Co-Locating Code and Data for Energy-Efficient CPUs

by

Mark J. Sutherland

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Mark J. Sutherland
Abstract
Co-Locating Code and Data for Energy-Efficient CPUs

Mark J. Sutherland
Master of Applied Science
Graduate Department of The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2016

The end of Dennard scaling has brought energy savings to the forefront of processor design. When coupled with massive datasets, traditional memory architectures are becoming a source of excess energy expenditure. In this thesis, we propose to conserve energy through the co-location of code and data, which limits data movement on and off the chip. To that end, we design a data prefetcher called Tempo that reduces useless prefetches by 18% and increases timeliness by 43%. Tempo boosts the performance of the state of the art Spatial Memory Streaming prefetcher by up to 20% (2.54% average). In contrast to data prefetching, we also describe an execution paradigm called Anti-Fetching, which dynamically moves code off-chip to a Near-Data Processor. Using Anti-Fetching on a set of graph processing benchmarks provides up to 31% energy savings in the memory hierarchy (11% average) over an efficient CPU baseline.
This thesis is dedicated to my fellow saints at Trinity Grace Church. All of you belong to my first family in the Lord Jesus Christ our Saviour, and I would not have been able to complete this thesis without your friendship, support, and prayer.
Acknowledgements

First and foremost, I want to acknowledge the guidance and contributions of my supervisor Natalie Enright Jerger. Working with her has been an honour and a pleasure, and I want to thank her for teaching me a huge amount about the exciting field of computer architecture, while listening to and tolerating my many crazy ideas. I also wish to thank my advisory committee - Profs. Jason Anderson, Andreas Moshovos, and Frank Kschischang for their valuable feedback on this work.

Everyone in the NEJ and AM groups, thank you so much for all the contributions you have made to my MASc studies here - both knowingly and unknowingly. I want to extend a special thanks to Joshua San Miguel: I appreciated every minute we spent discussing our evaluations, results, and future ideas. I’m very thankful for having a successful like-minded colleague at the next desk who was always willing to provide a quick sanity check of my results, and then inevitably engage in our own real-life version of intellectual reductionism as our discussion quickly turned to the nature of truth and reality.

Sam(wise), your friendship throughout this 2 years has been irreplaceable. Thanks for all the company, shared meals, and conversation after stressful weeks or close to deadlines - I’m going to miss you. Remember “…the thought, clear and cold, that in the end the Shadow [is] only a small and passing thing: there [is] light and high beauty forever beyond its reach.” One day, I hope to be able to say the words “I am glad you are here with me. Here, at the end of all things Sam.”

Finally, I wish to thank my Mom, Dad, and Uncle Russ for encouraging me to continue pursuing academic studies, and being an encouragement to me all of these years. You are some of my closest friends, and I would not have been able to do this without you.
List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Simulation configurations provided by DPC-2</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>IPC for All Prefetchers on mcf</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Prefetch Hits and MSHR Hits in mcf</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Hardware Storage Requirements for Tempo</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison of hardware implementations of the AGT and PHT</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison of Multiplier Types</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>Hardware Costs for Dynamic Degree Controller based on AMAT</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Parameters Used to Simulate Anti-Fetching</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Cache Parameters Obtained from ARM Cortex A7 [5] and Intel Core (model codes-17H) [37]</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of predicted vs. measured prediction accuracy for achieving break-even total energy</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Full-Application Basic Block Table Hit Rates</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Average Number of Live-in Registers and Bus Transactions Per Offload in cascades, centrality, and kronem</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Latencies of a Typical CPU Memory Hierarchy in ns ........................................ 4
2.2 Basic Floorplan of a DRAM Array ............................................................... 8
2.3 Instruction Level Parallelism in an Out-of-Order x86 Processor .................... 11

3.1 Example of repeated access patterns in an OS page. Reproduced from the original SMS authors [75] ............................................................. 13
3.2 Useless and Untimely prefetches issued by SMS ........................................... 14
3.3 Active Generation Table in SMS, comprised of Accumulation and Filter tables [75] .. 14
3.4 Example access pattern that shows multiple generations, as well as overlapping patterns. [75] 15
3.5 Block Diagram of the Pattern History Table, showing how prefetches are generated from combining a trigger access and existing spatial pattern [75] .................... 15
3.6 Histogram of prefetches residing in each spatial pattern for our SMS baseline ........ 16
3.7 Example repetitive access pattern in time, observed from mcf .......................... 17
3.8 Decomposition of a PHT entry into multiple tempo-based patterns .................. 17
3.9 Updating the AGT based on a hash of the local tempo buffer ......................... 18
3.10 Tempo Prefetching Walkthrough .................................................................. 19
3.11 IPC for SPEC CPU2006 benchmarks ......................................................... 21
3.12 Useless and Untimely prefetch comparisons between SMS and Tempo ................ 22
3.13 IPC for selected SPEC CPU2006 benchmarks, measured in low bandwidth mode . 23
3.14 Percentage Improvements in Accuracy and Coverage for Tempo .................. 23
3.15 Histograms of possible and issued prefetches in mcf. We do not include the trigger access in the bins on the X axis ................................................... 24
3.16 Speedup using the Dynamic Prefetch Degree Controller of §3.3.2 ................... 25
3.17 Hardware Block Diagram for Dynamic AMAT Calculation Described in §3.3.2 .... 27

4.1 Comparison of Data Layouts in Memory ....................................................... 32
4.2 Cache Miss Rates Compared for SNAP and PARSEC Applications ................. 33
4.3 Cache Lines Exhibiting Zero Reuse in SNAP Benchmarks ............................... 33
4.4 Frequency of Top Basic Blocks in SNAP Applications .................................... 34
4.5 Basic Block Latency Breakdown for netstat ............................................... 34
4.6 Overall System Block Diagram ..................................................................... 35
4.7 Modified Processor Core Diagram. New hardware structures are shaded with a pattern 36
4.8 Training Live In/Out Values in a sample BBL from netstat ......................... 37
4.9 Training a Bloom Predictor based on Cache Reuse ...................................... 39
Chapter 1

Introduction

The field of computer architecture is currently in an exciting transitional era; in recent years, computers have grown in complexity, having undergone a transformation from incrementally improving general purpose processors to more complex heterogeneous systems that potentially include specialized accelerators, approximate machines, or mixed signal co-processors. This shift is primarily due to the failure of physical technology scaling, combined with the difficulties presented by the massive data-sets that characterize many of today’s applications. In the past, every process node shrink invariably gave architects access to twice as many transistors and approximately 40% increase in frequency, all while maintaining the same total chip power. However, this scaling recipe, termed Dennard’s Law [21] has broken down due to fundamental limits on total chip power, particularly due to sub-threshold leakage constraints that are inherent with low supply voltage [12].

In addition to the end of supply voltage scaling, single-chip microprocessors have created the dark silicon problem [25], which means that a significant portion of the chip must be powered off or left dark to keep the power consumption within the proper envelope. Using all of the transistors on-chip would lead to overheating and permanently damaging the chip itself; every modern architecture is constrained in this way. Hence, hardware systems are taking on polarized characteristics, either scaling up towards complex warehouse-scale clusters [8] that rely on massive amounts of parallelism, or specialized devices connected to the “Internet of Things”. Although both of these systems have provided a way for designers to continue offering performance gains despite technology restraints, they still have one major problem: dealing with the exponential scaling of data being collected from sensors and/or uploaded by users. This trend is referred to as the rise of big data.

Common big data workloads include video encoding, natural language processing (such as Apple’s Siri and Microsoft’s Cortana), and neural networks; these applications are memory bound due to their performance being constrained by limitations in data throughput rather than on-chip computational power. Models of microprocessor performance from Borkar and Chien have separated out the 1000x performance gap between the 1.5µ processors of the 1980’s and the 32nm processors of today, and shown that only 10x of that has been due to noteworthy advances in microarchitecture [13]. Additionally, they demonstrate that dedicating huge fractions of die space to large caches is a necessary trend in standard uniprocessor designs, because it is more power-efficient than complex energy-intensive techniques that often have limited performance gains. Although their conclusions are consistent with current industry processors [20], [37], cache sizes cannot match the growth rates of user data, requiring designers to search
for alternative solutions when dealing with large data footprints.

This problem is exacerbated by the gap between the speed of main memory and the speed of on-chip computing. Classically referred to as the memory wall, this trend implies that the number of cycles to access memory in the case of a cache miss scales exponentially over time [83]. Prefetching is a well-established technique that attempts to address the memory wall issue, by speculatively loading the caches in advance with data that we think will be used by the processor. If the prefetcher predicts the correct data far enough in advance, the processor will not incur the penalty of a cache miss. Accurately predicting the memory access patterns of programs is an ongoing research problem to this day, with large potential upsides as well as challenges in limiting the penalties due to mispredictions. Although prefetching can effectively limit memory latency when done properly, it does not address the dark silicon or big data issues, since it assumes that all data transformation and compute is going to be performed in the microprocessor pipeline.

A re-emerging technique termed near-data processing (NDP) addresses all three previously mentioned trends - dark silicon, big data, and the memory wall - by executing simple computation patterns off the chip in a decentralized fashion. Using NDP is an efficient design technique to minimize on chip power, because of the large raw cost of actually moving data across the chip pins [44]. We believe that using NDP provides a scalable long term solution that removes latency and throughput pressure on the memory system, as well as enables systems designers to match their compute capabilities to the pace of data growth.

This thesis focuses on limiting data movement by co-locating the data and code that form a program through two separate mechanisms: firstly, an improved data prefetcher that speculatively loads on chip caches without saturating the memory system. Secondly, a hardware architecture for dynamically selecting code that would benefit from being executed near memory, and offloading it to a simple near data processor. Our architecture enables general purpose cores to share work with the NDP units, saving energy in the memory hierarchy by reducing movement, and requires no change to application binaries. Both designs share the primary goal of reducing the amount of energy spent moving data on and off chip, which we believe is a key goal given the energy constrained state of systems-on-chip.

1.1 Overall Contributions

In this work, we have accomplished the following key contributions:

- Design a data prefetcher that uses a combination of spatial and temporal information to significantly reduce untimely prefetches in a uniprocessor environment.

- Evaluate our data prefetcher using the SPEC CPU2006 workloads, and perform a detailed benchmark study that demonstrates the specific improvements we make.

- Profile the Stanford Network Analysis Platform (SNAP) graph processing workloads and demonstrate an opportunity to improve energy efficiency through utilizing NDP.

- Implement a novel NDP architecture that selects code from existing binaries to offload to be executed near memory.

- Use both oracle and realistic methods to evaluate the energy savings and performance for our NDP architecture.
Chapter 1. Introduction

This thesis has led to a publication in the program of the 2nd Data Prefetching Championship \cite{77} and a paper currently under submission to the IEEE Symposium on High Performance Computer Architecture.

1.2 Organization

We divide this document into 5 chapters including this one. In Chapter 2 we discuss the relevant background in the fields of prefetching, and near data processing, as well as compare our design(s) to related contributions that already exist in the literature. Chapter 3 describes Tempo, our hybrid design data prefetcher. We show how our design reduces prefetches that arrive at the caches either too late to cover demand misses fully, or are not used at all. In Chapter 4 we describe our NDP architecture and evaluate its performance and energy tradeoffs. Finally, Chapter 5 concludes and discusses possible future directions for this work.
Chapter 2

Background and Related Work

The canonical microprocessor memory system is structured as shown in Figure 2.1, where moving towards the base of the pyramid trades access latency for storage capacity. All operations take place on-chip, with data being provided from and being written to a very small number of registers. These registers can operate at the same speed as the processor itself. Since most processors only contain between 32 and 192 registers, data that overflows the register file is kept in on-chip caches that vary in size and management policy. Modern processors have up to three levels of cache, with the common feature that all data supplied to the registers is supplied by the first level (L1) cache, which rarely exceeds 64kB in size \[20\] [42]. At this point in the hierarchy, there is a significant latency gap - the first, second (L2), and last-level (LLC) caches often have access times between 2 and 20 cycles \[37\], whereas loading data from the Random Access Memory (RAM) can take up to 100 CPU cycles, assuming a row buffer miss and a 2:1 clock frequency gap between the CPU and DRAM \[54\].

This technique of caching data works well when the target application executes many reads that are coupled together. The classic example of coupled reads occurs when a program decides to scan and accumulate all of the values in a long list. Since multiple values can fit into the same cache entry, the cost of a cache miss on the first element is amortized over the rest of the elements to this cache entry, since they all incur hits.

![Figure 2.1: Latencies of a Typical CPU Memory Hierarchy in ns](image-url)
In contrast, programs that have poor cache locality will not only execute very slowly, but also cause the chip to wastefully expend energy moving data back and forth between the RAM and the on-chip caches. Architects have proposed many different solutions to bridge the latency gap between the processor and the memory, which generally can be placed into two categories: either improving the characteristics of the on-chip caches themselves, or proposing a method where the latency of accessing memory is overlapped with other critical-path computations and therefore appears smaller. The on-chip caches can be improved by better replacement policies [69] or relaxing inclusivity requirements [40], whereas other methods such as the Load Slice Core [17] focus on issuing as many misses in parallel as the program allows. The first method we discuss is called prefetching, and falls into the second category.

2.1 Data Prefetching

At its core, prefetching is a simple idea, if one can predict the next addresses that the CPU will access, then one can ensure that the data stored there is pre-loaded into the caches. Therefore the main memory latency is, in full or in part, hidden from the program because the data was fetched before the processor actually needed it. Although many prefetchers boast impressive performance gains, over-prefetching is dangerous, as these speculative loads have the potential to evict useful data from the caches and increase the number of memory reads required. In this section, we provide a short primer on key prefetching terminology and survey the existing corpus of prefetching techniques.

2.1.1 Key Terminology

**Miss Status Holding Register (MSHR):** A structure that tracks cache line information when a miss occurs, but before the data is filled into the cache from the lower levels of the memory hierarchy. Using MSHRs allows the cache to store the required state about this missed line and process other misses behind it in the request queue.

**Prefetch Hit:** A cache hit that is directly induced by the line being loaded in advance by the prefetcher. These hits completely hide the latency of all levels in the hierarchy beneath this one. A prefetch hit is also referred to as a *timely prefetch*.

**Outstanding Prefetch Hit:** A prefetch that has been issued to the lower level of the memory hierarchy (e.g. from the L2 cache to the LLC), but has not yet returned the correct data. All prefetches need to occupy an MSHR to correctly be handled by the memory system. The key difference between an outstanding hit and a regular prefetch hit is that an outstanding hit only hides a portion of the lower level access latency. Outstanding prefetch hits are also referred to as *late prefetches*.

**Useless Prefetch:** These types of prefetches are issued to the cache controller, but either never incur a reference by the core before being evicted from the cache, or are never actually completed due to MSHR overflow. The cost paid for issuing useless prefetches is excess off-chip bandwidth to fetch the data, and energy to write it into the local cache. If the cache controller drops these prefetches, we do not incur the bandwidth cost, but have increased pressure on shared on-chip resources such as MSHRs.

**Cache Pollution:** A prefetcher can cause cache pollution by issuing prefetches that evict useful data from the local cache. In this case, we incur the energy cost for both the prefetch and re-fetching the data that was evicted. Cache pollution is harmful to the execution time of the program.

**Prefetch Degree:** The number of prefetch requests issued for each regular demand miss. Adjusting this number allows the systems architect to control how much load a prefetcher places on the memory.
system supporting it.

**Prefetcher Accuracy:** The accuracy of a prefetcher is defined as: 
\[ Acc = \frac{N_{pHits}}{N_{prefetches}} \]
where \(N_{pHits}\) represents the number of cache hits to prefetched blocks, and \(N_{prefetches}\) is the total number of prefetches issued.

**Prefetcher Coverage:** The coverage of a prefetcher is defined as: 
\[ Cov = \frac{N_{pHits}}{N_{misses}} \]
where \(N_{pHits}\) represents the number of cache hits to prefetched blocks, and \(N_{misses}\) is the total number of misses seen by the cache. This metric shows the fraction of total cache misses that the prefetcher is able to eliminate.

