Hardware Acceleration for Unstructured Big Data and Natural Language Processing

by

Prateek Tandon

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2015

Doctoral Committee:

Associate Professor Thomas F. Wenisch, Chair
Assistant Professor Michael J. Cafarella
Professor Dennis M. Sylvester
Assistant Professor Linjia Tang

TABLE OF CONTENTS

LIST OF FIGURES iv

LIST OF TABLES vi

LIST OF APPENDICES vii

ABSTRACT viii

CHAPTER

I. Introduction 1

1.1 HAWK: Hardware Support for Unstructured Log Processing . 2
1.2 An Accelerator for Similarity Measurement in Natural Language Processing 4

II. HAWK: Hardware Support for Unstructured Log Processing 7

2.1 Introduction 8
2.2 Problem Description 12
2.2.1 Desiderata for a Log Processing System 13
2.2.2 Regular Expression Parsing 17
2.2.3 Conventional Solutions 17
2.3 Background 18
2.4 HAWK in Principle 21
2.4.1 Preliminaries 22
2.4.2 Key Idea 23
2.4.3 Design Overview 25
2.5 HAWK Architecture 26
2.5.1 Compiler 27
2.5.2 Pattern Automata 29
2.5.3 Intermediate Match Unit 31
2.5.4 Field Alignment Unit 32
2.6 Experimental Results 34

2.6.1 Experimental Setup . . . 34
2.6.2 Performance . . . 37
2.6.3 ASIC Area and Power . . . 41
2.6.4 FPGA Prototype . . . 43
2.7 Related Work . . . 44
2.8 Impact of Technology Trends on the HAWK Design . . . 45
2.9 Conclusion . . . 47

III. A Software Implementation of HAWK's Functionality . . . 48

3.1 Introduction . . . 49
3.2 Design . . . 51
3.2.1 Compiler . . . 51
3.2.2 Pattern Matcher . . . 52
3.3 Experiments and Results . . . 54
3.3.1 Experiments . . . 54
3.3.2 Results . . . 55
3.4 Conclusion . . . 59

IV. A Hardware Accelerator for Similarity Measurement in Natural Language Processing . . . 61

4.1 Introduction . . . 62
4.2 Related Work . . . 64
4.3 Design . . . 65
4.3.1 Constructing a Semantic Search Index . . . 65
4.3.2 Accelerator Architecture . . . 67
4.4 Methodology . . . 70
4.4.1 Simulation . . . 70
4.4.2 Timing, Power, and Area Analysis . . . 72
4.5 Results . . . 72
4.5.1 Performance . . . 73
4.5.2 Area and Energy . . . 76
4.6 Conclusion . . . 77

V. Conclusion . . . 78

APPENDICES . . . 81

BIBLIOGRAPHY . . . 126

LIST OF FIGURES

Figure

2.1	A sample log file	13
2.2	An Aho-Corasick pattern matching automaton	19
2.3	Block diagram of the accelerator architecture	22
2.4	The multicharacter bit-split pattern matching automata compilation algorithm	27
2.5	Three-step compiler operation for a 4-wide accelerator and three search terms (W=4, S=3)	28
2.6	Operation of the major string matching subunits over three cycles	30
2.7	Query performance for the single pattern search task on synthetic data, across varying selectivities	38
2.8	Query performance on real-world text data, for varying numbers of search patterns	39
2.9	Query performance for complex predicates task, across varying selectivities	40
2.10	Area requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip)	41
2.11	Power requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip)	42
3.1	Functionality of the software pattern matcher	52
3.2	Processing rates for W=1	55
3.3	Processing rates for W=2	56
3.4	Processing rates for W=4	56
3.5	Processing rates for W=2 (Line graph)	57
4.1	High-level description of semantic search index construction	66
4.2	Accelerator block diagram	67
4.3	Speedup	73
4.4	L2 and memory mus utilization	74
A.1	Experimental configurations for characterizing the costs associated with remote accesses	88
A.2	Runtime breakdown on the Server and representative Reader for the CPU-intensive microbenchmark	90

A.3 Runtime breakdown on the Server and representative Reader for the I/O-intensive microbenchmark 91
A.4 Random data placement in a Hadoop cluster 96
A.5 Partitioned data placement 97
A.6 Selective replication of popular data blocks within partitions 97
A.7 Local access fraction as a function of cluster availability 102
A.8 Local access fraction versus number of replicas for 10% cluster availability 102
A.9 Effect of selective replication of popular blocks on local access fraction 103
A.10 Percentage of remote accesses as a function of data placement policy on a real Hadoop cluster 105
B.1 Memory and inter-cache bandwidth utilization for non-SPEC benchmarks 112
B.2 Memory and inter-cache bandwidth utilization for SPEC benchmarks 113
B.3 Average L1 miss latency for the non-SPEC benchmarks given symmetric memory read and write latencies 117
B.4 Average L1 miss latency for the SPEC benchmarks given symmetric memory read and write latencies 117
B.5 Percent change in average L1 miss latency for the non-SPEC benchmarks under conditions of asymmetric memory read and write latencies 118
B.6 Percent change in average L1 miss latency for the SPEC benchmarks for asymmetric memory read and write latencies 119
B.7 Percent energy change when adding an L2 for the non-SPEC benchmarks 122
B.8 Percent energy change when adding an L2 for the SPEC benchmarks 123

LIST OF TABLES

Table

2.1	Notation associated with HAWK design	20
2.2	Provisioned storage — per bit-split state machine, and total	29
2.3	Server specifications	34
2.4	Component area and power needs for 1-wide and 32-wide configurations	41
3.1	Notation associated with software implementation	51
4.1	Simulation parameters	69
4.2	Statistics for Twitter and Wikipedia datasets	70
4.3	Synthesis results	70
4.4	Power and area results	76
A.1	Runtimes for various I/O configurations	89
B.1	Memory access latency inflection points	120

LIST OF APPENDICES

Appendix

A. Minimizing Remote Accesses in MapReduce Clusters 82

B. PicoServer Revisited: On the Profitability of Eliminating Intermediate Cache Levels 107

ABSTRACT

Hardware Acceleration for Unstructured Big Data and Natural Language Processing

by

Prateek Tandon

Chair: Thomas F. Wenisch

The confluence of the rapid growth in electronic data in recent years, and the renewed interest in domain-specific hardware accelerators presents exciting technical opportunities. Traditional scale-out solutions for processing the vast amounts of text data have been shown to be energy- and cost-inefficient. In contrast, custom hardware accelerators can provide higher throughputs, lower latencies, and significant energy savings. In this thesis, we present a set of hardware accelerators for unstructured big-data processing and natural language processing.

The first accelerator, called HAWK, targets unstructured log processing and fits within the context of string search. HAWK consumes data at a fixed 32 GB/s, a scan rate an order of magnitude higher than standard in-memory databases and text processing tools. HAWK's scan engine requires no control flow or caches; hence, the hardware scan pipeline never stalls and can operate at a fixed 1 GHz frequency processing 32 input characters per clock cycle. The accelerator's design avoids the cache misses, branch prediction misses, and other aspects of CPUs that make performance unpredictable and require area-intensive hardware to mitigate. HAWK also leverages

a novel formulation of the state machines that implement the scan operation, thereby facilitating a hardware implementation that can process many characters concurrently while keeping on-chip storage requirements relatively small. In the HAWK design, 32 consecutive characters are conceptually concatenated into a single symbol, allowing a single state transition to process all 32 characters. Naïvely transforming the input alphabet in this way leads to intractible state machines—the number of outgoing edges from each state is too large to enable fixed-latency transitions. We leverage the concept of bit-split state machines, wherein the original machine is replaced with a vector of state machines that each process only a bit of input. As a result, each per-bit state requires only two outgoing edges. Another novel feature of the HAWK design is that the accelerator searches for strings and numbers in a manner that is agnostic to the location of field and record delimiters in the input log file. The mapping between matched strings/numbers and fields is done *after-the-fact* using a specialized field alignment unit. In its pareto-optimal configuration, HAWK requires 45 mm^2 in 45 nm technology and consumes 22 W during operation.

The second accelerator we propose targets similarity measurement in natural language processing, and in contrast to HAWK, fits within the realm of semantic search. This accelerator addresses one of the most data- and compute-intensive kernels that arises in many NLP applications: calculating similarity measures between millions (or even billions) of text fragments. We develop this accelerator in the context of a motivating NLP application: constructing an index for semantic search (search based on similarity of concepts rather than string matching) over massive text corpora. Constructing the search index nominally requires a distance calculation (e.g., cosine similarity) between all document pairs, and is hence quadratic in the number of documents; this distance calculation is the primary computational bottleneck of the application. As an example, over half a billion new tweets are posted to Twitter daily, implying roughly 10^{17} distance calculations per day. An optimized C implemen-

tation of this distance calculation kernel running on Xeon-class cores, would require a cluster of over 2000 servers, each with 32 cores, to compare one day's tweets within a 24-hour turnaround time. Instead, our accelerator for similarity measurement can be integrated alongside a multicore processor, connected to its last-level cache, to perform these distance calculations with extreme energy efficiency at the bandwidth limit of the cache interface. By leveraging the latency hiding concepts of multi-threading and simple scheduling mechanisms, we maximize functional unit utilization. Our accelerator provides 36×-42× speedup over optimized software running on server-class cores, while requiring 56×-58× lower energy, and only 1.3% of the area.

CHAPTER I

Introduction

Electronic text data volumes have undergone explosive growth in recent years, and this trend is expected to continue [89]. Every day, over 200 billion emails are transmitted, and over 500 million tweets are published [85]. In 2010, Facebook was generating of over 130 TB of log data per day; by 2012, this number had grown to 500 TB [100]. Most of these large data corpora are typically processed using scale-out solutions (for example, using large clusters of servers running software frameworks such as Hadoop). However, such scale-out processing solutions incur high energy and hardware costs [59].

From a hardware perspective, domain-specific accelerators have been the focus of renewed interest recently. Technology trends indicate that transistor dimensions will likely continue to scale for several technology generations. However, the anticipated end of CMOS voltage (a.k.a. Dennard) scaling has led experts to predict the advent of dark silicon; that is, that much of a chip must be powered off at any time [22, 39, 86, 104]. This forecast has renewed interest in domain specific hardware accelerators that can create value from otherwise dark portions of a chip. Several recent designs to accelerate aspects of data management have already been proposed [55, 112]. Further, recently-announced CPU chip designs include field-programmable gate array (FPGA) elements [27]; these FPGA elements are designed

to act as programmable accelerator blocks for time-consuming tasks. The confluence of the aforementioned hardware technology developments makes domain-specific hardware accelerators even more practical and promising.

1.1 HAWK: Hardware Support for Unstructured Log Processing

Many forms of electronic text are inherently high-velocity. Examples of such high-velocity data include unstructured electronic text log data, such as system logs, social media updates, web documents, blog posts, and news article [89]. These textual logs can hold useful information for time-sensitive domains, such as diagnosing distributed system failures, online ad repricing, and financial intelligence. Queries on these high-velocity text data are often *ad hoc*, *highly-selective*, and *latency-intolerant*. That is, a system designed to answer such queries may not know the workload ahead of time; the queries can usually ignore the vast majority of the overall corpus; and user interactivity requires fast query answers that reflect up-to-the-second data.

Time constraints and the varied workloads in the high-velocity data space often make index construction impractical. In such scenarios, an ad-hoc query system's performance directly depends on its ability to scan and select from the contents of memory. When performing an in-memory scan-and-select on traditional modern hardware, *memory bandwidth* is a crucial performance bottleneck as it sets an upper bound on the speed of the scan. We find that existing systems and tools do not come anywhere close to saturating available memory bandwidth. For example, a state-of-the-art in-memory database can scan at best 2 GB/s of data, far short of the 17 GB/s RAM-to-CPU DDR3 channel offered by modern architectures. Non-database textual tools, such as the *grep* and *awk* commands, perform even worse, sometimes by more than an order of magnitude. The gap arises because all of these tools must execute,

on average, many instructions for each character of input they scan. Thus instruction execution throughput, rather than memory bandwidth, becomes the performance limiter. As memory bandwidths continue to improve (e.g., with the proliferation of DDR4), the gap between instruction execute rate and available memory bandwidth is likely to grow further.

In this context, we propose a combination of a custom hardware accelerator, which we refer to as *HAWK*, and an accompanying software query compiler for performing selections and scans over in-memory text data. HAWK is designed with an objective of processing in-memory text at a fixed rate of 32 GB/s, faster than the data rate of a single-channel DDR3-2133 system. HAWK's scan engine requires no control flow or caches; hence, the hardware scan pipeline never stalls and can operate at a fixed 1 GHz frequency processing 32 input characters per clock cycle. The accelerator's design avoids the cache misses, branch prediction misses, and other aspects of CPUs that make performance unpredictable and require area-intensive hardware to mitigate.

HAWK also leverages a novel formulation of the state machines that implement the scan operation, thereby facilitating a hardware implementation that can process many characters concurrently while keeping on-chip storage requirements relatively small. In the HAWK design, 32 consecutive characters are conceptually concatenated into a single symbol, allowing a single state transition to process all 32 characters. Naïvely transforming the input alphabet in this way leads to intractible state machines—the number of outgoing edges from each state is too large to enable fixed-latency transitions. So, we leverage the concept of bit-split state machines [101], wherein the original machine is replaced with a vector of state machines that each processes only a bit of input. As a result, each per-bit state requires only two outgoing transitions. Matches are reported when the vector of machines have all recognized the same search string. Another novel feature of the HAWK design is that the accelerator searches for strings and numbers in a manner that is agnostic to the location of field and record

delimiters in the input log file. The mapping between matched strings/numbers and fields is done *after-the-fact* using a specialized field alignment unit.

Through hardware simulation and real-world testing of software solutions, we demonstrate that our proposed system can saturate modern memory bandwidths and obtain scan rates that are an order of magnitude higher than standard in-memory databases and tools. Indeed, our scan operations are fast enough that they are competitive with software solutions that utilize pre-computed indices for many of our target queries. HAWK, in its pareto-optimal configuration, requires 45 mm^2 in 45 nm technology and consumes 22 W during operation. Both the power and area figures of HAWK are well within the corresponding envelopes of a server-class chip.

1.2 An Accelerator for Similarity Measurement in Natural Language Processing

The second target domain for accelerators in this thesis is natural language processing (NLP). We propose and evaluate a hardware accelerator that addresses one of the most data- and compute-intensive kernels that arises in many NLP applications: calculating similarity measures between millions (or even billions) of text fragments [18, 26, 31, 91, 96]. We develop this accelerator in the context of a motivating NLP application: constructing an index for semantic search (search based on similarity of concepts rather than string matching) over massive text corpora such as Twitter feeds, Wikipedia articles, logs, text messages, or medical records. The objective of this application is to construct an index where queries for one search term (e.g., "Ted Cruz") can locate related content in documents that share no words in common (e.g., documents containing "GOP candidate"). The intuition underlying semantic search is that the relationship among documents can be discovered automatically by clustering on words appearing in many documents (e.g., "GOP" frequently appearing

in documents also containing "Cruz"). Such a search index can be constructed by generating a graph where nodes represent documents (such as tweets) and edges represent their pairwise similarity according to some distance measure (e.g., the number of words in common) [38, 82]. A semantic search can then be performed by using exact text matching to locate a node of interest in this graph, and, thereafter, using breadth-first search, random walks, or clustering to navigate to related nodes.

Constructing the search graph nominally requires a distance calculation (e.g., cosine similarity) between all document pairs, and is hence quadratic in the number of documents. This distance calculation is the primary computational bottleneck of the application. As an example, over half a billion new tweets are posted to Twitter daily [105], implying roughly 10^{17} distance calculations per day, and this rate continues to grow. Clever pre-filtering can reduce the required number of comparisons by an order of magnitude; nevertheless, achieving the required throughput on conventional hardware remains expensive. For example, based on our measured results of an optimized C implementation of this distance calculation kernel running on Xeon-class cores, we estimate that a cluster of over 2000 servers, each with 32 cores is required to compare one day's tweets within a 24-hour turnaround time.

Instead, we develop an accelerator for similarity measurement can be integrated alongside a multicore processor, connected to its last-level cache, to perform these distance calculations with extreme energy efficiency at the bandwidth limit of the cache interface. The accelerator performs only the distance calculation kernel; other algorithm steps that grow linearly in the number of documents and are easily completed in software. Our design leverages the latency hiding concepts of multi-threading and simple scheduling mechanisms to maximize functional unit utilization.

We evaluate the design through a combination of cycle-accurate simulation in the gem5 framework (performance analysis) and RTL-level synthesis (energy analysis). For Twitter and Wikipedia datasets, our accelerator enables 36×-42× speedup over a

baseline software implementation of the distance measurement kernel on a Xeon-like core, while requiring 56×-58× lower energy.

The remainder of this document is structured as follows: Chapter II describes HAWK, the accelerator that supports ad-hoc queries on large datasets. Chapter III describes a software implementation of HAWK's functionality. Chapter IV describes the design of the accelerator that supports calculating similarity measures. The appendices describe work that, while not directly related, touches upon topics that are relevant to the area of big data. Appendix A describes a study on improving MapReduce performance by reducing the number of remote accesses. Appendix B describes an investigation into the profitability of removing intermediate cache levels given the advent of 3D-stacked memory; this study is performed in the context of data-centric and scientific workloads.

CHAPTER II

HAWK: Hardware Support for Unstructured Log Processing

Rapidly processing high-velocity text data is critical for many technical and business applications. Traditional software-based tools for processing these large text corpora use memory bandwidth inefficiently due to software overheads and thus fall far short of peak scan rates possible on modern memory systems. In this chapter, we present HAWK, a custom hardware accelerator for ad hoc queries against large in-memory logs. HAWK comprises a stall-free hardware pipeline that scans input data at a constant rate, examining multiple input characters in parallel in a single accelerator clock cycle. We describe a 1 GHz 32-character-wide HAWK design targeting ASIC implementation in 45 nm manufacturing technology that processes data at 32 GB/s—faster than most extant memory systems—which requires 42% of the area and 35% of the power budget of an Intel Xeon-class processor in the same technology. This ASIC design outperforms software solutions by as much as two orders of magnitude. We further demonstrate a scaled-down FPGA proof-of-concept that operates at 50 MHz with 4-wide parallelism (200 MB/s). Even at this reduced rate, the prototype outperforms software grep by 6.5× for large multi-pattern scans.

2.1 Introduction

High-velocity electronic text log data—such as system logs, social media updates, web documents, blog posts, and news articles—have undergone explosive growth in recent years [89]. These textual logs can hold useful information for time-sensitive domains, such as diagnosing distributed system failures, online ad pricing, and financial intelligence. For example, a system administrator might want to find all HTTP log entries that mention a certain URL. A financial intelligence application might search for spikes in the number of Tweets that contain the phrase *can't find a job*. Queries on this high-velocity text data are often *ad hoc*, *highly-selective*, and *latency-intolerant*. That is, the workload is not known ahead of time; the queries often ignore the vast majority of the corpus; and query answers must be generated quickly and reflect up-to-the-second data.

Memory-resident databases have recently become a popular architectural solution, not simply for transactional [68, 98] workloads but for analytical ones [72, 95, 97, 115] as well. Storing data in RAM admits extremely fast random seeks and fast scan behavior, potentially making such databases good matches for *ad hoc* and *latency-intolerant* log query systems. Although RAM storage costs are higher than other technologies, they are falling over time and are likely already acceptable for many datasets (*e.g.*, Twitter's own search engine now stores recent data in RAM [28]).

Because time constraints and varied workloads make index construction impractical, an ad hoc log query system's performance will depend on its ability to scan and select from the contents of memory. When performing an in-memory scan-and-select on traditional modern hardware, *memory bandwidth*—the rate at which the architecture supports transfers from RAM to the CPU for processing—sets an upper bound on the speed of the scan.

Unfortunately, existing systems and tools do not come anywhere close to saturating available memory bandwidth. For example, for a state-of-the-art in-memory

database, we measure a peak scan rate of 2 GB/s of data, far short of the 17 GB/s RAM-to-CPU DDR3 channel offered by modern architectures. Non-database textual tools, such as *grep* and *awk*, perform even worse, sometimes by orders of magnitude. The gap arises because all of these tools must execute many instructions, on average, for each character of input they scan. Thus instruction execution throughput, rather than memory bandwidth, becomes the performance limiter. Nor is it clear that growth in CPU cores can solve the problem, as memory bandwidths also continue to improve (*e.g.*, with the proliferation of DDR4).

System Goal — There are many questions to answer when building an in-memory analytical database, but in this chapter we narrowly focus on just one: *can we saturate memory bandwidth when processing text log queries?*[1] If so, the resulting system could be used directly in *grep*- and *awk*-style tools, and integrated as a query processing component in memory-resident relational systems.

We are interested in designs that include both software and hardware elements. Although hardware accelerators have had a mixed history in data management systems, there is reason to be newly optimistic about their future. The anticipated end of CMOS voltage scaling (*a.k.a.* Dennard scaling) has led experts to predict the advent of chips with "dark silicon"; that is, chips that are designed to have a substantial portion powered off at any given time [22, 39, 86, 104]. This forecast has renewed interest in domain specific hardware accelerators that can create value from otherwise dark portions of a chip—accelerators powered only when especially needed. Researchers have recently proposed several hardware designs tailored for data management [55, 112]. Further, recently-announced chip designs include *field programmable gate array (FPGA)* elements [27], making a domain-specific hardware accelerator — implemented in FPGAs — even more practical and promising. There

[1]Note that although we are motivated primarily by text log processing, general streaming data query processing has many of the same requirements.

has also been substantial recent interest in using FPGAs for database query processing [48, 75, 106, 109].

Technical Challenge — It is not surprising that current software systems on standard cores perform poorly. Most text processing systems use pattern matching state machines as a central abstraction, and standard cores that implement these machines in software can require tens of instructions per character of input. Further, there is a central challenge in efficiently representing state machines for large alphabets and complex queries; the resulting transition matrices are sparse, large, and randomly accessed, leading to poor hardware cache performance.

In this work, we set an objective of processing in-memory ASCII text at 32 *gigacharacters per second* (GC/s), corresponding to a 32 GB/s data rate from memory—a convenient power of two expected to be within the typical capability of near-future high-end servers incorporating several DDR3 or DDR4 memory channels. We investigate whether a custom hardware component can reach this performance level, and how much power and silicon area it takes. Achieving this processing rate with conventional multicore parallelism (e.g., by sharding the text log data into subsets, one per core) is infeasible; our measurements of a state-of-the-art in-memory database suggest that chips would require nearly 20× more cores than are currently commonplace in order to reach the target level of performance.

Our Approach — We propose a combination of a custom hardware accelerator and an accompanying software query compiler for performing selection queries over in-memory text data. When the user's query arrives, our compiler creates a pattern matching finite state automaton that encodes the query and transmits it to the custom hardware component; the hardware accelerator then executes it, recording the memory addresses of all text elements that satisfy the query. This list of results

can then be used by the larger data management software to present results to the user, or as intermediate results in a larger query plan.

We exploit two central observations to obtain fast processing while still using a reasonable hardware resource budget. First, our accelerator is designed to operate at a fixed scan rate: it always scans and selects text data at the same rate, regardless of the data or the query, streaming data sequentially from memory at 32 GB/s. We can achieve such performance predictability because the scan engine requires no control flow or caches; hence, the hardware scan pipeline never stalls and can operate at a fixed 1 GHz frequency, processing 32 input characters per clock cycle. Our approach allows us to avoid the cache misses, branch mispredictions, and other aspects of CPUs that make performance unpredictable and require area-intensive hardware to mitigate.

Second, we use a novel formulation of the automata that implement the scan operation, thereby enabling a hardware implementation that can process many characters concurrently while keeping on-chip storage requirements relatively small. We conceptually concatenate 32 consecutive characters into a single symbol, allowing a single state transition to process all 32 characters. Naïvely transforming the input alphabet in this way leads to intractable state machines—the number of outgoing edges from each state is too large to enable fixed-latency transitions. So, we leverage the concept of *bit-split pattern matching automata* [101], wherein the original automaton is replaced with a vector of automata that each process only a bit of input. As a result, each per-bit state requires only two outgoing transitions. Matches are reported when the vector of automata have all recognized the same search pattern.

