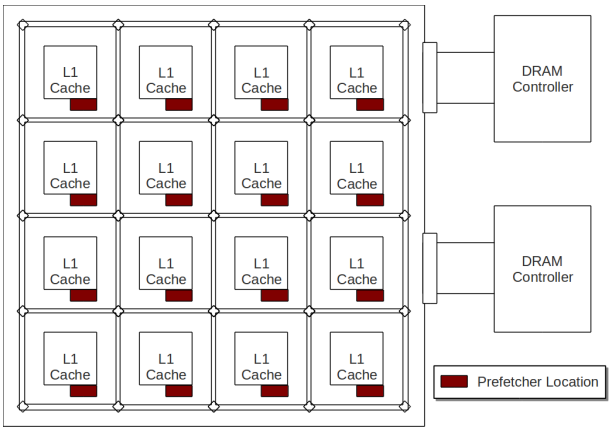

Figure 4.1: A depiction of the potential prefetcher location: At the Tile Cache

Placement of prefetchers alongside individual tile caches (see Figure 4.1) would suggest multiple prefetchers, each tied to a specific tile. These prefetchers could fetch into local prefetch buffers or into the d-caches of their tiles.

There would be potential advantages to this setup. Firstly, prefetchers could fetch either into a buffer positioned on their tile or directly into their tile's d-cache. Both of these buffer locations would promote low latency to prefetched data. Secondly, communication of miss streams and mode switch packets from tiles would be low in latency and would not require use of the chip's networks.

Local tile prefetchers would have several disadvantages. Firstly, they would compete for valuable processor real estate and cooling. As can be seen in Figure 4.1, the prefetcher would be competing for the same chip real estate on each tile as the general-purpose processor, the c-cores, and the L1 cache. Any prefetcher logic placed on a tile would have to be put there in place of other, perhaps more valuable hardware, and the heat generated by such logic would contribute to the dissipation demands on the processor. Secondly, a proximal prefetched data storage system would put extra stress on the chip's networks, some of which could be unnecessary since some prefetches

could never be used. Thirdly, if local prefetch buffers were used, they would have to be kept coherent; this would put a significant burden on the chip's networks as well as the complexity of the prefetcher and its integration with the rest of the chip. Conversely, if the tiles' d-caches were used as prefetch buffers, then prefetched data would inevitably evict program requested data, creating a conflict of interest between the prefetcher and the program.

On-tile logic would not be necessary for an on-tile buffer or d-cache storage of prefetched data. Prefetched data could be delivered to such proximal locations by off-tile logic via the GDN or the MDN. This could help to alleviate some of the real estate and power budget contention issues associated with local prefetchers.

From the perspective of prefetcher development as an optional system to augment the functionality of GreenDroid, a set of prefetchers designed to be local to each tile, would be much less flexible in their design. Integration and removal of such prefetchers would be far more complex than would that of a prefetcher which did not have machinery amongst the chip's tiles. Local prefetchers would also burden the c-core development system with additional complexity, since it would have to set up each prefetcher on each tile so that it could work with and share space with the particular c-cores placed on that tile.

4.1.2 At the Edge of the Mesh Network

This location would be on the GreenDroid chip itself between the side of the mesh network and the external DRAM ports (see Figure 4.2).

This location would have advantages over the previous. Firstly, the prefetcher would be able to make requests to the DRAM without using on-chip networks. This could reduce the burden of unused prefetch requests on those networks significantly.

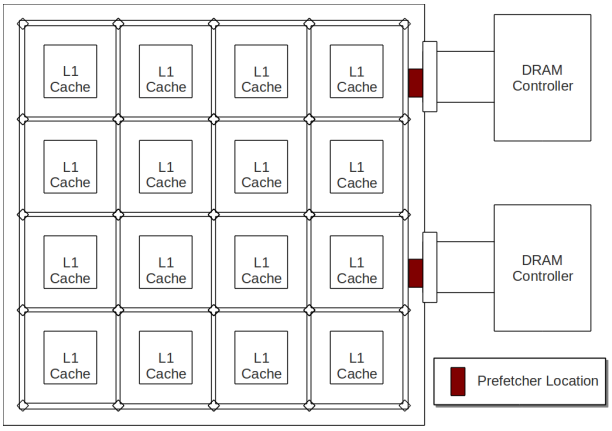

Figure 4.2: A depiction of the potential prefetcher location: At the Edge of the Mesh Network

Secondly, using a single prefetch buffer would eliminate the need for buffer coherence without causing the prefetched data to contend with the program requested data for d-cache locations. Thirdly, by moving all of the prefetcher's hardware outside of the tiles, several complexity issues associated with on-tile hardware placement (discussed above) could be mitigated.

There would be disadvantages of this system above as well. There would be higher latency between each tile and its prefetcher logic and buffer since all communication between the tiles and the prefetcher would have to go over on-chip networks. Also, prefetcher logic communication (such as mode switch packets) would put a small burden on those networks, although this would be expected to be trivial, since mode switches would not occur frequently. Similarly to the previous location, in this location, the prefetcher would be limited in physical size and complexity, as it would still be competing with other components for processor real estate. While the complexity regarding chip integration could be reduced, there would still be issues: the prefetcher's design would be tied to that of the rest of the chip, complicating independent development, and its footprint would have to be taken into account whenever a chip's c-cores changed.

While the prefetch buffer capacity would be limited by this placement, the latency between tiles and the prefetcher would be similar to the average latency between two tiles, which would be low compared to the latency between a tile and a location in DRAM.

Using underutilized on-chip tile caches to hold victim entries from neighboring tiles is an idea that has been considered by other GreenDroid teams. This prefetcher location could benefit from such a setup since the victim cache network packet protocol could potentially be hijacked by the prefetcher, allowing it to use underutilized caches as prefetch targets without having to deal with cache contention or local prefetch buffer coherence. For prefetches with a high likelihood of use, this could help to push the prefetched data closer to the tiles which would need them. It could also prove to be an effective use of otherwise dormant on-chip hardware. By using a hybrid technique wherein prefetched data could be stored in a prefetch buffer or in victim cache areas, on-chip network contention could be reduced by limiting the pushing of high-value prefetches to times when the network availability were evaluated to be low; such evaluation would be a complex issue of its own since the decentralized nature of the network would make it difficult to know where traffic might be occurring. These advantages would depend on a system which is not yet in place; if that system were implemented, the effectiveness of this location could be reevaluated.

4.1.3 At the DRAM Controller

This location would be nearly the same as the edge of the mesh network, the difference being that it would be off-chip (see Figure 4.3).

There would be several advantages to moving the prefetcher off-chip. Firstly, it could effect savings of on-chip real estate with relatively small differences in tile-to-

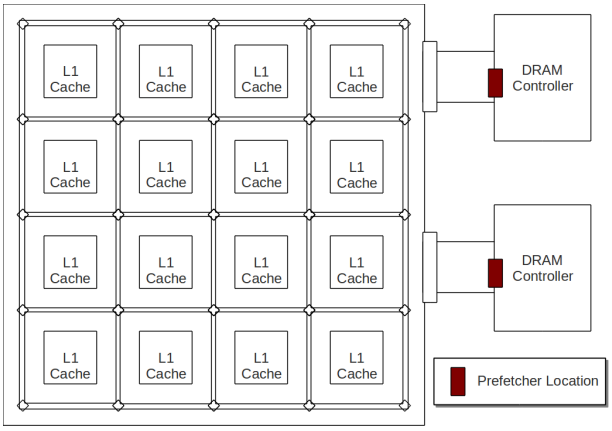

Figure 4.3: A depiction of the potential prefetcher location: At the DRAM Controller

prefetcher-buffer latency. It would not change the prefetcher's capabilities with regards to its access to the DRAM since it would be essentially in the same place on the tile-to-DRAM path.

The primary disadvantages would be that it would result in slightly higher latency to the prefetch buffer than an on-chip solution, and that it would not lend itself as well to prefetching in to underutilized on-chip data caches (if a system conducive to this functionality were implemented on GreenDroid).

4.1.4 Final Decision

Because of the increase in demands on the chip's resources imposed by local prefetchers per tile, and the low expected latency overhead of an off-chip prefetcher, the location at the DRAM controller was chosen for the prefetcher. This location allowed for independent development of the prefetcher and the GreenDroid chip, providing for a logistically adventagious decoupling.

4.2 Number

Three primary concerns affect the number of prefetchers to use for this multi-tile processor:

1. The scalability of the prefetching system should be tied to that of the DRAM.
2. The processor's memory request stream should be communicated to the prefetcher efficiently.
3. The memory request stream data should be partitioned so that it can be effectively used to make next address calculations for the set of active tiles.

The prefetcher can either see the processor request stream as coming from a set of unrelated tasks or as coming from a single task being performed by a series of related threads. In a massively parallel processor, many threads are expected to be in execution at once, and those threads are not necessarily expected to be working on related data. In the case of GreenDroid, there is a large set of distinct processing regions, each with their own processing elements; however, the expectation is that processes will use GreenDroid's processing regions to migrate around the chip in order to gain access to different sets of c-cores, and that fewer unrelated threads will be executed in parallel than might be expected from other systems with so many processing cores.

4.2.1 Independent Logic per Tile

In this setup, the prefetcher would differentiate between the stream of addresses being requested on a per tile basis. This could be accomplished either by using multiple prefetchers, each specifically tied to a certain tile or set of tiles, or by using a single

prefetcher, which would treat streams coming from different tiles or sets of tiles differently.

This setup would be advantageous in several ways and particularly under certain usage conditions. The need to differentiate between streams of requests coming from different tiles would arise primarily if the streams were unrelated and treating them as being related would cause confusion in pattern recognition. Under any usage conditions, if the prefetch buffer were divided into pieces, each tied to an individual tile, it could speed up buffer searching, since each sub-buffer would be smaller.

This setup would suffer if cache miss streams from different tiles were related, since each predictor would see a smaller portion of the the overall stream. This would mean that each predictor would have less information than were available about the stream for which it were predicting.

4.2.2 Singular Logic for All of Main Memory

This could potentially be the simplest set up. The prefetcher would make decisions based on a singular stream of processor requested memory addresses, without differentiating them by the tile that asked for them.

This system would work best if all of the memory accesses from the chip were related; i.e. multiple tiles were performing subtasks with overlapping working sets. Since GreenDroid is designed to use its many tiles to perform different subtasks at different times, mostly pertaining to a small set of programs, it is expected that the GreenDroid chip will often be using its many tiles to work on a single job or a small set of jobs.

The primary disadvantage of this setup would be its scalability. As the demands on maim memory increased, and multiple DRAM controllers became involved, the speed of the singular logic of this system could become a bottleneck.