### 2.1.2 Data Prefetching Techniques

The three most common pieces of information that prefetchers use to make predictions are: each memory instruction’s program counter, patterns within the stream of addresses being referenced by the CPU, and the order those accesses follow. We refer to these three sources of information as **PC**, **spatial**, and **temporal**, respectively. In general, most prefetchers are primarily spatial, relying on the fundamental observation that most data is stored and accessed in regular patterns. It is also less costly to store spatial than temporal information, because spatial patterns can more easily be expressed relative to an arbitrary base.

The classical Stride prefetcher [18] is the most well known example of a purely spatial prefetcher, which has been included in commercial architectures such as IBM POWER4/5 [78, 41] and Intel Nehalem [35] due to its simplicity. Although stride designs can capture simple constant offset access patterns with very limited hardware requirements, they are easily defeated by more complex memory behaviour.

On the temporal side, the Global History Buffer (GHB) and spin-off designs [58, 22] keep a circular buffer of previous memory references, and periodically scan through it for patterns. If a match is detected in the current processor execution stream, the prefetcher then issues fetches to the future blocks. The addresses of the future blocks vary, and can either be based on absolute virtual address, or the difference between virtual addresses. This is an important distinction, as the authors of GHB show correlations based on exact addresses are rare [59], whereas using relative address offsets yields much higher coverage.

More complex data prefetchers make use of additional information such as the application’s code or the processor’s memory space. Two of these designs are Linearized Irregular Access Correlation [39] and Spatio-Temporal Streaming [74]. Both of these prefetchers store megabytes of metadata to capture and search through extremely long address streams. These techniques must virtualize their prediction data and store it in the off-chip memory in order to provide high prediction accuracy, and have tie-ins with the operating system’s page management algorithm as well as the processor’s Translation Lookaside Buffer (TLB). Although extra storage footprint may be acceptable as memory systems continue to scale in capacity, techniques with virtualized metadata have seen limited adoption in industry.

Generally, prefetchers that use code correlation are designed to function with a specific type of workload, which provides a much smaller use case for the designer. We choose to survey these types of prefetchers because of the rise of what is termed the 10x10 optimization [13]. The 10x10 optimization is a design trend where instead of spending resources on expensive general purpose processors, architects might place 10 specialized accelerators on one chip, each of which take 10% of the overall execution time originally allocated to the general core. These processors often exhibit increasing heterogeneity and begin to share characteristics such as restricted memory spaces and/or explicit communication primitives. In a heterogeneous system where applications are tightly coupled to the cores, it is intuitive to design the
prefetchers to capture the workloads of these applications as well.

A particular example of this design trend is the Spatial Memory Streaming (SMS) prefetcher \[75\], which is specifically targeted to improve the performance of commercial and database workloads. This prefetcher uses both code correlated and spatial information to capture repeated access patterns that occur over much larger regions of memory. Not only can SMS predict very complex access patterns such as buffer allocation and tree traversal, the metadata that it stores does not scale with the working set of the application, a desirable feature when architecting systems for big-data workloads.

2.1.3 Designs Focusing on Timeliness

The success of any prefetcher is ultimately tied to how timely the prefetches are. No matter how sophisticated the mechanism or how high the accuracy, the prefetches must arrive at the memory system in time to overlap the memory access latency with the processor’s execution. Srinath et al. propose to use dynamically gathered statistics such as pollution and accuracy, as well as estimations for timeliness to improve the caching performance of a stream-based prefetcher \[76\]. Their work, Feedback Directed Prefetching (FDP), saves memory bandwidth by eliminating prefetches harmful to the system, and also by inserting lines with a low locality of reference into lower positions in the LRU stack. FDP serves as an additional control mechanism that can be applied to any prefetcher, and does not depend on any specific applications or use cases.

In contrast, BuMP is a server RAM prefetcher that identifies high-density DRAM pages and triggers bulk transfers into the DRAM’s open row \[80\]. Although this design does not target prefetch timeliness by attempting to minimize latency throughout the memory system, we discuss it because of the observation that memory transfers in the same physical page are strongly correlated with code that accesses them.

The design which bears the most similarity to our prefetching work is the Time-Aware Stride (TAS) prefetcher \[85\], which embeds miss counters in the memory access streams observed by a standard Stride prefetcher \[18\]. TAS allows the prefetcher to determine how far ahead of the current memory access to begin issuing prefetches, and even the capability to predict the accesses which are localized around the next program counter. The TAS prefetcher relies on constructing a past ordering of program counters and the time between observing those accesses. We distinguish our work by the fact that our design does not use absolute miss counters, and instead focuses on choosing the most accurate prefetches out of a series of accesses inside the same physical page.

Furthermore, prefetching in general is apathetic to how many times each piece of data will be used by the processor. In contrast to moving data onto the chip in advance, Near Data Processing (NDP) proposes to move certain segments of code off chip, and execute them in compute units placed near to the DRAM itself. The next section covers the key fundamentals of NDP.

2.2 Near Data Processing

Near Data Processing (NDP) is an old idea that originally was proposed to overcome pin and packaging limitations on multi-board computer systems of ages past. In that era, a memory access would require a board-to-board transfer, costing the processor thousands of idle cycles. However, due to the increases in on-chip integration enabled by Moore’s Law and fabrication limits, NDP saw limited adoption. As we previously discussed, the end of Dennard scaling when combined with the exponential trend in big data provides a strong motivation for designers to reconsider NDP.
Chapter 2. Background and Related Work

Figure 2.2: Basic Floorplan of a DRAM Array

2.2.1 Building Blocks for NDP

A memory cell can be built in a variety of ways, but the traditional design uses 1 read/write transistor to select the cell, and 1 capacitor to store the value. This is referred to as the 1T1C style. Combining these cells into a large array with row and column lines and sense amplifiers, as shown in Figure 2.2, is a way in which large libraries of storage can be built. Fundamentally, building a large DRAM array requires chip foundries to combine small CMOS logic for the selection transistors with physically larger capacitors, which require a large dielectric well and surrounding plates. This requires a complex fabrication process which fundamentally uses more difficult deposition and etching steps. However, existing DRAM chips already have solved the problem of combining the two processes, since the memory cells also need to be combined with traditional CMOS logic for the sense amplifiers, row buffers, and the controller itself.

Adding an NDP to this chip style is straightforward in concept - when designing the memory modules, architects will place compute units on the same chip as the DRAM itself, and provision low cost data pathways between the compute units and memory controller. This is done because the central principle of NDP is to restrict the amount of energy spent on moving data to the location where it is operated on. These compute units can be as simple or complex as required by the system designer, with examples in literature that range from single-bit boolean operations [31] to reconfigure FPGA-like logic blocks [61].

The challenge for systems architects is to determine the relative granularity at which NDP is most applicable, particularly when integrating NDP with existing full-stack architectures. The two approaches are generally divided by the amount of action that is required in software in order for a program to utilize the NDP. Different granularities have their own advantages and disadvantages. Architectures that are explicitly configured and invoked by the programmer often have much larger scope, with the ability to handle complex loops, functions, and possibly entire programs. In contrast, software-transparent architectures operate at a much finer granularity, and are invisible to the upper layers of the system stack.

The large design space for NDPS forms a continuum of different approaches, each with varying degrees of application support required to use them. This trade-off is very similar to the balance that exists between general purpose processors and custom accelerator logic. NDP architectures that target specific applications and require programmer intervention can obtain energy and performance improvements of orders of magnitude, but often require complex changes to multiple layers of the system stack, not limited
to the compiler, ISA, or operating system. General-purpose NDPs must work within the constraints that already exist inside commodity hardware, and must operate with less available information about the program that is currently executing. This means that although their raw improvements are less remarkable, any program that exhibits their target behaviour can benefit. In the rest of this section, we explain a few specific examples of both types of NDP, software-managed and transparent.

### 2.2.2 Software-Managed NDP

The very first NDP designs were simple and provided simple fixed functions; the Terasys NDP array [31] placed single-bit ALUs in line with the DRAM columns in order to execute bit-parallel operations with massive throughput. In order to use the Terasys array, programmers needed to transform their code from standard sequential code into a language the authors called bit-parallel C (dBC). dBC incorporates specific low-level knowledge into the programmer’s code, such as requiring them to specify the widths of variables and explicitly communicate between multiple in-memory processors. Additionally, these single-bit ALUs disrupt the regularity of the DRAM cell layout, leading to a lower overall density and a less attractive implementation overall.

UC Berkeley’s Intelligent RAM (IRAM) project [63, 15] pushed the frontiers by configuring the IRAM as a simple vector processor that is tightly coupled with the actual memory cells themselves. Active Pages [61] was the first design to place code fragments in physical memory locations close to the data to be operated on; this design placed FPGA-like logic blocks near the memory arrays, and used them to execute simple functions comprised of arithmetic and memory indirection operations. All of the above designs must explicitly be operated by directives from the programmer and compiler, and therefore require significant changes in the way programmers think about writing code. Modern NDP architectures have expanded into the realms of 3D stacking and logic layer integration [2, 3, 68] and significantly grown in scope - but still require changes to the programming model, and often the compiler and operating system as well. Additionally, many remove or restrict the use of useful abstractions such as virtual memory and unified addressing [84, 68, 2].

FlexRAM [43] was the first design to present a full stack implementation where the NDP was not simply used as a co-processor for special applications, but is designed to appear as a standard DRAM chip that has added functionality if the programmer wishes to use it. Although they also re-write target applications to best utilize their architecture, they preserve general-purpose execution in the core and do not introduce an entirely separate space of memory that is only enabled for NDP usage. This architecture serves as a middle ground between techniques that are purely software managed, and the techniques we will next discuss, which are invisible to the programmer.

### 2.2.3 Transparent NDP

Dynamic and invisible NDP systems are still an emerging research area in computer architecture. A recent proposal from Hashemi et al. [32] targets pointer chasing workloads in out-of-order (O3) processor cores, and dynamically transfers small traces of instructions to a compute-enabled memory controller. Their fundamental contribution is to identify the fact that communication delay forms a non-trivial fraction of memory access delay inside a large chip multiprocessor. Therefore, pointer chasing behaviour incurs a latency overhead while the first load result traverses through all levels of the cache hierarchy back to the core. They add a less aggressive out-of-order core to the memory controller, and use it to
execute the slice of instructions that depends on the first load, as soon as the result is available.

Other works in NDP literature focus on formalizing hardware support for memory coherence, consistency, and synchronization [29]. These systems are important to allow NDP systems to integrate with existing large-scale computing systems, where designers can expose certain portions of the NDP to end-users through APIs or virtualization. This is particularly relevant given recent interest in warehouse-scale computing which often employ tens of thousands of server processors and memory nodes.

### 2.2.4 Scale-Out Computing

A scale-out system is one that gains compute capacity along with the addition of more storage. This is a desirable trend that stands in contrast to traditional scale-up systems, where more storage only allows the system to handle more data, without the ability to process it faster. Modern datacenters are based around complex O3 processor architectures which closely tracked transistor scaling trends, and software evolution has tended towards boosting usability and redundancy rather than low-level performance.

In an attempt to provide applications that track the trend of warehouse-scale computing combined with big data, both industry [81] and academia [27, 47] have provided open-source suites to help evaluate and guide architecture trends. These workloads are distinguished by extremely large working sets, more complex communication patterns, and significant increases in the instruction footprint. Furthermore, modern processors are awkwardly provisioned to deal with the unique demands of scale-out computing [27, 42], and therefore, we are seeing significant shifts in microarchitecture that particularly effect the memory system.

Existing microarchitectures are designed to maximize instruction-level parallelism (ILP), which is a simple concept illustrated in Figure 2.3. We show a simple x86 fragment with two independent instruction slices, both of which involve memory accesses. Register dependencies are marked with arrows and text showing the register which holds the dependent value. Instructions $I_0$ through $I_2$ form a dependent chain through register dependencies, and we assume that $I_2$ incurs a main memory access. Although a traditional processor would be stalled by this event, using out-of-order issue (O3) and allowing multiple outstanding cache misses (memory-level parallelism) means that issuing $I_3$ and $I_5$ can take place independently while the memory system is waiting for the load into the $rdx$ to complete.

Classical architectures made use of the extra transistors available at every node shrink by increasing the instruction scope that the processor can use to schedule independent instructions. This trend has continued to the point where the instruction scheduler window has become large enough that it restricts the clock period of the CPU.

Detailed characterizations show that when executing scale-out workloads, processors that are mainly focused on performance gains through ILP and MLP do not fully utilize these structures. This is due to two main factors: firstly, hiding memory latencies is only attainable when the two memory instructions are independent. Modern workloads often contain a high number of dependent accesses, where subsequent accesses must wait for the first one to complete, eliminating all potential to issue memory accesses concurrently. Secondly, the working set size of scale-out applications has vastly outstripped the amount of data that can be cached on-chip. Shared last-level cache (LLC) sizes often take up over half of the die itself, sometimes yielding upwards of 25MB of LLC capacity [20]. However, algorithms that process billions of Web sites (such as Google’s PageRank) or trillions of GPS locations (such as Facebook’s location matching) need to process orders of magnitude more data than can even fit on large disk arrays.

Instead of using die space for advanced processor structures and/or large LLC’s that prove to be
under-utilized, architects have proposed to use multiple low-power cores with in-order instruction issue \[27, 49\]. This principle uses threads and intelligent work sharing primitives in software to divide work amongst the cores, and relies on overall memory bandwidth for performance, rather than low latency for each instruction in a single stream. Since the ultimate goal of scale-out systems is to add compute capability at the same time as memory capacity, we believe that near-data processing fits as a solution. With compute enabled memory, threading and work sharing primitives can execute highly parallel code in-memory at massive scale.

Our work in this thesis focuses on a simple in-order core baseline and improves its energy expenditure by using near-data processing. Due to our focus on generality and energy efficiency, we believe our design is applicable to the massively scaled datacentres of the future.
Chapter 3

The Tempo Prefetcher

This chapter presents Tempo, an update to the state-of-the-art Spatial Memory Streaming (SMS) prefetcher that increases the number of timely prefetches that are issued into the L2 caches. We first provide an overview of our design (§3.1), before motivating our specific contribution to the mature field of data prefetching in §3.2. Then, we describe the mechanism by which we implement Tempo, and how our implementation integrates into existing microarchitectures (§3.3). Finally, we evaluate Tempo and describe how our design improves caching efficiency and performance over a regular SMS baseline (§3.4).

3.1 Overview

The key contribution we make with Tempo is to add temporal information to the SMS prefetcher while keeping the metadata small enough to be kept on-chip. Our design features a novel banked implementation of SMS that further classifies cache accesses within the same physical page based on the repetitive miss latency present in the local access stream. Evaluated on the SPEC CPU2006 benchmark suite, Tempo reduces useless prefetches by 17.6%, and achieves 1.45% and 2.57% increase in IPC on high and low bandwidth memory configurations over the baseline SMS design.

3.2 Motivation

In a modern, energy-constrained design space, a good prefetcher should prioritize improving IPC through accurate and timely prefetches, rather than simply opting for a very aggressive configuration that may waste energy due to increased contention for resources in the memory system. Srinath et al. demonstrate the over-prediction inherent in many simple prefetchers, proposing Feedback Directed Prefetching (FDP) which increases performance and reduces required memory bandwidth by dynamically monitoring cache pollution and adjusting prefetch aggressiveness \cite{feedbackDirectedPrefetching} using simple metrics such as prefetch accuracy and timeliness, collected over time in the hardware.

The Spatial Memory Streaming (SMS) prefetcher \cite{spatialMemoryStreaming} significantly improves prediction accuracy and partially addresses over-predictions for commercial workloads with non-trivial access patterns such as pointer-chasing or non-fixed offset traversals of a physical page. Particularly important is the fact that many of these commercial workloads show code-correlated access patterns. Although other prefetchers already use the program counter (PC) to identify the access patterns on a per-instruction basis \cite{programCounterBasedPrefetching}, Tempo...
Chapter 3. The Tempo Prefetcher

Figure 3.1: Example of repeated access patterns in an OS page. Reproduced from the original SMS authors [75].