Contributions and Outline — The core contributions of this chapter are as follows:

1. We describe a typical log processing query workload, describe known possible solutions (that are unsuitable), and provide some background information about conventional approaches (Sections 2.2 and 2.7).

2. We propose HAWK, a hardware accelerator design with a fixed scan-and-select processing rate. HAWK employs *automata sharding* to break the user's query across many parallel processing elements. The design is orthogonal to standard data sharding (*i.e.*, breaking the dataset into independent parts for parallel processing), and can be combined with that approach if desired (Sections 2.4 and 2.5).

3. We describe a 1 GHz 32-character-wide HAWK design targeting ASIC implementation in a 45 nm manufacturing technology. The accelerator requires 42% of the area and 35% of the power budget of a contemporary server-class processor. This design is capable of saturating near-future memory bandwidth, outperforming current software solutions by orders of magnitude. Indeed, our scan operations are fast enough that they are often competitive with software solutions that utilize pre-computed indexes.

4. We validate our ASIC design with a scaled-down FPGA prototype. The FPGA prototype is a 4-wide HAWK design and operates at 50 MHz. Even at this greatly reduced processing rate, the FPGA design outperforms *grep* by 6.5x for challenging multi-pattern scans.

We cover related work in Section 2.7 and conclude in Section 2.9.

2.2 Problem Description

We focus on the single problem of fast in-memory scans of textual and log-style data, a crucial task for a range of data management tools, including in-memory

```
www.pbs.org/nature.html; 72; 06:32:09; opera; linux; 131.24.0.7; 13,789,432; 3125
www.pbs.org/frontline.html; 41; 07:14:15; safari; osx; 187.98.32.1; 762,989,123; 7412
www.pbs.org/peg+cat.html; 156; 08:47:45; firefox; osx; 243.56.171.53; 432,404; 6780
www.pbs.org/dinosaur_train.html; 23; 11:11:11; ie; windows; 54.12.87.10; 55,764; 904
www.pbs.org/nova.html; 32; 16:56:21; safari; osx; 212.63.75,31; 314,573; 510
...
```

Figure 2.1: A sample log file.

relational databases performing in-situ data processing and log processing tools such as *Splunk* [10]. Figure 2.1 shows a brief example of such data.

We are particularly interested in settings where log data arrive quickly and must be queried rapidly. Examples of such workloads include network security analytics, debugging and performance analysis of distributed applications, online advertising clickstreams, financial trading applications, and multiplayer online games. More speculative applications could include news discovery and trend analysis from Twitter or other text sources. The query workload is a mixture of standing queries that can be precompiled, and *ad hoc* ones driven by humans or by automated responses to previous query results. The actual queries involve primarily field-level tokenization plus string equality tests.

In this section, we cover the user-facing desiderata of such a system, including the data model and query language. Then, we consider traditional software solutions for such queries and why hardware acceleration is desirable.

2.2.1 Desiderata for a Log Processing System

We now briefly describe the types of data and queries that our system aims to manage.

Data Characteristics — The text to be queried is log-style information derived from Web servers or other log output from server-style software. We imagine a single textual dataset that represents a set of records, each consisting of a number of fields. Delimiters specify the end of each record and each field; the number of fields per record is variable. We believe that string equality tests will be sufficient for the vast

majority of these queries. Because the text arrives very rapidly in response to external system activity, there is no premade indexing structure (*e.g.*, a B+ Tree) available. The logs are append-style, so records are sorted by arrival time.

The log may be the result of using a log consolidation system, such as Apache Flume [1]. Such systems will collect log data from multiple servers in a deployed system and deposit them into a single location for processing. However, consolidation does not substantially change the core query processing challenge.

Standard sharing formats such as JSON are increasingly common but still create non-trivial computational serialization and deserialization overhead when applied at large scale. As a result, they are generally only used for relatively-rare "interface-level" data communications, and are not standard for bulk logs. However, if the user does want to process JSON with our proposed hardware, doing so is possible using the filter-style deployment described in Section 2.2.2.

It some cases it might be possible to enable fast string-equality log queries by constructing an inverted index over the data. However, for high data rates, this approach is likely to be insufficient: inverted index construction on Wikipedia-scale datasets takes hours. Incremental "near-real-time" indexing is possible, but is generally only applied to modest text updates.

Query Language — The data processing system must answer selection and projection queries over the aforementioned data. Fields are simply referred to by their field number. For example, for the data in Figure 2.1, we might want to ask:

```
SELECT $3, $5 WHERE $7 = 200 AND
  ($5="132.199.200.201" OR $5="100.202.444.1")
```

The system uses default field and record delimiters, but the user can specify them explicitly if needed:

```
SELECT $3, $5 WHERE $7 = 200 AND
```

```
($5="132.199.200.201" OR $5="100.202.444.1")
FIELD_DELIM = '/'
RECORD_DELIM = '\n'
```

The system must support boolean predicates on numeric fields (`=`, `<>`, `>`, `<`, `<=`, `=<`) and textual ones (equality and `LIKE`).

Query Workload — We assume queries that have four salient characteristics. First, they are *ad hoc*, possibly written in response to ongoing shifts in the incoming log data, such as in financial trading, social media intelligence, or network log analysis. This changing workload means that even if we were to have the time to create an index in advance, it would not be clear as to which indexes to construct.

Second, queries are *time-sensitive*: the user expects an answer as soon as possible, perhaps so the user can exploit the quick-moving logged phenomenon that caused her to write the query in the first place. This need for fast answers further undermines the case for an index: the user cannot wait for the upfront indexing cost.

Third, queries are *highly selective*: the vast majority of the log data will be irrelevant to the user. The user is primarily interested in a small number of very relevant rows in the log. As a result, although our system offers projections, it is *not* designed primarily for the large aggregations that motivate columnar storage systems.

Fourth, queries may *entail many equality tests*: we believe that when querying logs, it will be especially useful for query authors to test a field against a large number of constants. For example, imagine the user wants to see all log entries from a list of suspicious users:

```
SELECT $1, $2, $3 WHERE $3 = 'user1' OR $3 = 'user2'
       OR $3 = 'user3' OR ...
```

or imagine a website administrator wants to examine latency statistics from a handful of "problem URLs":

```
SELECT $1, $4, WHERE $1 = '/foo.html'
       OR $1 = '/bar.html' OR ...
```

If we assume the list of string constants—the set of usernames or the set of problematic URLs—is derived from a relation, these queries can be thought of as implementing a semijoin between a column of data in the log and a notional relation from elsewhere [35]. This use case is so common that we have explicit support for it in both the query language and the execution runtime. For example, the user can thus more compactly write:

```
SELECT $1, $4 WHERE $1 = {"problemurls.txt"}
```

for a query logically equivalent to the one above.

When integrating HAWK with the software stack and interacting with the user, we envision at least two possible scenarios. The first usage scenario involves close integration with a data management tool. When the database engine encounters an ad hoc query, the query is handed off to the accelerator for processing, potentially freeing up the server cores for other processing tasks. Once the accelerator has completed execution, it returns pointers in memory to the concrete results. The database then retakes control and examines the results either for further processing (such as aggregation) or to return to the user. This scenario can be generalized to include non-database text processing software, such as *grep* and *awk*.

The second usage scenario involves a stand-alone deployment, in which a user submits queries directly to the accelerator (via a minimal systems software interface) and the accelerator returns responses directly to the user. In either case, the RDBMS software and the user cannot interact entirely directly with the hardware. Rather, they use the hardware-specific query compiler we describe in Section 2.5.1.

2.2.2 Regular Expression Parsing

Processing regular expressions is not a core goal: they are not required for many log processing tasks, and our hardware-based approach does not lend itself to the arbitrarily-deep stacks that regexp repetitions enable. The hardware natively supports exact string comparisons including an arbitrary number of single-character wildcards. However, it is possible to build a complete regular expression processing system on top of our proposed mechanism. We first implement using HAWK all of the equality testing driven components of the regular expression. Any strings that pass this "prefilter" are then examined with a more traditional software stack for full regular expression processing.

2.2.3 Conventional Solutions

Today, scan operations like those we consider are typically processed entirely in software. Simple text processing is often performed with command-line tools like *grep* and *awk*, while more complex scan predicates are more efficiently processed in column-store relational databases, such as *MonetDB* [68] and *Vertica* [56]. Keyword search is typically performed using specialized tools with pre-computed indexes, such as *Lucene* [69] or the *Yahoo S4* framework [76].

However, software-implemented scans fall well short of the theoretical peak memory bandwidth available on modern hardware because scan algorithms must execute numerous instructions (typically tens, and sometimes hundreds) per byte scanned. Furthermore, conventional text scanning algorithms require large state transition table data structures that cause many cache misses. For our design goal of 32 GC/s, and a target accelerator clock frequency of 1 GHz, our system must process 32 characters each clock cycle. Given a conventional core's typical processing rates of at most a few instructions per cycle, and many stalls due to cache misses, we would potentially require hundreds of cores to reach our desired level of performance.

Indexes are clearly effective, but are also time-consuming and burdensome to compute. Traditional index generation is prohibitive for time-sensitive, ad hoc queries. Moreover, indexes rapidly become stale for high-velocity sources and are expensive to update.

Hardware-based solutions have been marketed for related applications, for example, IBM Netezza's data analytics appliances, which make use of FPGAs alongside traditional compute cores to speed up data analytics [48]. Our accelerator design could be deployed on such an integrated FPGA system. Some data management systems have turned to graphics processing units (GPUs) to accelerate scans. However, prior work has shown that GPUs are ill-suited for string matching problems [117], as these algorithms do not map well to the *single instruction multiple thread (SIMT)* parallelism offered by GPUs. Rather than rely on SIMT parallelism, our accelerator, instead, is designed to efficiently implement the finite state automata that underlie text scans; in particular, our accelerator incurs no stalls and avoids cache misses.

In short, existing software and hardware solutions are unlikely to reach our goal of fully saturating memory bandwidths during scan—the most promising extant solution is perhaps the FPGA-driven technique. Therefore, the main topic of this chapter is how we can use dedicated hardware to support the aforementioned query language at our target processing rate.

2.3 Background

We briefly describe the classical algorithm for scanning text corpora, on which HAWK is based. The **Aho-Corasick algorithm** [15] is a widely used approach for scanning a text corpus for multiple search terms or *patterns* (denoted by the set S). Its asymptotic running time is linear in the sum of the searched text and pattern lengths. The algorithm encodes all the search patterns in a finite automaton that consumes the input text one character at a time.

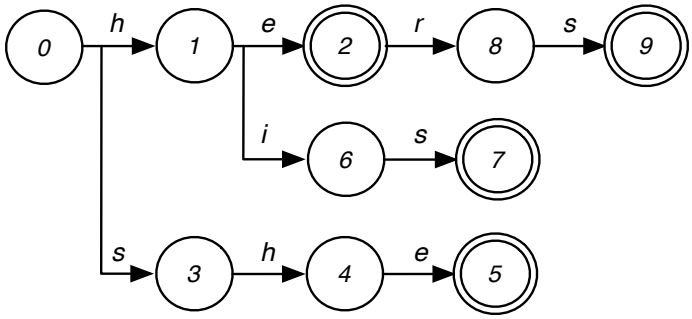


Figure 2.2: An Aho-Corasick pattern matching automaton. Search patterns are *he*, *she*, *his*, and *hers*. States *2*, *5*, *7*, and *9* are accepting states.

The Aho-Corasick automaton M is a 5-tuple $(Q, \alpha, \delta, q_0, A)$ comprising:

1. A finite set of states Q: Each state q in the automaton represents the longest prefix of patterns that match the recently consumed input characters.

2. A finite alphabet α.

3. A transition function ($\delta : Q \times \alpha \rightarrow Q$): The automaton's transition matrix comprises two sets of edges, which, together, are closed over α. The *goto function* $g(q, \alpha_i)$ encodes transition edges from state q for input characters α_i, thereby extending the length of the matching prefix. These edges form a trie (prefix tree) of all patterns accepted by the automaton. The *failure function* $f(q, \alpha_i)$ encodes transition edges for input characters that do not extend a match.

4. A start state $q_0 \in Q$, or the *root node*.

5. A set of accepting states A: A state is accepting if it consumes the last character of a pattern. An *output function output(q)* associates matching patterns with every state q. Note that an accepting state may emit multiple matches if several patterns share a common suffix.

Figure 2.2 shows an example of an Aho-Corasick trie for the patterns 'he', 'she', 'his' and 'hers' (failure edges are not shown for simplicity).

Two challenges arise when seeking to use classical Aho-Corasick automata to meet our performance objective: (1) achieving deterministic lookup time, and (2) consuming input fast enough. To aid in our description of these challenges, we leverage the notation in Table 2.1.

Parameter	*Symbol*
Alphabet	α
Set of search patterns	S
Set of states in pattern matching automaton	Q
Characters evaluated per cycle (accelerator width)	W

Table 2.1: Notation associated with HAWK design.

Deterministic lookup time — A key challenge in implementing Aho-Corasick automata lies in the representation of the state transition functions, as various representations trade off space for time.

The transition functions can be compactly represented using various tree data structures, resulting in lookup time logarithmic in the number of edges that do not point to the root node (which do not need to be explicitly represented). Alternatively, the entire transition matrix can be encoded in a hash table, achieving amortized constant lookup time with a roughly constant space overhead relative to the most compact tree.

However, recall that our objective is to process input characters at a constant rate, without any possibility of stalls in the hardware pipeline. We require deterministic time per state transition to allow multiple automata to operate in lockstep on the same input stream. (As will become clear later, operating multiple automata in lockstep on the same input is central to our design). Hence, neither logarithmic nor amortized constant transition time are sufficient.

Deterministic transition time is easily achieved if the transition function for each state is fully enumerated as a lookup table, provided the resulting lookup table is small enough to be accessed with constant latency (e.g., by loading it into an on-

chip scratchpad memory). However, this representation results in an explosion in the space requirement for the machine: the required memory grows with $O(|\alpha| \cdot |Q| \cdot log(|Q|))$. This storage requirement rapidly outstrips what is feasible in dedicated on-chip storage. Storing transition tables in cacheable memory, as in a software implementation, again leads to non-deterministic access time.

Consuming multiple characters — A second challenge arises in consuming input characters fast enough to match our design target of 32 GC/s. If only one character is processed per state transition, then the automaton must process state transitions at 32 GHz. However, there is no feasible memory structure that can be randomly accessed to determine the next state at this rate.

Instead, the automaton must consume multiple characters in a single transition. The automaton can be reformulated to consume the input W characters at a time, resulting in an input alphabet size of $|\alpha|^W$. However, this larger alphabet size leads to intractable hardware—storage requirements grow due to an increase in the number of outgoing transitions per state on the order of $O(|\alpha|^W \cdot \log_2 |Q|)$. Moreover, the automaton must still accept patterns that are arbitrarily aligned with respect to the window of W bytes consumed in each transition. Accounting for these alignments leads to $|Q| = O(|S| \cdot W)$ states. Hence, storage scales exponentially with W as $O(|S| \cdot W \cdot |\alpha|^W \cdot \log_2(|S| \cdot W))$.

HAWK uses a representation of Aho-Corasick automata that addresses the aforementioned challenges. In the next section, we discuss the principle of HAWK's operation, and detail the corresponding hardware design.

2.4 HAWK in Principle

We now describe our proposed system for processing text log queries at rates that meet or exceed memory bandwidth. We first describe the central ideas that underlie

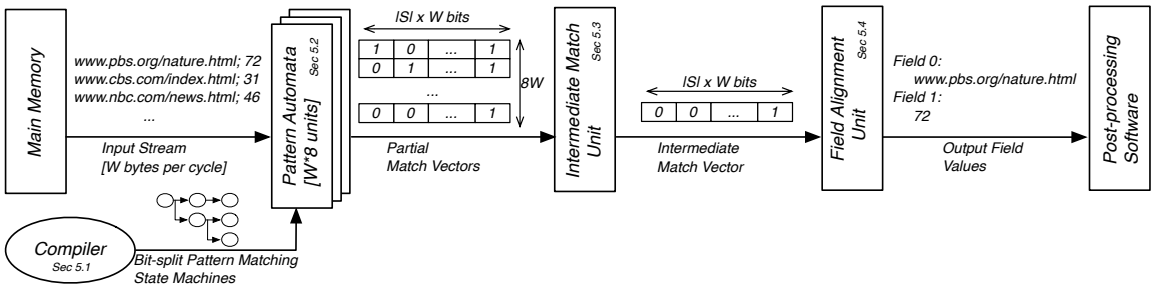


Figure 2.3: Block diagram of the accelerator architecture.

the HAWK architecture. We then step through the architecture at a high-level before describing its core components: the query compiler, the pattern automaton units, the intermediate match unit, and the field alignment unit.

2.4.1 Preliminaries

Recall that we propose a *fixed scan rate* system, meaning that the amount of input processed is the same for each clock cycle: HAWK has no pipeline stalls or other variable-time operations. Since semiconductor manufacturing technology will limit our clock frequency (we target a 1 GHz clock), the only way to obtain arbitrary scanning capacity with our design is to increase the number of characters that can be processed at each clock cycle.

There are multiple possible deployment settings for our architecture: integrating into existing server systems as an on-chip accelerator (like integrated GPUs), or as a plug-in replacement for a CPU chip, or "programmed" into reconfigurable logic in a CPU-FPGA hybrid [27]. The most appropriate packaging depends on workload and manufacturing technology details that are outside the scope of this chapter.

An *accelerator instance* is a sub-system of on-chip components that processes a compiled query on a single text stream. It is possible to build a system comprising multiple accelerator instances to scale processing capability. We define an accelerator instance's *width* W as the number of characters processed per cycle. An accelerator instance that processes one character per cycle is called *1-wide*, and an instance that

processes 32 characters per cycle is called *32-wide*. Thus, if our design target is 32 GB/s of scanning capacity, and the clock has a 1 GHz frequency, we could deploy either a single *32-wide* accelerator instance, or 32 *1-wide* accelerator instances. When deploying HAWK, an architect must decide *how many* accelerator instances should be manufactured, and of *what width*.

A common technique in data management systems is *data sharding*, in which the target data (in this case, the log text we want to query) is split over many processing elements and processed in parallel. Our architecture allows for data sharding—in which each accelerator instance independently processes a separate shard of the log text, sharing available memory bandwidth—but it is not the primary contribution of our work. More interestingly, our architecture enables *automata sharding*, in which the user's query is split over multiple accelerator instances processing a single input text stream in lockstep. Automata sharding enables HAWK to process queries of increasing complexity (i.e., increasing numbers of distinct search patterns) despite fixed hardware resources in each accelerator instance. HAWK is designed to make automata sharding possible.

2.4.2 Key Idea

The key idea that enables HAWK to achieve wide, fixed-rate scanning is our reformulation of the classic Aho-Corasick automaton to process W characters per step with tractable storage. As previously explained, simply increasing the input alphabet to $|\alpha|^W$ rapidly leads to intractable automata. Instead, we extend the concept of *bit-split pattern matching automata* [101] to reduce total storage requirements and partition large automata across multiple, small hardware units. Tan and Sherwood propose splitting a byte-based ($|\alpha| = 2^8 = 256$) Aho-Corasick automaton into a vector of eight automata that each process a single bit of the input character. Each state in the original automaton thus corresponds to a vector of states in the bit-split

automata. Similarly, each bit-split state maps to a set of patterns accepted in that state. When all eight automata accept the same pattern, a match is emitted.

Bit-split automata conserve storage in three ways. First, the number of transitions per state is drastically reduced to 2, making it trivial to store the transition matrix in a lookup table. Second, reduced fan-out from each state and skew in the input alphabet (ASCII text has little variation in high-order bit positions) results in increased prefix overlap. Third, the transition function of each automaton is distinct. Hence, the automata can be partitioned in separate storage and state IDs can be reused across automata, reducing the number of bits required to distinguish states.

Our contribution is to extend the bit-split automata to process W characters per step. Instead of the eight automata that would be used in the bit-split setting (one automaton per bit in a byte), our formulation requires $W \times 8$ automata to process W characters per step. Increasing W introduces the new challenge of addressing the alignment of patterns with respect to the W-character window scanned at each step; we cover this issue in detail in later sections.

Extending the bit-split approach to $W > 1$ results in exponential storage savings relative to widening conventional byte-based automata. The number of states in a single-bit machine is bounded in the length of the longest search term L_{max}. Since the automaton is a binary tree, the total number of nodes cannot exceed $2^{L_{max}+1} - 1$. The key observation we make is that the length of the longest search pattern is divided by W, so each bit-split automaton sees a pattern no longer than $\frac{L_{max}}{W} + P$, with P being at most two characters added for alignment of the search term in the W-character window. We find $|Q|$ for a single bit machine scales as $O(2^{\lceil \frac{L_{max}}{W} + P + 1 \rceil}) = O(1)$ in W. The storage in the bit-split automata grows as $O(|S| \cdot W)$ to overcome the aforementioned alignment issue (reasons for this storage increase will become clear in subsequent sections). With $W \times 8$ bit-split machines, the total storage scales as

$O(8 \cdot |S| \cdot W^2)$, thereby effecting exponential storage savings compared to the byte-based automaton.

2.4.3 Design Overview

We now describe our proposed system in detail.

Figure 2.3 shows a high-level block diagram of our accelerator design. At query time, the system compiles the user's query and sends the compiled query description to each accelerator instance. Each instance then scans the in-memory text log as a stream, constantly outputting matches that should be sent to higher-level software components for further processing (say, to display on the screen or to add to an aggregate computation).

The major components of our design are:

- A *compiler* that transforms the user's query into a form the hardware expects for query processing—a set of *bit-split pattern matching automata*. These automata reflect the predicates in the user's query.

- *Pattern automaton* hardware units that maintain and advance the bit-split automata. At each cycle, each pattern automaton unit consumes a single bit of in-memory text input. Because each automaton consumes only one bit at a time, it cannot tell by itself whether a pattern has matched. After consuming a bit, each automaton emits a *partial match vector (PMV)* representing the set of patterns that *might* have matched, based on the bit and the automaton's current state. For an accelerator instance of width W, there are $W \times 8$ pattern automaton units. For a query of $|S|$ patterns, the partial match vector requires $|S| \times W$ bits.

- The *intermediate match* hardware unit consumes PMVs from the pattern automata processing each bit position to determine their intersection. At each

clock cycle, the intermediate match unit consumes *W×8* PMVs, performing a logical AND operation over the bit-vectors to produce a single *intermediate match vector (IMV)* output. The *IMV* is the same length as the PMVs: $|S| \times W$.

- Finally, the *field alignment* unit determines the field within which each match indicated by the IMV is located. Pattern matching in all of the preceding steps takes place without regard to delimiter locations, and therefore, of fields and records in the input log file. This *after-the-fact mapping of match locations to fields*, which is a novel feature of our design, allows us to avoid testing on field identity during pattern matching, and thereby avoids the conditionals and branch behavior that would undermine our fixed-rate scan design. If the field alignment unit finds that the IMV indicates a match for a field number that the user's query requested, then it returns the resulting *final match vector* (FMV) to the database software for post-processing. To simplify our design, we cap the number of fields allowed in any record to 32—a number sufficient for most real-world log datasets.

Note that each accelerator instance supports searching for 128 distinct patterns. Therefore, a device that has 32 *1-wide* accelerator instances can process up to 32×128 patterns, a device with 16 *2-wide* instances can process up to 16×128 distinct patterns, and a device with a single *32-wide* instance can process up to 1×128 distinct patterns. By varying the number of instances and their width, the designer can trade off pattern constraints, per-stream processing rate, and, as we shall see later, area and power requirements (see Section 2.6.3).

2.5 HAWK Architecture

We now describe the four elements of HAWK highlighted in Figure 2.3 in detail.

Input: Query **K** and architecture width W
Output: Bit split automata set M.

```
1: S = shard( sort( ⋃ predicates(K) ) )
2: S' = []
3: for each s ∈ S do
4:     for i = 1 to W do
5:         S'.append(pad(s, i, W))
6:
7: Automata set M = {}
8: for each s ∈ S' do
9:     for i = 0 to len(s) do
10:        for bit b ∈ s[i] do
11:            M[i MOD W].addNode(b)
12:
13: for each m ∈ M do
14:     makeDFA(m)
15:     for each q ∈ M.states do
16:         makePMV(q)
```

Figure 2.4: The multicharacter bit-split pattern matching automata compilation algorithm.