4.2.3 Independent Logic per DRAM Controller

This setup would be similar to the singular logic in that it would not differentiate between requesting tiles in the partitioning of the processor request stream; instead, it would consider all members of each stream of addresses coming to an individual DRAM controller to be related. It would be different, from the aforementioned in that it would scale with the number of DRAM controllers. This setup would be advantageous since GreenDroid's main memory access system scales in the same way.

In order for a synchronized setup to work with data sets that arbitrarily spanned multiple DRAMs, all prefetchers would need either to receive all synchronization messages, or an extra layer of communication between predictors would have to be implemented in order to effect synchronization. This could provide a bottleneck that were more restrictive that that affecting the DRAM under those certain circumstances.

4.2.4 Final Decision

Since most of the tiles in GreenDroid are expected to work on parts of the same problem, and the prefetcher will, in this iteration, not be prefetching data directly to different buffers (or caches) for each tile, the best options seem to involve the consideration of address streams from different tiles as related. Independent Logic per DRAM Controller can scale with the main memory access system. For this reason, tying the prefetcher's logical systems and buffers to individual DRAM controllers was chosen. This connection between prefetchers and DRAM controllers can be easily accomplished because of the chosen location of prefetcher elements in the memory system: off chip, just before the DRAM controllers.

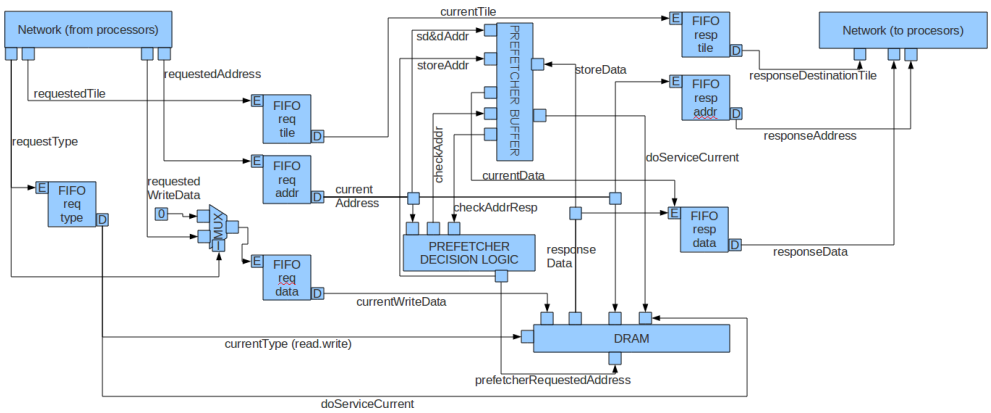

Figure 4.4: A prefetcher datapath diagram (prefetcher internals).

4.3 Prediction Logic

Some prior work has indicated that stride and related prefetchers, while simple and space efficient, can also be very effective, even given little prior information about data structure and access patterns [16,23,25,37]. Other work has sought to use software devices, such as *helper threads* [8,11,27,47,48] and explicit prefetch commands added to code by a specialized compiler [34], to help define order where a generic prefetcher might have trouble.

4.3.1 Condensed Stride

The *condensed stride* prefetcher logic seeks as many addresses as it can find between the most recent processor requested address and each of the previous requested addresses within a certain time frame such that the difference between the two addresses in question is within a specified maximum distance. It is called "condensed" because it keeps a small amount of state for detecting streams, only allowing itself to prefetch for readily visible streams; its simplicity is intended to reduce its footprint and its power requirements.

4.3.2 Null

The *Null* prefetcher makes no requests. It never makes any prefetch decisions, and it does not use the prefetch buffer as a conflict cache, enforcing the condition that its buffer is always empty. It suffers no latency due to prefetch buffer searches. This prefetcher is used as a baseline for cycle time comparisons of the system with different prefetcher setups to the system without any prefetcher effects.

4.4 Aggressiveness

Prefetcher aggressiveness determines the frequency with which the prefetcher will be allowed to issue prefetch requests. In a memory system where DRAM contention can be a significant impediment to performance, prefetcher aggressiveness can be of primary concern.

The prefetcher can be throttled in two ways: Firstly, its prediction logic may only attempt to issue prefetch requests when they are deemed prudent by the prediction mechanism. For example, a stride prefetcher may only make fetches for addresses that it believes with high confidence will be needed by the processor. Secondly, the prefetcher may be disallowed from making prefetch requests when it runs the risk of contending with the processor for DRAM access. This second type of throttling may be effected externally to the prefetcher's decision logic, since it is intrinsically linked to the demands on the memory system rather than to the prediction mechanism. Both of these types of throttling may be used to determine the ultimate level of prefetcher aggressiveness. The second type is the focus of this section, since it is a global concern, which may apply in the same way to any prefetcher logic used. The first method is covered on a per decision logic basis, since it is much more directly tied to the specific decision logic in question,

and will generally differ between different prediction logic setups working in the same memory system.

4.4.1 Strong: Continual Stream

This setup would allow the prefetcher to issue a prefetch request for every cycle in which it had a calculated address ready, provided that the DRAM queue could accommodate a new fetch request (in the case of GreenDroid the maximum queue length would be one, so prefetch requests would only be able to go out when there were no other outstanding DRAM requests). It could allow the prefetcher to get significantly ahead of the processor when it were confident about its prefetch addresses; this could be beneficial, since it could result in a buffer full of useful addresses. However, if the prefetcher were largely inaccurate and still requesting addresses, this setup could cause the processor to be delayed when its requests missed in the prefetch buffer. In a pipelined or otherwise multi-ported DRAM, a nearly continual stream of prefetcher requests could be less dangerous, since new requests would be processed while outstanding requests were still in the pipeline.

4.4.2 Relaxed: DRAM Free

In this setup, the prefetcher would be restricted to making requests only when there were no outstanding processor requests that had missed in the cache. This would be a conservative strategy, aiming to minimize the number of cycles that the processor waited due to outstanding prefetch requests. While something closer to a continual stream could be effective in a fully pipelined design, when used with a single-ported, unpipelined DRAM, it could potentially block the processor from making needed requests for periods equal to the maximum DRAM latency. This could effectively double

the DRAM latency for the processor if the prefetcher's predictions were completely inaccurate. Under the belief that that some requests are unpredictable by any prefetcher, this would be a much more sensible aggressiveness setting for use with a single-ported, unpipelined DRAM.

4.4.3 Final Decision

Since GreenDroid is using a single-ported, unpipelined DRAM, its prefetcher will use the *DRAM Free* notion of aggressiveness. This prefetching system endeavors never to be in the way of the processor. With an unpipelined DRAM, the prefetcher could easily end up blocking the processor routinely by having too aggressive a prefetcher. A DRAM Free aggressiveness level will help to minimize that danger.

4.5 Conflict Cache

The idea of using the prefetcher buffer as a conflict cache was added after having run some tests of the simulation system. Certain stride simulations were reporting more buffer hits than prefetch requests. This suggested that the test in question (summing three arrays) was suffering in a noticeable way from conflict cache misses, and that the prefetcher buffer was helping to alleviate that problem by supplying its buffered cache lines several times in a row. Since a DRAM Free prefetcher is restricted to making requests only when the processor is making none, its buffered values could become "outdated" after a long stream of processor requested buffer misses. For this reason, when the prefetcher is dormant (whenever the processor is missing in its cache) its buffer is filled by processor requested cache lines, allowing it to act as a fully associative conflict cache for the processor.

4.6 Condensed-Stride Logic

4.6.1 Description

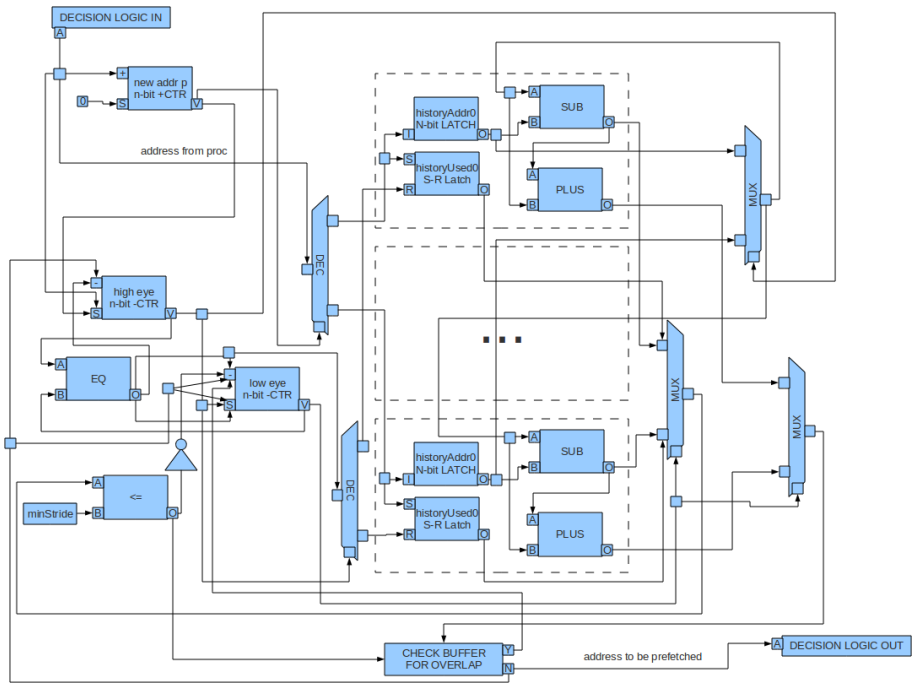

Figure 4.5: A datapath depicting the layout of the Condensed-Stride prefetcher next prefetch address calculation logic.

The condensed stride prefetcher logic attempts to detect progressive memory access patterns defined by standard stride lengths. It keeps a record of the last 32 addresses to be requested for a read. For each entry, a single bit defines whether that address will be used as a base for finding standard stride-length memory accesses. The prefetcher also stores two indexes: a high eye and a low eye. In order to calculate a new address to be prefetched, the prefetcher measures the difference between the address at the location of the high eye and all other addresses stored in the history. It then decrements the low eye from the high eye until it finds an address for which the difference is small enough to prefetch; at this point, it checks the prefetcher buffer to make sure it won't fetch an

address which is already in the buffer. Once an acceptable stride length is found, the address at the high eye is marked as unusable as a base and the high eye is decremented. If no acceptable stride lengths are found for a high eye address, the prefetcher marks the address at the high eye as unusable as a base and decrements the high eye. At this point one address calculation is complete. If a stride length has been determined, a prefetch is made for the address at twice the stride length from the address at the high eye. Once all addresses have been marked unusable as a bases, the prefetcher stops making requests until new data is made available by the processor.