SMS associates the PC initiating a memory fetch with a more lengthy access pattern. Hereafter, we refer to these as spatial patterns, borrowing terminology from the original authors.

Figure 3.1 shows an example of how an application may access a very specific set of memory addresses, that are sparsely distributed throughout a large region of memory (in this figure, an operating system page). However, SMS lacks information about the order in which those blocks are accessed, and issues prefetches in strictly increasing address order [75]. This is a significant impediment to the performance of SMS on scientific and engineering workloads, as their memory access patterns exhibit stronger correlations in time than purely in space and are captured well by temporally-ordered prefetchers based on a Global History Buffer (GHB) [59, 22].

Later work by the authors of SMS adds temporal information in the form of a GHB that tracks the ordering of the spatial patterns themselves, rather than the individual constituent memory accesses. This Spatio-Temporal prefetcher achieves approximately 3% speedup over the original SMS design [74] and requires megabytes of off-chip storage for its temporal metadata. One of our goals with this work is to maintain the temporal information inside SMS, while keeping the storage space low enough so that it can be stored on-chip.

In keeping with the overall motivation of our work, we studied the behaviour of SMS in a single-processor environment, and tracked the usage characteristics of its cache lines. In Figure 3.2 we displayed the fraction of SMS’ prefetches that are never referenced by the CPU before being evicted, as well as the fraction that are either early or late. On this set of applications, using a 32kB SMS prefetcher, we measured that up to 79% of SMS’ prefetches are evicted without incurring a reference or are unable to be processed by the L2 cache controller due to contention. Furthermore, up to 25% of the prefetches are untimely and do not reach the core in time. Tempo is designed as an energy efficient prefetcher that minimizes untimely prefetches in high-contention scenarios by only issuing the requests that are the most likely to be processed in time by the system.

3.2.1 Spatial Memory Streaming Base Design

Tempo is built as an addition, with SMS serving as the foundational infrastructure. Therefore, in this section we first explain the design and operation of SMS itself. The SMS prefetcher has two main components, the Active Generation Table (AGT), which records patterns and trains the predictor, and the Pattern History Table (PHT), which stores the patterns for future access. We begin with discussing the function of the AGT, whose layout is shown in Figure 3.3.

†The full details of our methodology are found in §3.4.1.
Chapter 3. The Tempo Prefetcher

In Figure 3.2, we define the concept of a pattern generation. The access to block A (regardless of A’s location inside the page) is referred to as the trigger access, which is the first access in the spatial pattern. The length of a pattern generation is defined from the trigger access to when the first block in this pattern is evicted from the cache. This choice is key because it means that all of the blocks inside a spatial generation are simultaneously present in the cache. In the figure, we see two separate generations of Pattern A, and an overlapping generation of Pattern B. Spatial Pattern A is repeated in Figure 3.3, which shows a step by step walkthrough of the trigger access ∈ and pattern training ∈ ∈ ∈, concluding on an eviction ∈.

The AGT stores bit vectors, where every bit corresponds to a cache block in that operating system page; each entry is tagged with the upper bits of the address that correspond to the page number. In the original SMS design, the Filter Table is used to remove accesses from code that only incurs a single memory access and therefore should not allocate a new slot in the AGT. This means that in order for a new pattern to start the training process, it first must be allocated inside the filter table, and receive a second access that transfers it to the AGT. On a conflict miss in the AGT, an access from the Filter Table is dropped, preserving the patterns inside the AGT to continue the training process.

Once a new pattern has been trained inside the AGT and transferred to the PHT, observing a trigger access begins the prefetching process. Figure 3.4 shows a walkthrough of generating prefetches on a PHT hit. First, the spatial pattern is copied to a prediction register, and the page base address is extracted...
Chapter 3. The Tempo Prefetcher

Figure 3.4: Example access pattern that shows multiple generations, as well as overlapping patterns.

Figure 3.5: Block Diagram of the Pattern History Table, showing how prefetches are generated from combining a trigger access and existing spatial pattern.

from the original trigger access. Then, a new prefetch is generated for each cache block indicated in the spatial pattern vector. These prefetches are issued on a 1-per-cycle basis.

Although storing prefetches in this manner is efficient storage-wise, it has its drawbacks, particularly with finite memory system resources. With standard 4kB memory pages and 64B cache blocks, there are 64 blocks per page, all of which may eventually have a prefetch issued. However, with OS superpages becoming a common option for systems designers, the amount of blocks that potentially need to be prefetched will only increase. Since SMS orders its prefetches by increasing address, naively issuing requests in that order has the potential to fill all of the MSHRs inside the CPU with prefetches that are actually referenced further in the future, or not at all. Furthermore, issuing prefetch requests for all of the cache blocks designated in the PHT can lead to an over-saturation of the memory system, where actual demand requests are starved for cache resources due to the numerous prefetches. We tackle both of these issues with Tempo.

We demonstrate this difficulty inherent in SMS in Figure 3.6 with an example taken from the application milc. This figure shows a histogram of how many prefetches are possible in each page region, as well as how many are actually issued. In this figure, the X-axis corresponds to how many blocks were recorded (ie. how many bits were set) in the last instance of this spatial pattern, and therefore how many prefetches are possible. Bins 6, 21, and 22 show similar behaviour, where the possible prefetches
Figure 3.6: Histogram of prefetches residing in each spatial pattern for our SMS baseline.

are not all issued to the memory system due to contention on the number of L2 MSHRs. In all three of those bins, we see the potential prefetches in bin number $N$ being spread over the lower bins numbered $[0, N-1]$ due to contention on the L2 read queue and MSHRs. Approximately 2000 (15%) of the spatial generations in bin 6 demonstrate MSHR contention, and we see the prefetches divided over bins 0-5 approximately equally. In contrast, the spatial patterns containing either 21 or 22 possible prefetches both manage to only issue approximately 10 of them. *Tempo* addresses this problem by providing a notion of which blocks will likely return to the processor in time, and issues those prefetches first.

### 3.3 The Tempo Prefetcher

The *Tempo* prefetcher further classifies the cache accesses within a spatial region into multiple sets, based on the relative speed with which the CPU is generating L2 misses. Adding this information during training gives the prefetcher the capability to request the most timely or useful blocks at prefetch time. Our notion of timeliness is connected to the number of intervening cache misses that take place between two accesses in the same spatial generation. Conceptually, we demonstrate this idea in Figure 3.7, which shows an example pattern observed in the benchmark *mcf*. Along the timeline (increasing rightwards), we mark the hits and misses seen by the L2 cache, and label each interval with the number of misses from other spatial generations that occurred between the two interval endpoints. Between the first two accesses, there were six intervening misses, and between the second and third accesses, there were thirty-one. From this point forwards, we add a global miss counter to the L2 cache, and define the cache time delta as the difference in the counter’s value after two cache accesses.

By gathering memory traces and statistics from the applications comprising SPEC CPU2006, we observe the same PC generating misses in the L2 cache with varied cache time deltas, hereafter referred to as tempos. Using the same example from Figure 3.7, the first three cache accesses may be to cache blocks A, A+5, and A+15 with the miss counter equal to 2, 8, and 39; this access sequence has tempos of 6 and 31, respectively. The point of assigning tempos to an access stream is to determine the contention in the memory system between multiple spatial patterns. If the cache hierarchy is saturated with requests and the L2 miss counter is ticking quickly, it is very likely that the lower levels of the memory hierarchy will be slow to respond to any prefetch requests. Therefore, at prefetch time, *Tempo* can intelligently prioritize the cache blocks that are likely to arrive at the core in time to be referenced by the core. We more fully explore this concept in §3.4.3 with a detailed example.

---

2This name was directly inspired by Chazelle, Simmons, and Teller’s foundational work: *Whiplash*
3.3.1 Tempo Decomposition

We now explain how we integrate the concept of cache time deltas into training the Tempo prefetcher. In Figure 3.8 we show how a dense pattern representing a large number of memory accesses can be filtered into multiple pattern vectors, each representing a distinct access tempo. All of the references in the unfiltered pattern are present in the decomposed slices. This is implemented in hardware by splitting both the AGT and PHT into multiple disjoint slices, where the AGT only sets the bits for the cache accesses that were recorded by that PC with that particular tempo. Our tempo values are similar to those of the TAS prefetcher [85]: a cache access tempo of less than 4 is classified as fast, from 4-9 is medium, from 9-15 is slow, and over 15 is very slow. We selected these values through heuristics and experimenting over the large space of possible values.

In order to localize the current access tempo being exhibited by each outstanding spatial pattern being trained, we introduce a small number of local tempo buffers (LTBs) into the AGT training process. The LTBs are indexed by a hash involving only adds and XORs of the PC, and store the last three tempos observed for that particular PC. Therefore, as we train a new spatial region generation, each L2 cache access calculates the average tempo inside its respective LTB, and sets the correct bit in the AGT slice which corresponds to that LTB value. Figure 3.9 shows this process. Each PC will hash into one of 32 LTB’s, calculate the average of the last 3 observed access tempos, and record the access to the current cache block in the pattern vector that corresponds to this access tempo. Upon observing an eviction to any cache block in this spatial region, all slices of the AGT currently trained are sent to the PHT. Now that we have defined the manner in which we classify accesses in Tempo, we move to discussing how we generate prefetches.

3.3.2 Prefetching Based on Local and Memory Tempo

The process of generating prefetch addresses is similar to how addresses are classified based on access tempo. We make the key observation that we can choose which address candidates to issue as prefetches based on the current utilization of the memory system’s resources and load. Tempo has already classified the blocks recorded during this spatial generation into one of four distinct tempos, but that in itself is not enough to ensure the right prefetches are issued. This is because the memory system may respond to
our prefetches at different rates. For example, if the lower-level cache and/or main memory is currently heavily loaded with many outstanding requests, it will naturally respond to our demand accesses and prefetches more slowly, and vice versa if it has many free resources to handle incoming requests.

Therefore, as we defined the notion of the current access tempo for a particular PC, we also define the memory tempo by storing a checkpoint of the global miss counter (§3.3) in every L2 MSHR when it is allocated for a new demand request. When a demand request is filled into the cache and the MSHR is freed, we calculate the delta between the current miss counter and the value returned from the MSHR. This memory tempo is stored in a single global tempo buffer (GTB) which tracks the last seven observed memory tempos. We use this memory tempo information when Tempo observes a trigger access; the current local access tempo (from the LTB’s) is compared against the current tempo of the memory system (in the GTB), allowing us to make an informed decision about which slices of the current spatial region that we wish to use for prefetching. Figure 3.10 further explains this process with a step by step walkthrough:

1. The current PC sees a demand request to address B+2.
2. We look up the current tempo for this PC in the LTB, and get an average tempo of 6.
3. All elements of the GTB are averaged to obtain a memory tempo of 11.
4. Therefore, the PC is rushing ahead of the tempo at which the memory is returning demand requests, making it likely that if we issue aggressive prefetches from the fast and medium slices, they will not return in time to cover the incoming demand miss.
5. Tempo therefore chooses to issue prefetches from the slow and very slow slices of the PHT, since it is likely that these cache blocks will be returned to the L2 cache in time to cover demand misses.

Also note that the inverse may be true for the case where the GTB is returning requests faster than the current PC is issuing them (this is referred to as the PC dragging behind the memory’s tempo). In this case, we can issue prefetches more aggressively from the faster tempo slices, knowing that the memory system will likely return these blocks in time for the upcoming demand accesses.

What differentiates the Tempo prefetcher from other existing priority schemes is the fact that our priority scheme is proactive rather than retroactive. Feedback Directed Prefetching [76] relies heavily on statistics such as prefetch accuracy and cache pollution to dynamically throttle its stream-based prefetcher based on the characteristics of the memory system. Although we do include a very simple throttling mechanism in Tempo based on average memory access time, the defining characteristic of our prefetcher is how we actively measure the access and memory tempos during training, and use that
Dynamic Aggressiveness Control

In addition to the structures presented in Sections 3.3.1 and 3.3.2, we add a dynamic aggressiveness controller that adjusts *Tempo’s* prefetch degree based on the average memory access time (AMAT) calculated during demand misses. Two 16-bit registers track the AMAT for all cache accesses initiated by the core. These registers keep an exponential moving average (EMA) for the last 500 demand requests, for the purposes of dynamically adjusting *Tempo’s* maximum prefetch degree. We use 16-bit registers because of the amount of time required to return from memory is discretized by the CPU clock, and generally will range from 20-100 clock ticks. Half-precision registers allow a dynamic range of up to 65,535 for the EMA, which easily meets our requirements. We implement the EMA with a two hardware multiplies and an addition, where one register holds the current EMA, and the other stores the access latency seen by the last demand request. Therefore, we compute the EMA via the following expression, where $\alpha$ represents a decaying parameter (0.5 in our design) and $t_{new}$ is the new demand request latency:

$$EMA_{new} = \alpha \cdot t_{new} + (1 - \alpha) \cdot EMA_{old}$$

Since we also need to store the CPU’s cycle count in each demand MSHR to update the AMAT, we add a 32-bit cycle counter to each of the 16 L2 MSHRs in our hardware budget.

When the AMAT for the last 500 demands has increased over what we previously had observed, we reduce the prefetch degree by 1, and vice versa if the AMAT has decreased. Both adjustments to the prefetch degree require a 5-cycle difference in the AMAT, otherwise the degree is not changed. We chose this value empirically by sweeping all thresholds from 1-10 and choosing the value with the highest average IPC. The performance improvements of the degree controller presented in this section are given in §3.4.4.
3.4 Evaluation and Results

In this section, we describe our evaluation infrastructure, and quantify the specific results that demonstrate Tempo’s improvements. We begin by describing our simulation methodology.

3.4.1 Methodology

We evaluate our prefetcher using a trace-based infrastructure provided as a part of the 2nd Data Prefetching Championship (DPC-2) [65]. Our prefetcher trains on the L2 access stream, prefetches into the L2 cache, and can request its prefetches to be filled to the LLC if the MSHR occupancy is greater than 10 out of the 16 available at the L2. The DPC-2 framework models an aggressive 6-wide pipeline that can issue two loads and one store per every cycle, as well as a three level inclusive cache hierarchy, with the following characteristics:

- 16kB, 8-way L1 cache, with 4-cycle hit latency and 8 outstanding requests to the L2.
- 128kB, 8-way L2, with 10 cycle access latency to the tag array. The L2 can have 16 outstanding requests to the LLC.
- 1 MB, 16-way LLC, with a 20 cycle access latency.

Our prefetcher trains on the L2 access stream, prefetches are filled into the L2 cache, and can request its prefetches to be sent to the LLC which does not occupy an L2 MSHR. In all of the experiments in this section, we disable the dynamic mechanism that adjusts the prefetch degree based on the AMAT, to isolate the additional pattern filtering of Tempo. We directly discuss the affects of the dynamic mechanism for adjusting AMAT in §3.4.4. In the first sections of our evaluation, we disable the dynamic prefetch degree controller because of the fact that the baseline SMS prefetcher is not equipped with this functionality. Therefore to provide the most fair comparison, we only add our Tempo contribution, since our dynamic degree controller also has the capability to reduce excess prefetches when turned on.

In order to simulate the performance of the cache hierarchy, we use a combination of built-in simulator features with our own custom cache model. We utilize the DPC-2 simulator [65] to provide the hashing and mapping functions that map each cache access to a set and way of the L2, and model the rest of the parameters such as LRU replacement and MSHRs ourselves. The competition infrastructure provides four target “modes” that model different system configurations. We enumerate the differences of each mode in Table 3.1. In this work, we focus on Modes 1 and 3, as they provide the most insight into the contributions of our Tempo prefetcher.

Using the Pin [51] tool provided in the DPC-2 framework [65], we generate traces for all applications comprising SPEC CPU2006 [33], compiled with GCC 4.0. We fast forward each benchmark for 10 billion instructions into the region of interest, before gathering a trace of 100 million instructions.