2.5.1 Compiler

HAWK must first compile the user's query into pattern-matching automata. Figure 2.5 conceptually depicts compilation for a 4-wide accelerator. Figure 2.4 provides details of the compilation algorithm. The compiler's input is a query in the form described in Section 2.2. After parsing the query, the compiler determines the set of all patterns S, which is the union of the patterns sought across all fields in the WHERE clause. S is sorted lexicographically and then sharded across accelerator instances (Line 1). Sharding S lexicographically maximizes prefix sharing within each bit-split automaton, reducing their sizes.

Next, the compiler must transform S to account for all possible alignments of each pattern within the W-character window processed each cycle. The compiler forms a new set S' wherein each pattern in S is padded on the front and back with wildcard characters to a length that is a multiple of W, forming W patterns for all possible alignments with respect to the W-character window (Lines 2-5). Figure 2.5

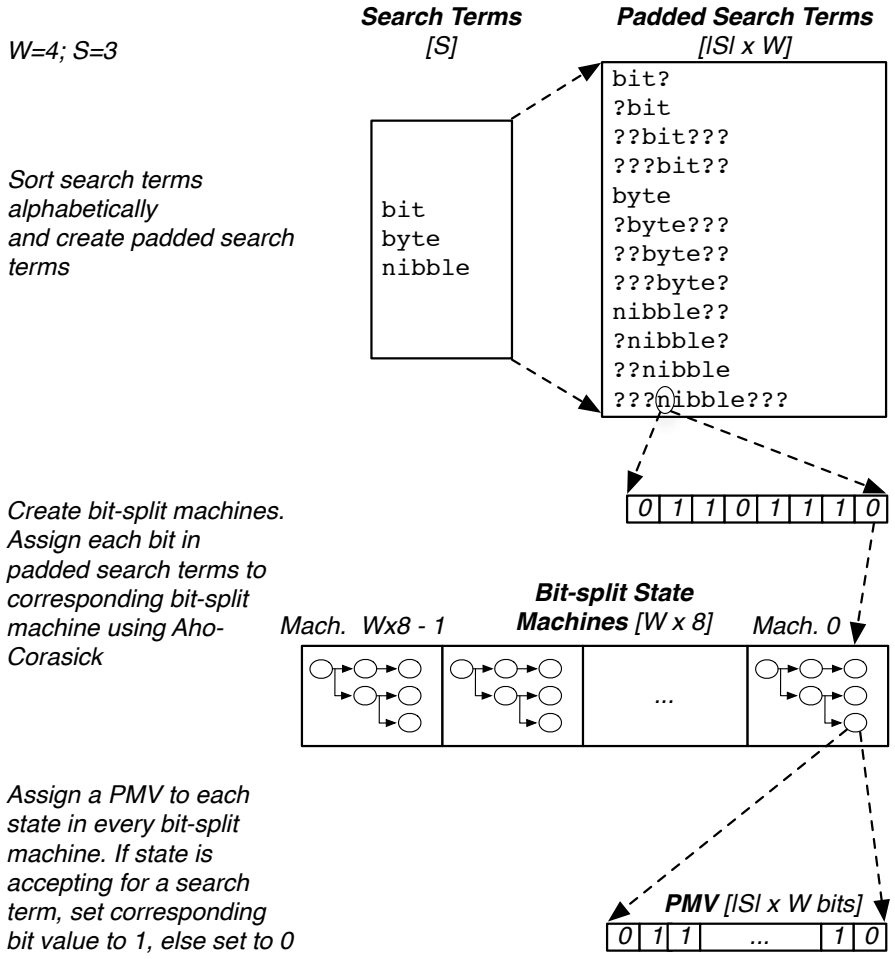


Figure 2.5: Three-step compiler operation for a 4-wide accelerator and three search terms (W=4, S=3).

shows an example of this padding for S={bit, byte, nibble} and W=4. For a machine where W=1, there is just one possible pattern alignment in the window; no padding is required.

The compiler then generates bit-split automata for the padded search patterns in S'. We generate these bit-split automata according to the algorithm proposed by Tan and Sherwood [101] (summarized in Lines 7-11). A total of $W\times8$ such automata are generated, one per input stream bit processed each cycle. Each state in these automata has only two outgoing edges, hence, the transition matrix is easy to represent in hardware. Automata are encoded as transition tables indexed by the

state number. Each entry is a 3-tuple comprising the next state for inputs bits of zero and one and the PMV for the state.

Each state's PMV represents the set of padded patterns in S' that are accepted by that automaton in that state. The compiler assigns each alignment of each pattern a distinct bit position in the PMV (Line 16). It is important to note that the hardware does not store S' directly. Rather, patterns are represented solely as bits in the PMV.

Accelerator Width (W)	*1*	*2*	*4*	*8*	*16*	*32*
Per Bit-split Machine Storage (KB)	74.8	69.6	33.5	16.5	16.4	32.8
Total Storage (MB)	0.6	1.11	1.07	1.06	2.1	8.4

Table 2.2: Provisioned storage. Per bit-split state machine, and total.

2.5.2 Pattern Automata

The *pattern automata*, shown in the first panel of Figure 2.6, each process a single bit-split automaton. Each cycle, they each consume one bit from the input stream, determine the next state, and output one PMV indicating possible matches at that bit position.

Consider the pattern automaton responsible for bit 0 of the $W \times 8$-bit input stream (from Figure 2.6). In *cycle 0*, the automaton's current state is 0. The combination of the current state and the incoming bit value indicates a lookup table entry; in this case, the incoming bit value is 0, so the lookup table indicates a next state of 1. The pattern automaton advances to this state and emits its associated PMV to the intermediate match unit for processing in the next cycle.

The transition table and PMV associated with each state are held in dedicated on-chip storage. We use dedicated storage to ensure each pattern automaton can determine its next state and output PMV in constant time. (Accesses may be pipelined over several clock cycles, but, our implementation requires only a single cycle at 1 GHz frequency.).

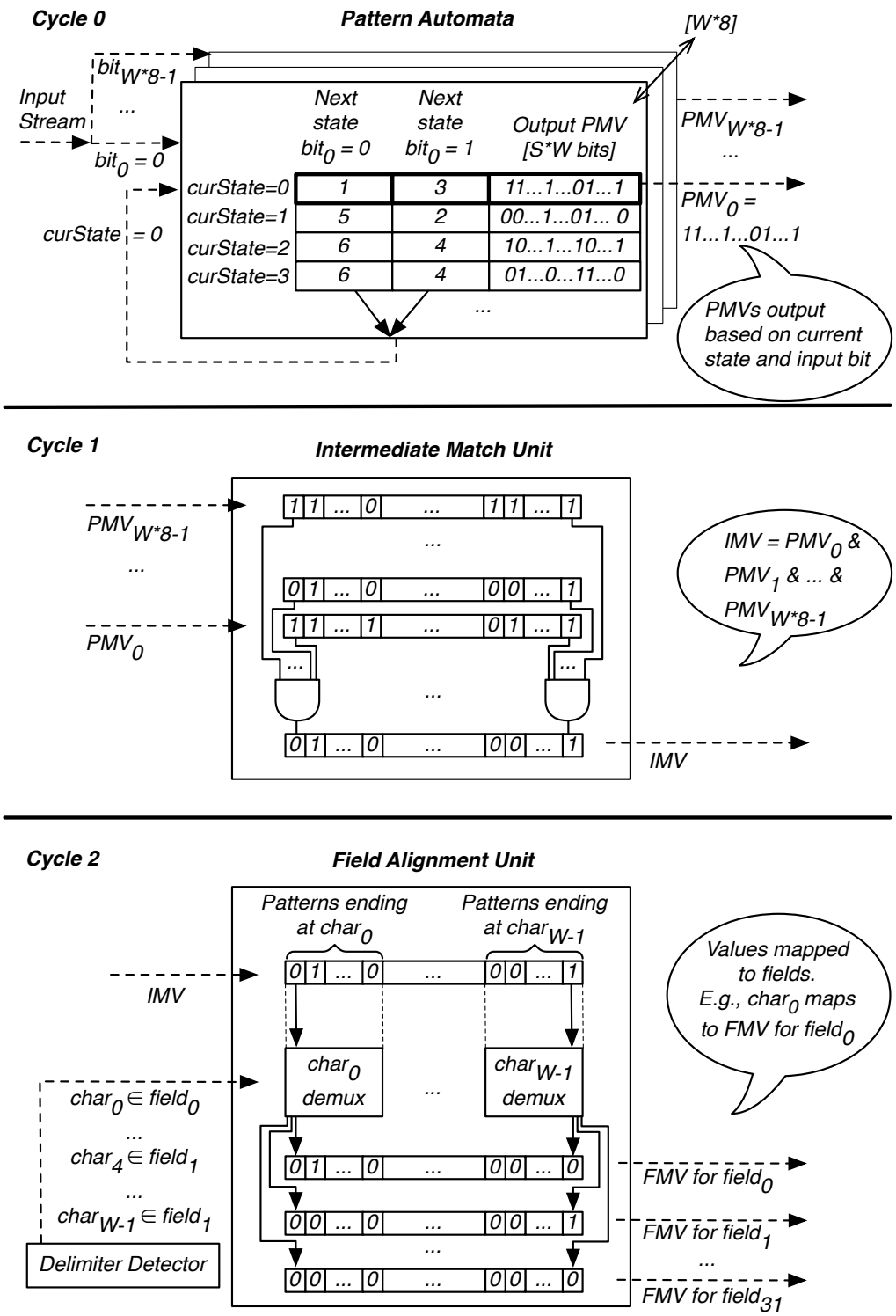


Figure 2.6: Operation of the major string matching subunits over three cycles.

We determine storage requirements for pattern automata empirically. We select 128 search terms at random from an English dictionary and observe the number of

states generated per automaton. We then round the maximum number of states required by any automaton to the next power of 2, and provision this storage for all automata. (Note that if the query workload were to systematically include longer strings, such as e-commerce URLs, then storage requirements would be correspondingly higher.)

Table 2.2 shows the *per-automaton* and *total* storage allocation for a range of accelerator widths. The storage requirement per pattern automaton is comparable to a first-level data cache of a conventional CPU. We observe a few interesting trends. First, the per-automaton-storage is minimal for W=8 and W=16. Whereas the number of patterns grows with W (a consequence of our padding scheme), the number of states in each automaton shrinks due to an effective reduction in pattern length (a consequence of processing multiple characters simultaneously). At the same time, as the number of patterns grows, the PMV width increases. The reduction in states dominates the larger PMV widths until W=16. Beyond that point, the impact of increased PMV widths starts to dominate.

Note that we conservatively provision the same storage for all automata, despite the fact that ASCII is highly skewed and results in far more prefix sharing in high-order bit positions. This decision allows our accelerator to support non-ASCII representations and ensures symmetry in the hardware, which facilitates layout.

2.5.3 Intermediate Match Unit

The *intermediate match* unit (the middle panel of Figure 2.6) calculates the intersection of the PMVs. A pattern is present at a particular location in the input stream only if it is reported in the PMVs of *all* pattern automata. The intermediate match unit is a wide and deep network of AND gates that computes the conjunction of the $W \times 8$ $|S| \times W$-bit PMVs. The result of this operation is the $|S| \times W$-bit wide intermediate match vector, which is sent to the next processing stage. As with the

pattern automata, the intermediate match unit's execution can be pipelined over an arbitrary number of clock cycles without impacting the throughput of the accelerator instance, but our 32-wide ASIC implementation requires only a single cycle. In our FPGA prototype, we integrate the pattern automata and intermediate match unit and pipeline them over 32 cycles, which simplifies delay balancing across pipeline stages.

Figure 2.6 shows that the PMVs generated by the pattern automata in *cycle 0* are visible to the intermediate match unit in *cycle 1*. The intermediate match unit performs a bitwise AND operation on all W×8 $|S| \times W$-bit PMVs and yields an IMV. In our example, the second and last bits of all PMVs are set, indicating that the padded patterns corresponding to these entries have been matched by all bit-split automata (i.e., true matches). The intermediate match unit, therefore, outputs an IMV with these bits set.

2.5.4 Field Alignment Unit

HAWK's operation so far has ignored the *locations* of matches between the log text and the user's query; it can detect a match, but cannot tell whether the match is in the correct tuple field. The *field alignment* unit (the bottom panel of Figure 2.6) reconstructs the association between pattern matches and fields. The output of the field alignment unit is an array of field match vectors (FMVs), one per field. Each vector has a bit per padded search pattern ($|S| \times W$ bits); this allows the user to determine the exact location of the matching pattern within the input stream. Bit i in FMV j indicates whether pattern i matched field j and the pattern's location within the input stream.

The field alignment unit receives two inputs. The first input is the $|S| \times W$-bit IMV output from the intermediate match unit. This vector represents the patterns identified as true matches.

The second input comes from a specialized *delimiter detector* that is preloaded with user-specified delimiter characters. (The hardware design for the delimiter detector is straightforward and is not detailed here for brevity. It is essentially a simple single-character version of pattern matching.) Each cycle, the delimiter detector emits a field ID for every character in the W-character window corresponding to the current IMV (overall, W field IDs).

Search patterns that end at a particular character location belong to the field indicated by the delimiter detector. Recall that bit positions in the PMVs (and hence, the IMV) identify the end-location of each padded search pattern within the current W-character window (see Section 2.5.1). Thus for every end-location, the field alignment unit maps corresponding IMV bits to the correct field ID, and the respective FMV. The operation of the field alignment unit is a demultiplexing operation (see Figure 2.6).

In *cycle 2*, the field alignment unit evaluates the window processed by the pattern automata in *cycle 0*, and by the intermediate match unit in *cycle 1*. In our example, the IMV's second and last bits are set, indicating that the corresponding patterns ending at character_0 and character_{W-1} have matched in *some* fields. The delimiter detector indicates that character_0 is in field_0, and character_{W-1} is in field_1. Thus, the patterns ending at character_0 are mapped to the FMV for field_0, and the patterns ending at character_{W-1} are mapped to the FMV for field_1. The mapped FMVs are subsequently sent to the post-processing software.

The field alignment unit hardware entails 32 AND operations for each bit of the IMV. Compared to the pattern matching automata, area and power overheads are minor.

Processor	Dual socket Intel E5630
	16 threads @ 2.53 GHz
Caches	256 KB L1, 1 MB L2, 12 MB L3
Memory Capacity	128 GB
Memory Type	Dual-channel DDR3-800
Max. Mem. Bandwidth	12.8 GB/s

Table 2.3: Server specifications.

2.6 Experimental Results

We have three metrics of success when evaluating HAWK. The most straightforward is query processing performance when compared to conventional solutions on a modern server. The remaining metrics describe HAWK's area and power requirements, the two hardware resource constraints that matter most to chip designers. We will show that when given hardware resources that are a fraction of those used by a Xeon chip, an ASIC HAWK can reach its goal of 32 GC/s and can comfortably beat conventional query processing times, sometimes by multiple orders of magnitude. Furthermore, we validate the HAWK design through proof-of-concept implementation in an FPGA prototype with scaled down frequency and width and demonstrate that even this drastically down-scaled design still can outperform software.

2.6.1 Experimental Setup

We compare HAWK's performance against four traditional text querying tools: *awk*, *grep*, *MonetDB* [68], and *Lucene* [69]. We run all conventional software on a Xeon-class server, with specs described in Table 2.3. We preload datasets into memory, running an initial throwaway experiment to ensure data is hot. We repeat all experiments five times and report average performance.

We design a HAWK ASIC in the Verilog hardware description language. Fabricating an actual ASIC is beyond the scope of this thesis; instead, we estimate performance, area, and power of an ASIC design using Synopsys' DesignWare IP suite [99], which includes tools that give timing, area, and power estimates. (Syn-

thesis estimates of area and power from such tools are part of conventional practice when testing novel hardware designs.)

Synthesizing an ASIC design entails choosing a target manufacturing technology for the device. We target a commercial 45 nm manufacturing technology with a nominal operating voltage of 0.72 V, and design for a clock frequency of 1 GHz. The details are less important than the observation that this technology is somewhat out of date; it is two generations behind the manufacturing technology used in the state-of-the-art Xeon chip for our conventional software performance measurements. However, the 45 nm technology is the newest ASIC process to which we have access. Since power and area scale with the manufacturing technology, we compare HAWK's power and area against a prior-generation Intel processor manufactured in the same technology.

The FPGA HAWK prototype is tested on an Altera Arria V ST platform. Due to FPGA resource constraints, we build a single 4-wide HAWK accelerator instance. We use the block RAMs available on the FPGA to store the state transition tables and PMVs of the pattern matching automata. In the aggregate, the automata use roughly half of these RAMs; there are insufficient RAMs for an 8-wide accelerator instance. Because of global wiring required to operate the distributed RAMs, we restrict clock frequency to 50 MHz. Thus, the prototype achieves a scan rate of 200 MB/sec.

Because of limited memory capacity and overheads in accessing off-chip memory on our FPGA platform, we instead generate synthetic log files directly on the FPGA. Our log generator produces a random byte-stream (via a linear feedback shift register) and periodically inserts a randomly selected search term from a lookup table. We validate that the accelerator correctly locates all matches.

The HAWK compiler is written in C. For the large memory-resident datasets we expect to process, query compilation time is negligible relative to the runtime. Since the primary focus of this chapter is on string pattern matching, our compiler software

does not currently handle numeric fields automatically; we compile numeric queries by hand. However, extending it to handle numeric predicates is straightforward.

Our evaluation considers three example use cases for HAWK that stress various aspects of its functionality. In each case, we compare to the relevant software alternatives.

2.6.1.1 Single Pattern Search

We first consider the simplest possible task: a scan through the input text for a single, fixed string. We generate a synthetic 64 GB dataset comprising 100-byte lines. We use the text log synthesis method described by Pavlo et al., for a similar experiment [78]. We formulate the synthetic data to include target strings that match a notional user query with selectivities of 10%, 1%, 0.1%, 0.01%, and 0.001%. We time the queries needed to search for each of these strings and report matching lines. We compare HAWK against a relational column-store database (*MonetDB*) and the UNIX *grep* tool. For *MonetDB*, we load the data into the database prior to query execution.

2.6.1.2 Multiple Pattern Search

Next, we consider a semijoin-like task, wherein HAWK searches for multiple patterns in a real-world dataset, namely, the Wikipedia data dump (49 GB). We select patterns at random from an English dictionary; we vary their number from one to 128. We compare against an inverted text index query processor (*Lucene*) and again *grep*. For *Lucene*, we create the inverted index prior to query execution; indexing time is not included in the performance comparison. *Lucene* and *grep* handle certain small tokenization issues differently; to ensure they yield exactly the same search results, we make some small formatting changes to the input Wikipedia text. We execute

grep with the *-Fw* option, which optimizes its execution for patterns that contain no wildcards.

2.6.1.3 Complex Predicates

Finally, we consider queries on a webserver-like log of the form *<Source IP, Destination URL, Date, Ad Revenue, User Agent, Country, Language, Search Word, Duration>*. This dataset is also based on a format proposed by Pavlo and co-authors [78]. A *complex query* has selection criteria for multiple columns in the log. It takes the following form[2]:

```
SELECT COUNT(*) FROM dataset WHERE (
(Date in specified range)
AND (Ad Revenue within range)
AND (User Agent LIKE value2 OR User Agent LIKE ...)
AND (Country LIKE value4 OR Country LIKE ...)
AND (Language LIKE value6 OR Language LIKE ...)
AND (Search Word LIKE value8 OR Search Word LIKE ...)
AND (Duration within range)).
```

We tune the various query parameters to achieve selectivities of 10%, 1%, 0.1%, 0.01%, and 0.001%. We compare against equivalent queries executed with the relational column-store (*MonetDB*) and the UNIX tool *awk*.

2.6.2 Performance

We contrast the performance of HAWK to various software tools in GC/s. By design, the HAWK ASIC always achieves a performance of 32 GC/s, and there is no sensitivity to query selectivity or the number of patterns (provided the query fits

[2]We add the COUNT element to the query so that *MonetDB* does not incur extra overhead in actually returning the concrete result tuples, but rather incurs only trivial aggregation costs.

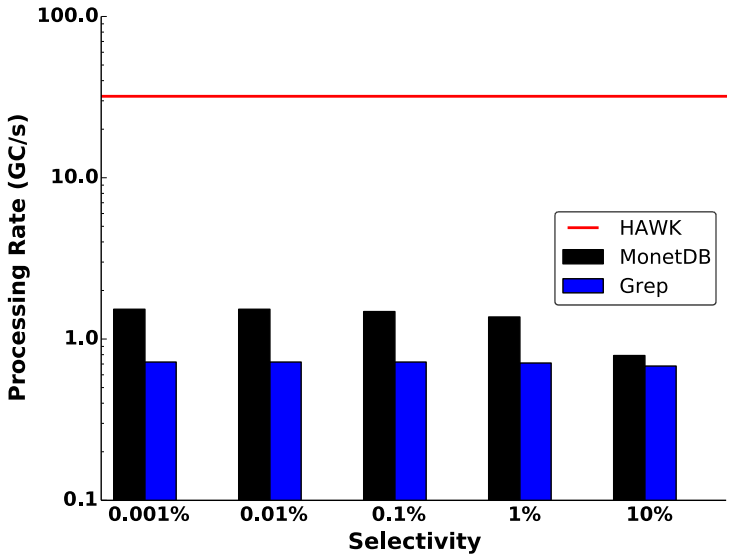


Figure 2.7: Query performance for the single pattern search task on synthetic data, across varying selectivities.

within the available automaton state and PMV capacity). In contrast, the software tools show sensitivity to both these parameters, so we vary them in our experiments.

2.6.2.1 Single Pattern Search

Figure 2.7 compares HAWK's single pattern search performance against *MonetDB* and *grep*. We find that HAWK's constant 32 GC/s performance is over an order of magnitude better than either software tool, and neither comes close to saturating memory bandwidth. *MonetDB*'s performance suffers somewhat when selectivity is high (above 1%), but neither *grep* nor *MonetDB* exhibit much sensitivity at lower selectivities.

2.6.2.2 Multiple Pattern Search

Figure 2.8 compares HAWK against *Lucene* and *grep* when searching for multiple randomly-chosen words in the Wikipedia dataset. For *Lucene*, we explore query formulations that search for multiple patterns in a single query or execute separate queries in parallel and report the best result.

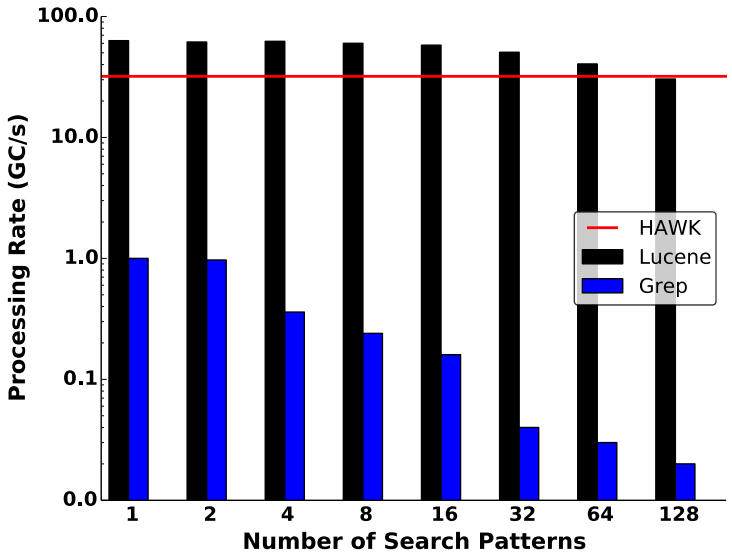


Figure 2.8: Query performance on real-world text data, for varying numbers of search patterns.

Grep's performance is poor: its already poor performance for single-pattern search (1 GC/s) drops precipitously as the number of patterns increases, to as little as 20 megacharacters/s in the 128-word case. Unsurprisingly, because it uses an index and does not actually scan the input text, *Lucene* provides the highest performance. We report its performance by dividing query execution time by the size of the data set to obtain an equivalent GC/s scan rate. Note that this equivalent scan rate exceeds available memory bandwidth in many cases (i.e., no scan-based approach can reach this performance).