Traditionally, stride detecting prefetchers use saturating counters to detect and begin fetching for request streams. Conversely, this condensed stride prefetcher does not use saturating counters; instead, it assumes a stream for the maximally temporally proximal pairs of processor requested addresses within the minimum stride distance from each other. This setup may result in the issuance of more extraneous prefetch requests when numerically proximal requests that are not parts of streams are issued in close temporal proximity, but it should also cause streams to begin being fetched for more quickly. Since each request address may only be paired with one other address while functioning as the high eye, and the request address history buffer is as large as the prefetch buffer, the prefetcher will never out pace any processor request stream by more requests than can fit in the prefetch buffer.

The name "condensed-stride" refers to the condensed footprint of this prefetcher. It was designed to keep less state than conventional stride prefetchers. It also was designed to be easily added and/or removed to/from the GreenDroid system with as little overhauling of other systems as possible. Often stride prefetchers attempt to associate request streams with different program execution locations by tying their counters to data they receive from the program counter (see [1]). By contrast, this condensed-stride prefetcher does not have such data available, since making it available could require sig-

nificant modifications to the memory request system, including the packet system of the MDN (or the use of some of other network(s)) to communicate such information, and this prefetcher was designed to function without making major modifications to other systems on the chip.

4.6.2 Implementability and Simulation Tuning Numbers

The next address calculation restarts whenever and only when the high eye moves. Whenever this happens, the state of all of the calculated numbers in the prefetcher needs to be updated. The longest path to such an update passes through one multiplexer, one subtracter, and one adder; for this path, we will allow two cycles. Once the state has been updated, the low eye must traverse all of the history cells. This traversal could potentially be done more efficiently using a binary tournament tree; however, we would still need to enqueue accesses to the prefetcher buffer to check for overlap. We allow six cycles for the traversal plus how ever many cycles are necessary for the various checks of the prefetcher buffer. We also add an additional cycle for each time the low eye traverses the history cells completely, resulting in a reduction of the high eye without any address having been produced. Thus, we give the condensed-stride prefetcher decision logic eight to nine cycles for each traversal of the low eye plus a penalty for each time it checks the prefetcher buffer.

Chapter 5

Related Work

This project is related to other work on memory prefetching. Some prominent hardware and software prefetching works are discussed in this section. Most pertain in some way to software prefetching via explicit compiler-generated prefetch commands or via blocking or non-blocking helper threads or to hardware prefetching using machine learning techniques.

5.1 Stride Prefetchers

[1] discusses the design of stride prefetchers which incorporate the use of the program counter (PC) to differentiate between different memory access streams. Such associations can increase the accuracy of stride prefetchers since many streams are accessed by the same PC location as a program iterates through a loop or executes a recursion. In this paper, the authors discuss methods of limiting the amount of data that needs to be sent along with each memory request. Instead of sending whole PC values, they send only the four least significant bits; their results indicate that such hashed data can be used effectively to gain similar advantages to those gained by using the full PC.

The condensed-stride prefetcher by contrast, does not use the PC at all in determining its streams. It avoids the use of the PC because the sending of any PC data along with every memory request would eliminate its easy integration (and/or deintegration) into the GreenDroid system. The CS prefetcher looks for stream access by temporal and address-based proximity of requests only. This works well for certain applications, but it may work less well in other applications than a prefetcher that took the PC into account might.

5.2 Markov Prefetching

In their 1997 paper, "Prefetching Using Markov Predictors," [24] Doug Joseph and Dirk Grunwald discussed the idea of using a Markov-like model for performing a form of correlation-based prefetching. This concept was designed to prefetch for access patterns that were not conducive to stride or stream prefetchers. Their model made use of correlations between series of fetches, training a limited-overhead Markov model on cache misses, and then prefetching based on the model's predicted following addresses. Their system gave scores to potential prefetch addresses based on their expected likelihood of request by the processor, placing them in a priority queue based on those scores. Processor requested addresses were placed in the same queue with the highest priority. By fetching addresses dequeued from this priority queue, this system was able to throttle itself to reduce memory contention with the processor while also increasing its prefetch usefulness ratio.

5.3 Assisted Execution

Michael Dubois and Yonh Ho Song introduced, in their 1998 paper [11], the idea of "assisted execution," which involved the creation of extra assisting "nano-threads." These light-weight threads would run on the same processing core as the main thread, making speculative memory requests ahead of the main thread. In this way, when they were active, they would end up performing some prefetching for the main thread. Since they would be separate threads, they could be suspended while waiting for their memory requests to be serviced without upsetting the computational critical path of the main thread. These threads, for example, could easily be instructed to proceed with a series of contiguously addressed prefetches whenever the main thread were about to traverse an array. The authors reported "Simulation results on several SPEC95 benchmarks show that sequential and stride prefetching implemented with nano-thread technology performs just as well as ideal hardware prefetchers." [11]

5.4 Speculative Precomputation

More work on software prefetching was published in 2001 by Jamison D. Collins, et al. [10]. This work discussed the idea of Speculative Precomputation (SP): the use of idle hardware thread contexts to execute pieces of code speculatively, in an attempt to push data cache misses to occur before the data will be needed by the main thread. This system is designed to leverage otherwise unused thread contexts, for executing applications which are largely sequential, making little use of multi-threaded hardware. This type of software prefetching targets systems in which all code executes on a general-purpose processing core, wherein there is no specialized knowledge of any particular code segments. However, it is still fundamentally conducive to an architecture such as

GreenDroid with many available processing cores, especially for applications which are otherwise difficult to parallelize.

A complete heterogeneous prefetching solution for GreenDroid could include something such as Speculative Precomputation for code segments which were run on the general-purpose processor. Speculative Precomputation threads could be executed on nearby tiles in order to reduce the number of caches misses for these low-information situations. This would require that tiles be able to fetch from adjacent tiles' L1 caches, which is a potential capability for future GreenDroid releases (see 2.2.3). Since SP generates extra, non-essential threads, it could increase power consumption, the minimization of which, in the case of GreenDroid, is of primary concern. Because power constraints for GreenDroid are set more to maximize battery life than to minimize processor heat dissipation (meaning that the processor will not necessarily overheat if extra threads are added), the processor could potentially run with added prefetching mechanisms such as SP only when its host phone were plugged into an external power source.

5.5 Software-Controlled Pre-Execution

Also in 2001, Chi-Keung Luk wrote a paper discussing the idea of software-controlled pre-execution in SMT processors [31]. The system illustrated differs from the previous one in that the software prefetching threads are spawned by a software-controlled mechanism as opposed to an automatic hardware controlled system, which spawns speculative threads whenever a cache miss occurs in the main thread.

Speculative execution-based helper threads are incapable of achieving speedups of greater than 2:1 in processing time, since they stall whenever they initiate a fetch from the DRAM. In order for such a system to achieve greater than a 2x speedup, multiple threads need to be used, each speculating on a different code segment , so that when

one thread is stalled, another can be fetching ahead of it. By staggering such speculative threads, full sections of the program could have been fetched for by the time their predecessors finished executing.

5.6 Prefetcher Throttling

In "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," we see the same non-competition notion as DRAM Free being employed: "region prefetches are scheduled to be issued only when the Rambus channels are otherwise idle." [29]

Chapter 6

Simulation

This chapter discusses the process by which the prefetcher was simulated on the GreenDroid cycle accurate simulator and outlines the benefits shown by the use of this prefetcher. We begin with an introduction to the simulator itself. We then discuss the process by which specifics of the simulations were chosen and the challenges that were faced in the implementation. Finally, we conclude by illustrating and discussing the results obtained.

6.1 Software Environment

The GreenDroid cycle accurate simulator is the framework in which the simulation of the prefetcher design was performed. In this section, we will discuss the simulator as the means by which tests and benchmarks were performed.

6.1.1 Cycle Accurate Simulator

Operation of GreenDroid is simulated one cycle at a time. Each device is described in a .bc file, using the *bc* language. Devices are defined using an initialization, or *init* function and a calculation, or *calc*, function, which are each registered with the environment by a call to a native registration function. The state of each device is stored in a struct-like data structure, which is initially populated by the init function, and is ultimately passed to the calc function, when the chip simulation starts. The calc function operates in a continuous loop, yielding after each cycle using an explicit call to the built-in *yield* primitive. For each cycle of the simulation, each device is executed starting at its last call to yield and stopping at its next. Devices are generally coded to perform the appropriate computations for the cycle at hand between successive calls to yield; sometimes devices may simply yield for a certain number of cycles, and then perform their computations for all of the elapsed cycles at once.

User Code

The compiler included with the simulator environment is capable of compiling programs written in C which make use of standard C libraries as well as specialized GreenDroid libraries to run on GreenDroid under the Linux-basedoperating system.

In hardware simulation, allowing software running on the simulated hardware to report to the user of the simulator is a challenge. Communication itself can strongly affect the simulation results. For example, one could use the simulation environment to monitor the contents of some existing register, thereby allowing the software to use that register as a means of communication with the user. This would, however, affect the register's primary function as a part of the simulated hardware. Fortunately, GreenDroid's simulator has a low overhead means through which programs can communi-

cate with the user: The architecture includes space for special processor instructions which allow the program running on the processor to deliver one from a set of messages, each containing one integer value, to the user. This communication mechanism requires a minimal cycle overhead. GreenDroid's C libraries expose this functionality with the functions: *raw_test_pass(int)*, *raw_test_fail(int)*, and *raw_test_done(int)*. These functions are instrumental in debugging user code written for the simulator and in returning user code measured results when performing hardware tests. When called, raw_test_pass(int) communicates the pass message, the int value specified, and information about the source of the call. raw_test_fail(int) and raw_test_done(int) deliver their messages in similar ways. Unlike raw_test_pass(int), both of the other calls result in a termination of the simulation at the point of the call.

In order to categorize statistical data during the prefetcher simulation, the code running on the processor must be able to communicate section breaks to the prefetcher's statistical machinery; without this ability, the prefetcher would be unable to differentiate between the periods when each processor job were being carried out, and would thus be unable to collect data regarding its performance on different code segments. To accomplish such sectioning, the user code piggy-backs on the mechanism used to request mode switches from the prefetcher. Using a special protocol for section definition added to the GDN communication mechanism, the user code is able to tell the prefetcher's statistical mechanisms when it begins and ends significant code sections. Although this mechanism comes with a slight GDN overhead, it is low in the grand scheme of a large test, since section divisions rarely occur.

Hardware Description

The simulator is divided into devices. Each device has its own event loop. Because the prefetcher and the DRAM work so closely, the prefetcher description source code was added to the same source file that already contained the DRAM source. The prefetcher's event loop describes its next address calculation and fetching functions. The addition of knowledge to the prefetcher from the processor's address stream and the seeking of prefetched data within the prefetcher buffer upon processor requests, however, is set within the DRAM's event loop. In this way, the prefetcher's operations are split between two event loops, since some of them are essentially autonomous, while others re influenced by or have influence on the DRAM itself.