Our evaluation has three main prongs: firstly, we show that Tempo is effective in reducing the amount of useless and untimely prefetches issued by SMS. Secondly, we show that restricting available memory bandwidth magnifies Tempo’s performance gains over SMS, since Tempo can select the essential prefetches out of the large potential set. Finally, we evaluate the hardware storage cost of Tempo and compare it to SMS, as well as show that our design easily integrates with a standard L2 cache architecture.
3.4.2 Performance and Timeliness Results

To evaluate the performance of Tempo, we compare its performance against a state of the art 32kB SMS prefetcher using retired instructions per cycle (IPC), the number of useless prefetches, and prefetch timeliness. We quantify prefetch timeliness by comparing how many prefetches result in hits inside the cache (timely) versus inside the MSHRs (untimely) or after the line has already been accessed (late). The baseline SMS prefetcher represents the current state-of-the-art design, as we compared our implementation against the version submitted by the original authors [28] to the previous Data Prefetching Championship. In Figure 3.11, we compare the absolute IPCs achieved by Tempo and SMS, both compared to a baseline that does not include any prefetching. These results are displayed for Mode 1 of the simulated memory hierarchy. We have omitted comparisons for gromacs, namd, povray, calculix, and omnetpp, as we see essentially no variation in IPC for these applications with any of the prefetchers available in the simulation infrastructure, or that we implemented to compare against.

Tempo achieves an average IPC increase of 1.45% over SMS for these applications. We notice that for applications such as perlbench, sjeng, and h264, both SMS and Tempo obtain less than a 1% speedup over a baseline no-prefetching design. However, by capturing the interleaved tempo behaviour that we originally observed in Figure 3.7, we obtain a 20.4% IPC increase on the most memory intensive benchmark, mcf. To investigate the performance characteristics further, we augmented the DPC-2 framework with our own cache simulator and gathered more detailed statistics for the four applications mentioned above that see limited speedup. Our statistics are: the raw miss rates exhibited by each application, the average memory access time on L2 misses, and the number of L2 MSHRs allocated. Note that our cache simulator only is able to model the L2 and LLC tag arrays, since the simulator only exposes the L2 request stream to the prefetcher.

perlbench and sjeng have L2 hit rates in excess of 91%, and therefore our prefetchers have limited opportunity to improve the average access latency. Furthermore, the average memory access time (AMAT) of these low-opportunity applications is within 3% of the LLC hit latency (20.58 cycles versus 20 cycles), which demonstrates to us that the vast majority of L2 misses incur a hit in the LLC. To verify this data, we repeated these experiments when using Mode 2 of the simulator in Table 3.1, which restricts the LLC size to 256kB instead of the regular 1MB. With this configuration, the AMAT increases by less than 2% over the regular 1MB LLC, demonstrating that these applications cache most of their entire working set on chip and show limited opportunity for prefetching.
In Figure 3.12, we compare the fraction of prefetches classified as useless and untimely between SMS and Tempo, with each bar normalized to the total number of prefetches per application. Useless prefetches are undesirable for two reasons: they may simply never be referenced by the processor, which wastes energy in moving an unused block into a higher level cache, or potentially cause harmful cache pollution by evicting cache blocks which are still useful. On average, we reduce the amount of useless prefetches being issued to the memory system by 18% and the number of untimely prefetches by 43%. We find that our reductions in useless prefetches are primarily due to the elimination of prefetch requests from the faster tempo slices, particularly when the memory system is congested and operating in the very slow tempo range.

However, we notice that in two benchmarks (libquantum and omnetpp), the number of useless prefetches actually increases. In both applications, Tempo issues a far greater number of prefetches than SMS due to the fact that AGT and PHT entries have a potentially longer lifetime than in an equally sized SMS prefetcher. In Tempo, each of the AGT and PHT slices can individually remain active as long as this PC is still generating requests at the matching tempo, where SMS evicts the entire pattern vector on the first eviction in the AGT, or tag conflict in the PHT. This discrepancy of how pattern vectors are managed between the two prefetchers gives Tempo the potential to possess more prefetch-generating PHT hits over time. In order to rectify this issue, we could simply enforce that each spatial generation must move its PHT slices between the AGT and PHT in lockstep, rather than the current design which allows them to be decoupled. Figure 3.13 compares the performance of Tempo and SMS in DPC-2’s low memory bandwidth configuration of 400 MT/s. When operating in this restricted memory system, Tempo attains an IPC increase of 2.57% over SMS for these benchmarks, due to the fact that it judiciously issues far more timely prefetches than SMS.

We also report the improvements in accuracy and coverage for Tempo over an equivalently sized SMS predictor in Figure 3.14. On average, we improve prefetching accuracy by 5.9%, and coverage by 9.4%. Our IPC improvements in the benchmarks astar and lesleis3d occur because Tempo increases prefetch coverage by 86.6% and 67.4% respectively. In order to determine the reasons why coverage increases by so much for these two applications, we inspect the spatial patterns recorded for each program counter in the application, as well as tracking the differences in prefetches issued and returned between SMS and Tempo.

In lesleis3d, we see one particular application phase that is responsible for this behaviour - two program counters (PCs) often traverse entire pages in an interleaved fashion, where one PC accesses one block
Chapter 3. The Tempo Prefetcher

The access patterns of astar are far more complex, and we were unable to find any obvious patterns when the memory accesses were localized per-PC. Instead, we chose to track the access patterns at a page granularity, recording all of the different program counters that requested a certain page, as well as the cache blocks accessed. By displaying the access patterns in this fashion, we were able to see that many of the physical pages incur an access to every block, but the accesses are split across approximately 20 different PC’s, making it difficult for code-correlated prefetchers such as SMS to gain full knowledge of the access pattern. Tempo shares this difficulty in the training process.

The coverage benefits come from the fact that Tempo is able to share the available resources at the L2 amongst the different PCs. When the fully populated spatial patterns are ready to prefetch, SMS sends requests for all of the prefetches in the PHT, up to the maximum number that can be placed into the MSHRs (16). However, Tempo divides the linear page scanning into its four slices, and only issues prefetches for the matching tempos according to the decoding process described in §3.3.2. In this particular application, issuing less prefetches per-PC preserves space in the MSHRs and request queues for the other PCs which follow this one. This behaviour, although beneficial, was not by design.
Although we achieve general increases in both coverage and accuracy through this behaviour, we note that our largest single-benchmark IPC improvement of 20.4% occurs in \emph{mcf}, which only has 3.3% and 1.1% improvements in accuracy and coverage, respectively. To explain the large IPC improvements in \emph{mcf} despite modest gains in accuracy and coverage, the next section prevents a detailed study of this application’s behaviour.

### 3.4.3 Detailed Study

We choose the \emph{mcf} benchmark not just because it exhibits the behaviour that \emph{Tempo} targets, but also that it is the most difficult application of these to accelerate with prefetching. Table 3.2 shows the IPC attained by all of the different prefetchers available in the DPC-2 simulation framework for comparison. The Best-Offset prefetcher \cite{53} was the overall winner of DPC-2, and the Access Map Pattern prefetcher \cite{38} was the overall winner of DPC-1. The trivial next-line prefetcher outperforms all designs on \emph{mcf}. Otherwise, \emph{Tempo} is the best performing prefetcher on this benchmark.

Figure 3.15 shows the histograms demonstrating the prefetches possible and issued, per page, in SMS and \emph{Tempo}. In SMS, the most commonly trained spatial pattern has 2 prefetches per page, not including the trigger access itself. This occurs because of the access pattern in \emph{mcf}, which also explains the reason that next-line prefetching performs the best in this benchmark. \emph{mcf} accesses blocks that are located across many different physical pages, and the page number sequences demonstrate a simple

Table 3.2: IPC for All Prefetchers on \emph{mcf}

<table>
<thead>
<tr>
<th>Prefetcher</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0.138</td>
</tr>
<tr>
<td>b-off</td>
<td>0.145</td>
</tr>
<tr>
<td>nline</td>
<td>0.189</td>
</tr>
<tr>
<td>stream</td>
<td>0.138</td>
</tr>
<tr>
<td>ampm</td>
<td>0.138</td>
</tr>
<tr>
<td>sms</td>
<td>0.137</td>
</tr>
<tr>
<td>tempo</td>
<td>0.168</td>
</tr>
</tbody>
</table>

Figure 3.15: Histograms of possible and issued prefetches in \emph{mcf}. We do not include the trigger access in the bins on the X axis.
Chapter 3. The Tempo Prefetcher

### 3.4.4 Dynamic Aggressiveness Control

This section isolates the effects of our mechanism to control Tempo’s prefetch degree based on the AMAT, originally described in §3.3.2. Our goal is to determine the additional speedup that is directly attributable to our dynamic degree controller. We used Mode 1 of the simulator and performed identical executions of the trace, with and without our prefetch degree controller enabled. The speedup for the configuration using the dynamic controller is given in Figure 3.16.

The speedups are limited for our dynamic controller, ranging from 0.92 to 1.1, with an average of 1.02. In the benchmarks bzip2 and gamess, we obtain the 10% speedup because of the high overall accuracy.
Data Structure | Components | Budget (B)
---|---|---
AGT (64 entries, 4 slices) 2-way Associative | Tags: 64b addr - 12b Pattern Vectors: 63 Offset: 12 PC:14 | 36096b × 2 = 9024B
PHT (256 entries, 4 slices), 2-way Assoc | Tags: 14b PC Pattern Vectors: 63 | 78848b × 2 = 19712B
Extra MSHR Storage (32) | Tags: 64b addr - 12 Valid: 1 Time Counter: 10 Cycle Counter: 64b | 254B
Local Tempo Buffers (32) | Each: 3-wide Time Counter: 10b | 120B
Global Tempo Buffer (1) | 7 entries Time Counter: 10b | 9B
AMAT Registers (2) | Cycle Counters: 64b | 16B
**Total:** | | **29 135B**

Table 3.4: Hardware Storage Requirements for Tempo

<table>
<thead>
<tr>
<th>Structure</th>
<th>Chip Area</th>
<th>Energy Per Access</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-hot AGT</td>
<td>0.04787 mm²</td>
<td>11.3 pJ</td>
<td>0.334 ns</td>
</tr>
<tr>
<td>3-bit AGT</td>
<td>0.04341 mm²</td>
<td>11.0 pJ</td>
<td>0.333 ns</td>
</tr>
<tr>
<td>1-hot PHT</td>
<td>0.11257 mm²</td>
<td>29.6 pJ</td>
<td>0.395 ns</td>
</tr>
<tr>
<td>3-bit PHT</td>
<td>0.08429 mm²</td>
<td>21.1 pJ</td>
<td>0.339 ns</td>
</tr>
<tr>
<td>128kB L2 Cache</td>
<td>1.2799 mm²</td>
<td>102.3 pJ</td>
<td>1.187 ns</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of hardware implementations of the AGT and PHT.

of both SMS and Tempo. Both of these benchmarks keep the prefetcher at its maximum degree, filling both the L2 MSHRs and sending the rest of the requests to the LLC directly. Although we show in the next section that the hardware overhead of our controller is limited, we argue that its benefits are small enough to leave it out of any hardware implementation.

### 3.4.5 Hardware Design and Budget

Table 3.4 breaks down the hardware budget for Tempo. Our design was implemented to fit inside a 32kB storage limit for inclusion in DPC-2. For simplicity’s sake, the easiest way to store the tempo information is to replicate each of the SMS structures four times. However, we could represent each slice more economically by using a three bit counter for every cache block, where different values would represent this cache block being accessed by different tempos, saving 25% of the raw storage bits.

Using CACTI 6.5 [56] to model both the AGT and PHT, we tested the energy and area overheads of the following two configurations, assuming both structures are implemented as set-associative caches:

- The naive implementation shown in Table 3.4 where each tempo slice is represented using 1-hot encoding.
- Using a three-bit counter to represent the different tempos each block may be associated with.

The results for energy, area, and cycle time for both the AGT and PHT are shown in Table 3.5. We also include the numbers for the L2 cache itself for comparison. Using a 3-bit encoding allows us to save 33% of PHT area and energy, with approximately the same cycle time. The only extra complexity that
Figure 3.17: Hardware Block Diagram for Dynamic AMAT Calculation Described in §3.3.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Speed</th>
<th>Power</th>
<th>Area</th>
<th>Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Multiplier</td>
<td>Slowest</td>
<td>Highest</td>
<td>Highest</td>
<td>Regular</td>
</tr>
<tr>
<td>Wallace Tree</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Irregular</td>
</tr>
<tr>
<td>Booth’s Multiplier</td>
<td>Fast</td>
<td>Low</td>
<td>Lowest</td>
<td>Irregular</td>
</tr>
<tr>
<td>Vedic Multiplier</td>
<td>Fastest</td>
<td>Medium</td>
<td>Low</td>
<td>Irregular</td>
</tr>
</tbody>
</table>

Table 3.6: Comparison of Multiplier Types.

is required to support generating prefetch addresses with the 3-bit encoding is the addition of a decoder
to the Prediction Register in Figure 3.5. Since implementing a 3-bit decoder only requires simple AND

gates, we choose to use the 3-bit encoding for Tempo.

In order to implement the dynamic degree controller of §3.3.2, we add simple circuitry to compute
the 500-request EMA. Figure 3.17 shows a block diagram of the hardware circuit required to support an
EMA calculation and a comparison of two values. The components whose widths are parametrized in this
section are shaded in a darker colour. Input parameters are on the left of the figure, and both registers
holding the input EMA and the old EMA are shown on the right, feeding into the final comparator.
Note that the above circuit is very similar to a fused multiply-add (FMA), which for general terms $a, b, c$
is given by:

$$a \leftarrow a + b \times c$$

However, due to the additional multiplication of $1 - \alpha$, we cannot use a single dedicated FMA for our
computation, instead requiring two n-bit multipliers and an n-bit adder.

In order to choose the proper type of floating point multiplier to build, we present a short comparison
of multiplier array structures in Table 3.6. All of the multipliers listed are well documented in circuits
literature [4, 70, 19, 82], and we extracted the high level comparison in Table 3.6 by comparing the
scaling characteristics from published works at multiple technology nodes, implemented both on ASICs
and FPGAs. Using the Vedic multiplier allows us to attain a very fast parallel multiply at the cost of
a more complex layout. However, since the EMA feedback occurs on an L2 miss, latency itself is not
the most critical factor, as the CPU itself will be waiting upwards of 20 clock cycles to return the data
(the minimum latency of the L2). Therefore, we choose to set power as our key optimization target due
to the overall theme of this thesis work. Recent proposals have demonstrated capability to combine the
benefits of Vedic arithmetic with the speed benefits of fast adders (Brent-Kung) for the carry trees [45].
We choose to use these designs because they attain both high speed and low overall power.

Since 16x16 multipliers are a de-facto standard for implementing higher-precision computations, we use power and area numbers for a state of the art half-precision (16x16) multiplier synthesized in 90nm [45], and use scaling factors of $\left(\frac{45}{90}\right)^2$ for the area/power, and $\frac{45}{90}$ for delay [21] to normalize it to our 45nm cache technology. We present these costs in Table 3.7 as well as the final cost of the entire EMA unit, which adds the delay, power, and area of a single 16-bit fast carry-save adder [24, 72]. Compared to the area of the Tempo prefetcher itself, we see that the EMA unit consumes approximately 1000x less chip area, and introduces approximately 0.0001% power overhead when introduced to an aggressive O3 core operating at a total envelope of approximately 80W. Although this EMA design fits the constraints of our target design, we ultimately believe that implementing it in any prefetcher is not required due to the limited impact that the dynamic controller has on performance (see § 3.4.4).