Remarkably, however, our results show that, when the number of patterns is large, a HAWK ASIC is competitive with Lucene *even though HAWK does not have access to a precomputed inverted index.* In the 128-pattern case, *Lucene's* performance of 30.4 GC/s falls short of the 32 GC/s performance of HAWK. At best, *Lucene* outperforms HAWK by a factor of two for this data set size (its advantage may grow for larger data sets, since HAWK's runtime is linear in the dataset size). Of course, these measurements do not include the 30 minutes of pre-query processing time that *Lucene* requires to build the index. (As a point of comparison, our automata compile

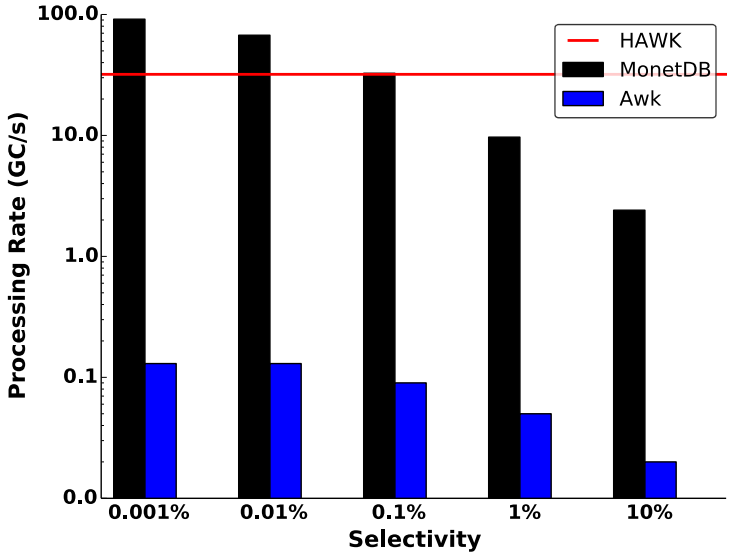
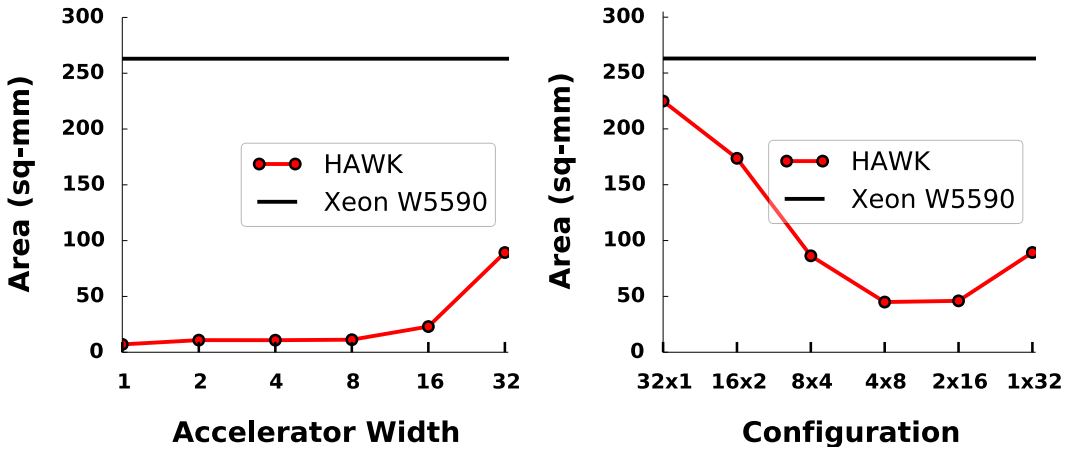


Figure 2.9: Query performance for complex predicates task, across varying selectivities.

times are on the order of seconds for all tested scenarios. We used cached query plans for all *MonetDB* queries.) As a result, even though *Lucene*'s query processing times are faster when the set of patterns is small, HAWK is a better fit in our target ad hoc scenario, in which the text corpus is changing rapidly enough to make indexing impractical.

2.6.2.3 Complex Predicates

Figure 2.9 compares HAWK, *MonetDB*, and *awk* on the complex queries described in Section 2.6.1.3. *MonetDB* performance spans a 45× range as selectivity changes from 10% to 0.001%. When selectivity is low, *MonetDB* can order the evaluation of the query predicates to rapidly rule out most tuples, avoiding the need to access most data in the database. For 0.001% selectivity, it outperforms HAWK by 3×. However, for queries that admit more tuples in the answer, where *MonetDB* must more frequently examine large text fields, HAWK provides superior performance, with more than 10× advantage at 10% selectivity. The performance of *awk* is not competitive.



(a) Single-Unit (b) Multi-Unit, 32GC/s

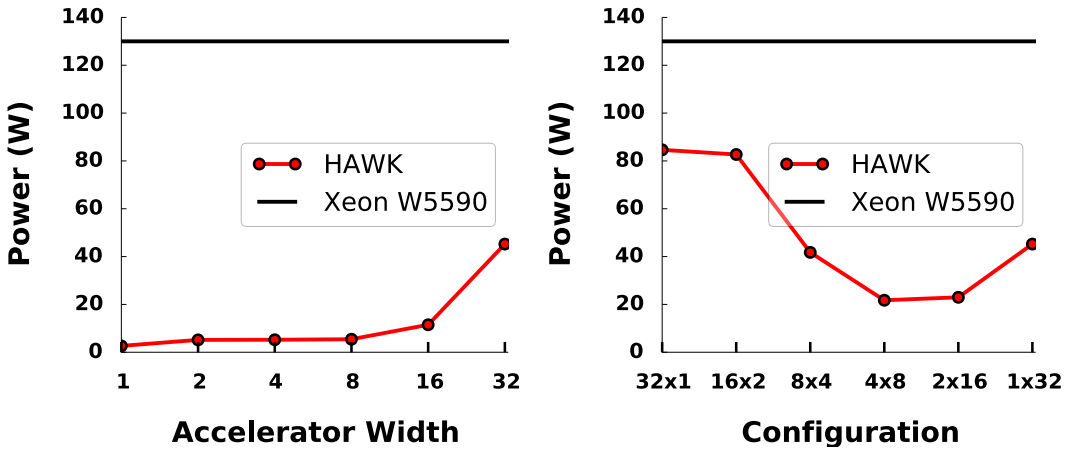
Figure 2.10: Area requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip).

2.6.3 ASIC Area and Power

We report a breakdown of an ASIC HAWK instance's per-sub-component area and power estimates for two extreme design points, *1-wide* and *32-wide*, in Table 2.4. For both designs, the pattern automata account for the vast majority of area and power consumption. Pattern automata area and power are dominated by the large storage structures required for the state transition matrix and PMVs. We can see here the impact that state machine size has on the implementation. Even with the drastic savings afforded by the bit-split technique, the automata storage requirements are still large; without the technique, they would render the accelerator impractical.

	1-wide		*32-wide*	
Unit	*Area* (mm^2)	*Power* (mW)	*Area* (mm^2)	*Power* (mW)
Pattern Automata	5.7	2602	86	44,563
Intermediate Match Unit	< 0.1	< 1	< 1	35
Field Alignment Unit	< 1	14	1	448
Delimiter Detector	1.1	< 1	< 1	< 1
Numeric Units	< 0.1	1	< 1	39
Other Control Logic	0.2	26	1	146
Total	7.1	2644	89	45,231

Table 2.4: Component area and power needs for 1-wide and 32-wide configurations.



(a) Single Unit (b) Multi-Unit 32GC/s

Figure 2.11: Power requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip).

Figures 2.10 and 2.11 compare the area and power requirements of ASIC HAWK to an Intel Xeon W5590 chip [7]. That chip uses the same generation of 45 nm manufacturing technology as our synthesized design. We find that a 1-wide HAWK instance requires only 3% of the area and 2% of the power of the Xeon chip. A 32-wide HAWK requires 42% of the area and 35% of the power of the Xeon processor. Although these values are high, they would improve when using more modern manufacturing technology; a 32-wide HAWK instance might occupy roughly one-sixth the area of a modern server-class chip.

Figures 2.10 and 2.11 also reveal an interesting trend. The 8-wide (*4×8*) and 16-wide (*2×16*) HAWK configurations utilize resources more efficiently (better performance per area or watt) than other configurations. This saddle point arises due to two opposing trends. Initially, as width W increases from 1, the maximum padded pattern length (L_{max}) per bit-split automaton decreases rapidly. Since each bit-split automaton is a binary tree, lower L_{max} yields a shallower tree (*i.e.*, fewer states) with more prefix sharing across patterns. Overall, the reduced number of states translates into reduced storage costs.

However, as W continues to grow, L_{max} saturates at a minimum while the set of padded patterns, S', grows proportionally to $|S| \times W$. Each pattern requires a distinct bit in the PMV, which increases the storage cost per state. Above $W = 16$, the increased area and power requirements of the wide match vectors outweigh the savings from reduced L_{max}, and total resource requirements increase.

Overall, the 8-wide and 16-wide configurations strike the best balance between these opposing phenomena. It is more efficient to replace one 32-wide accelerator with four 8-wide accelerators or two 16-wide accelerators. We find that the *4×8* configuration, which exhibits the lowest area and power costs, requires approximately 0.5× area and 0.48× power compared to the 32-wide accelerator, while maintaining the same performance. Compared to the W5590, the *4×8* configuration occupies about 0.21× the area and requires 0.17× the power. From a deployment perspective, we recommend using four 8-wide accelerators (*4×8*) to obtain the best performance-efficiency trade-off.

2.6.4 FPGA Prototype

We validate the HAWK hardware design through our FPGA prototype. As previously noted, the prototype is restricted to 4-wide accelerator instance operating at a 50 MHz clock frequency, providing a fixed scan rate of 200 MB/sec. As with the ASIC design, the storage requirements of pattern automata dominate resource requirements on the FPGA.

We program the accelerator instance to search for the same 64 search terms as in the multiple pattern search task described in Section 2.6.1.2. Although it is 160× slower than our ASIC design, the FPGA prototype nevertheless remains faster than *grep* for this search task by 6.5×, as *grep* slows drastically when searching for multiple patterns. Whereas *grep* achieves nearly a 1 GB/s scan rate for a single pattern, it slows to 30 MB/s when searching for 64 search terms. (Note that this is

still faster than searching for the terms sequentially in multiple passes, but only by a small factor). With better provisioning of on-chip block RAMs, both the width and clock frequency of the FPGA prototype could be improved, increasing its advantage over scanning in software.

2.7 Related Work

There are several areas of work relevant to HAWK.

String Matching — Multiple hardware-based designs have been proposed to accomplish multicharacter Aho-Corasick processing. Chen and Wang [29] propose a multicharacter transition Aho-Corasick string matching architecture using nondeterministic finite automata (NFA). Pao and co-authors [77] propose a memory-efficient pipelined implementation of the Aho-Corasick algorithm. However, neither work aims to meet or exceed available memory bandwidth.

Some elements of our approach have been used in the past. Hua et al. [47] present a string matching algorithm that operates on variable-stride blocks instead of single bytes; their work is inspired in part by how humans read text as patterns instead of single characters. van Lunteren et al. [66] use transition rules stored using balanced routing tables; this technique provides a fast hash lookup to determine next states. Taking a different approach, Bremler-Barr and co-authors [25], encode states such that all transitions to a specific state can be represented by a single prefix that defines a set of current states. However, we are not aware of any previous work that uses our approach of combining bit-split automata with multiple-character-width processing.

Processing Logs — Processing text logs is an important workload that has dedicated commercial data tools [10] and is a common use case for distributed data platforms such as Hadoop [2] and Spark [115]. In-memory data management systems have also become quite popular [68, 95, 98, 115].

Databases and FPGAs — A large amount of research has focused on using FPGAs to improve database and text processing. Mueller et al. explore general query compilation and processing with FPGAs [75]. Teubner et al. propose *skeleton automata* for avoiding expensive FPGA compilation costs [106]. The project with goals most similar to our own is probably that of Woods et al. [109], who examine the use of FPGAs for detecting network events at gigabit speeds. Although this project also focuses on the problem of string matching, it has a lower performance target, does not have our fixed-processing rate design goal, and is technically distinct. IBM Netezza [48] is the best-known commercial project in this area.

2.8 Impact of Technology Trends on the HAWK Design

We now discuss the impact of future technology trends on the HAWK design. In general, we expect that future technology trends will drive the demand for domain-specific accelerators such as HAWK. We also expect that the learnings from the HAWK design can be applied to other domain-specific accelerators. The algorithmic and design contributions will continue to hold true in the face of newer technologies; however, the design tradeoffs may change to some extent.

First, we consider the effect of new memory technologies. The falling prices of DRAM along with higher memory densities have spurred the rise of applications that work on large in-memory datasets [3]. Recall that HAWK is also targeted towards processing in-memory logs. Technologies like 3D-stacked memories (see Appendix B) and Intel's 3D XPoint memories [6] offer the promise of significantly higher bandwidths and lower latencies. Such improvements in memory technology will provide further impetus for domain-specific accelerators like HAWK. As noted earlier, traditional software solutions demonstrate large gaps between the available and utilized memory bandwidths; with the aforementioned newer memory solutions, these gaps

are likely to grow even larger. Therefore, accelerators such as HAWK can serve a valuable role in processing in-memory data.

Next, we consider the impact of technology scaling. As shown in Table 2.4, HAWK's pattern automata units dominate the area and power requirements, with the pattern automata being dominated by memory elements. A limitation of our methodology is that in the absence of access to an industrial memory compiler, we use flip-flops to represent these memory elements. Based on anecdotal evidence, we believe that our area and power estimates would be about 8× lower if we were to use SRAMs instead of flip-flop-based memories. We expect that with the use of SRAMs, the pareto-optimal design which currently occupies an area of 45 mm^2, will, in fact, take up about 6 mm^2. Scaling down to current process nodes will offer further advantages. We estimate that in 14 nm, the area of the aforementioned configuration will be about 1-2 mm^2. We also expect that the power requirement of the pareto-optimal configuration will be approximately 1 W. Therefore, with appropriate memory elements, and access to newer process libraries, HAWK would move from being a plug-in replacement for a server chip, to an accelerator in the dark silicon domain[3]. Moving into the dark silicon domain will provide an additional advantage for HAWK—multiple instances of HAWK can be placed without much overhead, and these instances can then, in the aggregate, saturate memory bandwidth. These multiple instances of HAWK can operate by evaluating separate sets of search terms, or separate chunks of the input stream, or a combination of the two.

Finally, we consider improvements in the CPU design space that may make software solutions running on CPUs competitive with HAWK. A recent trend in CPU technology is the availability of wider SIMD units and registers. In the context of

[3]Technology trends indicate that transistor dimensions will continue to scale for several technology generations. However, the anticipated end of CMOS voltage scaling has led to predictions of the advent of "dark silicon"—i.e., much of a chip must be powered off at any time to stay within power budgets. This forecast has sparked interest in domain specific hardware accelerators that drastically improve the energy-efficiency of compute intensive tasks to create value from otherwise dark portions of a chip.

software-based pattern matching using multi-character bit-split automata, the wider SIMD units can allow for larger PMVs to be supported, and, therefore, larger numbers of search terms and/or larger accelerator widths. However, pattern automata updates will still need to be performed using traditional hardware units, and these automata updates will suffer from performance degradation due to larger automata (more cache misses and stalls). Therefore, it is unlikely that the improvements in CPU-SIMD technology will make software solutions as proposed in Chapter III competitive with HAWK. That said, GPUs may offer ample SIMT parallelism and latency-hiding to merit the implementation of multi-character bit-split pattern matching automata as described in Chapter III.

2.9 Conclusion

High-velocity text log data have undergone explosive growth in recent years. Data management systems that rely on index-driven approaches cannot apply to this workload, and conventional scan-based mechanisms do not come close to exploiting the full capacity of modern hardware architectures. We have shown that our HAWK accelerator can process data at a constant rate of 32 GB/s. We have also shown that HAWK is often better than state-of-the-art software solutions for text processing.

CHAPTER III

A Software Implementation of HAWK's Functionality

In Chapter II we explore HAWK, an accelerator for unstructured log processing. In the same chapter, we note that grep's performance for exact pattern matching degrades drastically as the number of search patterns increases. For example, grep demonstrates processing rates of almost 1 GC/s for one or two search terms, but this processing rate drops to 20 MC/s for 128 search terms. Grep's behavior can be attributed to the underlying mechanism it uses for pattern matching. Grep makes use of the Boyer-Moore/Commentz-Walter algorithms, and for a small number of search terms, is able to skip over large portions of the input stream. However, as the number of search patterns increases, grep is unable to skip over large portions of the input stream, and must do more comparisons. In this context, we leverage the techniques and learnings from the HAWK design to explore a software implementation of HAWK that overcomes grep's deficiencies. We compare our software's performance against grep to identify performance inflection points beyond which our design exceeds grep's performance. We demonstrate that for a large number of search terms (i.e., 16 or above), our software implementation provides processing rates of over 90 MC/s; in comparison, grep achieves processing rates of under 30 MC/s. However, for fewer than 16 search terms, grep provides far better performance than our design. We also

find that traditional Aho-Corasick automata provide superior performance compared to bit-split automata. In other words, the costs of implementing bit-split automata outweigh the benefits.

3.1 Introduction

HAWK, as described in Chapter II, is an application-specific integrated circuit (ASIC). However, ASIC design is expensive, and due to its application specific nature, limited in terms of usage. An alternative avenue for implementing HAWK is using a field-programmable gate array (FPGA) platform. We do, in fact, implement HAWK on an FPGA as a proof-of-concept. However, FPGA boards can often be expensive and difficult to use.

In light of the above drawbacks of a hardware-based solution, investigating a software-based implementation can be valuable. The typical method of searching for strings in software is using *grep*. However, as previously shown in Section 2.6, *grep* shows poor memory bandwidth utilization when searching for large numbers of search terms (i.e., over 16). For example, when searching for one or two patterns, grep demonstrates a scan rate of about 1 GC/s; however, when searching for 128 patterns, the scan rate drops to 20 MB/s. *Grep's* performance degradation can be attributed to its underlying algorithm for exact pattern matching, the Boyer-Moore/Commentz-Walter algorithms [12, 24, 32]. These algorithms leverage characteristics of the search patterns to skip over potentially multi-character windows of the input stream where a match cannot possibly be found. Since a smaller number of search terms allows for larger skip windows, only a small portion of the input stream needs to be evaluated. As the number of search terms grows, increasingly larger portions of the input stream have to be examined for potential matches, and at the same time, an increased number of comparisons needs to be made.

To study whether memory bandwidth can be more effectively utilized using the techniques of Aho-Corasick [15] and bit-split machines [101], in conjunction with processing multiple characters per step, we implement key aspects of the HAWK design in software. Our software implementation makes use of the Aho-Corasick algorithm which, in contrast to Boyer-Moore/Commentz-Walter, generates a pattern matching state machine for all the search terms, and evaluates all input characters in a manner agnostic to the characteristics of the search patterns. Further, Aho-Corasick only needs to make one state machine transition per input character. Therefore, we expect that for a large number of search terms, Aho-Corasick, and hence our implementation, should demonstrate at least similar performance compared to *grep*.

Further, as shown in Chapter II, bit-split pattern matching automata provide storage benefits over traditional byte-based Aho-Corasick pattern matching automata. In a software context, smaller pattern automata can contribute towards improved cache locality. We also evaluate whether the bit-split technique provides any advantages, and investigate the potential benefits of evaluating multiple characters of the input stream within each processing step.

The major goals of our investigation into a software implementation are:

- Identify inflection points beyond which our software implementation demonstrates higher performance compared to *grep*.
- Evaluate the performance of bit-split designs and various accelerator widths.

Overall, we make the following contributions in this chapter. We demonstrate the design of a software-based compiler and pattern matcher that leverage the techniques of Aho-Corasick and bit-split automata. Our design exhibits a processing rate of over 90 MC/s for 16 or more randomly selected search terms; in comparison, *grep* demonstrates processing rates of under 30 MC/s. However, for 8 or fewer search terms, *grep* is significantly superior to our solution. Additionally, we find that the

costs of implementing bit-split pattern automata outweigh the associated benefits. In other words, traditional Aho-Corasick style byte-based pattern automata offer higher performance in software.

The remainder of the chapter is structured as follows. Section 3.2 describes the details of the software implementation, Section 3.3 discusses results, and Section 3.4 concludes.

3.2 Design

Parameter	*Symbol*
Characters evaluated per processing step (accelerator width)	W
Set of search patterns	S
Number of bits each automaton is responsible for	$\|B\|$

Table 3.1: Notation associated with software implementation.

The software implementation of HAWK's functionality consists of two aspects—a compiler and a pattern matcher. The compiler, as previously described, takes as input a set of user-defined search terms. The compiler then converts the search terms into a set of bit-split pattern matching automata. The pattern matching automata are then utilized by the pattern matcher during the pattern matching process. We restrict our software implementation to single-threaded code since we consider a single instance of *grep* as our baseline for comparison. Further, since we restrict ourselves to exact pattern matching without introducing the notion of fields, we do not implement stages of the HAWK pipeline such as the field alignment unit. Table 3.1 lists the notation used in this chapter.

3.2.1 Compiler

We implement the compiler described in Figure 2.4 and Figure 2.5 using C. The compiler takes in a user query and generates pattern matching state machines. In the context of the software implementation, we limit the user queries to searches for exact

```
Input: Bit split automata set M, input stream S
Output: Match vector v.
1: readAutomata(M)
2: for each character window w ∈ S do
3:     bitset b = splitIntoBits(w)
4:     v.setAllBits()
5:     for each automaton m ∈ M do
6:         transitionRule t = m[currentState]
7:         m.nextState = t.nextState[b[m]]
8:         v = v & t.pmv
9:     if (v > 0) recordMatch()
```

Figure 3.1: Functionality of the software pattern matcher.

patterns. The compiler first sorts the search terms alphabetically and creates padded search terms if needed. Next, the compiler generates bit-split automata using the Aho-Corasick algorithm. Finally, each state in every bit-split automaton is assigned a partial match vector (PMV); the PMV represents the search terms that the said state accepts.

3.2.2 Pattern Matcher

The pattern matcher implements the functionality of HAWK's pattern automata and intermediate match unit; Figure 3.1 illustrates the functionality at a high level. The pattern matcher first reads in the compiled pattern matching automata into its internal data structures (line 1). Each pattern automaton is represented as an array of transition rules. Each transition rule is addressable by the automaton's current state, and consists of an array of all the possible next states that the automaton can transition to given an input value. Every transition rule also has associated with it, a partial match vector (PMV) that specifies the search patterns that the current state accepts. Section 2.4 provides more details on the format of the transition rules.

During each processing step, the pattern matcher reads in, and processes, characters from the input stream in increments of the accelerator width, W (line 2). For example, for $W = 2$, the pattern matcher reads and processes two characters at a

time. The matcher supports bit-split pattern matching automata that can be responsible for a variable number of bits—either single bits, or a set of bits (up to 8 bits). For example, for $|B| = 1$ each automaton is responsible for one bit in the bit representation of the W character window, and for $|B| = 8$ each automaton is responsible for eight bits in the bit representation of the W character window. Note that the $W = 1$, $|B| = 8$ configuration corresponds to byte-based Aho-Corasick. Once the required characters have been read in from the input stream, the character window is parsed, and the relevant bits are sent to each pattern matching automaton (line 3).

Subsequently, each automaton selects a transition rule corresponding to its current state (line 6), and selects its next state from the transition rule based on the input bit(s) (line 7). Finally, the automaton ANDs the PMV contained in the current transition rule with the PMV generated by the previously evaluated automaton (line 8). The match vector that is generated after all the automata have been evaluated represents the search terms that all the pattern automata agree upon as having matched (line 9).

A major distinction between the operation of HAWK and the software implementation is that HAWK's operation is inherently parallel, whereas the software is inherently sequential. More specifically, HAWK evaluates all automata in parallel and then ANDs together all the PMVs generated. The software, on the other hand, evaluates each automaton separately and performs an AND between two PMVs in every step; this evaluation and ANDing is repeated until all automata have been processed.