The prefetcher code is designed so that its calc routine can choose to invoke some particular subroutine depending on the prefetcher type (Condensed-Stride or Null). The prefetcher is allowed to calculate a new prefetch address based on a next address calculation timer. The timer is decremented at every prefetcher calc loop yield call; when it reaches 0, a calculation is performed, and if there are no outstanding requests to the DRAM, the address produced is prefetched into the prefetch buffer. In order to enact the DRAM Free aggressiveness level, the prefetcher keeps track of the current state of processor DRAM fetch request activity. The prefetcher calc time counter may be decremented when the DRAM is not free, simulating a next prefetch address calculation, but prefetches may not be issued.Prefetch buffer find latency is simulated by yielding once in the DRAM calc loop for each cycle of prefetch buffer latency. DRAM latency is simulated in a similar way using the appropriate DRAM latency for the type of fetch occurring (see Section 6.3 for a description of the different DRAM latencies used).

bc Language

The bc language used to describe devices is based on C. It has several syntactical additions which improve ease of development. Firstly, it has an implied data type which can be used by declaring a variable as global or local. Using this data type, it allows for the creation of hashmap data structures, with special "." syntax for adding and accessing their fields. It also allows for function pointers to be added to hashmap structures using &fn syntax. This means that hashmaps can be used similarly to C++ objects, since a "self" variable is passed as the first argument to any function which is called from a hashmap pointer.

6.2 System Constraints

In order to test this system, a set of assumptions was required. Since final performance numbers, c-core configurations, and demands on GreenDroid were not known, many assumptions had to be made to simulate an appropriate system for testing of prefetcher configurations.

The following performance numbers for the memory system were imposed by the simulation environment:

- Word width: 8 bytes
- Cache line width: 8 words
- L1 Cache size per tile: 32 kilobytes (4096 cache lines)

Based on the specs for the Xilinx Virtex-6 ML-605 DRAM controller to be used with GreenDroid, the following performance numbers were chosen for the prefetcher:

- DRAM max concurrent open rows: 4
- DRAM read latency (new row: RAS + CAS latency): 73 cycles
- DRAM read latency (open row: CAS latency only): 27 cycles
- DRAM write penalty: 70 cycles

In order to effect a conservative simulation, with the Null prefetcher, the CAS latency was used for all reads. This is the same latency given to the condensed-stride prefetcher because it is expected that most streams will make multiple requests on the same row in short succession, so the row will stay opened and no RAS operation will need to be performed. This situation should pertain for the most part to both the prefetcher and the processor, so for these addresses it is prudent to give the same read latency to both.Under this setup, there is no particular advantage given to the prefetcher or the processor based on expectations of the proximity of the addresses it will request.

Finally, the following performance choices were made for the prefetcher itself. These were considered conservative choices based on the maximum amount of time that the prefetcher would be expected to take to generate addresses and search its (small) prefetch buffer. These conservative estimates were made based on the design datapath (see Figure 4.5).

- Prefetcher next address calculation time: 8 cycles^{+}
- Prefetcher buffer find latency: 4 cycles

$^{+}$ One extra cycle is taken for each full swing of the low eye that does not result in the generation of a valid prefetch address, but serves only to invalidate the use of the high eye on the current high eye address; if no valid next address is found, another iteration must be made before a new address is generated. One times the buffer find latency is also required for each lookup in the prefetcher buffer.

The simulator's DRAM module is capable of fetching two 8 word cache lines at once. There is one DRAM, and it is non-pipelined, so the system must wait for an entire fetch to finish before initiating another. While this is itself a performance limiter (see discussion of this limitation in Section 3.4), it also presents a special challenge, since it forces the prefetcher to be extra judicious with its prefetch requests, since they are capable of blocking processor requests for a period equal to the maximum latency of the DRAM.

6.3 Simulation Methodology

Simulation was performed using two different prefetcher setups: *condensed stride* and null. The Null prefetcher was used to determine, roughly, how many cycles were saved by the condensed ctride prefetcher over no prefetcher at all. The Null prefetcher did not make any fetches, had a zero cycle buffer find latency, and always had an empty prefetch buffer; it did not ever contend with the processor for DRAM fetch time, nor did it offer any speedups to the processor.

Since the DRAM is capable of keeping up to four rows open simultaneously, we assume that the stride prefetcher will most often be able to make request from open rows, so we allow its prefetch requests to be made using the open row DRAM latency.

The condensed stride prefetcher was defined so that it always fetched two cache lines at once and stored them in its buffer. The entire prefetcher system ran with a DRAM Free (see Section 4.4.2) level of aggressiveness, only making prefetch requests when the processor had no outstanding requests of its own.

6.4 Implementation

This section discusses the implementation details for the selected testing operations, commenting on the challenges discovered and the specific methods used to overcome those challenges.

6.4.1 Stride Detection

The prefetcher used condensed stride detection based logic to make prefetch decisions. This prefetcher was developed to detect a series of stride pattern memory access streams which could be occurring in an interlaced fashion. It was designed to be able to ignore some level of noise in the request stream (i.e. requests not conforming to any of the stride access series).

The condensed stride logic keeps a list of previous processor memory requests. Whenever a new processor request is seen by the prefetcher, it compares the requested address (its starting point) to each of the entries in its knowledge base in reverse chronological order, seeking the pair of addresses with the smallest difference. When it finds an address pair with a smaller difference than it has thus far seen, it adds the difference (positive or negative) to the requested address to produce a potential prefetch address. It then checks the prefetch buffer for that address; if that address is found, then the difference at hand is considered nonviable. If the address is not found, it considers that address the best prefetch candidate thus far found. It then resumes its search for the smallest viable address difference. If it finishes its search and the smallest viable address difference it has found is higher than some maximum acceptable stride length value, it declares the latest processor requested address unfruitful, and proceeds to perform the same search again, using the newest address in the knowledge base that is still considered fruitful as its starting point. This next iteration is considered a new address calculation, requiring

the same amount of time as the last. When all addresses in the knowledge base have been declared unfruitful, the prefetcher becomes dormant. In this way, the condensed stride prefetcher throttles its own use of the prefetcher buffer in order to keep itself from getting too far ahead of the processor and overwriting useful entries which have yet to be used.

6.4.2 Prefetcher Buffer as Conflict Cache

A series of simple programs and standard benchmarks were run using the condensed stride detection prefetcher in its various stages of development. One such program traversed three arrays at once, building up a sum from their collective members. The statistics gathered indicated that there were significantly more hits in the prefetch buffer than there were prefetch requests, indicating that certain addresses were being requested from the buffer multiple times having only been added to the buffer once. It was determined that this was due to caching conflicts between the arrays, and that the prefetch buffer was thus inadvertently acting as a conflict cache in this instance. From there, the ability to use the prefetch buffer as a conflict cache was added; processor requests that missed in the buffer would be stored once they had been fetched from the DRAM. The overhead from this system was all in buffer usage, since the fetching of these addresses would occur in either case. It was found to be somewhat effective for certain tasks. When the stride detecting prefetcher were in an accurate phase, it would rarely use its buffer space for conflict entries, since most of the processor requests would hit in the buffer. Only at times when the prefetcher were not accurate, or were issuing few requests of its own, due to a low rate of pattern detection, would a significant portion of the buffer space be used to hold conflict entries. Thus, for this type of prefetcher, which had spells of pattern detection and spells of low information, conflict caching was

accomplished with little overall overhead.

6.5 Selected Benchmark Results

This section outlines the selected benchmarks and discusses their relevance to this prefetcher. In this section, we discuss the results obtained during the simulation runs. For each of the benchmarks, we present two graphs:

1. A graph depicting the prefetcher buffer hit ratio for memory request cache misses over the course of the benchmark's execution. This graph includes a scatter plot which shows the local hit ratio for small set of temporally proximal memory requests. It also includes a connected curve plotting the cumulative hit ratio for all requests issued between the beginning of the program execution and the time associated with the x coordinate on the graph. The local hit ratio scatter plot can be used to get an idea about where the hit ratio hovered, and how consistent it was for small time intervals, and the cumulative hit ratio can be used to see how the prefetcher's performance evolved over time and what effect it had overall.

2. A graph depicting the number of cycles taken to reach each numbered cache miss when the benchmark was run with the condensed stride prefetcher versus when it was run the null prefetcher. Since no changes were made to the cache configuration between these two scenarios, each cache miss indicates a specific, unique, consistent location within the execution record of the program. Thus, for any given cache miss index, a lower number of elapsed cycles indicates that the program has approached the location of that cache miss in its execution more quickly (using fewer processor cycles).

Benchmarks from two categories were used to test the effectiveness of the prefetcher: the PROJECT series, which included a set of benchmarks designed for this project, and the SPEC CINT2000 series.

6.5.1 PROJECT Benchmarks

The PROJECT benchmarks implement common computational tasks. Each was designed to run within a few hours. The test results were obtained by running the benchmarks to completion.

1.traversal

The 1.traversal benchmark begins by creating four array, one of which is equal in size to the L2 cache of the tile on which it runs. It proceeds to traverse the cache array, reading from it in order, so as to clear the cache. It then sums the values of all of the integers in the the other three arrays by traversing them in an interlaced manner, reading one integer from each at a time.

In this simplest of benchmarks, we see exceptional hit ratios for the condensed stride prefetcher. This is expected, since this benchmark performs multi-array traversal making uniform-length strides along each of its arrays. The prefetcher appears to have no trouble detecting and anticipating these stride access patterns.

We see a reasonable gain in cycle performance when using the prefetcher versus when not using it. This performance gain is, of course, limited by the effect of memory access delay on the computation. In the case of this benchmark, much of the computation involves the performing of load operations from main memory. However, our performance gain is limited by the relative speed of the DRAM used with respect to the speed of the processor itself.