3.4.6 Tempo Summary

In this section, we present Tempo, a novel addition to the Spatial Memory Streaming prefetcher which utilizes repetitive patterns in cache access timings in order to filter out prefetches that are unlikely to reach the cache in time to be referenced. We implement Tempo using simple set-associative structures, and obtain 1.45% and 2.57% increase in IPC on high and low bandwidth memory configurations over the current best SMS design. Additionally, on the most memory intensive benchmark, we obtain a 20.4% improvement in IPC.
Chapter 4

Anti-Fetching: A Dynamic NDP Architecture

Although data prefetching can be an effectual technique in hiding lower-level memory latency, it continues to enforce the traditional computing paradigm that all of the data computations will take place in the processor. In this chapter, we present Anti-Fetching, an architecture which dynamically offloads code fragments to a near-data processor, saving energy in the memory hierarchy by selectively loading the on-chip caches with data that will incur reuse. We iteratively construct packages of instructions in the CPU front end, and identify which instruction packages should be executed near memory based on the application’s cache reuse characteristics. We also design an energy efficient architecture to execute these instruction packages, issuing memory requests directly to the DRAM controller rather than using the cache hierarchy. We evaluate Anti-Fetching with a set of graph processing workloads, and demonstrate up to 31% energy savings in the memory hierarchy (11% average) over an in-order CPU baseline.

Firstly, we present an overview of our design in §4.1 and provide the motivation for our contribution in §4.2. Then, we describe both the on and off-chip elements of our new architecture in §4.3 and §4.4 before presenting our experimental evaluation in §4.5.

4.1 Overview

Anti-Fetching is a general hardware-only NDP architecture that operates transparently at runtime, and does not require any explicit effort from the programmer or compiler to use. Our architecture executes existing binaries and extracts energy savings through dynamically offloading user code to a small in-order processing pipeline close to the DRAM. It uses very simple table-based training structures with approximately 3KB of hardware overhead to identify common basic blocks that are generating memory accesses with no reuse inside the on-chip caches. Using the basic block as the smallest unit of computation that can be offloaded to our NDP pipeline eliminates the need for complex control logic near the memory, and simplifies the speculation on whether or not to offload a section of code.

Our key contributions are:

- Profiling the Stanford Network Analysis Platform (SNAP) applications [47, 46], and demonstrating that we can accelerate them using a runtime NDP approach.
• Designing a microarchitecture that dynamically captures the common basic blocks of a program and communicates with our near-data processor.

• Using the reuse characteristics in the cache hierarchy to determine whether or not to offload these basic blocks into memory.

• Proposing and evaluating an analytical model which describes the prediction accuracy in terms of the memory hierarchy design and application reuse rate.

• Utilizing an oracle offloading predictor to obtain 11% average energy savings (31% maximum) without degrading application performance, and evaluating the penalties arising from mispredictions.

4.2 Motivation

Energy and power constraints are primary concerns in chip design due to the unique combination of massive data sets and a limited future for CMOS technology scaling. Not only does the dark silicon problem [25] mean that every type of architecture is energy limited, memory latency and bandwidth continue to be an issue for a wide variety of architectures. Memory-bound applications have proven difficult to accelerate by applying classical architecture techniques such as speculation, pipelining, and instruction level parallelism. These techniques not only have limited efficacy in speeding up memory-bound applications, but also have the downside of further increasing power consumption. In contrast, near-data processing (NDP) is an energy-efficient design principle that places compute units much closer to the memory banks, which limits energy inefficient data movement across long wires.

Using NDP techniques is a resurgent topic in architecture literature, due to the energy savings from reducing data movement across the chip’s pins. To quantify just how expensive data movement has become, consider that a single off-chip access can cost up to $200 \times$ the amount of energy as a double-precision floating point divide [44]. The most common solution to this problem is to cache as much data on the chip as possible. Unfortunately, this approach is not as effective on big data applications, which have far less temporal locality [7] and therefore large numbers of dead cache lines.

Despite this trend, general purpose processors continue to dedicate more on-chip real estate to large last level caches (LLC); for example, Intel’s most recent Haswell server chips have between 25 and 40MB of LLC space [20], which can consume up to 50% of the total die area. The success of this approach is entirely a function of the inherent amount of locality inside the target applications. Instead, leveraging NDP paradigms can save power by allowing designers to rely on smaller cores augmented by compute-enabled memory. This frees up die area, allowing designers to either shrink the chip itself, or dedicate those extra transistors to more efficient uses.

Past work that contrasts in order (IO) and out of order (O3) cores has shown that IO cores have a higher performance per watt when dealing with applications that are limited by available ILP or memory footprint [10]. This occurs primarily due to under-utilization of the expensive structures required to feed a superscalar O3 processor with instructions [27]. Instead, architects have turned to throughput-oriented computing that intelligently trades the improved single-threaded performance of an O3 core, for the additional throughput of many IO cores with less area and/or power budget. We choose IO cores because of their low power consumption, and improve their performance/energy through the use of a novel NDP mechanism. Our work focuses on using IO cores as basic building blocks for a hybrid
architecture that can execute code either on-chip or near-memory, depending on the application’s cache patterns.

Computer architects have proposed many different improvements to accelerate low-MLP workloads, with contributions in both the pure hardware [2, 57, 6] and full-stack [3, 68, 36] domains. Although many of these architectures use NDP [2, 3, 68, 36] to speed up workloads with complex memory behaviour, they have a very important shortcoming: all of them require large changes to infrastructure such as the operating system and compiler. Additionally, many remove or restrict the use of abstractions such as virtual memory and unified addressing [84, 68, 2]. We believe that these changes are a significant barrier to widespread adoption of NDP techniques, particularly to code that is written by an average programmer. Re-writing legacy code to run on an accelerator, or any non-traditional architecture is not only difficult, but a significant investment of time. For NDP techniques to actually achieve widespread adoption, hardware architects should design systems that can make use of emerging memory and logic integration in a way that is completely invisible to the programmer. Put another way, generality remains important for both software and hardware design.

Near-data processing techniques also have the potential to operate in harmony with emerging datacentre designs that make use of in-memory databases [60], virtualized DRAM [62], and slave nodes that only exist to store large datasets in their main memory [1]. These techniques decrease database latency and improve energy by storing datasets in-memory that traditionally would be swapped to disk. In parallel with moving the data from disk to memory, we can reduce the energy expenditure even further by executing the code close to the data itself. To demonstrate the opportunity to use NDP to improve the efficiency of workloads with massive data sets, we select a family of applications that have both complex memory patterns and scalable datasets - the example applications in the Stanford SNAP graph processing library [47]. Commercial graph analytics are common workloads that execute at the datacentre scale, and also have proven difficult to accelerate with general purpose hardware techniques. We now shift to discussing the difficulties of executing graph workloads with large data footprints using traditional cache hierarchies.

4.2.1 Limited Locality in Graph Applications

Graphs are ubiquitous abstractions for representing data points and their connections, and are particularly useful in the fields of data mining, pathfinding, and convex optimization problems. Although any graph can be minimally represented by a node and edge list file, a challenge for software designers is to provide useful abstractions to users that allows them to perform complex analysis tasks on giga-scale graphs with reasonable computation times.

There are a plethora of software techniques and frameworks available for accelerating graph processing [50, 79]. Designing computer hardware to intelligently execute these parallel algorithms remains complex, particularly due to the complex memory accesses inherent in most non-trivial graph algorithms. For many years, computer architects have been able to achieve performance boosts from using larger or more associative cache hierarchies. However, the efficacy of this approach is very low when executing a workload with limited spatial and temporal locality.

To illustrate this problem, we investigated the memory layouts of an arbitrarily connected graph in the Stanford SNAP graph processing library [47], and compared it to the data structure in raytrace, a graphics workload from PARSEC-3.0 [9]. The canonical data structure in raytrace is a templated C++ class called an RTBox, which represents a 3D environment where objects interact with a light
source and are rendered by a camera plane. The internal data points of an RTBox are simply stored in a single dimensional vector of objects, with index functions written in standard C-style row-major ordering. This leads to the data layout shown in Figure 4.1a, where accesses to the neighbouring cells (shown in green) will likely result in cache hits due to spatial locality. Tracing light rays through a 3D scene is also naturally a spatial problem, which assists in boosting cache efficiency when executing this algorithm on traditional CPU architectures.

In contrast, the SNAP example applications [47] represent their graphs in a node and edge list format, where each unique node is stored in an unordered hash table. The unordered nature of the table is important since graph processing algorithms tend to demonstrate “edge-based” locality, where nodes that are connected by edges are often accessed together. Figure 4.1b shows the locality exhibited with this form of data representation. Even though the actual hash table itself is stored in memory as a simple array of structures, relationships between the nodes are based on their mutual edge connections, which can point anywhere in the global memory space.

To demonstrate the raw differences in cache efficiency between these data layout formats, we utilize Pin [51] to gather the miss rates of both the PARSEC and SNAP benchmark suites when executing on a simple in-order core\footnote{Our full methodology can be found in §4.5.1}. We corrected for the raw size of the input by choosing a graph from the public SNAP datasets [46] that is similar in size to PARSEC’s simlarge input. Figure 4.2 displays the number of misses per thousand instructions (MPKI) at each level of the cache.

As expected, we see better caching performance in the PARSEC workloads; only two applications have an L2 MPKI exceeding 1.5, where SNAP applications such as cliques, centrality, and netstat have very similar MPKI in both the L1 and L2. We refer to this behaviour as full misses, since all levels of the hierarchy are bypassed and data is fetched from main memory. Given that memory latency is often on the order of 100 CPU cycles [54], increasing the L2 MPKI by 5 misses can slow down a basic IO core operating at 1 IPC by up to 50%. Our goal in this thesis is to execute the code that surrounds these full misses with a near-data processor. This technique has the potential to eliminate wasted cache lookups for low-locality accesses by issuing the memory operation directly and executing the surrounding code in a near-data processing pipeline.

Figure 4.3 shows the fraction of lines that do not result in any cache hits during their lifetime in the cache. We immediately notice that in most of the applications, 80% of lines brought into the L2 are never used again; instead of bringing these lines into the caches, we propose to operate on them
using NDP, reducing the energy spent on both tag comparisons and data loads into the caches. Next, we characterize the code patterns exhibited by the SNAP workload suite, which informs our design decisions about how to select candidate sections of code for offloading.

### 4.2.2 Common Basic Blocks

Significant work already exists that focuses on identifying and accelerating loops that generate a large number of memory accesses [57, 6, 55]. In our work, we focus on single basic blocks (BBLs), which are groups of instructions that have a single entry and single exit point. Choosing to focus on basic blocks enables us to simplify our NDP design, since a BBL contains no branches or control flow transfers. This design choice means that our NDP architecture will not be constrained by control speculation, and branch decisions can be made using the existing prediction logic in the processor.

An alternative instruction scope to focus on could be identifying repeated basic blocks in a tight loop. Although tight loops are simple to identify using branch predictor hardware [73], we would need to explicitly specify the loop trip count before sending a BBL to execute in the NDP. This means we must either introduce an ISA modification to always store the trip count in a specific register, or provision the compiler to communicate the trip count in another location. In keeping with our goal to achieve a
dynamic near data processing architecture with existing binary compatibility, we choose to maintain the focus on single BBLs.

We profile our applications by breaking all instructions inside the region of interest (ROI) down by basic block. In Figure 4.4 we display the percentage of instructions in the most common 50 basic blocks of the community and netstat applications. We notice that community executes upwards of 75% of its dynamic instructions in just two basic blocks of the entire code footprint. Netstat has a flatter distribution which is characteristic of most of the SNAP applications; however, this application still spends almost 50% of its instructions in its top 5 most common basic blocks. Approximately half of the SNAP applications see this type of basic block activity, where upwards of 25% of the code is concentrated within the top ten basic blocks.

Finally, in Figure 4.5 we sub-divide the execution time of each basic block of netstat by counting the number of arithmetic and memory operations, and recording the latency of each memory access using the same cache parameters as in Table 4.1b. We see that many of the top ten basic blocks have a significant fraction of their execution time comprised of L1 and L2 miss latency. This is common behaviour that exists across the majority of the SNAP benchmarks. In the next section, we design a microarchitecture that is capable of identifying these common memory-intensive basic blocks, and then show how these BBLs are offloaded to memory.
4.3 Anti-Fetching Microarchitecture

In this section, we propose a method to dynamically identify basic blocks that are candidates to be offloaded and processed near the DRAM. Our modified processor frontend utilizes well-understood structures such as the instruction decoders, branch predictor, and physical register map table to build these BBL candidates. We also demonstrate how our architecture utilizes each BBL’s cache reuse characteristics to decide which candidates will be offloaded to the near data processor. At a high level, we attempt to build our architecture with the smallest amount of changes required to a standard single-board CPU design. We propose an architecture which is shown in the block diagram of Figure 4.6, preserving the on-chip memory controller that is standard for modern architectures. Our contributions instead modify the data transmission and reception that occurs between the Core, Caches, and DRAM banks.

This section pertains to the modifications we make to the CPU core, and is structured around the block diagram in Figure 4.7. We dedicate each of the following subsections to explaining the components labelled (a)-(d), beginning with the BBL Sequence Queue (a), which works in harmony with the branch predictor to assign each BBL a unique identifier/key and provides a method for us to offload BBLs in a non-speculative state.

4.3.1 Identifying Non-Speculative BBLs

Since our goal is to develop a transparent mechanism for executing code in the NDP, we begin by assigning a unique identifier to each BBL: the PC of the first instruction in the BBL. We make this design decision because of the axiom that any BBL has a single entry and exit point. The difficulty with using control instructions to delineate blocks of instructions is the branch predictor inherent in every modern processor. Even a very simple in-order core might have multiple branches, and thus BBLs, outstanding in the pipeline. Speculatively offloading BBLs to execute in memory has potentially massive performance drawbacks on the event of a misprediction, because any memory writes in the
offending BBL have to be cancelled. Although O3 processors manage this problem using associative load-store queues and only writing to the cache upon instruction retire, this solution is intractable when we need to initiate extra off-chip transfers to recover from branch mispredictions. Therefore, we make the design choice to delay our offload until all branch conditions are resolved and we know the authoritative direction and target of this branch.

Upon fetching any branch instruction, we read the output of the Branch Target Buffer (BTB) as a prediction of what BBL is going to be executed next. That target prediction is then fed into the BBL Predictor which, if there is a hit, will decide whether or not to offload this BBL to the NDP. Finally, we write the tuple $d|t|o$ corresponding to this branch’s direction, target, and offload prediction into the BBL Sequence Queue. Unconditional branches follow the same procedure, with the direction prediction being always taken.

After this branch goes through the regular process of being decoded and traversing the instruction queue, it eventually issues and has its direction and target resolved. At this point, our architecture takes the front of the BBL Sequence Queue (BSQ) and verifies the direction and target prediction; if both were correct, then we can take action on the offloading decision that was previously supplied by the BBL Predictor. On a mispredicted branch, we obtain the correct target and re-lookup the offload prediction for the new target.

### 4.3.2 Storing BBL Context

This section explains how our architecture interfaces with the existing pipeline structure to record all of the BBL context needed by the NDP. We choose to store all of the offloading information in a small associative table referred to as the Basic Block Context Table (BBT), which is Component in Figure 4.7. Table 4.1a shows a hardware cost breakdown of all the components of the BBT, which is comparable in size to a small branch predictor. §4.5.6 explains and evaluates our choice of a 64x2 BBT.
We begin with a discussion of how we choose to identify the instructions that make up a BBL.

We observe that attempting to store all of the instructions comprising a BBL in a hardware table leads to difficulties in sizing the table, due to the fact that BBLs vary in storage requirements. This problem is exacerbated when designing for variable length ISAs such as x86. Although a potential solution is to provision hardware to store the maximum number of bytes for all the BBL’s instructions, we observed that the most common instruction size across all SNAP applications is 2B (mean 2.64B). The worst-case provisioning approach means that on average, only $\frac{1}{8}$ of the BBT’s space would be used.