Since multiple pattern automata are updated sequentially in the software implementation, optimizing portions of the code becomes imperative. To aid in fast matching, we represent the partial match vectors (PMVs) using 128-bit MMX registers; these 128-bit registers provide fast AND operations using SSE instructions (*_mm_and_si128*). During the phase when the pattern automata are being read into

the matcher, we utilize the *_mm_loadu_si128* instruction to load the PMVs. The width of the MMX registers, however, places an upper bound on the number of search terms, $|S|$, and the accelerator width, W, that can be implemented. Use of 256- and 512-bit registers available on more modern architectures can allow for larger values of $|S|$ and W. Finally, to reduce overheads associated with reading the input stream, we leverage memory-mapped file I/O (*mmap*).

3.3 Experiments and Results

We now present the results of the performance comparison between our software implementation and *grep*. Recall that the main goals of our investigation into a software implementation are to identify inflection points beyond which the software implementation demonstrates higher performance than *grep*, and to evaluate the performance of various bit-split configurations and accelerator widths. We vary various design parameters and report the performance of our software implementation and *grep* in megacharacters per second or *MC/s*.

3.3.1 Experiments

While comparing the performance of our software solution against *grep*, we vary the following parameters as powers of two:

- Accelerator width ($W = 1$ to 4)
- Number of search terms ($|S| = 1$ to 128)
- Bits each individual automaton is responsible for ($|B| = 1$ to 8, with 8 representing an automaton responsible for an entire character, i.e., equivalent to Aho-Corasick for $W = 1$)

Since we utilize 128-bit MMX registers, we restrict the number of search terms, $|S|$, to a maximum of 64 for an accelerator width of two ($W = 2$), and to 32 for an

accelerator width of four ($W = 4$). The search terms themselves are picked randomly from the English dictionary. We ensure that the same search terms are used while evaluating our software implementation, and for *grep*.

Note that for *grep*, we only vary the number of search terms, $|S|$. Further, to ensure that the outputs of the software and *grep* are equivalent, we run *grep* with '*-Fo*' and pipe the results to word count (*wc*). The input stream we use is the same 49 GB Wikipedia data dump utilized in Section 2.6. We repeat all experiments five times and report average performance.

3.3.2 Results

We find that the best performance afforded by our accelerator is for $W = 2$ and $|B| = 8$; this configuration strikes a balance between the number of characters read from the input stream, and the number of next state updates and PMV ANDs performed in sequence during each processing step. For $W = 1$, the overhead of reading the input stream dominates, whereas for $W = 4$, the costs associated with updating a minimum of four automata and ANDing together their associated PMVs dominate.

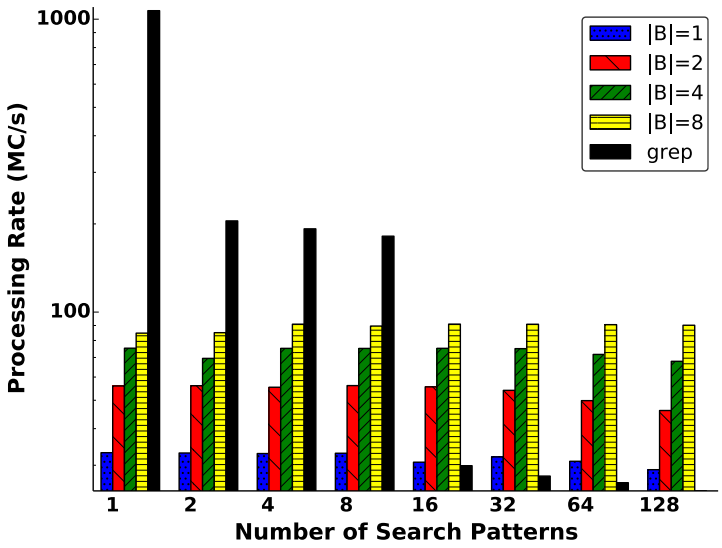


Figure 3.2: Processing rates for W=1.

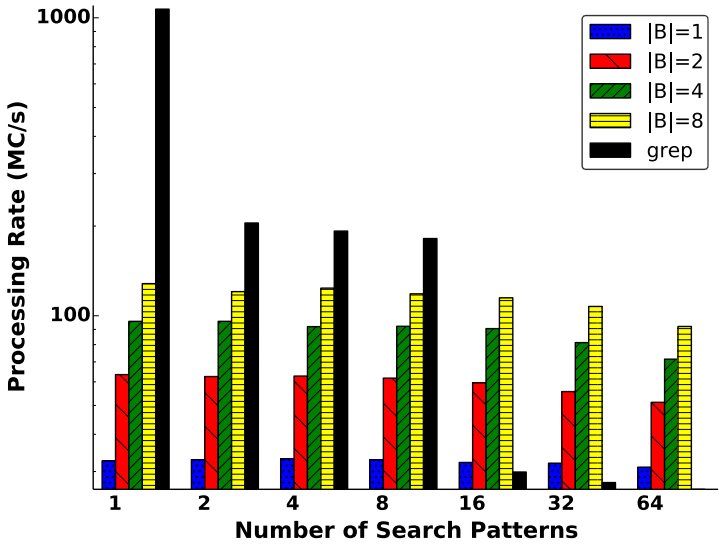


Figure 3.3: Processing rates for W=2.

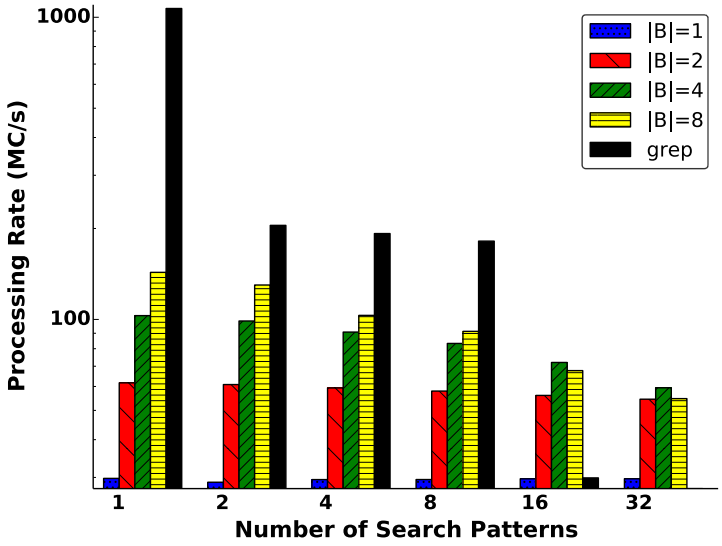


Figure 3.4: Processing rates for W=4.

Figure 3.2, Figure 3.3, and Figure 3.4 show the processing rates, in MC/s, of our software solution and *grep*. In general, for a smaller number of search terms (i.e., under 16) randomly picked from the English dictionary, *grep* shows superior performance compared to our software solution. This phenomenon can be explained by the fact that *grep* utilizes the Boyer-Moore/Commentz-Walter algorithms [24, 32] for exact string matching [12], with Commentz-Walter being used when searching for

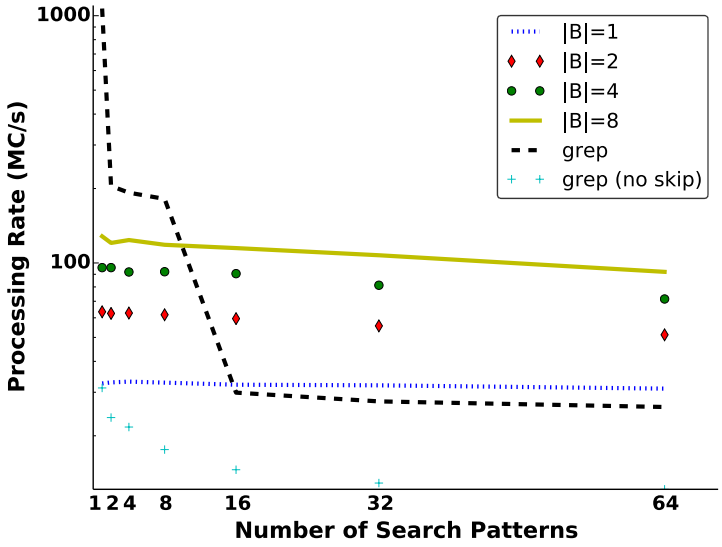


Figure 3.5: Processing rates for W=2 (Line graph).

more than one pattern. These algorithms leverage characteristics of the search terms to skip over portions of the input stream where a match cannot possibly be found. A smaller number of search terms allows for larger skips; typically, only a small fraction of the input stream needs to be evaluated. However, for more numerous search terms, larger portions of the input stream have to be examined for potential matches, and more comparisons need to be made. Further, in practical scenarios, Aho-Corasick has been demonstrated to be faster than Commentz-Walter [11, 36]. Our software implementation, on the other hand, evaluates each character in the input stream independent of the characteristics of, and the number of, search terms. For large numbers of search terms (i.e., 16 and over), our software implementation demonstrates better performance compared to *grep*.

Figure 3.5 shows the processing rates for our software implementation and *grep* for $W = 2$ as a line graph. Figure 3.5 also illustrates the performance of *grep* when all characters in the input stream are evaluated; the associated points on the graph are marked *"grep (no skip)"*. To force *grep* to evaluate every input character, we search for a variable number of single character patterns. We vary the number of

unique single characters we search for from one to 64 as powers of two. We find that *grep's* performance when evaluating all characters in the input stream is drastically worse than when searching for the same number of randomly selected English words. We attribute the difference in *grep's* performance to a higher likelihood of single characters being matched, and therefore, more post-processing operations (since each character is being searched for independently, a larger $|S|$ implies that the sum of the associated matches goes up). We further find that when searching for a single pattern, the processing rate achieved is between that of the 8 randomly-selected English pattern scenario and the 16 randomly-selected English pattern scenario; this gives credence to the conclusion that, typically, beyond $|S| = 16$, all characters in the input stream are being evaluated by *grep*. (Note: For the single pattern case, we report results for the character *'z'* which accounts for 0.16% characters in the input stream. We pick *'z'* because it has one of the lowest character frequencies in English [4], and therefore, post-processing in *grep* due to matches is minimized, while still maintaining a non-zero probability of a match in the input stream).

In general, the best performance for our software implementation is seen for 8 bits per automaton, i.e., for $|B| = 8$. Recall that compared to HAWK, our software implementation is sequential. In other words, the next states for all automata are evaluated sequentially (in contrast, HAWK evaluates each automaton in parallel) and the associated PMVs are also ANDed in a sequential manner. For $|B| = 1$ and $W = 1$, eight different automata are evaluated and eight PMVs are ANDed sequentially in every step; however, for $|B| = 8$, only one automaton needs to be evaluated, and no AND operations need to be performed; this lower number of operations contributes to improved performance. The overall conclusion from the aforementioned graphs is that Aho-Corasick is, in fact, superior to the bit-split approach in software. In other words, the costs of implementing bit-split automata in software overcome the associated benefits. Note that for $W > 1$ and $|B| = 8$, each automaton is responsible

for one character in the input stream. In other words, instead of naïvely extending the alphabet for $W > 1$, we save on storage by making each automaton responsible for one character. Overall, however, for small numbers of search terms, *grep* is a more feasible solution than our software implementation.

We also observe that as the number of search terms increases, the performance of the software implementation degrades (the trend is most visible in Figure 3.3 and Figure 3.4). This performance degradation can be attributed to two factors. First, the number of matches increases with larger $|S|$, thereby requiring more post-processing. For example, for $|S| = 4$, there are approximately 63,000 matches in the Wikipedia input stream; however, for $|S| = 64$, the number of matches increases to almost 7.3 million. Second, the pattern matching automata increase in size with increased numbers of search terms. The larger automata entail more cache misses and a correspondingly higher percentage of cycles stalled on data. For example, for $W = 2$ and $|S| = 4$, only 6% of cycles are stalled on data. But for $W = 2$ and $|S| = 64$, over 30% of cycles are stalled.

As stated above, the $W = 2$ and $|B| = 8$ configuration provides the best overall performance. However, we note that the peak performance improvement for $W = 2$ over $W = 1$, when considering a large number of strings, is about 20%. However, as $|S|$ increases, the performance advantage decreases to only about 2% for 64 search terms. Moving to $W = 4$, in fact, hurts performance due to the costs associated with evaluating a higher number of automata, and the larger sizes of these automata. In fact, for $W = 4$, the performance drop between $|B| = 4$ and $|B| = 8$ for $|S| > 8$ is a direct consequence of the increased automaton sizes.

3.4 Conclusion

In conclusion, we demonstrate a software implementation of HAWK. When run on a Xeon-class core, for a large number of search terms (i.e., 16 or above), we demon-

strate processing rates of over 90 MC/s; in comparison, *grep* demonstrates processing rates of under 30 MC/s. However, for fewer than 16 search terms, *grep* provides far better performance compared to our design. We also find that traditional Aho-Corasick automata provide superior performance compared to bit-split automata. In other words, the costs of implementing bit-split automata outweigh the benefits.

CHAPTER IV

A Hardware Accelerator for Similarity Measurement in Natural Language Processing

The continuation of Moore's law scaling, but in the absence of Dennard scaling, motivates an emphasis on energy-efficient accelerator-based designs for future applications. In natural language processing, the conventional approach to automatically analyze vast text collections—using scale-out processing—incurs high energy and hardware costs since the central compute-intensive step of similarity measurement often entails pair-wise, all-to-all comparisons. In this chapter, we describe a custom hardware accelerator for similarity measures that leverages data streaming, memory latency hiding, and parallel computation across variable-length threads. We evaluate our design through a combination of architectural simulation and RTL synthesis. When executing the dominant kernel in a semantic indexing application for documents, we demonstrate throughput gains of up to 42× and 58× lower energy per similarity-computation compared to an optimized software implementation, while requiring less than 1.3% of the area of a conventional core.

4.1 Introduction

Whereas technology trends indicate that transistor dimensions will likely continue to scale for several technology generations, the anticipated end of CMOS voltage (a.k.a. Dennard) scaling has led many researchers and industry observers to predict the advent of "dark silicon"; that is, that much of a chip must be powered off at any time [22, 39, 86, 104]. This forecast has renewed interest in domain specific hardware accelerators that drastically improve the energy-efficiency of compute intensive tasks to create value from otherwise dark portions of a chip.

One target domain for such accelerators is natural language processing (NLP). With the explosive growth in electronic text, such as emails, tweets, logs, news articles, and web documents, there is a growing need for efficient automatic text processing (e.g., summarization, indexing, and semantic search). The conventional approach to analyze vast text collections—scale-out processing on large clusters with frameworks such as Hadoop—incurs high costs in energy and hardware [59]. We propose and evaluate a hardware accelerator that addresses one of the most data- and compute-intensive kernels that arises in many NLP applications: calculating similarity measures between millions (or even billions) of text fragments [18, 26, 31, 91, 96].

We develop this accelerator in the context of a motivating NLP application: constructing an index for semantic search (search based on similarity of concepts rather than string matching) over massive text corpora such as Twitter feeds, Wikipedia articles, logs, text messages, or medical records. The objective of this application is to construct an index where queries for one search term (e.g., "Ted Cruz") can locate related content in documents that share no words in common (e.g., documents containing "GOP candidate"). The intuition underlying semantic search is that the relationship among documents can be discovered automatically by clustering on words appearing in many documents (e.g., "GOP" frequently appearing in documents also containing "Cruz"). Such a search index can be constructed by generating a graph

where nodes represent documents (such as tweets) and edges represent their pairwise similarity according to some distance measure (e.g., the number of words in common) [38,82]. A semantic search can then be performed by using exact text matching to locate a node of interest in this graph, and, thereafter, using breadth-first search, random walks, or clustering to navigate to related nodes.

Constructing the search graph nominally requires a distance calculation (e.g., cosine similarity) between all document pairs, and is hence quadratic in the number of documents. This distance calculation is the primary computational bottleneck of the application. As an example, over half a billion new tweets are posted to Twitter daily [105], implying roughly 10^{17} distance calculations per day, and this rate continues to grow. Clever pre-filtering can reduce the required number of comparisons by an order of magnitude; nevertheless, achieving the required throughput on conventional hardware remains expensive. For example, based on our measured results of an optimized C implementation of this distance calculation kernel running on Xeon-class cores, we estimate that a cluster of over 2000 servers, each with 32 cores is required to compare one day's tweets within a 24-hour turnaround time.

Instead, we develop an accelerator that can be integrated alongside a multicore processor, connected to its last-level cache, to perform these distance calculations with extreme energy efficiency at the bandwidth limit of the cache interface. The accelerator performs only the distance calculation kernel; other algorithm steps, such as tokenization, sorting, and pre-filtering, have runtimes that grow linearly in the number of documents and are easily completed in software. Our design is inspired by the latency hiding concepts of multi-threading and simple scheduling mechanisms to maximize functional unit utilization. The accelerator comprises a window of active threads (each corresponding to a single document pair), a simple round-robin functional unit scheduler, and three kinds of functional units: intersection detectors (XDs), which identify matching tokens (words) in documents; floating point multiply-

accumulate units (MACs), which perform distance calculations; and floating point multiply-divide units (MDIVs), which normalize the distance measure before it is written back to memory. We evaluate the design through a combination of cycle-accurate simulation in the gem5 framework (performance analysis) and RTL-level synthesis (energy analysis). For Twitter and Wikipedia datasets, our accelerator enables 36×-42× speedup over a baseline software implementation of the distance measurement kernel on a Xeon-like core, while requiring 56×-58× lower energy.

4.2 Related Work

Hardware accelerators for text processing, clustering, semantic search, and database applications have been the focus of extensive research in the architecture community. Tan and Sherwood present a specialized, high-throughput string matching architecture for intrusion detection and prevention [101]. Chen and Chien investigate low-power and flexible hardware architectures for k-means clustering [30]. Fushimi and Kitsuregawa describe a co-processor with hardware sorters for database applications [42], and Moscola et al. implement reconfigurable hardware that extracts semantic information from volumes of data in real-time [73]. Roy et al. present an algorithm for frequent item counting that leverages SIMD instructions [90].

Our accelerator relies on a fast set intersection detector, a topic of much prior work. Wu and co-authors demonstrate a GPU-based solution for set intersection detection on the CUDA platform [111]. Schlegel et al. propose an algorithm for sorted set intersection computation that speculatively executes comparisons between sets using SIMD instructions available on modern processors [93]. Ding and Konig develop linear space data structures to represent sets such that their intersection can be computed in a worst-case efficient way and within memory [34]. In contrast to these works, we propose custom hardware to perform set intersection that is particularly suited to the NLP domain.

Perera and Li have done extensive work in the area of hardware support for distance measurement computation [79, 80]. Their work targets FPGAs and smaller, fixed length vectors. Our proposed design, however, overcomes the drawbacks of FPGAs, and targets variable vector lengths, which is important when dealing with documents larger than a few words.

4.3 Design

We briefly describe the overall problem of constructing a semantic search index and then focus on the dominant kernel, distance calculation between documents, and how our hardware accelerates this operation.

4.3.1 Constructing a Semantic Search Index

The motivating context for our accelerator is the problem of constructing a semantic search index over snippets of text. We implement an algorithm based on the text similarity quantification work of Erkan and Radev [38]. The full application is described in informal psuedo-code in 4.1. In the first step, the textual documents are transformed into a vector representation to reduce their memory footprint. Each word in a document is replaced with a tuple comprising a token id and a weight that represents the information content of the word (based, e.g., on the word's a priori appearance frequency in English text). We then sort the tokens so that the set intersection of two documents can easily be determined with a merge join.

The similarity calculation step nominally must compare all documents pairs, however, documents that share no word in common have a similarity score of zero. The total number of comparisons can be reduced by an order of magnitude by first bucketizing documents (step 2), i.e., adding a pointer to the document into a bucket corresponding to each token in the document. Hence, each bucket contains only documents sharing at least one word in common.

```
[Step 1: Build vocabulary, tokenize, and sort]
1.  For all documents:
2.    Split into sub-strings at whitespace and punctuation
3.    Replace each sub-string with token id, IDF weight; add new tokens as needed
4.    Sort tokens in ascending order; replace duplicates with count
5.    Write out documents as sorted vectors of {token, weight = count * IDF}

[Step 2: Pre-filter and bucketize]
1.  For all documents:
2.    For all tokens in document:
3.      Insert pointer to document into a bucket corresponding to the token

[Step 3: Similarity calculation]
1.  For all buckets:
2.    For all document pairs d1, d2 in bucket:
3.      while d1 or d2 has more tokens:
4.        if d1.token == d2.token: //XD
5.          numerator = numerator + d1.weight * d2.weight //MAC
6.          pop front token from d1 and d2
7.        else:
8.          pop front token with lower token id
9.      similarity[d1,d2] = numerator / (||d1||×||d2|| ) //MDIV

[Step 4: Build similarity graph]
1.  Construct graph with node for each document and edges connecting documents
    with similarity >threshold
2.  Traverse graph (e.g., via random walk) to discover related documents and build
    index
```

Figure 4.1: High-level description of semantic search index construction.

The similarity calculation step then processes each bucket, calculating the similarity of each document pair via a merge join. Our hardware design accelerates this step. Following common practice [38,82], we use *cosine similarity* (the normalized dot product of the two weight vectors) as the distance measure, but our hardware architecture could easily implement other distance measures by replacing the multiply-accumulate operation with an appropriate alternative.

Once the complete similarity matrix of all document pairs has been calculated, the final step is to construct a graph where nodes correspond to documents, and edges connect together documents with similarity scores above some fixed threshold. Then, a conventional search index, mapping search terms to nodes for documents containing

those words, is constructed. Starting from these exact-word-match nodes, additional related documents can be discovered through traversal of the graph (e.g., via random walk).

Whereas GPUs are often used for problems that exhibit large-scale parallelism, they are not well-suited to calculating distance measures using the method described in 4.1. Because input documents vary in length, the merge-join set intersection operation does not lend itself to SIMT parallelism, since loop bounds for each document-pair depend on document length. It is unclear how to stage the input data to avoid substantial thread divergence and many idle GPU threads. It is also unclear how to lay out data in memory to enable coalesced accesses, which are crucial to high GPU performance.

4.3.2 Accelerator Architecture

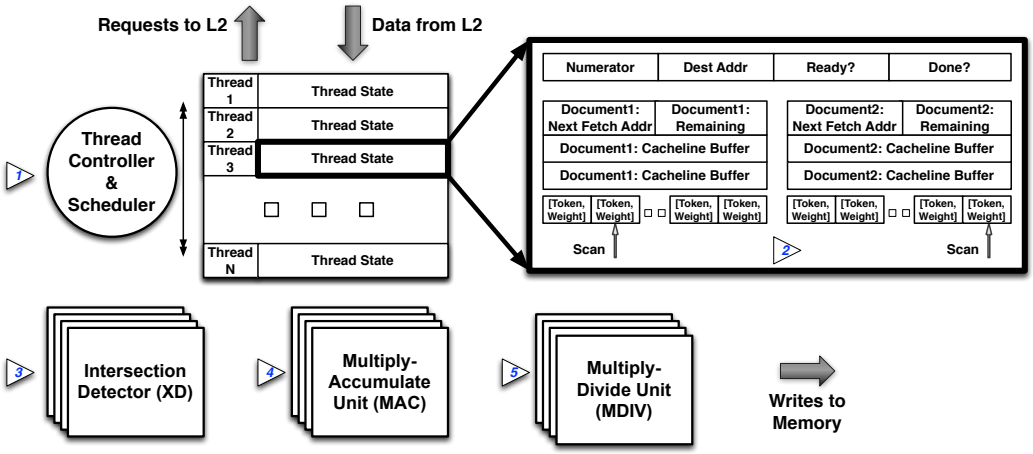


Figure 4.2: Accelerator block diagram.

Our accelerator implements step three of 4.1 entirely in hardware. Figure 4.2 shows a block diagram of our accelerator. The accelerator is connected to the L2 bus and reads from the system's L2 cache. Since the accelerator never reads its own output, it writes memory, via the L2 bus, with non-cacheable transactions that bypass L2. The CPU controls the accelerator by preparing a region of memory with an array of document-pair descriptors. Each descriptor contains the address of the vector

representing each document and a destination address for the similarity calculation result. The accelerator is activated through programmed I/Os that provide the start address and length of the descriptor array. The CPU can then sleep until an interprocessor interrupt from the accelerator is delivered to indicate completion.