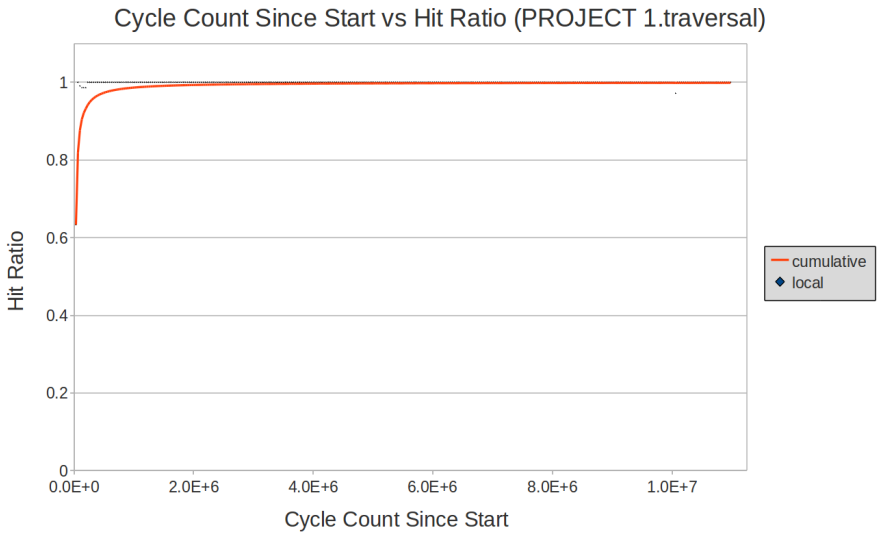

Figure 6.1: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the PROJECT 1.traversal benchmark. As expected, since this benchmark performs a simple traversal with a uniform stride length, the prefetcher performs nearly perfectly, maintaining nearly a 100% hit ratio over most of the course of benchmark.

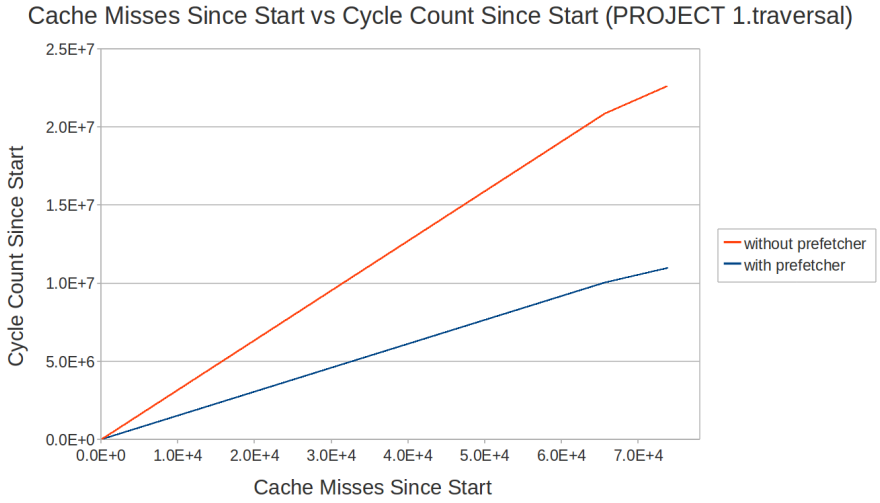

Figure 6.2: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the PROJECT 1.traversal benchmark. This graph illustrates significant speedups over the course of the benchmark. This is expected, since much of the computational time spent during this benchmark is owed to memory latency, so with a high prefetcher buffer hit rate, the computational time required to get to each cache miss should be lowered. In this case, by incorporating the prefetcher, each point in the computation was able to be reached in about half of the time.

2.qsro

The 2.qsro (QuickSort Reverse Order) benchmark creates two arrays. The first of these arrays is filled with values in descending order. The second array is then traversed in order to clear the cache. Finally, the original array is sorted using the C library qsort function.

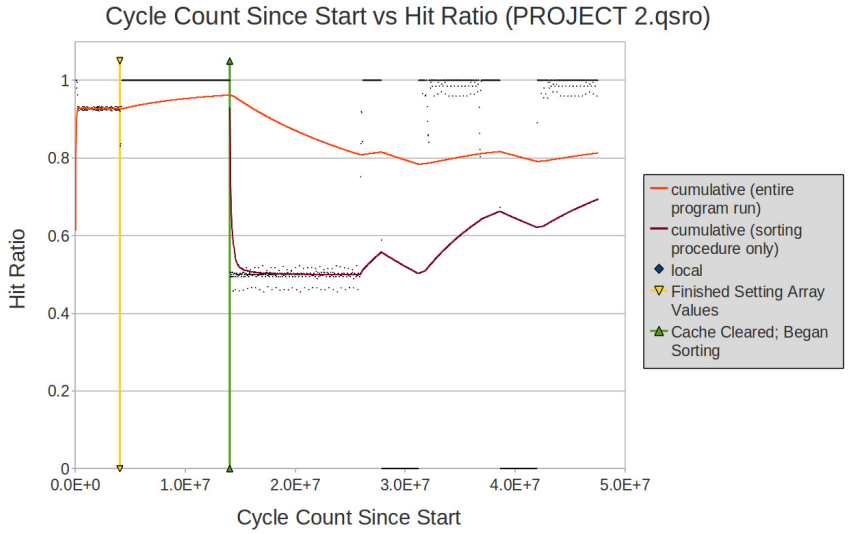

Figure 6.3: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the PROJECT 2.qsro benchmark. Here we see very high hit ratios while the original array is populated, and while the cache is cleared. During the sorting procedure, we see three zone types: one where the prefetcher tended to have about a 50% hit ratio, one where it tended to have about a 100% hit ratio, and one where it had about a 0% hit ratio. We presume that the 50% section came from a period where the procedure traversed the array, flipping the positions of every pair of values it saw (reverse-sorted array), resulting in a lot of hits and a lot of buffer value kills due to writes. For the 100% sections, we presume that these came from traversals of resulting fully sorted sections. Finally, for the 0% sections, we presume that these came from sections wherein the procedure randomly polled the array in an attempt to determine a good pivot point. These hit ratios would be as expected for such a procedure: very high during read traversals and very low during random accesses. The prefetcher reached nearly a 70% hit ratio over the course of the sorting procedure.

Since the values in the unsorted array are set in reverse order, we expect some regularity in the performance of QuickSort on the array. This regularity may come

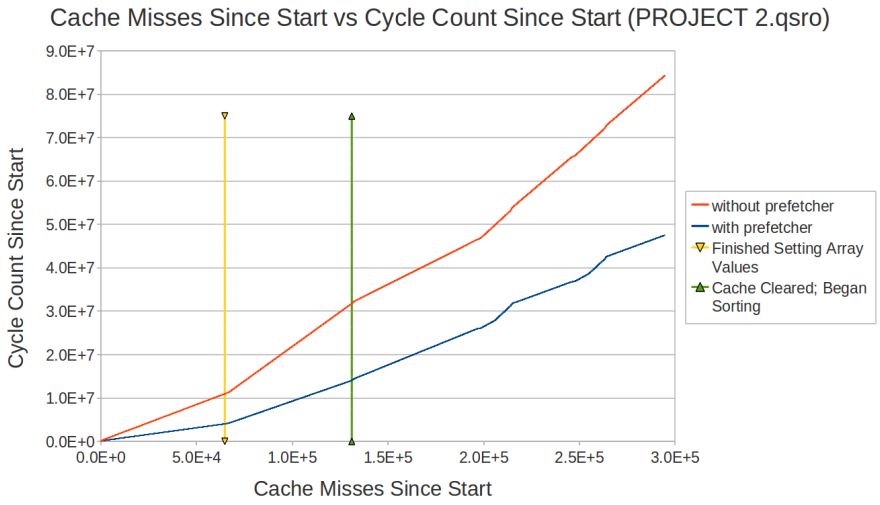

Figure 6.4: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the PROJECT 2.qsro benchmark. In this graph we see the run with the prefetcher out pacing the run without the prefetcher by about a five to three ratio during the sorting procedure. This speedup is effected in more in certain zones than in others, since the prefetcher hit ratio swings between about 0% and about 100% in different sections. The five to three ratio we observe is significant especially considering the prefetcher's wide swing of hit local ratio values.

from consistently good pivot point selection, since the data is laid out in the array in a simply patterned manner. QuickSort, when using good pivot points, begins by traversing large strips of memory. As seen in Figure 6.3, these traversals appear to result in high prefetcher buffer hit ratios. There are also other sections where the hit ratio is zero; we suspect that these are related to the search for a good pivot point, which may be performed at random throughout the array.

3.qsoo

The 2.qsro (QuickSort Out of Order) benchmark creates two arrays. The first of these arrays is filled with values out of order. These values are generated by multiplying the array index by a prime number which is about one third the size of the array (a power of two) and then taking that product modulo the array size. The second array is then traversed in order to clear the cache. Finally, the original array is sorted using the C library qsort function.

In contrast to 2.qsro, 3.qsoo has a very non-uniform prefetcher buffer hit ratio pattern. This seems to result from poorer pivot point selection. When poorer pivot points are selected, the same data needs to be partitioned and pivoted more times, and the partitions vary more in size and position. This results in a longer running time (as seen in Figures 6.5 and 6.6) and more pivot point selections. It may also result in a less uniform cache-hit/miss profile. These seem to be likely explanations for the lower uniformity in prefetcher buffer hit ratios seen for this benchmark. This could also explain the lower overall hit ratio for the prefetcher buffer, since the cache miss profile would be less uniform if the cache were populated less uniformly, which would make it more difficult to detect and fetch for stride-based accesses.

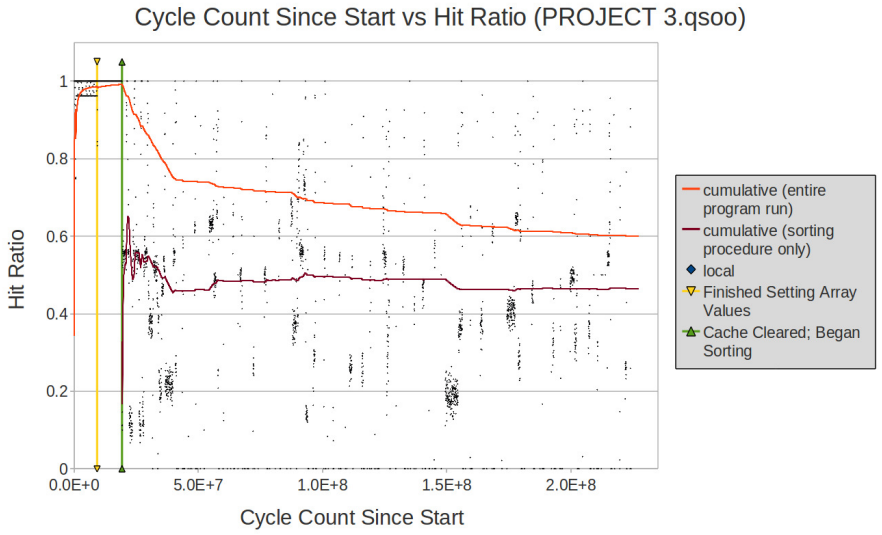

Figure 6.5: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the PROJECT 3.qsoo benchmark. In this graph, we do not see the consistent partitioning of hit ratios that we saw in the graph for PROJECT 2.qsro (Figure 6.3). The more random-eque ordering of the array would have caused the procedure to have much less consistency in the offsets and lengths of its traversals, and the numbers and configurations of its pivot point selections. Interestingly, this de-partitioning of the procedure's traversals only reduced the overall hit ratio of the prefetcher during the sorting procedure from nearly 70% to about 45%, indicating that even when traversals are not consistently long, the prefetcher can see reasonable hit ratios.