Instead, we choose to let the NDP itself fetch and decode the BBL’s instructions in the same manner as the processor. At offload time, we simply pass the correct fetch pointer (which is identical to the BBL tag) to the NDP, which enables it to read the relevant instructions either from a standard $\mu$-op cache, or directly from memory on a $\mu$-op cache miss. To fully eliminate this BBL from the processor’s instruction stream, we must provide the processor a restart point to begin fetching from when the NDP completes executing. The restart point for a BBL comprised of instructions $[I_0, ..., I_{N-1}]$ is given by the outcome of the control instruction $I_{N-1}$; this restart point will be calculated by the NDP and passed back to the processor.

### Identifying Registers

In order to maintain program correctness when offloading blocks to the NDP, we need to pass it the correct register values that are used inside a BBL, and receive its updates (if any) to the architectural registers after the BBL is finished executing. This problem simplifies to determining which registers are live-in and live-out for a BBL; the live-in registers will have their values passed over the chip interconnect at offload time, and vice versa for the live-out ones after the execution has finished. Before a BBL can be offloaded, it must go through the two phases of having its live in/out registers identified, and then being trained in the offloading predictor. The rest of this section deals with how we identify these registers during a BBL’s on-chip execution.

When the BSQ $\oplus$ supplies the resolved target for the next BBL that will be executed, we search the BBT $\ominus$ for this branch key. A miss in the BBT evicts the NMRU entry, and begins the register identification process for the current BBL. As we begin to issue instructions that belong to this current
BBL, we monitor the registers that are being read and written. Each BBT entry contains two bit vectors representing each processor register, and we use the outputs of the pipeline’s Issue stage to set or clear the correct bits based on the read and writes.

In Figure 4.8, we show an example of registers that are read being marked as live-ins, and those which are written marked as live-out. The BBL shown is the most common memory intensive basic block in netstat, whose latency breakdown can be viewed as the 6th bar from left in Figure 4.5. We show the training process by associating each instruction $I_0 - I_6$ with its own colour and subscript in the live in/out vectors. For example, when $I_1$ was issued, the processor sets rdx as both a live-in and live-out, because it is both read and written by this BBL. Although we focus on IO cores in this thesis, the same technique is applicable to O3 cores which use physical register renaming. However, the BBT must also record a pointer into the register map table which stores the correct value to send to the NDP at offload time.

### 4.3.3 Selecting BBL Offloads

The BBL Offload Predictor is the most performance critical part of our design. As previously shown in Figure 4.5, different BBLs have different memory behaviour, and our design’s performance heavily depends on executing the memory operations in the correct level of the memory hierarchy - using the caches when their hit rates will be high, and bypassing them for low locality operations.

We use cache reuse as our metric of choice to determine which BBLs to offload to our NDP unit. This metric is a first-order heuristic which simply distinguishes which BBLs have access patterns that result in hits in the on-chip caches. To track reuse using hardware structures, we add a 32-bit tag field and flag bit to every line in the L2 cache; the tag stores the identifier of the BBL that caused this line to be loaded, and the flag bit indicates whether this line has received a hit/reuse. Upon evicting a line from the L2, we check the reuse bit and accordingly train our predictors - if the line has not been reused, then the associated BBL will be considered as a candidate for executing using NDP. Adding a full 4B BBL tag and a reuse bit to each way translates to a 33 bit overhead, which translates to a 6% overhead in a cache with 64B data blocks, and 2.5% with 128B blocks.

§4.5.2 shows an analytical model for the prediction accuracy required to save energy overall with Anti-Fetching. Due to the fact that predictor accuracy is critical for performance, we initially assume the use of a oracle predictor which authoritatively determines whether or not a memory access will be reused. This allows us to focus on the efficacy of Anti-Fetching as a technique to save energy, given a way to separate out the reused lines perfectly. However, any realistic predictor can interface with the above mechanism to train its internal structures on the stream of blocks being evicted from the L2.

### Realistic Predictor Design

Designing a realistic BBL predictor is a balance between minimizing the hardware cost, and reducing the aliasing factor between basic blocks. Ideally, we would use a table which stores a unique entry for every candidate BBL, but this is unachievable in real hardware that is on the critical path of the core. We approach this problem by using a counting Bloom filter inside the core, which is queried by the BBL tag being supplied by the branch predictor.

We chose to use Bloom filters from the family of set-storage structures primarily because all lookups, insertions, and deletions proceed in constant time, which is critical for a structure that is tightly coupled
to the processor. Counting Bloom filters provide an extra feature over single-bit filters, as each entry in the filter is an N-bit up-down saturating counter. Changing the counter width and prediction threshold are two extra parameters that are used to adjust how aggressively the core offloads to the NDP units.

Figure 4.9 shows a simple example of how the predictor is trained based on evicted L2 cache lines, as well as showing the phenomena of BBLs aliasing inside the Bloom predictor itself. In this figure, we see the following sequence of events: Firstly, line $A$ associated with BBL $0xFE$ is evicted. When our training mechanism is invoked, we read the reuse bit and see that the line has been marked as reused; therefore, we pass the L2 tag $A$ through the hash functions (h), and then increment the counters inside the Bloom predictor. In contrast, upon evicting line $B$, we see that the line has not been reused. After hashing into the predictor, we decrement the matching counters. The middle bit in the array that is being both incremented (+1) and decremented (-1) demonstrates the aliasing phenomena which depends on the number of BBL tags and choice of hash function. We chose to use a CRC32 polynomial to compute our hash functions, eliminating the need to select the “best” bits from the BBL tag for direct indexing. Methods of implementing CRCs in hardware have been well documented in the literature [23, 30].

### 4.3.4 Sending Offload Requests

We transmit all offloading requests from the core through the regular LSU, in order to preserve memory access ordering. Each offload request transmits an *NDP packet* over the on/off chip interconnect, which is comprised of:

- The tag of the BBL being offloaded (4B).
- All register live-in values (1-8B depending on the register width encoded in the operand).
- The bit-vector representing which registers are live-out (4B for our implementation). This vector is *optional* and is often only transferred once per offloaded BBL.

The final structure in our modified processor core is the Offload Packet Cache (OPC) Mirror, which is simply a copy of which BBTs have their meta-data cached inside the NDP. We discuss the OPC itself more in §4.4. Since the live in/out vectors are identical for each dynamic instance of the BBL, we choose to cache a small number of them (16) inside the NDP as an optimization for eliminating dynamic traffic across the system bus. The structure of the OPC Mirror is a 16 entry direct mapped tag array, which simply stores the tags of which BBLs are resident in the OPC. Every offload request performs a lookup in the OPC Mirror. If this BBL is cached in the NDP, we can eliminate the live out vector.
transfer from the NDP packet, otherwise the full packet is required. We update the OPC Mirror on every offload request, which guarantees that it is identical to the tag array of the OPC itself.

Once the NDP invocation is sent via the LSU, we treat it as any other memory request that produces a register value, in the sense that we mark all of the live-out registers as busy, stalling the core if any other instruction attempts to read them. When the BBL finishes executing in the NDP, and has transferred its live out registers across the bus, we commit them to the register file and redirect the Fetch unit to resume execution at the correct spot.

### 4.4 Executing BBLs Near Memory

After an offload is initiated using the on-chip structures detailed in § 4.3, we need to stall the CPU core until the NDP unit indicates that it has completed execution, and the live-out values have been returned. We accomplish this by adding a special-purpose register that contains the restart address after returning from an NDP request. As previously mentioned, we are transferring the NDP packet to the memory using the same load-store unit and on/off chip fabric that services L2 miss requests and responses. Therefore, an NDP request is functionally the same to the CPU as any other stall that occurs due to memory latency.

On the memory side, we logically separate NDP packets from regular demand misses by adding a single bit which is passed over the communication fabric with every request. NDP packets need to be buffered near the NDP unit itself, whereas regular requests are sent directly to the memory transaction queue to be handled by the DRAM controller. Once an offloaded packet has traversed the communication fabric and arrived at the NDP unit, it then is written into a small tagged storage structure called the offloaded packet cache (OPC). We need this structure to store all of the packet’s context during its execution. The rest of this section explains how an NDP packet is processed.

#### 4.4.1 NDP Pipeline Design

Our NDP accelerator unit (NDPA) is designed for simplicity while maintaining the ability to execute a variety of BBLs. During the BBL extraction process explained in § 4.3.1, we exclude any BBL from being offloaded that contains floating point or vector instructions, as well as special ISA extensions such as AVX, SSE, and SHA. We support NDP execution of predicated instructions, since the NDP packet coming from the processor also contains the values of any relevant predicates. In Figure 4.10, we show a high level diagram of our NDPA and its interface to the physical DRAM. Our design is very similar to the on-chip core with a few key differences, notably in the structures used to fetch and retire instructions.
When a new NDP packet arrives from the core, we look up the BBL tag in the OPC; if a packet with this tag is already resident in the OPC, we can simply over-write the registers corresponding to the live-in registers sent from the CPU. In this case, the CPU would have hit the OPC Mirror and not transferred the live-out vector over the bus. The NDPA then begins fetching instructions from the fetch address passed by the core; by default, this instruction fetch is done directly from DRAM, unless we hit the \( \mu \)-op cache. After the \( \mu \)-ops corresponding to this BBL have been placed in the instruction queue, our NDPA executes them in the same fashion as a regular x86 processor. After the last instruction in this offloaded BBL is committed, we read the live-out bit vector resident in the OPC, and determine which registers need to be sent back to the processor core.

4.4.2 Interface to DDR Controller

All memory requests generated in the NDPA are sent directly to the memory controller as new DRAM transactions. Since the x86 ISA operates on register widths ranging from a byte to a double-word, we use the existing instruction decode step to determine the size of the data which needs to be returned from memory. Due to limitations in our simulation infrastructure, we evaluate offloaded DRAM requests as performing a double-word sized DRAM access, regardless of the register width encoded in the x86 memory operand. This means that we report a slightly greater DRAM energy expenditure than what would exist in a memory controller configured to issue smaller transactions.

Our architecture does not violate memory ordering because we stall the CPU core upon an offload request, and therefore the NDPA’s memory requests can simply be placed in the back of the memory transaction queue without needing additional checks for ordering. We modify the standard formatting of memory transactions by adding a single bit to identify their origin as either coming from the CPU or the NDPA. Once the memory request has completed, the memory controller forwards the response to either the CPU or the NDPA, and continues regular operation. Data that is returned to the NDPA is written into the Commit pipeline stage and can either be written back to the register file, or sent along the NDPA’s forwarding paths to be used in the Issue stage for subsequent instructions.

4.4.3 Continued Prediction Training

Once a BBL begins to be offloaded to the NDP, its memory requests are no longer issued to the cache hierarchy, and therefore are not able to be profiled by the realistic training method described in §4.3.3. This is simply because our method works by inspecting the reuse bit of lines being evicted from the L2, and therefore a BBL which is continually being offloaded will not be associated with any L2 accesses. Fundamentally, in order to usefully update the Bloom predictor via the training process of §4.3.3, we need to return some information back to the L2 cache in order to continually train the Bloom predictor.

To solve this problem, we choose to return 1/64 of the cache lines accessed by the NDPA back to the CPU, writing the data into the cache just as if the request had not been offloaded. This approach is inspired by Qureshi et al. [69] in their Set Dueling design for arbitrating between cache insertion policies in a shared LLC. Conceptually, if a BBL accesses lines which in future will be reused in the cache, we want these lines to be processed by our prediction training mechanism and this BBL to no longer incur offloads to the NDP. The difficulty now arises of choosing which lines should be returned. Since the NDP itself has no a priori knowledge of whether or not this line will incur any reuse, we treat all offloaded BBLs and lines equally. We use a simple 6-bit counter that is decremented every time
a BBL is executed in the NDPA, and return the current line when the counter hits 0. We chose the fraction of $1/64$ via experimentation, having tested fractions from $1/4$ to $1/256$ and selecting the one that provided the highest increase in accuracy to our realistic Bloom predictor in §4.3.3. Without a feedback mechanism to re-train the BBL predictor, we see offloading accuracies drop below 10% due to the lack of any training feedback.

### 4.4.4 BBL Offloading Walkthrough

In this section, we provide a step-by-step walkthrough of how a BBL is fetched, predicted as an offload, sent to the NDPA for execution, and returned to the processor along with its restart point. The overall process is displayed in Figure 4.11 using the same basic block that we displayed in the BBL training process of Figure 4.8. The table shows the cycle at which the instructions enter the IQ (labelled as Q), are issued in a 1-wide pipeline (I), finish execution (E), and commit their results (C). In this example, all three instructions enter the IQ at the same time due to a µ-op cache hit.

In Step 1, the processor decodes the new branch instruction $I_{n-1}$, which is passed to the branch and offload predictors. In this example, we assume that the branch is predicted taken, the prediction target hits in the BBT, and we have chosen to offload it. Step 2 verifies the direction and target prediction after the branch has resolved in CPU cycle 4, and removes $I_0$ and $I_1$ from the pipeline. We then initiate the BBL offload in step 3 by reading the live-in values from the register file, the live-out bit vector from the BBT, and sending the NDP packet over the bus.

The NDP packet is transferred over the interconnect to the DRAM controller, which reads the offload flag and redirects this packet to the NDPA. Step 4 shows the packet being written to the NDPA, and executed in the backend. After the NDPA completes executing this BBL, Step 5 encompasses the NDP request being returned to the processor, with the correct restart point. The CPU then fetches $I_r$ from the PC that is returned from the NDP, which is the first instruction in the next BBL in the control flow.

### Summary

In this section, we described an updated microarchitecture that enables a CPU to dynamically offload BBLs to a near-data processor, and presented a basic compute pipeline that is co-located with the DRAM, and executes offloads sent from the core. We use branch target addresses to identify BBLs, the processor’s decode stage to determine which registers are required for each BBL, and an oracle predictor...
to decide which BBLs are offloaded. Our NDP is a single-issue in-order pipeline that issues requests directly to DRAM, and returns the live-out registers as well as next instruction PC to the CPU. In the next section, we evaluate our design and present our experimental results.

## 4.5 Evaluation and Results

We evaluate our architecture using analytical and simulation techniques. Firstly, we present our methodology and discuss the system parameters modelled. Then, we propose a simple mathematical model that quantifies the tradeoff between accuracy of the BBL Predictor and the overall energy savings attained. We then show absolute performance and energy comparisons to our in-order baseline, and perform design space explorations on showing how reduced prediction accuracy affects Anti-Fetching. Additionally, we repeat these experiments using our realistic Bloom predictor described in §4.3.3 and quantify the prediction accuracy, performance, and energy expenditure when we use this predictor instead of an oracle. Additionally, we show experiments that demonstrate our BBT design captures a high percentage of BBLs while maintaining single-cycle access latency in the processor core. Finally, we present a discussion of the challenges and potential solutions that exist in order to apply Anti-Fetching to more complex systems.

### 4.5.1 Methodology

All of our experiments use Intel Pin [51] to instrument all basic blocks of the SNAP example applications. We fast forward over the section of the application that loads the raw graph file from memory, and run each application for a region of interest (ROI) of 100M memory instructions, which translates to 1-2B total user-mode instructions. Both our baseline core and our NDPA are simply modelled as constant 1 IPC pipelines, with the exception of load-to-use memory latency. We assume perfect branch prediction for all of our experiments.

To simulate the memory, we pass the NDP packets created by Pin to a custom frontend for DRAMSim 2.0 [71], where we extract the memory addresses for each in-memory instruction and issue transactions to DRAMSim. Although we model the relative instruction widths of each x86 operation, DRAMSim calculates the access energy equally regardless of the operand type, and therefore the energy numbers for our offloaded instructions are slightly higher than they should be. However, we estimate that the amount of instructions that access less than a word or double-word in the DRAM is extremely limited and therefore will not significantly impact our results.

See Table 4.1b for the cache sizing and memory timing parameters we used. The caches are sized to match those frequently included in efficient in-order cores [5, 37], and memory timings were extracted from a standard 4GB Micron DDR3-1866 part datasheet [54]. Both the BBT and OPC are small tagged tables, modelled as caches using CACTI 6.5 [56]. To model the power of both our baseline and NDPA, we used McPat 1.0 [48], removing the branch predictor and target buffer, data caches, and floating point units from the NDPA. The key parameters of our NDPA are also summarized in Table 4.1b.