The accelerator is architected much like an in-order core and has six major architectural blocks: a memory read interface, a thread controller/scheduler, intersection detectors (XDs), multiply-accumulate units (MACs), multiply-divide units (MDIVs), and finally, a memory write interface. As most NLP algorithms represent concepts like document similarity with floating-point values, we use floating-point functional units. We describe the operation of the accelerator by walking through a simple example. Numerical labels in Figure 4.2 correspond to the steps described below.

(1) Fill thread window. The thread controller maintains a window of active threads (each thread corresponding to a document pair specified in a descriptor), and schedules threads to functional units using a simple round-robin scheduler. Each thread window entry comprises thread status information (addresses for the next data fetches, destination address, remaining indices to be scanned, partial sum) and several cacheline-sized data blocks for each input document. If any thread window entry is empty, the controller fills it with the next descriptor in the input array. The controller then iterates over all active threads and issues requests to L2 to fill all available buffer space for each document. While our simulations ignore virtual memory, in a practical implementation, virtual addresses must be translated by either a dedicated TLB or by the TLB of a core neighboring the accelerator.

(2) Token data arrives. Once document data arrive for a particular thread, processing can begin. Each document is represented as an array of {*token, weight*} tuples. Each cycle, a ready thread arbitrates for an XD unit which will compare the next two tokens sent to it.

Parameter	*Range*
Number of XDs	1-32
Number of MACs	1-32
Number of MDIVs	1-32
Thread Window Size	1-64
XD Delay	1 cycle
MAC Delay	3 cycles
MDIV Delay	7 cycles

Table 4.1: Simulation parameters.

(3) XD comparison. The operation of the XD units is conceptually similar to a sort-merge join. In a particular cycle, the XD unit compares the token ids of the next tokens from each document. Recall that tokens in each document have been sorted. Hence, if the token ids do not match, the head pointer for the document with the smaller token is advanced and updated tokens are compared again in the next cycle. If the tokens match, the thread is marked and will arbitrate for a MAC unit in the next cycle.

(4) MAC calculation. When an XD unit reports a match, the MAC unit performs the floating-point multiply-accumulate to calculate an updated numerator for the document similarity computation. The input weight values from the two documents are multiplied and summed with the current numerator, and the result is stored back into the active thread entry.

(5) Normalization. When the end of either input document is reached, the merge-join set-intersection operation is complete. The thread then arbitrates for an MDIV unit to normalize the accumulated numerator by the product of the magnitudes of the input documents, and then arbitrates for the store interface unit to write its output value to the destination address in memory. The thread window entry is then freed.

Statistic	*Twitter*	*Wikipedia*
Number of documents in dataset	10,000,000	100,000
Number of documents in bucket of interest	2,500,000	15,000
Average document length (tokens)	9.3	511.5
Minimum document length (tokens)	2	24
Maximum document length (tokens)	39	4,107
Average number of intersections	1.01	200.9

Table 4.2: Statistics for Twitter and Wikipedia datasets.

Unit	*Area* (μm^2)	*Delay (ns)*	*Power (mW)*
XD	1,143	0.5	0.5
MAC	14,232	1.5	4.93
MDIV	18,216	3.5	3.95
Controller+Thread Window	293,878	0.5	69.4

Table 4.3: Synthesis results.

4.4 Methodology

We use a two-pronged approach to evaluate our design relative to an optimized software baseline. We measure performance of both the pure software implementation and hardware-accelerated kernel using the gem5 architectural simulator [20]. To investigate energy savings and area overheads, we implement our design in Verilog and synthesize using industrial 45 nm standard cells.

4.4.1 Simulation

To compare the performance of our accelerator to a CPU baseline, we extend the gem5 simulator with a device model for our accelerator. The accelerator connects to the L2 interface. It can read and write 64-byte cache blocks from L2 and is controlled via programmed I/O to special memory locations. We vary the hardware parameters of our design (number of threads, XDs, MACs, and MDIVs) to determine the minimum hardware needed to saturate L2 and/or main memory bandwidth, which ultimately limits the performance of the accelerator. Table 4.1 shows the various parameters of the design space we explore. We determine functional unit delays from synthesized timing results targeting a 2 GHz clock.

We contrast our hardware design with a SSE-accelerated C implementation of the cosine similarity kernel compiled with gcc -O3. We model a 4-wide out-of-order processor running at 2 GHz with 64 KB L1 caches and an 8 MB L2. As operating system interactions and I/O do not contribute significantly to the runtime of this workload, we use gem5's syscall emulation mode. Note that this simulation mode does not model virtual memory; nevertheless, because of the high data locality, TLB misses are unlikely to significantly affect the runtime of the baseline or hardware-accelerated execution. We validate that the CPU runtimes reported by gem5 are, on average, within 6% of the runtimes observed on a comparable 8-core Xeon-class server. For consistent energy comparisons between the CPU baseline and our hardware design, we report the gem5 results.

We construct benchmarks for Twitter and Wikipedia from databases of 10 million tweets and 100,000 articles respectively. Table 4.2 shows various statistics for the datasets we use. The software pre-processing steps of the semantic index construction algorithm (tokenization, sorting, and bucketizing) are performed offline in advance; our measurements focus only on the dominant distance calculation step. From the Twitter database, we select the most frequently occurring token (corresponding to the string *"RT"*) and construct a bucket of all tweets containing this token (2.5 million entries, requiring 6.25 trillion distance calculations). As it is impossible to process this vast dataset in simulation, we simulate only the first 5 million tweet-pair comparisons and use the first 1 million tweet-pairs for warm-up. We follow a similar bucketization process for the Wikipedia data, and simulate 50,000 article-pair comparisons with the first 2000 pairs used for warm-up.

When processing the entire data set, the document-pair comparisons are blocked to maximize L2 locality. Thus, the computation will alternate between one phase where a large fraction of document accesses miss to main memory and a much longer phase where a block of documents is resident in L2 and there are no main memory

accesses. The relative time spent in each phase depends on the relative size of the document bucket and the L2 cache. To ensure that our accelerator design hides latency and saturates available L2/memory bandwidth in both phases, we construct two test cases: *Fit*, wherein all documents are L2-resident, and *Spill*, wherein the L2 is empty and documents must be retrieved from memory. We report speedup of the accelerator relative to the CPU baseline for both phases. To avoid L2 cache pollution, and since the accelerator never reads its own output, outputs are written directly to main memory using uncacheable writes.

4.4.2 Timing, Power, and Area Analysis

We implement the accelerator in Verilog and synthesize using an industrial 45 nm standard cell library to obtain delay, area, and power results assuming a 0.72 V supply voltage. Table 4.3 shows the post-synthesis delays and areas for each of the sub-units of the accelerator (the thread window size is 6, the configuration we use in our final design). We use these synthesized delay results to set functional unit latencies within the gem5 model. Our floating-point multiply, multiply-accumulate, and divide units are from the Synopsys DesignWare IP suite [99]. The delays reported in the table are rounded up to the next 0.5 ns clock edge. We use system configuration and functional unit activity results from gem5 to generate estimates of CPU core and cache power using McPAT [61].

4.5 Results

The following subsections outline the performance and energy improvements afforded by our design.

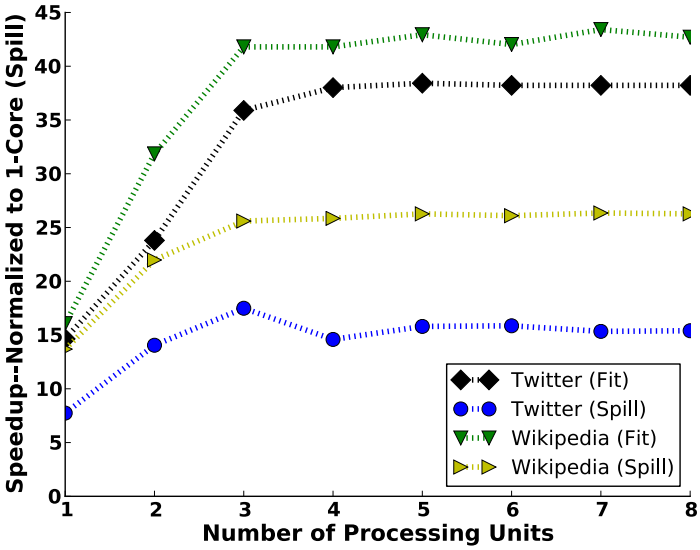
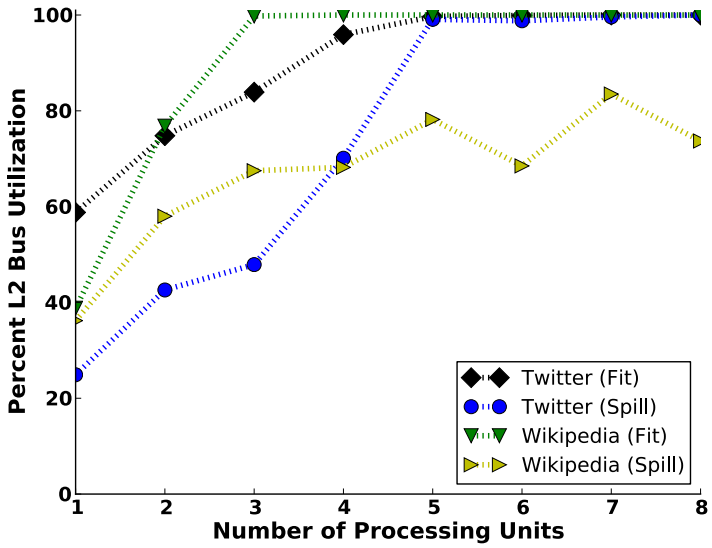


Figure 4.3: Speedup. Normalized to 1-core *CPU (Spill)* case.

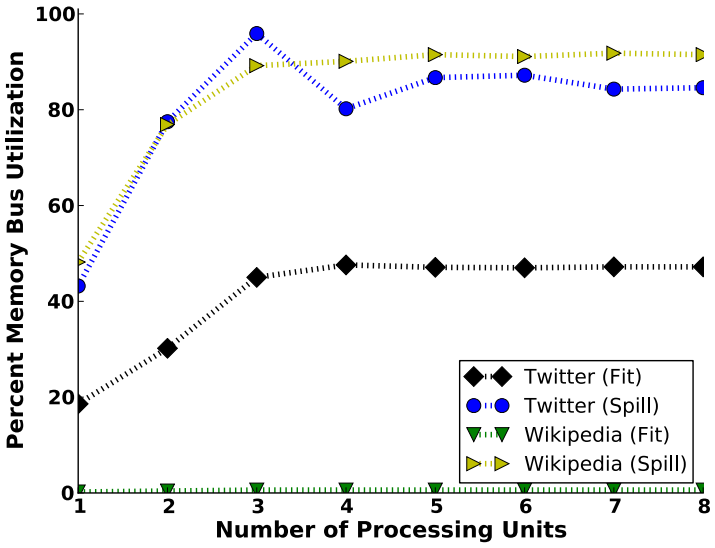
4.5.1 Performance

We first contrast the performance and performance scalability of the accelerator relative to the baseline cosine similarity software kernel running on conventional out-of-order CPU cores. Figure 4.3 shows the speedup provided by the accelerator for the Twitter and Wikipedia datasets for both the *Fit* and *Spill* scenarios normalized to each single-core *Spill* CPU baseline. On the horizontal axis, we vary the amount of hardware dedicated to the accelerator. To simplify presentation of the results, in this experiment, we vary the number of XD, MAC, MDIV units together, from 1 to 8. Each configuration has double the number of thread slots as XD units. These configurations all overprovision MAC and MDIV units relative to XD units; we further optimize the functional unit mix in subsequent experiments.

Through a combination of simulation, and experiments on a Xeon-class server, we verify that the baseline CPU performance scales roughly linearly with the number of CPU cores, up to about an 8.5× speedup with 8 cores for the *Twitter (Fit)* scenario over the single-core *Twitter (Spill)* case. CPU performance is limited because of the overheads of instruction execution (memory addressing, loop flow control, etc.).



(a) Percent Accelerator-L2 Bus Utilization



(b) Percent L2-Memory Bus Utilization

Figure 4.4: L2 and memory bus utilization. (a) L2 bus saturates for both *Spill* scenarios and the *Twitter (Spill)* case. (b) Memory bus is a bottleneck for the *Spill* configurations.

The accelerator enables substantial speedups. Even with only one of each functional unit, the accelerator can achieve speedups of 8× and 14× in the *Twitter (Spill)* and *Wikipedia (Spill)* scenarios respectively. In general, three XD units are required to achieve peak speedup. The accelerator improves performance because it eliminates all software overheads; e.g., in the *Twitter (Fit)* case, each XD unit can process a tweet-pair roughly every 24 clock cycles, while a CPU core on average requires 353 cycles to execute 469 instructions per tweet-pair.

Performance generally saturates beyond three XDs since the accelerator fully utilizes either the L2 bus for *Twitter (Fit and Spill)* and *Wikipedia (Fit)* scenarios, or main memory bandwidth for both *Spill* scenarios. We show the relevant bus utilization results in Figure 4.4. *Twitter (Spill)* is L2- or memory-bandwidth bound depending on the number of processing units deployed; this configuration also demonstrates decreased performance with more than three processing units due to destructive interference effects. *Wikipedia (Fit)* shows little memory traffic since the number of writes to memory is negligible, and all reads are serviced by the L2. As a point of comparison, for the CPU case, even with 8 cores, the L2 and memory bus utilizations peak at 8.5% for *Wikipedia (Fit)* and at 19% for *Twitter (Spill)* respectively.

We find that the pareto-optimal design, when considering performance, energy, and area in conjunction, consists of three XD units, two MAC units, one MDIV unit, and six thread slots. The Wikipedia dataset tends to favor slightly more functional units compared to the Twitter dataset due to its larger document size. Further, larger thread windows are favored in the *Spill* scenarios since they must maintain more outstanding accesses to the memory system to hide the long delay to access main memory.

Accelerator Configuration		
[XDs, MACs, MDIVs]	[3, 2, 1]	
Issue Window	6	
Area		
Core	24.88 mm^2	
Accelerator	0.31 mm^2	
Power		
Core	6 W	
Uncore	14.8 W	
Accelerator	0.43 W	
Fit Energy & Performance – CPU 1-core Baseline		
	Twitter	*Wikipedia*
Core Energy/Document-Pair	1.18 μJ	96.5 μJ
Chip Energy/Document-Pair	4.09 μJ	339.4 μJ
Fit Energy & Performance – Accelerator		
	Twitter	*Wikipedia*
Accelerator Energy/Document-Pair	2.12 nJ	170.9 nJ
Chip Energy/Document-Pair	72.7 nJ	5.8 μJ
Chip Energy Ratio (Core:Accelerator)	56.3:1	58.5:1

Table 4.4: Power and area results.

4.5.2 Area and Energy

Table 4.4 shows the area overhead and energy savings when using our accelerator in the [3 XD, 2 MAC, 1 MDIV] configuration with a thread window size of six. Note that, we assume that the accelerator is power-gated when cores are active and vice-versa. The accelerator only imposes an area overhead of 0.31 mm^2, less than 1.3% of the area of a core. Because of its simple microarchitecture and lack of instruction fetch/decode bottlenecks, the accelerator's power requirements are much lower than that of a core. The power savings translate to an even larger energy-efficiency gain, since the accelerator can also process document-pairs much faster (and hence incur less leakage overhead per processed document-pair). Overall, the accelerator improves energy efficiency by approximately two orders of magnitude relative to the CPU baseline.

4.6 Conclusion

The conventional approach of using scale-out methods to automatically analyze vast text collections incurs high energy and hardware costs since the central compute-intensive step of similarity measurement often entails pair-wise, all-to-all comparisons. We propose a custom hardware accelerator for similarity measures that leverages data streaming and parallel computation, and, due to its low-power requirements, utilizes dark silicon areas of the chip that would otherwise have to be powered down. Architectural simulations and RTL synthesis demonstrate throughput gains of up to 42× and 58× lower energy consumption compared to an optimized software implementation of cosine similarity calculation, while incurring minimal area overheads.

CHAPTER V

Conclusion

The confluence of the rapid growth in electronic data in recent years, and the renewed interest in domain-specific hardware accelerators presents exciting technical opportunities. Traditional scale-out solutions for processing the vast amounts of text data have been shown to be energy- and cost-inefficient. In contrast, custom hardware accelerators can provide higher throughputs, lower latencies, and significant energy savings. In this thesis, we have presented a set of hardware accelerators for unstructured big-data processing and natural language processing.

The first, an accelerator called HAWK, is targeted towards unstructured log processing. HAWK scans data at a fixed 32 GB/s, an order of magnitude higher than standard in-memory databases and tools. HAWK's scan engine requires no control flow or caches; hence, the hardware scan pipeline never stalls and can operate at a fixed 1 GHz frequency processing 32 input characters per clock cycle. The accelerator's design avoids the cache misses, branch prediction misses, and other aspects of CPUs that make performance unpredictable and require area-intensive hardware to mitigate. HAWK leverages a novel formulation of the state machines that implement the scan operation, thereby facilitating a hardware implementation that can process many characters concurrently while keeping on-chip storage requirements relatively small. In the HAWK design, 32 consecutive characters are conceptually

concatenated into a single symbol, allowing a single state transition to process all 32 characters. Naïvely transforming the input alphabet in this way leads to intractible state machines—the number of outgoing edges from each state is too large to enable fixed-latency transitions. We leverage the concept of bit-split state machines, wherein the original machine is replaced with a vector of state machines that each process only a bit of input. As a result, each per-bit state requires only two outgoing edges. Another novel feature of the HAWK design is that the accelerator searches for strings and numbers in a manner that is agnostic to the location of field and record delimiters in the input log file. The mapping between matched strings/numbers and fields is done *after-the-fact* using a specialized field matcher unit. In its pareto-optimal configuration, HAWK requires 45 mm^2 in 45 nm technology and consumes 22 W during operation.

The second accelerator targets similarity measurement in natural language processing. This accelerator addresses one of the most data- and compute-intensive kernels that arises in many NLP applications: calculating similarity measures between millions (or even billions) of text fragments. We develop this accelerator in the context of a motivating NLP application: constructing an index for semantic search (search based on similarity of concepts rather than string matching) over massive text corpora. Constructing the search index nominally requires a distance calculation (e.g., cosine similarity) between all document pairs, and is hence quadratic in the number of documents; this distance calculation is the primary computational bottleneck of the application. As an example, over half a billion new tweets are posted to Twitter daily, implying roughly 10^{17} distance calculations per day. An optimized C implementation of this distance calculation kernel running on Xeon-class cores, would require a cluster of over 2000 servers, each with 32 cores, to compare one day's tweets within a 24-hour turnaround time. Instead, our accelerator for similarity measurement can be integrated alongside a multicore processor, connected to its last-level

cache, to perform these distance calculations with extreme energy efficiency at the bandwidth limit of the cache interface. By leveraging the latency hiding concepts of multi-threading and simple scheduling mechanisms, we maximize functional unit utilization. Our accelerator provides 36×-42× speedup over optimized software running on server-class cores, while requiring 56×-58× lower energy, and only 1.3% of the area.

APPENDICES

APPENDIX A

Minimizing Remote Accesses in MapReduce Clusters

MapReduce, in particular Hadoop, is a popular framework for the distributed processing of large datasets on clusters of relatively inexpensive servers. Although Hadoop clusters are highly scalable and ensure data availability in the face of server failures, their efficiency is poor. We study data placement as a potential source of inefficiency. Despite networking improvements that have narrowed the performance gap between map tasks that access local or remote data, we find that nodes servicing remote HDFS requests see significant slowdowns of collocated map tasks due to interference effects, whereas nodes making these requests do not experience proportionate slowdowns. To reduce remote accesses, and thus avoid their destructive performance interference, we investigate an intelligent data placement policy we call 'partitioned data placement'. We find that, in an unconstrained cluster where a job's map tasks may be scheduled dynamically on any node over time, Hadoop's default random data placement is effective in avoiding remote accesses. However, when task placement is restricted by long-running jobs or other reservations, partitioned data placement substantially reduces remote access rates (e.g., by as much as 86% over random placement for a job allocated only one-third of a cluster).

A.1 Introduction

MapReduce [33] is a popular framework for the distributed processing of large datasets. One of the most popular implementations of the MapReduce programming model is Hadoop [2], an open-source Java implementation. Hadoop's design centers on affording scalability and availability of data. Hadoop provides scalability by making data management transparent to cluster administrators; this transparent data management allows the framework to support thousands of machines and petabytes of data. Hadoop ensures data availability (and scalability) by distributing three replicas of all data blocks that constitute a file randomly among distinct nodes. Whenever possible, Hadoop moves computation to data, as opposed to the more expensive option of moving data to computation. However, Hadoop's storage layer, the *Hadoop Distributed File System* or *HDFS* [23], facilitates remote data accesses when moving computation is not possible.

Until recently, network bandwidth has been a relatively scarce resource, and hence, conventional wisdom has held that remote data accesses should be minimized [33]. However, network performance improvements continue to outpace disk, which has led some researchers to argue that disk locality will soon be irrelevant in datacenter computing [17]. Indeed, we corroborate that this hypothesis holds even today when communicating over an unsaturated 1 Gb network—the performance gap between CPU-bound map tasks that access local and remote data (served from an idle node) is as little as 1.6%. Interestingly, however, we find that *servicing* remote HDFS requests disproportionally slows map tasks located on the same node, particularly under Linux's default "deadline" I/O scheduler (which biases scheduling to improve I/O performance; the fair I/O scheduler shrinks performance disparities at the cost of worse overall performance). For CPU-intensive map tasks, we find that *Reader* nodes, which access data from a remote node but serve no remote requests themselves, suffer only a 2% slowdown relative to local accesses. However, a *Server* node, which

services remote requests while also executing map tasks, suffers a 13% slowdown. Slowdowns are much larger for I/O-bound map tasks. Hence, we conclude that, unless MapReduce clusters use dedicated storage nodes, *remote accesses must still be minimized.*

Based on these observations, we investigate intelligent data placement as a potential avenue to reduce remote accesses. We focus our investigation on the "map" phase of MapReduce jobs as initial data placement is immaterial thereafter. Hadoop's scheduler is designed to assign map tasks to nodes such that they access data locally whenever possible. When a computation resource is assigned to a job, the scheduler scans the list of incomplete map tasks for that job to find any tasks that can access locally available data. Only if no such tasks are available will it schedule a task that must perform remote accesses. Hence, jobs with dedicated access to the entire cluster rarely incur remote accesses (remote accesses only arise at the end of the map phase, when few map tasks remain, or under substantial load imbalance, for example, due to server heterogeneity [14]). However, restrictions on task assignment, because of long-running tasks, prioritization among competing jobs, dedicated allocations, or other factors, can rapidly increase the number of remote accesses.

We contrast Hadoop's default random data placement policy against an extreme alternative, *partitioned data placement*, wherein a cluster is divided into partitions, each of which contains one replica of each data block. (Note that, since the number of replicas is unchanged and placement remains random within each partition, availability is, to first-order, unchanged). By segregating replicas, due simply to combinatorial effects, we increase the probability that a large fraction of distinct data blocks is available even within relatively small, randomly selected allocations of the cluster. We further consider the utility of adding additional replicas for frequently accessed blocks, to increase the probability that these blocks will be available locally in a busy cluster.

Our evaluation, through a combination of simulation of the Hadoop scheduling algorithm and validation on a small-scale test cluster, leads to mixed conclusions:

- When scheduling is unconstrained and task lengths are well-chosen to balance load and avoid long-running tasks, Hadoop's scheduler is highly effective in avoiding remote accesses regardless of data placement, as the job can migrate across nodes over time to process data blocks locally. Under an "Unconstrained" allocation scenario, Hadoop can achieve 98% local accesses.

- However, when task allocation is constrained to a subset of the cluster (e.g., because of long-running tasks, reserved nodes, restrictions arising from job priorities, power management [59], or other node allocation constraints), partitioned data placement substantially reduces remote data accesses. For example, under a "Restricted" allocation scenario where a job may execute on only one-third of nodes (selected at random), partitioned data placement reduces remote accesses by 86% over random data placement.

- We demonstrate that selective replication of frequently accessed blocks can further reduce remote accesses in *restricted allocation* scenarios.

This chapter is organized as follows: Section A.2 provides relevant background. Section A.3 delves into why reducing remote accesses is important even under unsaturated networks. Section A.4 explores the data placement policies considered in this research. Section A.5 provides experimental results for the performance of the data placement policies under different job scheduling scenarios, and Section A.6 concludes.