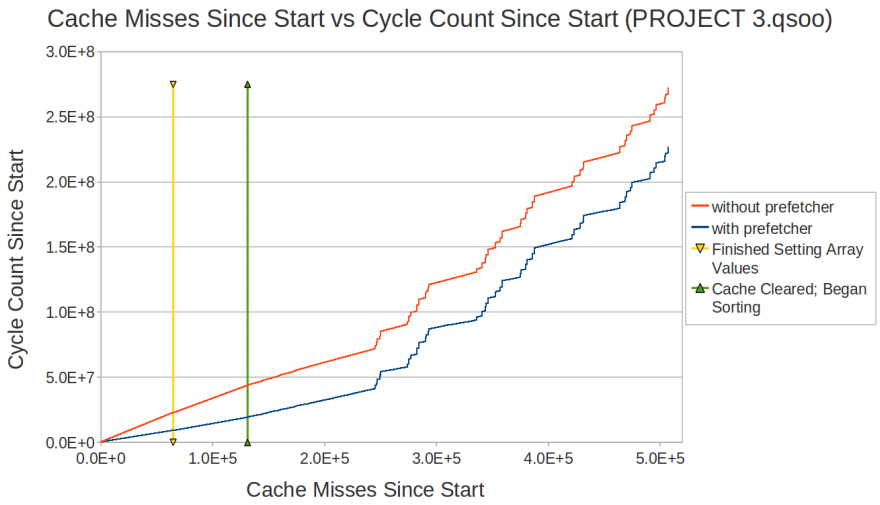

Figure 6.6: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the PROJECT 3.qsoo benchmark. Here we see a surprisingly low speedup of under 10% during the sorting procedure, given that the overall hit ratio for the prefetcher was about 45%. The execution of PROJECT 3.qsoo took significantly longer than that of PROJECT 2.qsro (as expected for QuickSort since pivot point selection would be harder with the supplied data, and the running time can be asymptotically larger for poor pivot point selection). The presumed reasons for the low speedup aside from the lower hit ratio are twofold: firstly, that during PROJECT 3.qsoo, the sorting procedure spent a higher portion of its time performing calculations in the processor and hitting in the cache and a lower portion waiting for memory requests than it did during PROJECT 2.qsro, and secondly, that where the hit ratio was lower, the prefetcher was being more inaccurate, and may have gotten in the way of the processor more.

4.bts

The 4.bts (Binary Tree Search) benchmark begins by creating two arrays. The first of these arrays is populated by a tree of nodes laid out in memory in breadth first search order; these nodes each contain an integer id, and a pointer to each of their left and right children. They are laid out by stepping several counters along the length of the array at different rates and placing nodes with appropriate pointers to other nodes within the memory space. Once the nodes have been laid out, the second array is traversed in order to clear the cache. Finally, the nodes are traversed again in depth first search order (out of order in memory) using a recursive function that relies on each of their left and right pointers, and their ids are summed.

In Figure 6.7, in the first partition, we see two distinct cache hit ratio zones pertaining to the generation of the BFS ordered tree in memory: to the left, we see the first phase of node creation, with an average hit ratio of over 90%; to the right, we see the second phase with a hit ratio of only about 50%. In both phases, the algorithm moves through the memory allocation very uniformly.

During the DFS portion of the execution, we see roughly a 70% hit ratio with a well contained local range surrounding 70% throughout the run. Considering the reconstruction pattern between a BFS and a DFS node ordering, this seems relatively high. Presumably, what is happening is that each of the bottom levels of the tree represents one stream as detected by the prefetcher. The prefetcher has a maximum limit to the stride lengths that it will detect, so the higher levels in the tree will not be detected as representing uniform stride distances, but the bottom levels will, as each level has all of its nodes at a uniform distance. So roughly 70% of the nodes are below the highest detected level.

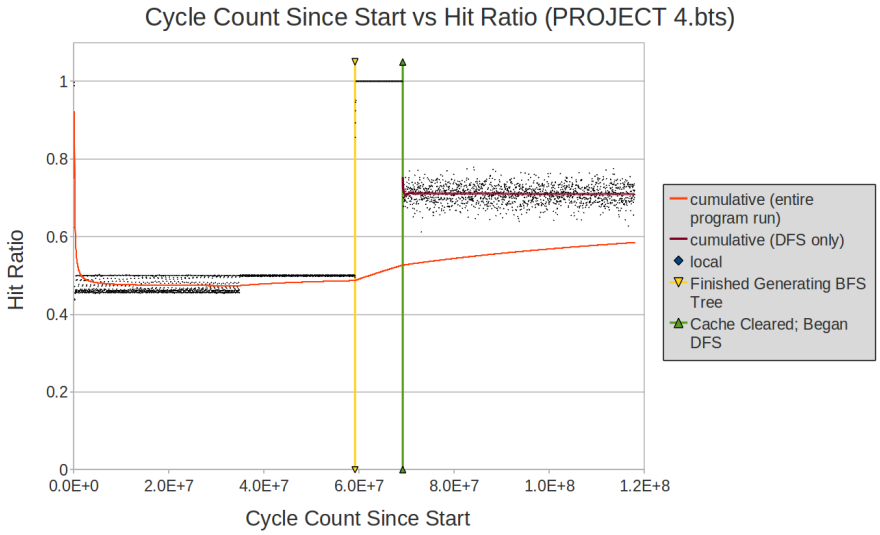

Figure 6.7: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the PROJECT 4.bts benchmark. Here we see around a 50% hit ratio during the original construction of the tree. This is somewhat surprising, since the majority of the work done during this time amounts to an array traversal. The working hypothesis is that the processor simply outran the prefetcher since it made so many requests so frequently. During the DFS, we actually see a higher hit ratio of over 70%. This is a particularly high hit ratio. In this case, the processor has more work to do outside of waiting for memory requests to be serviced, so it is less likely to out run the prefetcher. Also, since the tree is laid out in BFS-order, each level of the tree will have all of its nodes placed in memory at a consistent distance from eachother, and can thus be prefetched for as a unique request stream. The prefetcher is able to see a DFS on a BFS-ordered tree as a series of strided memory accesses. Since the prefetcher is able to detect up to 32 request streams, it should be able to prefetch for up to 32 levels in the tree, or until the first level in which the nodes are farther apart than the prefetcher's maximum allowable stride distance.

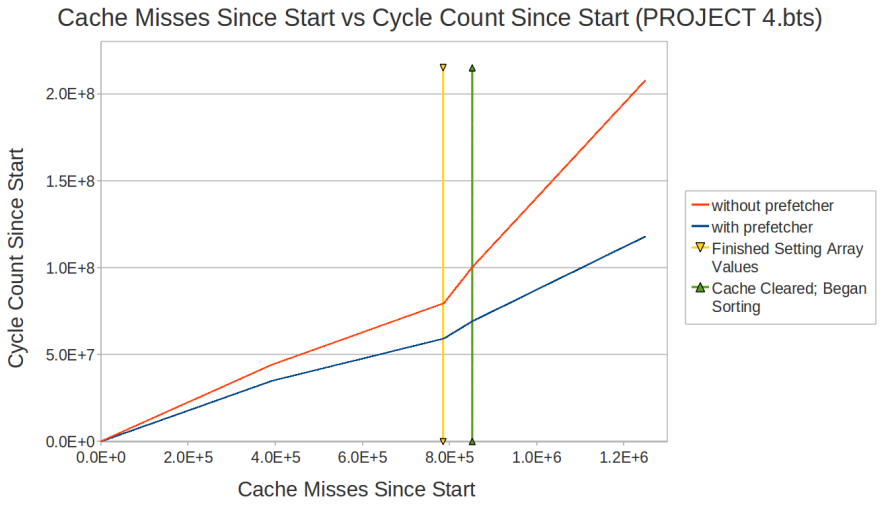

Figure 6.8: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the PROJECT 4.bts benchmark. Here we see more than a 2x speedup during the DFS portion of the execution. This speedup is much more than the 1/3x speedup we see during the construction of the tree. The two suspected culprits, aside from the higher hit ratio are, again, that more time would have been spent performing computations and hitting in the cache in the first section and less time waiting for memory requests, and that the more accurate prefetcher would have had impeded the processor less by making fewer extraneous prefetch requests.

6.5.2 SPEC CINT2000 Benchmarks

Each of the CINT2000 benchmarks was run for between one hour and several hours. Since the simulator was designed to simulate everything that happens in the processor, running them to completion could have taken days, and would have been infeasible. This makes the results difficult to interpret, since it is difficult to determine how far through the central algorithm of the benchmarks the simulation ultimately proceeded. Much of the early data comes from the startup portion of the benchmark programs.

175.vpr

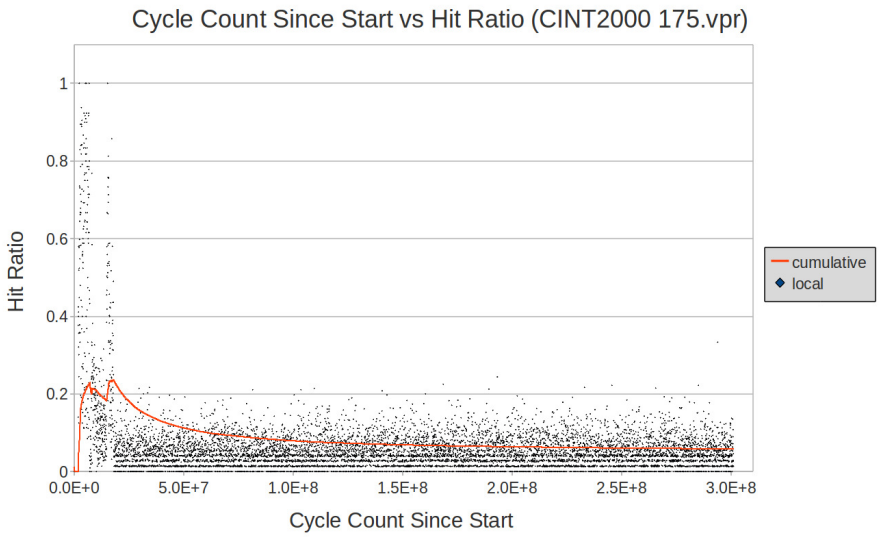

Figure 6.9: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the CINT2000 175.vpr benchmark. The prefetcher started strongly with a few segments of 100% hit ratio and an average hit ratio of about 20%, but after that initial success, it moved to a more consistent ratio averaging about 3%.