### 4.5.2 Analytical Model

The efficacy of our design depends primarily on two key parameters: how frequently the application reuses data in the caches, and the accuracy with which we offload BBLs to the NDP. Let \( N_{\text{lines}} \) represent the
## Anti-Fetching: A Dynamic NDP Architecture

### Data Structure Budget (B)

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Budget (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique BBL Tag</td>
<td>4B</td>
</tr>
<tr>
<td>MRU Flag</td>
<td>1 bit</td>
</tr>
<tr>
<td>Live In/Out Vectors</td>
<td>2 * 32 registers = 16B</td>
</tr>
<tr>
<td>Sets x Ways</td>
<td>64 x 2</td>
</tr>
<tr>
<td>Total</td>
<td>3104 B</td>
</tr>
</tbody>
</table>

(a) Hardware Budget for Basic Block Table

### CPU NDPA

<table>
<thead>
<tr>
<th>CPU</th>
<th>NDPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Width</td>
<td>2</td>
</tr>
<tr>
<td>Phys Registers</td>
<td>32</td>
</tr>
<tr>
<td>Integer ALUs</td>
<td>2</td>
</tr>
<tr>
<td>Split L1 I/D</td>
<td>64kB, 4-way, 3 cycle</td>
</tr>
<tr>
<td>Unified L2</td>
<td>2MB, 8-way, 14 cycle</td>
</tr>
<tr>
<td>Interconnect</td>
<td>400MHz Bus, 64-bit link</td>
</tr>
<tr>
<td>OPC</td>
<td>16 entries, 256B Total</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB, 933MHz, 8-bank</td>
</tr>
<tr>
<td>Timings (ns)</td>
<td>$t_{CL} = t_{RC/D} = t_{RP} = 13$</td>
</tr>
</tbody>
</table>

(b) Simulated Configuration for Cores, Caches, and Memory

### Table 4.1: Parameters Used to Simulate Anti-Fetching

<table>
<thead>
<tr>
<th>Core Type</th>
<th>L1 Parameters</th>
<th>L2 Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex A7</td>
<td>64kB, 64B blocks, 4-way</td>
<td>1MB, 64B blocks, 8-way</td>
</tr>
<tr>
<td>Intel Core, Models -17H</td>
<td>64kB, 64B blocks, 8-way</td>
<td>4MB, 64B blocks, 16-way</td>
</tr>
</tbody>
</table>

Table 4.2: Cache Parameters Obtained from ARM Cortex A7 [5] and Intel Core (model codes -17H) [37]

The number of cache accesses in a program, $P$ represent the accuracy of our BBL predictor, $R$ represent the percentage of lines that are reused in the caches, and $E_x$ represent the energy cost of accessing either the L1 cache, L2 cache, or DRAM. We can therefore express the total energy reduction of Anti-Fetching by the following expression:

$$E_{\text{reduction}} = N_{\text{lines}} * P(1 - R) * (E_{L1} + E_{L2})$$

$$-N_{\text{lines}} * R(1 - P) * (E_{\text{DRAM}} - E_{L1})$$

The two terms in Equation 4.1 correspond to: the number of non-reused lines we correctly Anti-Fetch into the NDP, thus saving accesses to all levels of the cache, and the number of lines that are incorrectly Anti-Fetched instead of being executed on-chip, incurring an extra DRAM access as a penalty. Note that in this equation, we do not directly consider the case of mispredicting a non-reused line, because this constitutes a lost opportunity rather than a penalty. For this simple model only, we do not include the other overheads such as BBT accesses or bus transfers, as they are minor contributors to the overall energy expenditure.

Simplifying Equation 4.1 and setting $E_{\text{reduction}} \geq 0$, we obtain the following inequality:

$$P \geq \frac{E_{\text{DRAM}} - E_{L1}}{E_{\text{DRAM}} - E_{L2} - 2E_{L1} + \frac{E_{L1} + E_{L2}}{R}}$$

In the limit, as the application’s reuse rate approaches 1, the prediction accuracy required to improve energy also approaches 1, and vice versa. Figure 4.12 shows the solutions to this inequality for two memory hierarchies that are representative of the simple caches inside the ARM Cortex A7, and the complex ones in a high performance Intel Core microarchitecture. The cache sizes and parameters used for the solutions are given in Table 4.2. We obtain the key parameters such as size and associativity from available datasheets [64, 37], and model the cache access energies using CACTI [56]. For obtaining $E_{\text{DRAM}}$, we use the frequently cited figure for DDR3 of 70 pJ/bit [67, 52] multiplied by a double-precision 8B access.
Figure 4.12: Predictor Accuracy Requirements Based on Equation 4.2.

According to this simplified model where we only consider the cost of accessing the storage structures themselves, prediction accuracy is paramount in order to extract energy savings. As the average energy expenditure in the cache hierarchy grows, the required prediction accuracy becomes more favourable for all values of $R$, as shown by the curve for complex caches in Figure 4.12.

**4.5.3 Oracle Prediction Results**

In order to satisfy the high accuracy standards predicted by our model, we gather our primary result using an oracle predictor that can perfectly remove all of the accesses that do not incur any reuse. Although not part of our analytical model, our simulations consider all energy in the memory hierarchy, including BBT accesses and bus transfers.

Figure 4.13 shows the energy savings in the memory hierarchy using Anti-Fetching with an oracle predictor, and the percentage of L2 accesses that we save by executing these basic blocks in our NDP.
Our architecture saves an average of 11% energy in the memory hierarchy, up to a maximum of 31% in centrality. This savings is directly proportional to the amount of L2 accesses we eliminate with Anti-Fetching. Our results demonstrate that specifically targeting the BBLs which do not benefit from caching is an efficient way to save energy from excess lookups in the memory hierarchy.

Figure 4.14 shows the speedup inside the ROI using perfect Anti-Fetching. All of the applications have their overall runtime affected by less than 1%. This is because the amount of BBLs affected by our perfect predictor occupy a very small fraction of the overall ROI execution time, as well as the fact that Anti-Fetching has a limited impact on the overall latency of an offloaded BBL. In the baseline case, a zero-reuse access incurs a miss at both levels of the cache hierarchy of approximately 4 and 14 cycles, given in Table 4.1b. The request is then sent to the memory itself, incurring the latency of the system bus, traversing the DRAM request queue, and then the memory read itself. In contrast, we model the latency for a BBL to be offloaded as the following:

- Resolving the offload prediction and looking up the target address in the BBT (1 cycle).
- Sending the NDP packet over the bus (2+ cycles). Recall that we are only sending the BBL tag, live-in registers, and potentially the vector representing the live-outs, not decoded instructions themselves.
- Executing the BBL in-memory ($N_{\text{inst}} + N_{\text{DRAM}}$ cycles), where $N_{\text{inst}}$ is the number of instructions in the BBL, and $N_{\text{DRAM}}$ is the latency of the actual DRAM request.
- Returning the relevant live-out registers (2+ cycles).

Comparing the latencies between a locally executed and offloaded BBL, we note that the number of cycles to execute the memory read is constant, since the same address is issued in both cases. Also, executing the BBL takes the same number of cycles in our constant 1 IPC model. Therefore, the latency difference between the baseline and offloaded case comes down to the difference in the number of clock ticks required to send values across the bus. A regular DRAM transaction incurs a 8B address request and 64B data response for a full cache block, which takes 9 clock ticks to complete, notwithstanding the latency of the memory response. Since both our BBL tag and live-out bit vector are 4B, we can
concatenate both of those into one 8B bus transaction. Since our evaluation faithfully models the variable register widths in x86, we calculate the number of bytes required for each live-in and live-out register and calculate how many registers can be “packed” into a single bus transaction using the system’s bus width (in our case, 8B).

This results in a variable number of bus transactions, which introduces a slight complexity in designing the bus protocol, since all of the components which share it must know how many cycles to wait before the bus becomes available again. Therefore, before sending the BBL tag, register values, or live in/out vectors, we transmit a single header transaction that identifies the following \( N \) cycles as an offloading request destined for the DRAM controller. Therefore, the final latency comparison between offloading and on-chip execution results in a comparison between the 18 cycle penalty of missing both on-chip caches, and the number of registers that have to be transferred across the bus, along with their respective widths. Depending on the number of registers, we may incur an overall latency benefit or penalty. We discuss communication between the core and the NDPA further in \( \S \) 4.5.7.

### 4.5.4 Reduced Prediction Accuracy

We now quantify the affects of increased misprediction rates, to verify our model presented in \( \S \) 4.5.2. To simulate a certain misprediction rate, we take the authoritative prediction from our oracle, and invert it with the specified misprediction frequency. We implement mispredictions by drawing a random variable \( R \) from a uniform random distribution in the interval \( [0, 1] \), and mispredicting if \( R \geq 1 - P \). This means we are not restricting our mispredictions to false negatives, but also simulating false positives where data that will be reused is not brought into the caches on its first reference, and will incur an extra DRAM the second time it is accessed.

Figure 4.15 shows the evolution in performance and full-system energy when we decrease the accuracy.
Figure 4.16: Prediction accuracies for 1kB Bloom filter compared to Infinite sized map. Zero reuse BBLs are separated.

of our oracle predictor. Since the applications we choose have reuse rates between 20-50%, and our baseline design is in-line with the Simple curve in Figure 4.12, we expect according to our analytical model that both metrics will be extremely sensitive to prediction accuracy.

In all applications shown, we notice that a significant slowdown occurs once the misprediction rate reaches 10%, because of the excess latency incurred by the wrongly predicted offloads. The energy expenditure follows a similar trend, albeit with slower growth. At the same 10% misprediction rate, our technique has an average energy overhead of 7% across all applications. Comparing this result to our mathematical model of Equation 4.2 and Figure 4.12, we see that the model slightly under-predicts the accuracy required in order to achieve an energy savings.

The application centrality has a L2 reuse rate of approximately 20% from looking back at Figure 4.3. Since our evaluated architecture nearly matches the cache and DRAM energy requirements plotted on the Simple line in Figure 4.12, we would expect that approximately 90% prediction accuracy would still yield energy savings. However, Figure 4.15b shows that a 90% prediction accuracy (shown as a 10% misprediction rate), actually incurs a 5% energy overhead. This is due to the fact that our model did not consider the energy cost of the extra structures such as the BBT and OPC required to support offloading, as well as the fact that Figure 4.15b considers energy expended in all sources of the chip, not just the cache hierarchy as the mathematical model does. Table 4.3 shows a comparison of the accuracies required for Anti-Fetching to expend the equal amount of energy as the baseline system, as predicted by our mathematical model and measured with simulation. The reuse rates are obtained from the same experiment as in Figure 4.3, and in every benchmark we see that our model slightly under-predicts the accuracy required to break even in total energy. We now turn to discussing our results for the realistic Bloom filter predictor described in § 4.3.3.

4.5.5 Results Using Bloom Prediction

In this section, we perform the same experiments as § 4.5.3 but substitute the realistic Bloom implementation of § 4.3.3 for the oracle predictor previously used. We provision our Bloom filter with 1kB of storage, and use 4-bit signed counters that need to be saturated at +6 to predict a BBL for offloading.
For hashing, we make use of the CRC polynomial given in the Ethernet, PNG, and Gzip implementations [14]. We first show our Bloom predictor’s accuracy in Figure 4.16, and compare it to a predictor based on an infinitely sized predictor, based on a std::map. Both predictors use the same saturating counter threshold. The first two bars for each benchmark show the overall accuracy, which takes into account predicting BBLs that should and should not be offloaded. We separate predictions for the BBLs that are offloaded by our oracle predictor in the last two bars. Our predictor averages 62% accuracy overall, and 57.9% on the BBLs that should be offloaded.

However, as we have already demonstrated, predicting offloads must be done with an extremely high accuracy in order to attain energy and performance gains. In many cases, the Bloom predictor actually outperforms the infinite sized predictor in total accuracy, but is only better for the zero-reuse BBLs in agmf and coda. Although saturating counters are extremely useful in branch direction prediction, applying them to reuse prediction is clearly unsuitable. This is because saturating arithmetic serves as a proxy for storing the “pseudo-last” state of the prediction key. Although branches tend to be strongly correlated with their last outcomes, we do not observe the same behaviour for cache reuse. Examining a trace of the oracle prediction outcomes for each block, we notice that the offloaded instances are interspersed throughout, and are unlikely to be captured with the outcome of a saturating counter.

Figure 4.17 shows a breakdown of execution cycles between the core, caches, DRAM, and NDPA for our baseline and Anti-Fetching design using the Bloom predictor. On average, using our Bloom predictor induces a 9% slowdown on our test set of applications, with the exception of forestfire (fore) which has a 13% speedup and motifs which has 7% speedup.

We show the energy usage in the memory hierarchy for Bloom prediction in Figure 4.18, with bars normalized to the baseline expenditure. Therefore, the height of the offloading bar represents the energy overhead from using a realistic but inaccurate predictor. On average, excess DRAM accesses consume 69.6% extra energy, which leads to the large overheads we see by using this predictor. This type of slowdown and energy excess is to be expected with low-accuracy predictors, as we showed before in §4.5.2 and §4.5.4.
We plot the fraction of cycles in which the NDPA is active in Figure 4.19 using realistic Bloom prediction. On average, the NDPA is used for less than 2% of the application ROI, with a maximum of 5.9% in cliques. In the oracle case, the NDPA would be used even less, due to our previous results that demonstrate that our hardware predictor sends approximately an order of magnitude more offloads to the NDPA. Comparing our cycle stacks of Figure 4.17 with the original reuse rates we profiled in Figure 4.2, we notice that applications such as centrality, cliques, and netstat which exhibited larger numbers of full cache misses are the same ones that incur significant slowdown and energy overhead in our final results. Although we attribute most of those adverse effects to the limited accuracy of our realistic predictor, we also acknowledge that applications such as cascades and infopath which we attain prediction accuracies over 95% see 0.5% and 12% slowdown, respectively. In order to identify the reasons for this behaviour, we now present a discussion of the behaviour of netstat, the same application whose basic block latencies we presented as motivation in Figure 4.4b and contrast it with motifs, an application which obtains both a speedup and energy savings using our architecture.
Discussion of Netstat and Motifs

Looking back at Figures 4.4b and 4.5, the BBLs which incur significant L2 miss latency are numbered 5-10 and 20-25. The first group occupies 20.8% of the overall dynamic instruction count in the program, and the second group approximately 3.5%. When using normal optimization flags, the basic blocks in both groups contain 3-5 XMM vector instructions per BBL. Since our NDPA does not support vector instructions and we exclude these blocks are from being offloaded, we re-compile netstat, still using -O3 optimization flags, but explicitly forbidding vector instructions. We then re-execute the experiment which simulates the cache behaviour of each BBL.

The BBLs which contained octa-word (32B) wide memory accesses for the vector instructions are converted into single-access additions which heavily use movsd x86 instructions to move data in and out of the core. This instruction is a built in “copy and increment” operation that moves a double-word to the destination operand, and increments the esi or source index register. Figure 4.20 shows the execution latency breakdowns of the original and de-vectorized programs side-by-side. The program which uses loops instead of vector instructions has approximately 4.5x the runtime of the vectorized version, since it does not make use of the parallelism offered by SIMD instructions. However, its L2 reuse fraction increases from 0.10 to 0.86, which is a byproduct of the fact that the core incurs cache hits to every datum but the first one stored in the same cache line. Effectively, vectorizing the program has introduced single-use cache lines, but reduces program runtime through the use of SIMD instructions. The overall amount of data transferred across the bus, however, remains the same, since the CPU still operates on the same amount of data.