A.2 Related Work

Data replication is widely used in distributed systems to improve performance when a system needs to scale in numbers and/or geographical area [102]. Replication

can increase data availability, and helps achieve load balancing in the presence of scaling. For geographically dispersed systems, replication can reduce communication latencies. Hadoop leverages replication to provide both availability and scalability. Further, Hadoop places two replicas of a data block on the same rack to save inter-rack bandwidth.

Caching is a special form of replication where a copy of the data under consideration is placed close to the client that is accessing the data. Caching has been used effectively in distributed file systems such as the *Andrew File System (AFS)* and *Coda* to minimize network traffic [46, 92]. Gwertzman and Seltzer have proposed a technique of server-initiated caching called *push caching* [44]. Under this technique, a server places temporary replicas of data closer to geographical regions from which large fractions of requests are arriving. Since replication and caching imply multiple copies of a data resource, modification of one copy creates consistency issues. Much research in the distributed systems field has been devoted to efficient consistency maintenance [74,102]. However, since Hadoop follows a write-once, read-many model for data (i.e., data files are immutable), maintaining consistency is not a concern.

In systems with distributed data replicas, achieving locality while maintaining fairness is a challenge. Isard and co-authors propose *Quincy*, a framework for scheduling concurrent distributed jobs with fine-grain resource sharing [49]. Quincy defines fairness in terms of disk-locality and can evict tasks to ensure fair distribution of disk-locality across jobs. Overall, the system improves both fairness and locality, achieving a 3.9x reduction in the amount of data transferred and a throughput increase of up to 40%.

Zaharia et al. create a fair-scheduler that maintains task locality and achieves almost 99% local accesses via *delay scheduling* [114]. Under delay scheduling, when a job that should be scheduled next under fair-scheduling cannot launch a data-local task, it stalls a small amount of time while allowing tasks from other jobs to be

scheduled. However, delay scheduling performs poorly in the presence of *long tasks* (nodes do not free up frequently enough for jobs to achieve locality) and *hotspots* (certain nodes are of interest to many jobs; for example, such nodes might contain a data block that many jobs require). The authors suggest long-task-balancing and hotspot replication as potential solutions, but do not implement either. In contrast to the authors' approach, we focus on how intelligent data placement can be used to maximize MapReduce efficiency in scenarios where node allocations are restricted.

Eltabakh and co-authors present *CoHadoop* [37], a lightweight extension of Hadoop that allows applications to control where data are stored. Applications give hints to CoHadoop that certain files are related and may be processed jointly; CoHadoop then tries to co-locate these files for improved efficiency. Ferguson and Fonseca [40] highlight the non-uniformity in data placement within Hadoop clusters, which can lead to performance degradation. They propose placing data on nodes in a round-robin fashion instead of Hadoop's default data placement, and demonstrate an 11.5% speedup for the sort benchmark.

Ahmad et al. [14] observe that MapReduce's built-in load balancing results in excessive and bursty network traffic, and that heterogeneity amplifies load imbalances. In response, the authors develop *Tarazu*, a set of optimizations to improve MapReduce performance on heterogeneous clusters. Xie et al. [113] study the effect of data placement in clusters of heterogeneous machines, and suggest placing more data on faster nodes to improve the percentage of local accesses. Zaharia et al. [116] also investigate MapReduce performance in heterogenous environments. The authors design a scheduling algorithm called *Longest Approximate Time to End (LATE)*, that is robust to heterogeneity and can improve Hadoop response times by a factor of two.

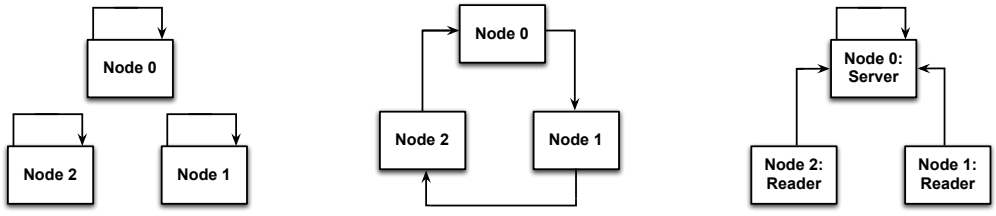
Ananthanarayanan et al. [16] observe that MapReduce frameworks use filesystems that replicate data uniformly to improve data availability and resilience. However, job logs from large production clusters show a wide disparity in data popularity. The

authors observe that machines and racks storing popular content become bottlenecks, thereby increasing the completion times of jobs accessing these data even when there are machines with spare cycles in the cluster. To address this problem, the authors propose a system called *Scarlett*. Scarlett accurately predicts file popularity using learned trends, and then selectively replicates blocks based on their popularity. In trace driven simulations and experiments on Hadoop and Dryad clusters, Scarlett alleviates hotspots and speeds up jobs by up to 20.2%. We explore the utility of selective replication in combination with partitioned data placement in subsequent sections.

Prior work has also shown that well-designed data placement might allow MapReduce clusters to be dynamically resized in response to load, in an effort to increase energy efficiency without compromising data availability [57, 59].

Finally, recent research demonstrates that application demands in production datacenters can generally be met by a network that is slightly oversubscribed [51]. However, as we show subsequently, even under unsaturated network conditions, remote accesses impose significant performance penalties on nodes that service these remote requests.

A.3 The Cost of Remote Accesses



(a) Local (b) Remote (c) Asymmetric

Figure A.1: Experimental configurations for characterizing the costs associated with remote accesses. Arrows indicate read requests.

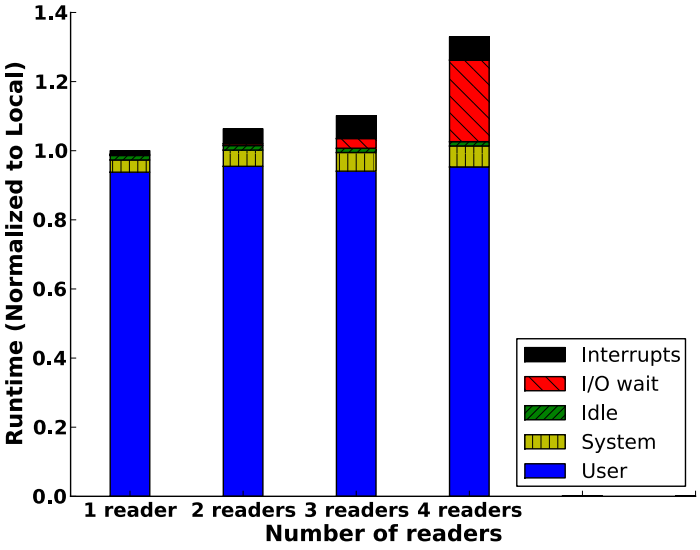
Microbenchmark	Local	Remote	Asymmetric	
			Reader	*Server*
CPU-intensive	1.0	1.1	1.02	1.13
I/O-intensive	1.0	1.3	2.65	3.14

Table A.1: Runtimes for various I/O configurations.
(Normalized to Local for each microbenchmark)

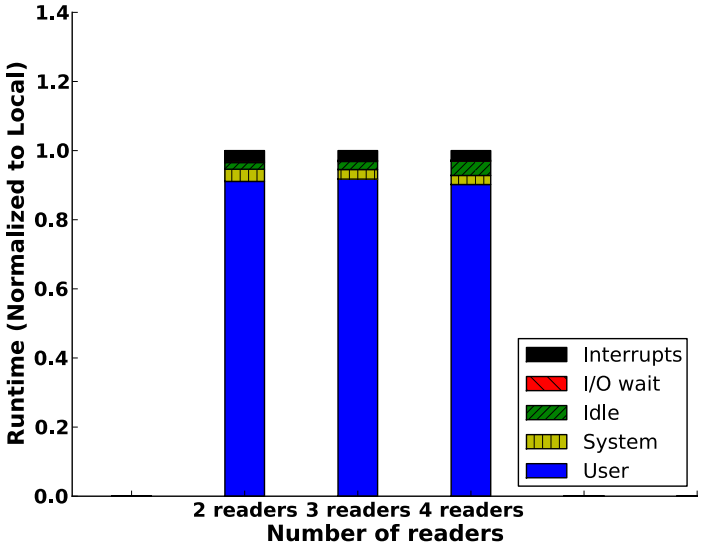
It is clear that remote accesses add to network traffic. When the network in a cluster is near saturation, each extra remote access contributes to longer latencies and even higher network traffic. Until recently, network bandwidth has been small compared to the combined disk bandwidth in a cluster; hence, relatively few simultaneous remote accesses can potentially constrain a network. It is therefore evident that remote accesses are best minimized under busy networks.

Perhaps more surprising, however, is our finding that remote accesses can cause performance penalties even in a low-latency network that is far from saturation (a scenario likely to become more prevalent as data center network topologies improve). As we will show, these performance penalties do not arise due to higher latency from retrieving data over the network. Indeed, to the contrary, we find that a map task accessing data locally or remotely experiences little difference in performance. Instead, we find the interference effect of *servicing* remote HDFS requests leads to a significant degradation of collocated map tasks. Hence, if all I/O can be performed locally, the peak throughput of a MapReduce cluster improves.

To study remote access overheads under unsaturated network conditions, we set up a small Hadoop cluster. We use Hadoop v0.21 on low-end servers representative of the low-cost systems often used for throughput computing clusters. Each server has eight 1.86 GHz Intel Xeon cores, and 16 GB of RAM running stock Ubuntu 10.10. This Linux release enables the "deadline" I/O scheduler (described later) by default. The inexpensive hard disks in these systems provide 50 MB/s sustained read



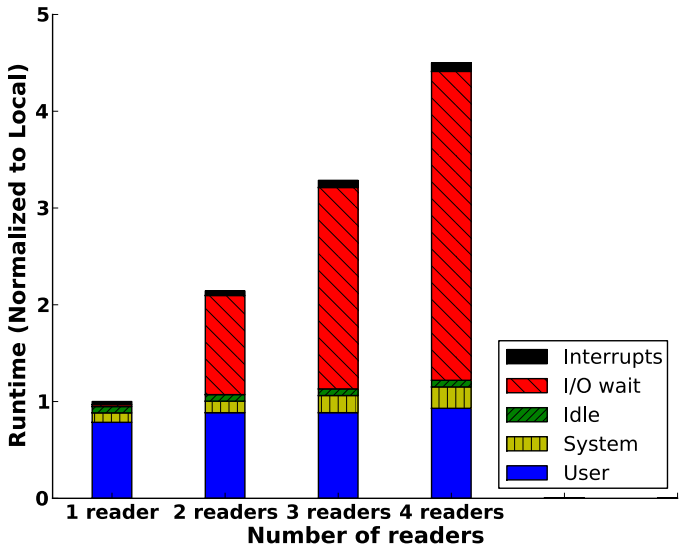
(a) Server



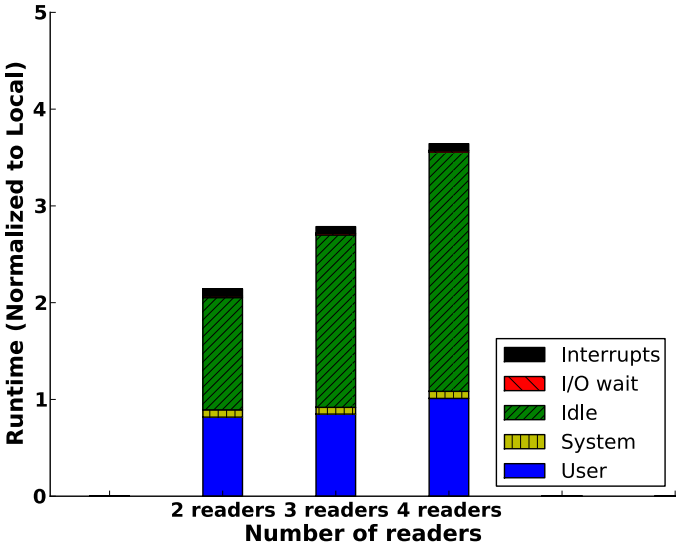
(b) Reader

Figure A.2: Runtime breakdown on the Server and representative Reader for the CPU-intensive microbenchmark. (a) As the number of readers increases, the CPU on the Server spends more time in I/O wait. (b) I/O wait associated with the read request is effectively masked on the Reader.

bandwidth over HDFS. The servers are connected via a dedicated gigabit Ethernet switch.



(a) Server



(b) Reader

Figure A.3: Runtime breakdown on the Server and representative Reader for the I/O-intensive microbenchmark. (a) As the number of readers increases, the Server spends more time in I/O wait. (b) I/O wait associated with the read request is not effectively masked on the Reader, and is seen as increased idle time.

We demonstrate the cost of remote accesses using a minimal Hadoop cluster of only three datanodes and a fourth dedicated namenode. While this configuration is not representative of typical MapReduce clusters, it allows us to isolate and easily

measure I/O performance effects. We use two microbenchmarks, CPU-intensive and I/O-intensive, respectively. The CPU-intensive microbenchmark runs the map task of the WordCount benchmark included with the Hadoop distribution. The I/O-intensive microbenchmark runs an empty map task. Note that since initial data placement does not matter beyond the map phase, we limit this experiment to only the map phase. The input file for both workloads is a 9.5 GB text file with 64 MB data blocks. Each datanode contains a complete copy of the file with the block replication factor set to one. We repeat experiments ten times and report averages; all reported results have 95% confidence intervals of 0.5% or better. We verify that network bandwidth never approaches saturation in any experiment. Further, to isolate the costs associated with remote accesses, we disable 7 of the 8 cores on the servers.

First, we perform a simple test contrasting the perfomance of map tasks that access local data against map tasks that access remote data served from an otherwise-idle node. This simple test isolates the impact of the network on map task performance. As predicted in recent literature [17], because the available network bandwidth exceeds the bandwidth of the disk and the data blocks transferred for each map task are large (64 MB), there is a negligble performance difference between local and remote accesses: the remote accesses incur only a 1.6% slowdown. Hence, one might conclude that remote accesses (and hence data placement) have no impact on performance.

However, this simple test neglects the interference effects of serving HDFS requests while concurrently running map tasks. We study these interference effects through three data access scenarios that are illustrated in Figure A.1. In the first scenario (*Local*), each datanode runs only map tasks that access local data; we normalize runtimes to this baseline. In the second scenario (*Remote*), each datanode runs only map tasks that access remote data from the node with the next higher ID (modulo the number of nodes); hence, all nodes act concurrently as both readers and servers. In the final scenario (*Asymmetric*), two datanodes (the *Readers*) access data located

on the third (the *Server*), which additionally runs its own map tasks that access data locally. In all scenarios, each map task accesses distinct files, and file system caches are initially empty.

Table A.1 shows the normalized runtimes for each combination of I/O scenario and microbenchmark. Runtimes are normalized to the *Local* case for each microbenchmark. For the CPU-intensive microbenchmark, we see in the *Remote* scenario that remote access results in a 10% performance penalty, considerably larger than the 1.6% penalty of traversing the network to access an idle HDFS server. In the *Asymmetric* scenario, we see a further interesting effect: the slowdown on the *Reader* nodes (which serve no remote HDFS requests) shrinks to only the network-traversal penalty, while the server node sees a disproportionate slowdown of 13%. For the I/O-intensive microbenchmark, the penalties are magnified. In the *Remote* scenario, the slowdown grows to 30%. The *Asymmetric* scenario sees even larger slowdowns. Overall, we observe that map tasks running on the *Server* node see a disproportionate slowdown (i.e., they are not receiving a fair share of disk bandwidth).

To explain the behavior observed above, we configure four nodes in the *Asymmetric* mode, and vary the number of nodes reading from the *Server* node from one to four (one of these readers is always present on the *Server*). Figure A.2 and Figure A.3 show the breakdown of runtime spent in various CPU states on the *Server* and a *Reader* for the CPU-intensive and I/O-intensive microbenchmarks respectively. As the number of readers accessing the *Server's* disk goes up, the *Server* spends increasing amounts of time in the CPU I/O-wait state. In other words, the CPU is stalled on I/O and starved for data to process. (To verify that the *Server's* CPU is stalled on I/O and does not have any other tasks to process, we run a separate CPU-intensive task on the *Server* and observe that the I/O wait gets transformed into user time.)

The increased runtime and I/O wait on the *Server* stems from inherent inefficiencies in HDFS. The HDFS client implementation is highly serialized for data reads [94]. In the absence of pipelining to overlap computation with I/O, the application waits for data transfer to complete before processing commences. Additionally, each datanode (the *Server* in this case) spawns one thread per client to manage disk access and network communication, with all threads accessing the disk concurrently; these threads consume valuable resources. Shafer et al. [94] observe that as the number of concurrent readers in HDFS increases from one to four, the aggregate read bandwidth reduces by 21% with *UFS2*, and by 42% with *ext4*. Further, the average run-length-before-seeking drops from over 4MB to well under 200kB. Since most I/O schedulers are designed for general-purpose workloads and attempt to provide fairness at a fine-grained granularity of a few hundred kilobytes, the disk is forced to switch between distinct data streams in the presence of multiple concurrent readers, thereby lowering aggregate bandwidth.

Our experimental observations show that for the CPU-intensive microbenchmark, the *Server* node experiences almost an 8x increase in disk-queue size when going from 2 readers to 4 readers—from 0.42 to 3.12. Further, the period between request issue and data reception on the *Server* quadruples—from 2.2ms to 8.9ms, thereby accounting for the I/O wait observed. Disk utilization is also observed to increase from 31% to 94%.

For the I/O-intensive microbenchmark, the *Server* experiences over a 2x increase in disk-queue size when going from 2 readers to 4 readers—from 2.2 to 4.8. As with the CPU-intensive microbenchmark, the period between request issue and data reception quadruples—from 2.2ms to 8.9ms. Disk utilization is seen to increase from about 92% to about 98%.

We note from Figure A.2 and Figure A.3 that the I/O wait seen on *Server* does not translate into an equivalent amount of idle time on the *Readers*. For the CPU-

intensive microbenchmark, a majority of the delay associated with I/O wait on the *Server* node is masked by computation on the *Readers*. However, this masking is not observed on the *Readers* for the I/O-intensive microbenchmark, and is translated into CPU-idle time. As a consequence, for the I/O-intensive microbenchmark, the *Readers* see large increases in runtime. Overall, we note that the slowdown in the *Remote* scenario can primarily be attributed to I/O stalls accounting for a larger fraction of the runtime. And the even larger slowdowns in the *Asymmetric* case can further be attributed to the saturation of the available bandwidth on the (single) disk serving all three nodes.

The central conclusion from these results is that *serving remote HDFS requests disproportionately delays locally-running map tasks*. The eventual source of this performance phenomenon can be traced to the interaction of the threading model of HDFS and the default I/O scheduler in recent versions of Linux, the "deadline" scheduler [8]. The deadline scheduler imposes a deadline on all I/O operations to ensure that no request gets starved, and aggressively reorders requests to ensure improvement in I/O performance. Because of these deadlines, a sleeping HDFS thread that receives an I/O completion (either a new request arriving over the network or data being returned from the disk) will preempt a map task nearly immediately to finish the I/O. In contrast, map tasks that issue an I/O request, block on the I/O, thereby freeing a core to allow HDFS to execute without disturbing other map tasks.

We can eliminate the unfairness caused by the deadline scheduler by instead switching Linux to use a *completely fair scheduler* [5]. However, we find that, when doing so, overall performance suffers: in the 4-reader case, while running the CPU-intensive microbenchmark, the *Server* node slows down by an additional 6%, while the *Reader* nodes slow down by 16%, to only about 6% faster than the *Server*. The difference in runtimes between the *Server* and the *Readers* with the completely fair scheduler is consistent with the extra resources the *Server* has to sacrifice in order

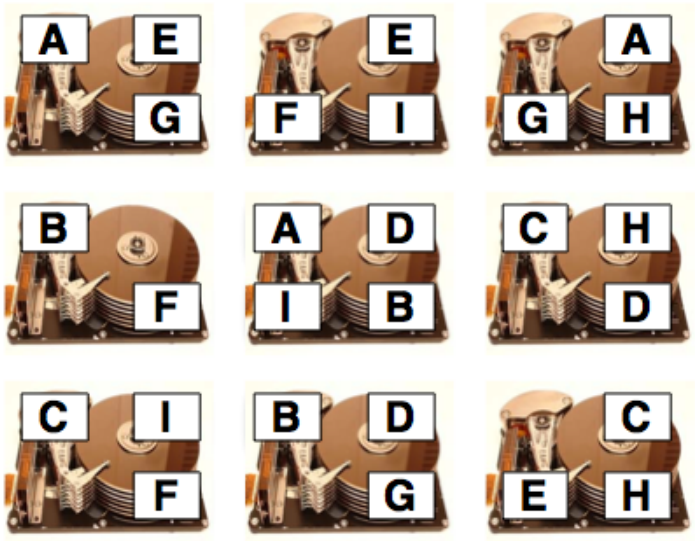


Figure A.4: Random data placement in a Hadoop cluster. Each block is replicated three times across the cluster. Each disk represents a node in the cluster.

to service the remote requests. We also note that with the completely fair scheduler, each *Reader* sees an increase in idle time that is consistent with the I/O wait time seen on the *Server*.

The aforementioned experiments demonstrate that nodes servicing remote read requests are slowed down more significantly than nodes making these requests. Overall, our observations indicate that reducing remote accesses can improve performance even in scenarios where the network is far from saturated, and not just in highly loaded clusters with multi-job loads, busy networks, and file fragmentation associated with multiple simultaneous writers. In the next sections, we explore intelligent data placement as one avenue to reduce remote accesses.

A.4 Data Placement

In this section, we propose *partitioned data placement* as an approach to reduce remote accesses. We first describe Hadoop's default *random data placement* as a baseline for comparison.

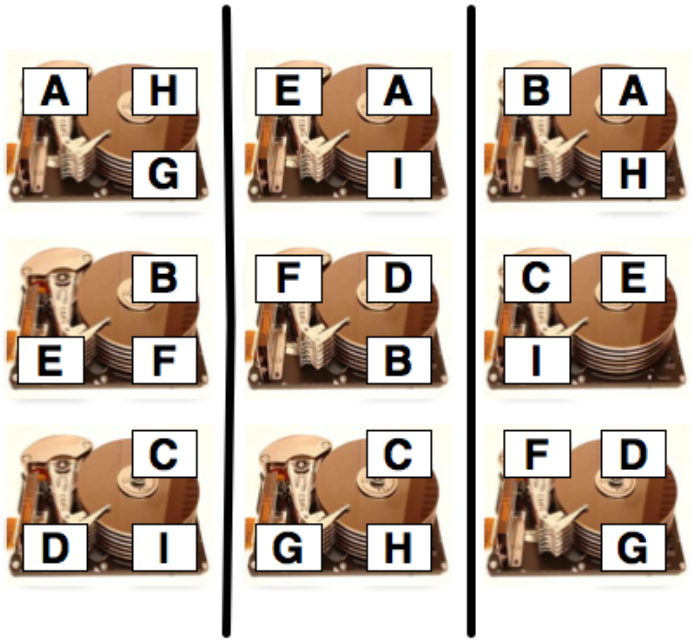


Figure A.5: Partitioned data placement. The cluster is partitioned into three sub-clusters (the vertical lines demarcate the partitions). Each partition contains one replica of each data block. Overall, the cluster still contains three replicas of each block.

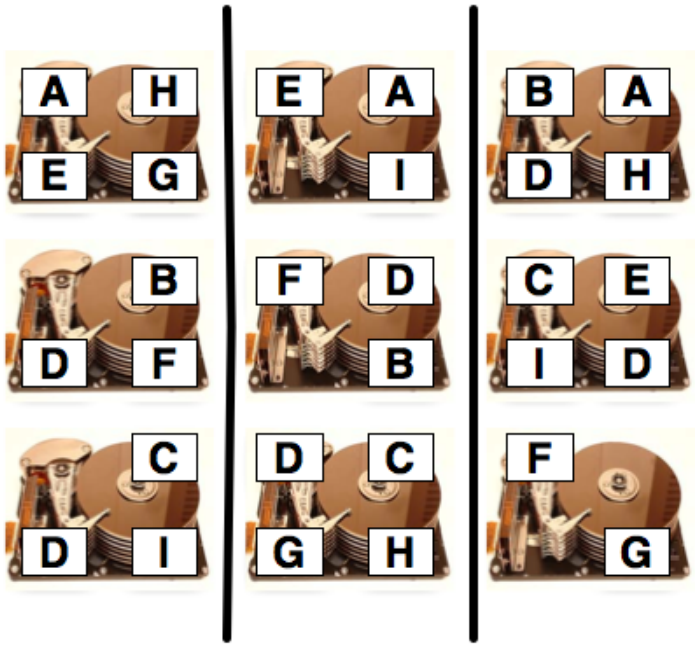


Figure A.6: Selective replication of popular data blocks within partitions. Block D is a popular block and is replicated twice within each partition. Non-popular blocks are replicated once per partition.