CINT2000 175.vpr received some but little help from the prefetcher during this initial portion of its execution. It turned out an overall hit ratio of about 6%. The graph runs only to a place where the hit ratio settles, but this benchmark was run for its entire

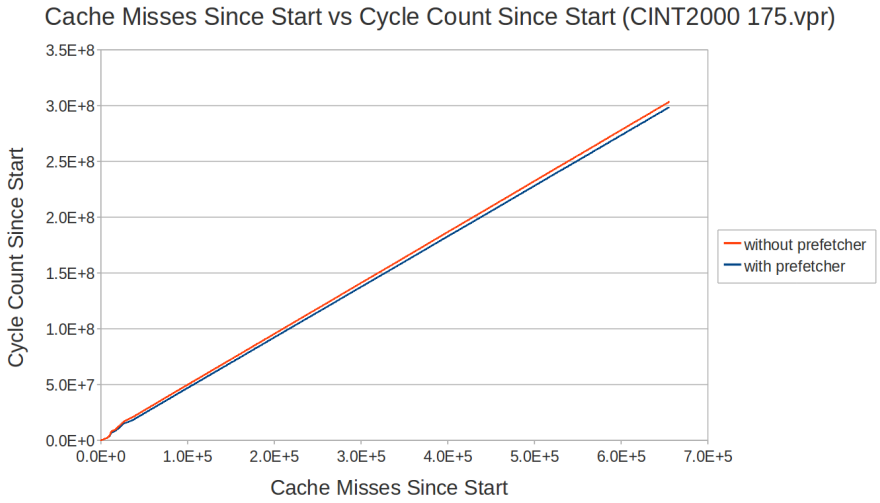

Figure 6.10: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the CINT2000 175.vpr benchmark. The prefetcher made its most substantial gains early on, and then with only a 3% hit ratio, it ended producing around a 2% cycle savings. While the prefetcher did not perform particularly strongly on this benchmark, it did ultimately save more cycles than it cost.

duration, and the recorded hit ratio for the benchmarks entire duration was 6.57%. Use of the prefetcher resulted in a slight decrease in overall cycles taken during the illustrated portion of the execution.

181.mcf

The 181.mcf benchmark performed the worst of any of the CINT2000 series during the measured portion of its run. During this portion of its run, it appears to have made almost no uniform memory accesses. This benchmarks serves as an example of the overhead of the prefetcher on applications which experience low hit rates while still inducing inaccurate prefetches by the prefetcher. As evidenced in Figure 6.12, without the prefetcher, the program actually ran faster to the end of the illustrated execution portion. This likely resulted from the processor having to wait for the prefetcher to finish its inaccurate fetches.

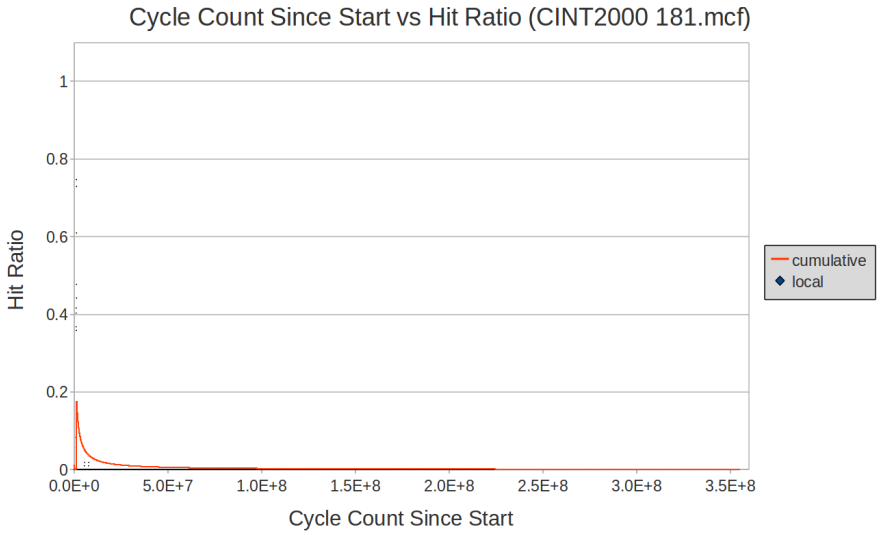

Figure 6.11: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the CINT2000 181.mcf benchmark. This graph depicts the prefetcher's worst performance on a selected benchmark. It initially had a few hits, but ultimately the hit ratio degraded to nearly 0%. This was presumably because 181.mcf performed nearly no standard stride length memory stream access.

256.bzip2

The CINT2000 256.bzip2 benchmark should find the the prefetcher relatively useful, since it moves over large swaths of data in a uniform manner in order to feed it to its compression algorithm.

The CINT2000 256.bzip2 benchmark was run for the longest period of any of the benchmarks used; thus, we see several distinct memory access regions. As expected, it has performed well in some of these regions. However, as is the case with many of these benchmarks, we do not know exactly what has gone on during these portions of the bzip2 run, so it is difficult to extrapolate what these numbers mean.

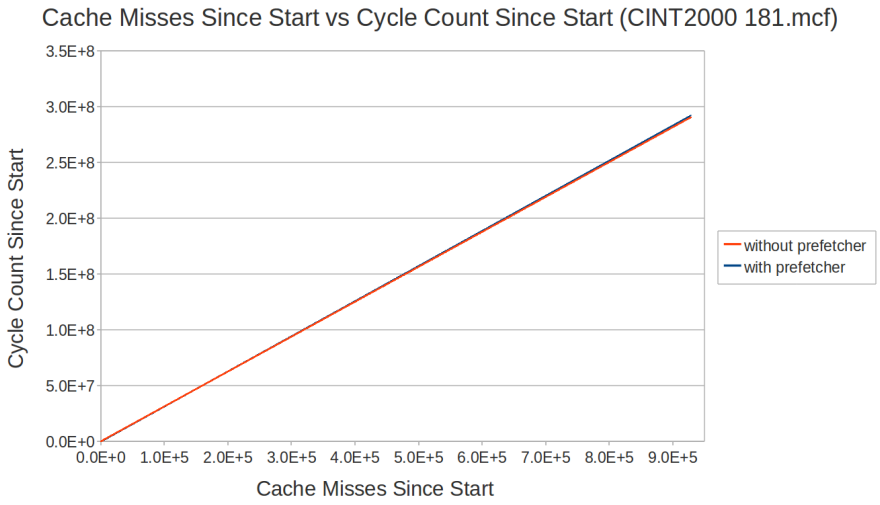

Figure 6.12: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the CINT2000 181.mcf benchmark. This graph is illustrative of a very bad case for the prefetcher. It shows how prefetcher overhead (due to processor/prefetcher memory contention) can actually make a run go more slowly if the hit ratio is very low. In this case, we see less than a 1% deficit due to prefetcher overhead, even with nearly a 0% overall hit ratio. Compared to the savings we saw in some of the PROJECT benchmarks, this is a low deficit.

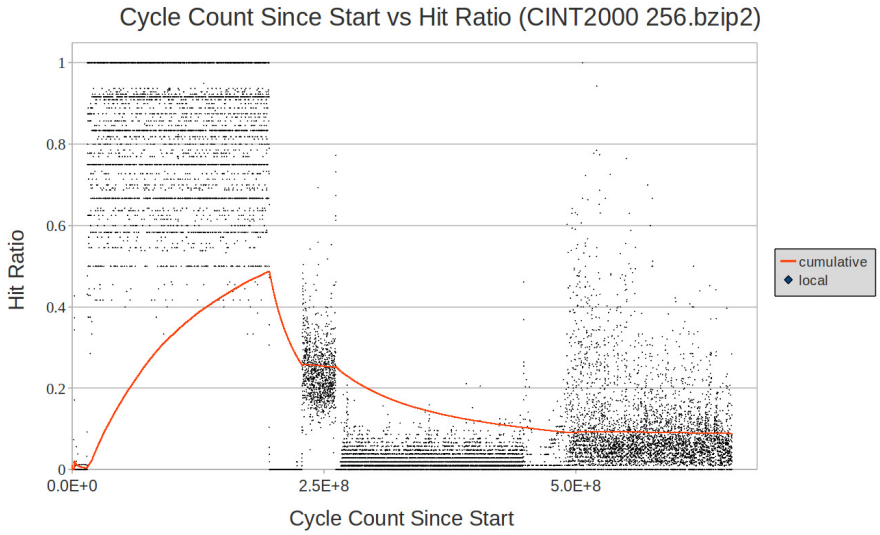

Figure 6.13: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the CINT2000 256.bzip2 benchmark. Here we see the prefetcher's best overall performance on any of the selected CINT2000 benchmarks; this is not a surprise, since the benchmark in question traverses larges swaths of data in order to compress them, presumably offering the prefetcher a set of standard stride length memory access streams to detect. We also see strongly defined partitioning of the prefetcher's performance; again, this is expected since this benchmark performs a series of differing jobs at different times (such as reading files, setting up different arrays, executing the compression algorithm ,etc.).

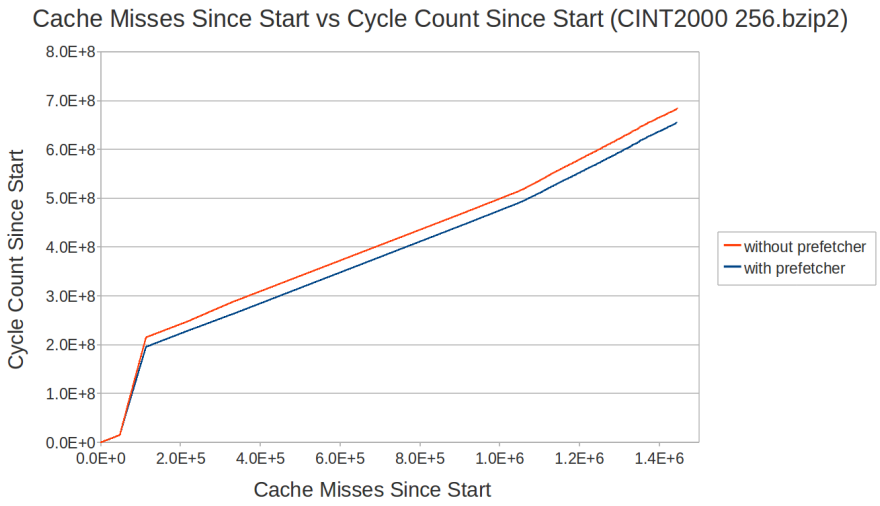

Figure 6.14: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the CINT2000 256.bzip2 benchmark. Here we see around a 4% overall cycle savings. This is lower than what we saw for some of the PROJECT benchmarks, but it is significant in this benchmark is more diverse in the types of work that it performs during its run (it has to read files, for example).