When we execute the non-vectorized code with Anti-Fetching, the utilization of our NDPA drops by an order of magnitude, which makes sense when we consider that our architecture relies on observing lines with no L2 reuse to consider BBLs for offloading. Therefore, we conclude that although our design is not entirely suitable for saving energy in programs heavily characterized by loops, the addition of

Figure 4.20: Comparison of Netstat Execution Time Breakdown When Disallowing Vectorization

\[
\text{mov} \quad \text{rax} , \quad \text{dword} \quad [r15+0x2c]  \\
\text{mov} \quad \text{rdx} , \quad \text{dword} \quad [rax+rdi+1]  \\
\text{cmp} \quad r14 , \quad rdx  \\
\text{jnl} \quad 0x407300
\]

Figure 4.21: Latency Critical Basic Block for Motifs
SIMD operations to our NDPA would have yielded higher utilization on benchmarks like netstat that perform wide memory accesses.

Motifs is an application that counts common subgraphs inside network nodes, and has two particular BBLs that show opportunity for offloading. We display the most common one in Listing 4.21 which consumes approximately 12% of the instructions in the program and spends 78% of its cycles waiting for memory. This BBL performs an array indirection operation, where the data inside r15 is read into rax, and then used as the address for the next instruction. Although we see in Figure 4.19 that motifs spends less than 1% of its cycles computing instructions in the NDP, Figure 4.17 shows that the speedup we obtain occurs primarily due to a reduction in cycles waiting for the DRAM. This finding is consistent with the fact that although the number of times this BBL is offloaded to the NDPA is very limited, the speedup we obtain comes from reducing the delay between the first and second memory transactions.

4.5.6 BBT Hardware Design

We now present our experimental results that show how we designed the BBT, and quantify the lost offloading opportunity from BBT misses on offload-predicted blocks. Sizing the BBT is an important design choice to provide the correct scope for both storing and offloading BBLs. A BBT that is too small may exclude certain BBLs from being considered for in-memory execution, since we will incur an energy penalty to re-train each BBL from scratch; a BBT that is too large has increased access time and leakage power, both of which are critical attributes to minimize in any chip design. Due to the fact that each entry costs 22B of storage (shown in Table 4.1a), increasing the number of sets quickly becomes expensive in terms of storage capacity as well as access time. At the same time, our goal is to minimize BBT misses, since each miss constitutes a lost offloading opportunity. To quantify this design tradeoff, we performed experiments with different BBL organizations, to determine the amount of offloading opportunity lost with finite-sized BBL storage. We chose to model the BBT as a simple cache-like structure where the unique BBL target address serves as the tag, and performed full application runs to determine the hit rates.

All set-associative designs use a simple NMRU replacement policy. Figure 4.22 shows the miss breakdown for both direct mapped and 2-way associative BBT designs, each with the same amount of raw capacity: 128 total BBL entries. We sub-divide the misses according to Hill’s 3C model for caches [34]. Both bars are normalized to the direct mapped design; the height of the 2-way bar shows the overall reduction in BBT misses attained by using an associative design. Table 4.4 shows the overall BBT hit rate for each benchmark, since reducing the number of conflict misses has a relative effect on the overall BBT hit rate. Although the cold misses are not directly visible in this graph, we see an average of 0.04% cold misses across all of the benchmarks. This occurs because of the large discrepancy between the number of unique BBLs in each application, and the number of times each BBL is dynamically executed. For example, we observe approximately 1500 cold misses per application, but upwards of 19 million conflict misses. Cold misses only depend on the raw code footprint of the application, where conflict and capacity misses are repeatedly incurred as the application continues running. On average, the direct mapped design has a hit rate of 79.7%, and the 2-way associative design of 84.9%. The improved hit rate is most apparent in kcores, infopath, and community. Although netstat and centrality show 81% and 74% reduction in conflict misses respectively, both of these benchmarks exhibit the extremely common BBL sequences of § 4.2 and have BBT hit rates in excess of 93%. Increasing the associativity to 4, yielding a 32x4 overall layout, did not provide significant miss reductions in our experiments.
Using CACTI [56] to model the BBT as a small cache, we see that the 64x2 organization requires approximately 50% higher energy per access, about 3.35pJ over the 2.50pJ of the direct mapped design. The leakage power between the two is approximately equal, as the overall data array size is the same, and the extra lookup cost comes from performing two tag comparisons in parallel. Given that these energy numbers are extremely small compared to the rest of the core itself, we choose the more associative BBT to minimize the miss rate, since BBLs incurring a miss cannot be offloaded. Both direct mapped and 2-way associative designs have an access time below 0.37ns, which is within the target 2GHz clock frequency of our CPU (clock period of 0.5ns).

Increasing the BBT size to 256 entries slows the cycle time to 0.87ns, and since the BBT is on the critical path of the processor during an offloading operation, we choose the 64x2 design to preserve clock frequency. Now that we have shown a 64x2 BBT satisfies the cycle time constraints for a 2GHz processor, we quantify the amount of offloading opportunity that we lose from using a finite-size BBT - recall that regardless of predictor, a BBL incurring a BBT miss cannot be offloaded. We gather this data by counting the number of oracle offload predictions, and cancelling the instances where the BBL being offloaded incurred a BBT miss. In Figure 4.23, we show the opportunity lost due to BBT misses, for both the direct mapped and 2-way associative designs. The dark bars labelled DMAP represent the increase in offloads lost due to the use of a direct mapped BBT over a 64x2 design. On average, we see that by using a 2-way associative BBT, we preserve 6% more offloading opportunity on average, up to a maximum of 16.1% in agmf and 14.1% in centrality.

4.5.7 Bus Communication

Finally, we evaluate the extra communication on the bus that takes place due to Anti-Fetching. When discussing the execution time effects of offloading a block in § 4.5.3, we demonstrated that the address request (8B) and data response (64B) of a regular DRAM request balances out the bus cost of sending the BBL tag (4B), its live-out vector (4B), as well as up to 64B of live in/out registers. Therefore, the energy savings of offloading a BBL using our oracle predictor is, where $E_{bus}$ represents the energy of a
single bus transaction, and \( w_r \) represents the byte-width of a live in/out register:

\[
E_{\text{savings}} = E_{\text{bus}} \times \left( \sum_{\text{offloads}} \left( 64 - \sum_{\text{regs}} w_r \right) \right)
\]  

(4.3)

Intuitively, as the number of registers we need to transmit increases, offloading a block becomes more costly in terms of both latency and energy. Since energy-consumption numbers for serial IO devices are difficult to obtain without acquiring industry IP such as Cadence Wide IO \cite{16}, we choose to evaluate our communications overhead by measuring the bus activity factor over 100k cycle intervals. The bus activity factor is simply defined as the number of clock ticks the bus is active, divided by the interval width (100k cycles). In computing the bus activity, we include all transfers from both offloads and regular DRAM transactions. We then use the dynamic power of the off-chip bus reported by McPat \cite{48} to estimate the extra energy expended due to the bus’ higher activity factor. Additionally, we track the total amount of data transferred over the bus during the application ROI. Figure 4.24 shows a comparison between total data (in MB) sent across the bus. We see that for all of the benchmarks, the baseline and offloaded case are approximately equal. This makes sense when we combine the benchmarks in Table 4.5 to Equation 4.3. On average, a DRAM request and data fill consume 9 bus cycles as previously described. All three benchmarks require less than 6 bus clocks to actually be offloaded, and this means that we obtain a net gain in latency from offloading these blocks. However, we do not save significant bandwidth because the number of blocks offloaded form less than 5% of all BBLs comprising the program. The same trend is true across all the benchmarks when using an oracle predictor.

In Figure 4.25 we show the time-evolution of the bus activity factor for \textit{cascades}, \textit{centrality}, and \textit{kronem}, compared between a baseline case with no offloading and oracle Anti-Fetching. From this chart, we make the following observations: firstly, the utilization factors for \textit{cascades} are relatively similar over the course of the entire execution. This is because this benchmark offloads less than 1.5% of its basic blocks. In the most offload-intensive application, \textit{centrality}, we see three phases of execution, which shift at approximately 2.5M and 15M cycles, respectively. Although the raw activity factors in all three phases are different, we see that the average bus activity factor is approximately doubled when we apply Anti-Fetching. The behaviour of \textit{kronem} is markedly different than the first two applications, in that the last 15M cycles report an average reduction in the bus activity factor of 45%. We attribute this behaviour to a single BBL that uses only one live-in register and requires no-live outs to be returned to the CPU,
Figure 4.24: Megabytes of data sent across the system bus in the application ROI. We include regular memory requests and offloads in this comparison.

<table>
<thead>
<tr>
<th>Application</th>
<th>Live-In Regs</th>
<th>Live-Out Regs</th>
<th>Bus Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>cascades</td>
<td>4.61</td>
<td>3.84</td>
<td>5.29</td>
</tr>
<tr>
<td>centrality</td>
<td>4.3</td>
<td>2.91</td>
<td>3.71</td>
</tr>
<tr>
<td>kronem</td>
<td>1.25</td>
<td>1</td>
<td>1.21</td>
</tr>
</tbody>
</table>

Table 4.5: Average Number of Live-in Registers and Bus Transactions Per Offload in cascades, centrality, and kronem.

and performs two dependent memory accesses in the DRAM. Executing this BBL on-chip would require multiple DRAM transactions for each memory request, but by offloading it, we save approximately half of the bus communication overhead.

Table 4.5 shows the average number of live-in registers, and excess bus clocks required for these three benchmarks, as given by the sum in Equation 4.3. The smaller number of excess transfers for kronem matches the behaviour we see in the latter half of Figure 4.25. In absolute terms, we notice that the absolute maximum activity factor in our experiments is 8.3%, which means that less than one tenth of the cycles executed in the CPU have an active transaction on the bus. We use McPat [48] configured with the simulation parameters in Table 4.1b to compare the relative impact on chip power that we incur from doubling the relative bus activity factor. Our in-order baseline core augmented with all structures required for Anti-Fetching consumes approximately 3.59W, of which 65.2 mW is comprised of bus power. Doubling the dynamic power consumed by the bus yields approximately a full-chip power overhead of 1.8%. We therefore conclude that our technique imposes modest overheads on the interconnect and chip itself.

Summary

In this section, we presented Anti-Fetching, an architecture for dynamic near data processing. Our architecture adds less than 4kB of hardware storage to the processor to train and gather all of the context required to transmit basic blocks from the CPU to be executed into the NDP. We proposed a simple analytical model to relate the prediction accuracy required to save energy to the application’s reuse rate. We then evaluated our architecture with a oracle and realistic predictors, demonstrated up to 31% energy savings in the memory hierarchy (11% average), and demonstrated that our analytical model accurately captures the accuracy-energy tradeoff of using our architecture. Finally, we discussed
a potential hardware implementation of the BBT which holds all of the offloading context, as well as quantified the bus communication overhead ensuing from using *Anti-Fetching*. 
Figure 4.25: Bus Activity Factor Evolution During Application ROI (Interval size 100k cycles)
Chapter 5

Future Work and Conclusion

Throughout this work, we have assumed uniprocessor baseline architectures, both for simplicity of evaluation, and because of the trend in literature towards simpler cores that rely on parallelism. Applying Anti-Fetching to a multi-core system would require offloading BBLs in a way that maintains coherent caches, and whatever memory consistency model that we choose to enforce. Since the NDPA essentially serves as another core with reduced functionality, new ordering points will have be set and maintained in the overall memory system. Although our ultimate vision is to see our proposed Anti-Fetching architecture applied to a more complex multi-core system, the immediate next step is to explore methods for mitigating the adverse effects of using realistic predictor designs. Given the results of our analytical model, we can either do this by boosting the accuracy of our predictors, or by increasing the amount of energy that we save on a correct offload.

Our architecture currently targets single basic blocks that execute in isolation from the overall context of the program, but we could expand our offloading framework to target sequences of basic blocks or loops. This idea coarsens the granularity of our NDP, and potentially allows us to capture streaming and scanning memory access patterns while limiting the energy spent on synchronizing the core and NDP after each basic block. We could also accomplish the goal of a coarser-grain NDP by interfacing with already existing operating system abstractions that deal with memory allocation. For example, memory-mapped files allow users to read and write data to specific segments of the memory space without incurring the overhead from paging. By adding compute-enabled file streams to the standard programming model, users could specify simple computation patterns which are to be executed in parallel on large regions of memory, and execute them on large arrays of NDPs similar to the one we describe in this thesis. Additionally, interfacing with the compiler may yield higher prediction accuracies, although this type of interface would no longer make our architecture transparent to the programmer and system stack.

If we wish to keep the same basic block level of offloading as presented in this thesis, the next step is to research realistic hardware predictors that can predict block reuse with high accuracy. We may be able to learn from relevant works in branch direction prediction, utilizing some form of global and/or local history of the sequence of cache misses generated by each BBL. Also, architects have proposed many dead-block predictors that attempt to determine which blocks the core is finished using, which is a similar problem to what we are attempting with offload prediction. By extending ideas such as these, we may be able to increase the raw accuracy of our realistic predictors and begin to see energy gains.
even without requiring oracle knowledge.

Another potential direction to pursue as future work for *Anti-Fetching* is to completely remove the backend of the on-chip CPU, and replace it with a centralized frontend whose only job is to dispatch batches of instructions to small processing units that are best equipped to handle them. This extends our observation that because of limited cache reuse, some computation is best performed off-chip. However, that concept can be extended to system where each layer of the caches, DRAM, non-volatile memory, and disk may have their own type of near-data processor, which is controlled by the centralized instruction front-end. This type of future architecture would have exclusively compute-enabled storage, where the amount of raw throughput scales with the overall system’s storage capacity, mitigating the effect of data footprints scaling faster than computing power. If possible, this type of approach would be extremely valuable in in the big data era.

Although the field of prefetching is mature, there is one clear avenue of future work on our *Tempo* prefetcher - exploring using our sliced Pattern History Table (PHT) for imposing an approximate order on the prefetches themselves. Recall that currently, our 4 slices simply store the accesses that occur with repeated patterns in cache access time. However, we recognize that our design does nothing to impose a notion of absolute order between any of the prefetches in the same page. In order to sort the prefetches in a page, we could utilize the existing slices of our PHT to individually store the first $N$ blocks, followed by the next $N$ and so on. This means that at prefetch time, we would prefetch all of the blocks in the first slice before any in the other slices, prioritizing the first few blocks that occurred references at training time.

5.1 Conclusion

In this work, we considered two methods of co-locating code and data to save chip energy: the *Tempo* prefetcher, and dynamically offloading code to be executed by a near data processor. Both methods rely on the concept that caching all data on the chip is not the best approach for achieving efficiently scalable computing. *Tempo* is an extension onto the state of the art Spatial Memory Streaming (SMS) prefetcher, and works by comparing the relative timings of the core’s L2 memory requests and the memory system’s demand responses. By comparing these two data, *Tempo* determines whether or not the memory system will likely prefetch cache blocks in time, and eliminates prefetches that are likely to be slowed down in the caches and DRAM because of higher resource contention. We compare *Tempo* against SMS using the SPEC CPU2006 applications and show an average reduction in useless prefetches by 17.6%, along with an average IPC increase of 2.54% (20.4% maximum).

Secondly, we propose *Anti-Fetching*, a novel mechanism for near-data processing (NDP) that saves energy by identifying cache blocks that are not reused by the processor core, and offloads the code that originally requested those blocks to a processor co-located with the DRAM. We profile the Stanford Network Analysis Platform (SNAP) example applications, and demonstrate that the trend of cache blocks with zero reuse is prevalent. We design on-chip structures that interface with existing microarchitecture to dynamically capture the common basic blocks in the SNAP applications, and choose which of these blocks should be sent to execute in the memory. Our structures come at the cost of less than 4kB of hardware. To execute these common basic blocks, we design a basic near-data processor architecture that is co-located with the DRAM itself, and describe how the core and NDP communicate for preserving program correctness. Using a perfect offloading predictor, *Anti-Fetching* obtains up to 31% (average
11\% energy savings in the memory hierarchy. We also demonstrate an overall reduction in data transfer across the bus interconnect of 1.9\%, which demonstrates that transferring register live in/outs across the bus does not induce excess traffic.
Bibliography