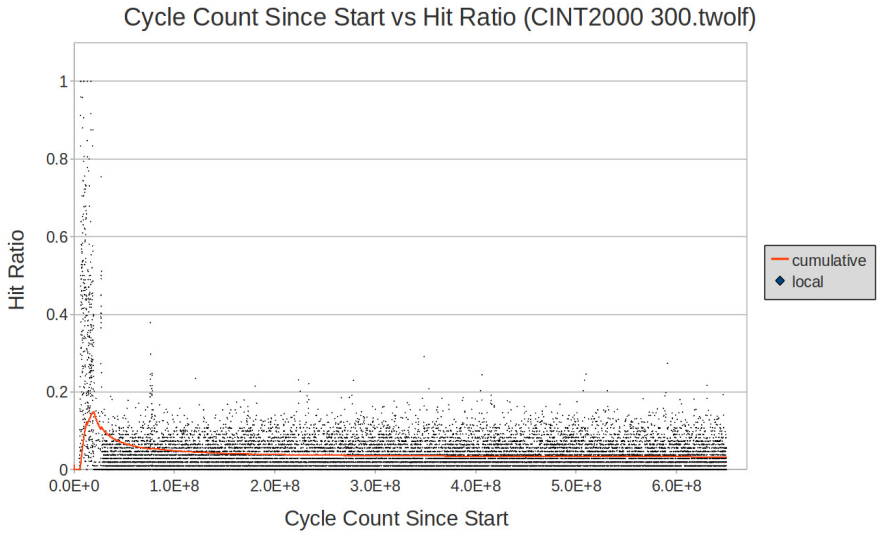

Figure 6.15: A graph depicting the local and cumulative prefetcher buffer hit ratios as a function of the number of elapsed cycles during a run of the CINT2000 300.twolf benchmark. Here we see a similar profile to that of CINT2000 175.vpr (see Figure 6.9): beginning more strongly and then teetering off into the single digits. While it is not a terribly strong showing, the prefetcher is still managing to make some gains.

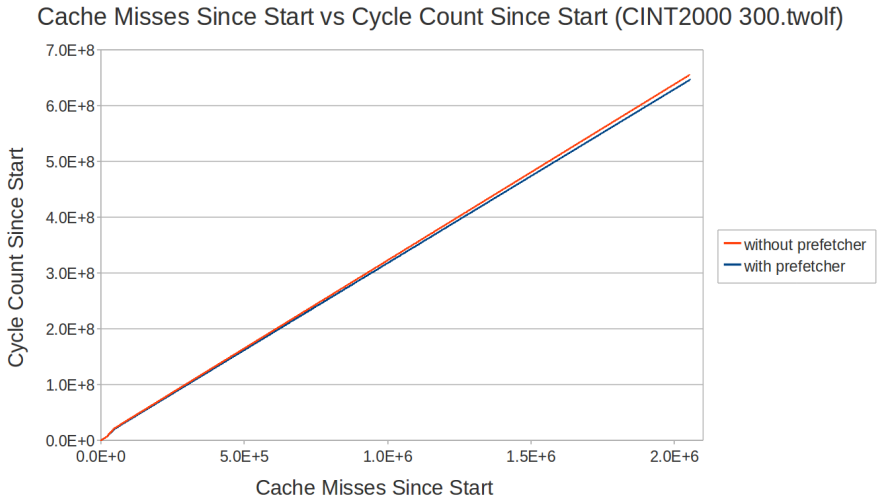

Figure 6.16: A graph depicting the number of cycles taken to reach each unique cache miss location during a run of the CINT2000 300.twolf benchmark. Similarly to what was seen in CINT2000 175.vpr (see Figure 6.10), we see a several percent gain in performance here. However, the majority of the gain comes at a point later in the run, seeming to indicate that the profile of processor to memory latency cycles spent differs between the two benchmarks.

300.twolf

CINT2000 300.twolf benchmark performed very similarly to CINT2000 175.vpr during the portion for which results were recorded. Again, it is hard to extrapolate further given this snippet of early results form the run.

6.5.3 Other Notes

Chapter 7

Conclusions

This project surveyed the current state of prefetcher technology, and illustrated a prefetcher design concept, intended to be used with UCSD's GreenDroid microprocessor. It went on to discuss the ways in which this concept was refined to work specifically with the layout of GreenDroid, and the process by which a prototype version of this prefetcher was simulated in the GreenDroid cycle accurate simulator. It also discussed related work, comparing and contrasting it to this project's design and illustrating ways in which some of these related concepts could be used alongside this project's design. Finally, in this section, the project concludes with a discussion of the simulation results and suggestions for future work.

7.1 Evaluation of Simulation Results

The effectiveness of a prefetcher would ultimately be largely dependent on the relative performance of the GreenDroid processor and its DRAM. The system tested found significant speedups over the Null prefetcher for the PROJECT benchmarks,

which ran to completion performing common computational tasks using standard algorithms to do so. By contrast, it found modest speedups for most of the SPEC CINT2000 benchmarks on which it was run. For the CINT2000 181.mcf benchmark, on which it had a very low hit rate, it decreased overall performance by a very slight margin.

This prefetcher has a small footprint and is easily integrable into the GreenDroid system. Its address calculation logic requires no information except for the cache miss address stream. In simulation, it exhibited high hit ratios and significant performance improvements for certain common computational tasks. It also made small performance improvements to most of the CINT2000 benchmarks tested and made a slight performance decrease for one of them.

7.2 Future Work

One note of concern is that when this prefetcher is being inaccurate, it can make a lot of extraneous prefetch requests. This could be a problem for a low energy system, since these requests effectively require the memory system to do more work than would otherwise be necessary. One way to deal with this situation might be to include a saturating counter which would keep track of the accuracy of the prefetcher's requests. When the counter were positive, the prefetcher would actually issue requests, and when it were negative, the prefetcher would calculate but not issue requests. This could also assist in keeping the prefetcher out of the way of the processor in situations where its accuracy were low (see the results of the SPEC CINT2000 181.mcf simulation).

Bibliography

[1] Hassan Al-Sukhni, James Holt, Daniel A. Connors, Mike Snyder, Matt Smittle, and Brian Grayson. The design of cost-effective stride-prefetching for modern processors. Technical report, Freescale Semiconductor, Inc. Austin, TX and Dept. of Electrical and Computer Engineering University of Colorado at Boulder.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] M. Annavaram, J.M. Patel, and E.S. Davidson. Data prefetching by dependence graph precomputation. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 52–61, 2001.

[4] J.-L. Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 176–186, nov. 1991.

[5] R. Bakalash and Zhong Xu. A barrel shift microsystem for parallel processing. In *Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on*, pages 223–229, nov 1990.

[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, feb. 2008.

[7] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 20(6):26–44, nov/dec 2000.

[8] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[10] J.D. Collins, Hong Wang, D.M. Tullsen, C. Hughes, Yong-Fong Lee, D. Lavery, and J.P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 14–25, 2001.

[11] Michael Dubois and Yong Ho Song. Assisted execution. Technical report, Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, California 90089-2562, October 1998.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[13] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, page 47, nov. 2004.

[14] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28:67–70, March 1996.

[15] M.J. Flynn, P. Hung, and K.W. Rudd. Deep submicron microprocessor design issues. *Micro, IEEE*, 19(4):11–22, jul-aug 1999.

[16] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 102–110, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[17] A. Garg and M.C. Huang. A performance-correctness explicitly-decoupled architecture. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 306–317, nov. 2008.

[18] M. Golden, S. Arekapudi, G. Dabney, M. Haertel, S. Hale, L. Herlinger, Y. Kim, K. McGrath, V. Palisetti, and M. Singh. A 2.6ghz dual-core 64bx86 microprocessor with ddr2 memory support. In *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*, pages 325–332, feb. 2006.

[19] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, march-april 2011.

[20] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 31(4):6–15, july-aug. 2011.

[21] H.P. Hofstee. Power-constrained microprocessor design. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 14–16, 2002.

[22] H.P. Hofstee. Power efficient processor architecture and the cell processor. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262, feb. 2005.

[23] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 1–11, New York, NY, USA, 2004. ACM.

[24] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 252–263, New York, NY, USA, 1997. ACM.

[25] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373, may 1990.

[26] R. Kalla, Balaram Sinharoy, and J.M. Tendler. Ibm power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, mar-apr 2004.

[27] D. Kim, S.S.-W. Liao, P.H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J.P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 27–38, march 2004.

[28] T. Kuroda. Cmos design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6–7, 2001.

[29] Wei-Fen Lin, S.K. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 301–312, 2001.

[30] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, Wisconsin 53706, May 1997.

[31] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 40–51, 2001.

[32] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread itanium processor. *Micro, IEEE*, 25(2):10–20, march-april 2005.

[33] G E Moore. Cramming more components onto integrated circuits, electronics, volume 38, number 8, april 19, 1965. *Also published online at ftpdownload intel comresearchsiliconmoorespaper pdf last visited on*, 7:2005, 1965.

[34] 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-V, pages 62–73, New York, NY, USA, 1992. ACM.

[35] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, march-april 2010.

[36] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, may 2008.

[37] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[38] Lu Peng, Jih-Kwon Peir, T.K. Prakash, Yen-Kuang Chen, and D. Koppelman. Memory performance and scalability of intel's and amd's dual-core processors: A case study. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 55–64, april 2007.

[39] Scott Ricketts. Efficient cache-coherent migration for heterogeneous coprocessors in dark silicon limited technology. Master's thesis, UCSD, 2011.

[40] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 37–48, 2001.

[41] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, R.D. Limaye, and S. Vora. A 65-nm dual-core multithreaded xeon reg; processor with 16-mb l3 cache. *Solid-State Circuits, IEEE Journal of*, 42(1):17–25, jan. 2007.

[42] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[43] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski, and P.G. Emma. Optimizing pipelines for power and performance. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 333–344, 2002.

[44] S. Swanson and M.B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *Communications Magazine, IEEE*, 49(4):112–119, april 2011.

[45] Michael Bedford Taylor. *Tiled microprocessors*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007. AAI0818397.

[46] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 205–218, New York, NY, USA, 2010. ACM.

[47] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 144–155, New York, NY, USA, 2004. ACM.

[48] P.H. Wang, J.D. Collins, Dongkeun Kim, B. Greene, Kai-Ming Chan, A.B. Yunus, T. Sych, S.F. Moore, J.P. Shen, and Hong Wang. Helper threads via virtual multithreading. *Micro, IEEE*, 24(6):74–82, nov.-dec. 2004.

[49] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 2–13, 2001.