A.4.1 Random Data Placement

Under random data placement, blocks are randomly distributed across nodes. Figure A.4 illustrates this data placement policy. Random data placement is a sim-

plication of Hadoop's default data placement scheme (Hadoop's default locality optimization seeks to collocate two of three data replicas within a single rack; rack locality is irrelevant when network distance has no performance impact, as in the scenarios we study). Although random data placement maintains data availability in the presence of disk failures, it can be inefficient from a performance perspective, especially when a job is restricted from executing on some machines within a cluster.

The drawbacks of random data placement under restricted task scheduling are illustrated using Figure A.4. Consider a cluster wherein only the three leftmost nodes may service a job (e.g., because other nodes are reserved). We define the *allocated cluster fraction* as the fraction of the cluster available to a job. In the example in the figure, the allocated cluster fraction is 3/9 or 33%. Under this scenario, it is clear that blocks D and H are not locally accessible within the available nodes, and hence must incur remote accesses.

A.4.2 Partitioned Data Placement

To reduce remote accesses when a job is restricted to a subset of the cluster, we propose *partitioned data placement*. In partitioned data placement, a cluster is divided into N partitions, with N being equal to the replication factor. Each partition contains exactly one replica of every data block; overall, the entire cluster contains N replicas of each block. Blocks are randomly assigned to nodes within a particular partition. Figure A.5 shows an example of this data placement policy for a cluster with three partitions.

When a job can be assigned an entire partition (or more), all data can be accessed locally. Hence, if the number of active jobs is less than the number of partitions, remote accesses will be rare (arising only due to load imbalance at the tail of the job). However, as subsequently demonstrated, partitioned data placement reduces remote accesses even for jobs that execute on a smaller number of nodes. The data

placement restrictions implied by partitioning reduce the probability of duplicate data blocks in a randomly selected subset of nodes, thus increasing the diversity of blocks available locally. To first order, partitioning does not sacrifice data availability since the overall number of replicas remains unchanged.

A.4.3 Replication

In addition to data placement, creating more replicas of a block can improve the probability the block will be available locally within a random subset of the cluster. But, adding extra replicas comes at a high storage cost (e.g., one extra replica per block implies a 33% storage increase). With knowledge of block access patterns, only the most popular (frequently accessed) blocks can be replicated, reducing replication costs while maintaining much of the benefit [16]. We explore the impact of selective replication similar to that proposed by Ananthanarayanan and co-authors in combination with partitioned placement. Figure A.6 illustrates the concept of *selective replication* in partitioned clusters; a popular block (in the example, block D), is replicated at a higher replication factor within each partition.

A.5 Results

We contrast random and partitioned data placement through a combination of simulation of the Hadoop scheduling algorithm and validation experiments on a small scale cluster. We use simulation to allow rapid exploration of the impacts of data placement policies on clusters much larger than the real clusters to which we have access.

We contrast random and partitioned data placement policies under two scenarios: *unconstrained* and *restricted* allocation. Under unconstrained allocation, the tasks that constitute a job may be scheduled on any node in the cluster. A job is granted an allocation that limits the maximum number of simultaneously executing tasks;

however, there is no restriction on the nodes that run these tasks. When multiple jobs execute concurrently, with a suitable scheduling discipline (e.g., round robin), over time, a job's tasks will visit a time-varying subset of nodes. Because the job migrates across the cluster over time, a large fraction of data blocks can be accessed locally at some point during the job's execution, even when the job is granted a small (simultaneous) allocation.

Under restricted allocation, we assign a job to execute within only a fixed (but randomly selected) subset of nodes. The restricted allocation scenario is representative of a variety of reasons that Hadoop jobs might be precluded from executing on some nodes. The simplest example is when nodes are explicitly reserved for certain jobs or users; however, restricted allocation might also arise because nodes are rendered unavailable due to long-running tasks, job priorities, power managment, or the job scheduling discipline.

A.5.1 Simulation Methodology

We model large-scale Hadoop clusters by extending the *BigHouse* data center simulation framework [70,71]. BigHouse is a parallel, stochastic discrete time simulation framework that represents datacenters via generalized queuing models driven by empirically-observed arrival and service distributions. Our simulation assigns MapReduce tasks lengths drawn from a service time distribution and assigns tasks to nodes in a manner similar to Hadoop's scheduler. When a task slot becomes available on a node, the scheduler checks to see if a local block required by the job is available on that node. If so, the newly created task is assigned this local block. If there are no local blocks that are awaiting processing, the scheduler picks a pending block from the closest remote node and assigns it to the new task.

We simulate a 60 node cluster that stores 10 files with up to 1200 blocks each. Block popularity is drawn from a Zipf distribution. Task execution times are drawn

either from an exponential distribution with rate parameter λ=1, or a gamma distribution with shape parameter k=2 and scale parameter θ=3. The baseline replication factor for both random and partitioned data placement is set to Hadoop's default of three. For partitioned data placement, we assume three partitions, i.e., one replica per partition.

A.5.1.1 Unconstrained Allocation

We first consider unconstrained allocation, wherein a task may be assigned to any node. Under this scenario, provided job lengths are reasonably balanced, the tasks constituting a job will be able to migrate across the cluster over time to visit each data block such that they can access it locally. We assume that jobs are sliced into tasks at the granularity of disk blocks; finer granularity can result in higher overhead from task startup and shutdown [94], while coarser granularity may restrict task scheduling flexibility. Recent research has suggested that relatively large data block sizes improve performance [78].

Under unconstrained allocation, both partitioned and random data placement perform similarly: approximately 98% of data blocks can be accessed locally (98.2% local accesses for partitioned, and 97.9% for random). On average, a job incurs its first remote access only after over 85% of tasks have been processed, i.e., towards the tail end of the job. Since tasks are scheduled on a per block basis, a job gets the opportunity to visit multiple nodes through the cluster, thereby increasing the likelihood that tasks will be scheduled on nodes that contain locally accessible blocks. The key take-away is that data placement has little impact when it is possible for a job to traverse the cluster and visit nearly all nodes over time. Even under a naive round-robin scheduler across competing jobs, almost 98% of accesses can be completed locally. Hence, under these circumstances, neither partitioned placement, nor other techniques (e.g., delay scheduling) are necessary.

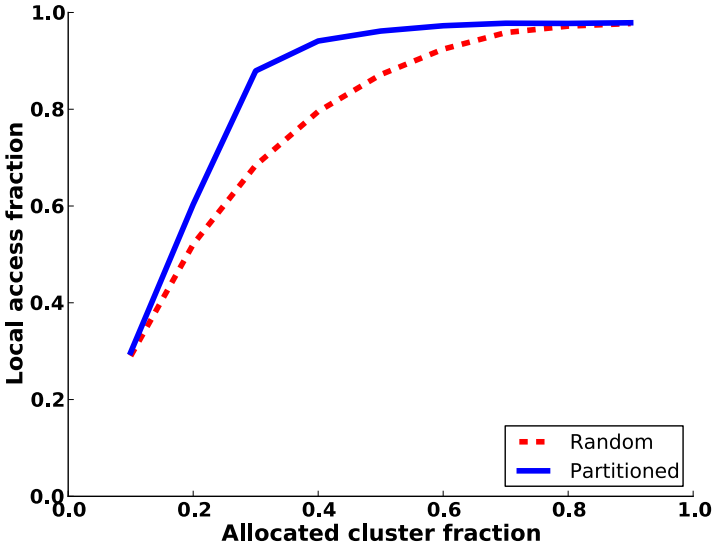


Figure A.7: Local access fraction as a function of cluster availability. Partitioned data placement dominates random data placement, especially for low cluster allocations.

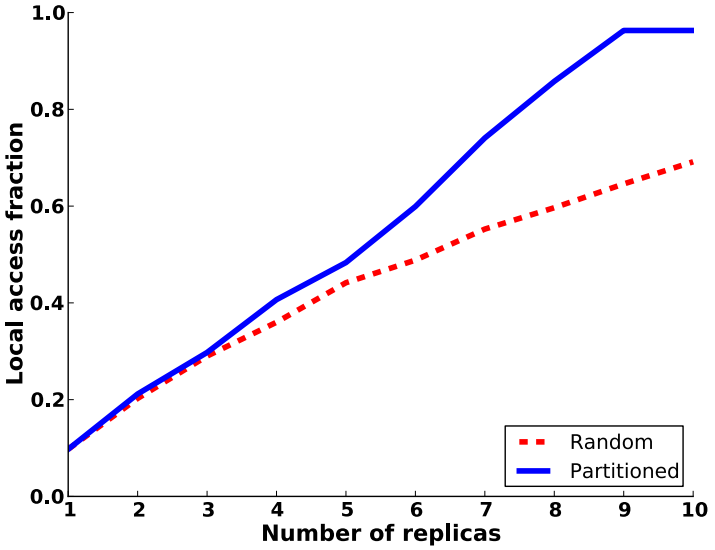


Figure A.8: Local access fraction versus number of replicas for 10% cluster availability. Adding more replicas increases the number of local accesses.

A.5.1.2 Restricted Allocation

Under restricted allocation scenarios, partitioned data placement can be effective in reducing remote accesses. Figure A.7 shows the fraction of blocks accessed locally for both random and partitioned data placement as a function of the fraction of the cluster allocated to a job (i.e., the fraction of nodes the job may visit), which we

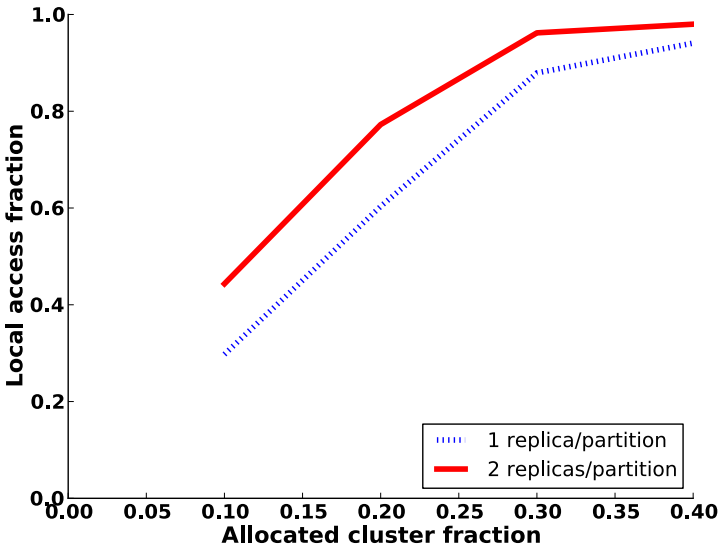


Figure A.9: Effect of selective replication of popular blocks on local access fraction. Selectively replicating popular blocks increases the number of local accesses with low storage overhead.

vary from 10% to 90%. Under random data placement, a job must be allocated nearly 80% of the cluster to avoid remote accesses. Stated another way, if more than 20% of a cluster is reserved and may not be used by a job, then the job will suffer an increased rate of remote data accesses. Hence, even relatively small allocations of much less than half the cluster can substantially impact local versus remote access rates under Hadoop's default placement scheme. Partitioned data placement substantially improves the fraction of local accesses for cluster allocations from 10% to 80%. Once the allocation fraction is larger than a partition (33% for the 3-partition cluster in this experiment), the local access fraction rapidly approaches 100%.

A.5.1.3 Replication

Next we study the effect of adding more replicas on local access rates under restricted allocations. We first consider the simple case where all blocks are replicated, ignoring popularity. Figure A.8 shows a graph of the local access fraction as a function

of the replication factor for 10% cluster allocation, sweeping the number of replicas from one (no replication) to ten. For the partitioned scheme, the number of replicas also corresponds to the number of partitions. As the number of replicas increases, the advantage of partitioned data placement over random data placement grows.

We next consider only selective replication of blocks that exceed a popularity threshold. To model this scenario, we increase the popularity of two files (corresponding to 22% of all blocks) by a factor drawn from a Zipf distribution relative to the remaining blocks. We provision twice as many replicas of blocks in the popular files. These additional replicas are again distributed across the partitions, such that there are now two replicas of the popular blocks per partition. Figure A.9 shows the effect of such selective replication, relative to a baseline of only a single replica per partition for all blocks. Selective replication increases the rate of local accesses by 10-20% over the relevant range of cluster allocations (above 40% allocation, nearly all accesses are local without additional replicas). Of course, selective replication results in a far lower storage overhead than naive replication of all blocks.

A.5.2 Validation on a Real Hadoop Cluster

We validate our simulation results via a small-scale test on a 10-node Hadoop cluster (nine datanodes and one dedicated namenode). We contrast Hadoop's default random data placement against partitioned data placement. In both cases, we distribute a 9.5 GB file across nodes in 64 MB blocks with a replication factor of three. We emulate restricted allocation by marking six randomly selected nodes as unavailable to execute tasks, corresponding to a 33% cluster allocation. We report the fraction of map tasks that access remote data.

Figure A.10 shows the percentage of remote accesses for each data placement policy. Over a series of ten trials, random data placement requires a maximum of 48.7% of map tasks to access remote data. On average, random data placement results in

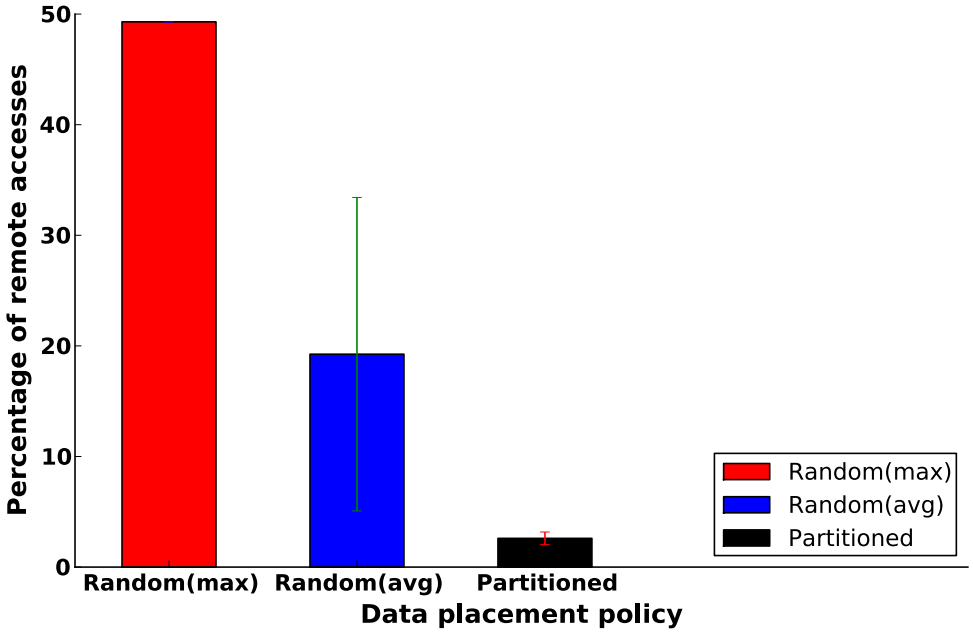


Figure A.10: Percentage of remote accesses as a function of data placement policy on a real Hadoop cluster. On average, partitioned data placement reduces the number of remote accesses by over 86% for a 33% cluster allocation.

19.2% remote accesses. Partitioned data placement reduces the average number of remote accesses to 2.6%, an 86% reduction compared to random data placement. In summary, our results show that the partitioned data placement policy reduces remote accessess relative to random data placement under restricted allocation scenarios. Additionally, increasing replication factors can further reduce remote accesses, especially for small allocations.

A.6 Conclusion

MapReduce, in particular Hadoop, is a popular framework for the distributed processing of large datasets on clusters of networked and relatively inexpensive servers. Whereas Hadoop clusters are highly scalable and ensure data availability in the face of server failures, their efficiency is poor. We demonstrate that remote accesses can cause significant performance degradation, even under unsaturated network conditions, due

to the disproportionate interference effects on nodes servicing remote HDFS requests. We study an intelligent data placement policy we call *partitioned data placement* as an avenue to reduce the number of remote data accesses, and the associated performance degradation, when task placement is restricted due to reasons such as long-running jobs or other reservations. During the course of our investigation, we find that partitioned data placement can reduce the number of remote data accesses by as much as 86% when a job is restricted to execute on only one-third of the nodes in a cluster.

APPENDIX B

PicoServer Revisited: On the Profitability of Eliminating Intermediate Cache Levels

The confluence of 3D stacking, emerging dense memory technologies, and low-voltage throughput-oriented many-core processors has sparked interest in single-chip servers as building blocks for scalable data-centric system design. These chips encapsulate an entire memory hierarchy within a 3D-stacked multi-die package. Stacking alters key assumptions of conventional hierarchy design, drastically increasing cross-layer bandwidth and reducing the latency ratio between successive layers. Hence, prior work, specifically PicoServer, suggests flattening the hierarchy, eliding intermediate caches that otherwise lengthen the critical path between L1 and stacked memory.

Although PicoServer argues for a flattened memory hierarchy for web serving workloads, it remains unclear if its conclusions hold more generally—particularly when considering metrics besides access latency—and more recent studies have often included intermediate caches. In this chapter, we investigate the bandwidth, latency, *and* energy filtering *afforded by an L2 cache. For data-centric scientific and server applications, we conclude: (1) 3D stacking provides copious bandwidth, hence L2 bandwidth filtering is moot; (2) although a flat hierarchy is optimal for access patterns with poor temporal locality, some workloads benefit from access latency reduction afforded*

by L2, an effect magnified by latency-intolerant in-order cores and for memory technologies with asymmetric read and write latencies; and (3) intermediate caches are rarely desirable from an energy perspective, and only if the cache is optimized for low leakage.

B.1 Introduction

The confluence of 3D stacking of logic and memory, emerging dense memory technologies, and low-voltage throughput-oriented many-core processors has sparked interest in single-chip servers as building blocks for scalable data-centric system design [87]. These single-chip servers encapsulate an entire memory hierarchy within a 3D-stacked multi-die package. However, 3D stacking alters key assumptions of conventional memory hierarchy design. For example, cross-layer bandwidth is drastically increased, and the latency ratio between successive layers is reduced [63]. In light of these inflections, prior work has suggested flattening the memory hierarchy, eliding intermediate caches that otherwise lengthen the critical path between L1 and stacked memory. For example, *PicoServer* proposes removing the L2 and re-allocating the area to additional cores on the chip [52].

Although the *PicoServer* study demonstrates both performance and energy advantages of a flattened memory hierarchy for web serving workloads, many recent studies of 3D stacked systems nevertheless have included intermediate cache levels between L1 and a stacked memory [21, 81, 108] or stacked last-level-cache [63, 64]. The lack of clarity regarding the utility of intermediate caches stems partially from uncertainty about the ultimate performance and energy characteristics of 3D stacked devices. For example, Loh describes how the physical design of DRAM can substantially alter its performance [63]. Liu and co-authors describe alternative 3D stacked DRAM-based design configurations that have up to a 2x difference in performance [62]. Similar uncertainty exists with other potential memory technologies; a survey by Lee and

co-authors indicates that published phase-change memory (PCM) performance estimates vary by up to 1.45x [58]. As another example of this uncertainty, the latency values cited by Li and co-authors [60] and Venkataraman and co-authors [107] vary by up to 2.5x for PCM.

In this chapter, we revisit the conclusions of the *PicoServer* study to identify the inflection points where an intermediate cache level becomes profitable. Whereas *PicoServer* focuses primarily on the latency impact of intermediate caches versus a flat hierarchy, it has also been observed that caches can act as bandwidth filters [43], and, more recently, as energy filters [54, 103] for lower hierarchy levels. Hence, we consider the utility of an intermediate cache from all three perspectives, *bandwidth*, *latency*, and *energy filtering*. In addition, we investigate design tradeoffs when memory read and write latencies are asymmetric, as is common for emerging memory technologies. We find memory parameter inflection points for a suite of eight applications, four from the SPEC CPU2006 suite (two each with good and poor relative temporal locality), and four from emerging data-centric workloads that motivate single-chip server design [87].

We make the following contributions:

- We characterize the bandwidth requirements of our workload suite, both with and without an L2 cache, and demonstrate that bandwidth filtering is moot given the large bandwidth provided by 3D-stacked memories. For these benchmarks, the highest bandwidth requirement without an L2 is approximately 29 GB/s with eight high-end cores, a requirement easily fulfilled by two DDR3 channels, and at least 5x below the projected bandwidth capability of 3D stacked memory.

- We show that although a flat hierarchy is optimal for access patterns with poor temporal locality, some scientific and data-centric workloads benefit from access latency reduction provided by intermediate cache levels, an effect that

is magnified by latency-intolerant in-order cores. We show that half the data-centric workloads do not require an L2 unless the stacked memory access latency is well over 12x the L2 access latency.

- We find that an L2 cache is rarely desirable from an energy perspective for data-centric workloads, even when the ratio of main memory to L2 dynamic access energy is high. Importantly, we find that it is critical for the L2 to have low leakage power for it to provide any energy gain, as L2 leakage can rapidly offset any energy savings achieved from conserving dynamic power.

The remainder of this chapter is organized as follows: Section B.2 discusses related work, and Section B.3 describes our experimental methodology. We consider the bandwidth, latency, and energy filtering impact of intermediate caches in Section B.4, Section B.5, and Section B.6 respectively. In Section B.7, we conclude.

B.2 Related Work

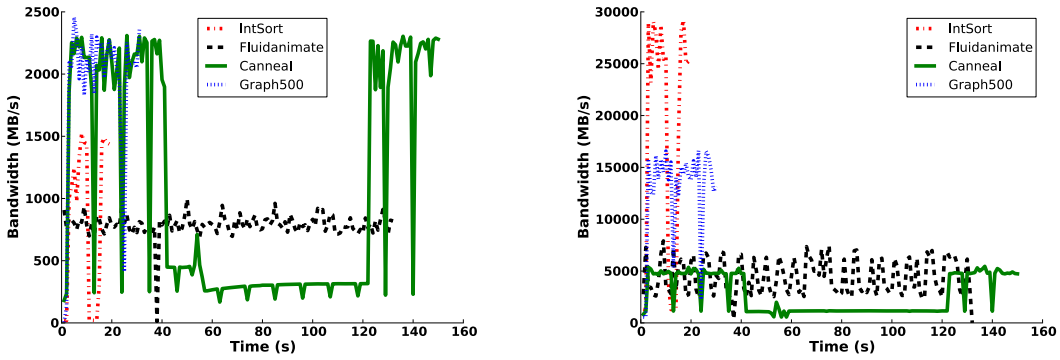
Our study is inspired by the disparity between the recommendation of *PicoServer* [52] to flatten the memory hierarchy and other more recent studies of 3D stacked memory hierarchies [21, 63, 64, 81, 108] which retain intermediate caches. *PicoServer* proposes an architecture where multiple simple, throughput-optimized, in-order processing cores are 3D-stacked with multiple DRAM dies that comprise main memory. *PicoServer* flattens the memory hierarchy by removing the L2 and re-allocating the area for additional cores. For the web-tier applications studied, the L2 provides little benefit, and the throughput advantage of re-allocating the area for additional cores is greater.

The *PicoServer* study focuses primarily on the latency impact of an L2 cache, observing that it slows the critical access path to the main memory. Bandwidth and energy filtering have been suggested as further motivations beyond latency hiding for

intermediate cache levels. These effects were not a focus of the *PicoServer* study. In this study, we consider whether these concerns might change the answer on whether or not to flatten the hierarcy. We consider data-centric and scientific workloads with larger memory footprints than the more network-centric benchmarks for which *PicoServer* was designed (the higher percentage of DMA and I/O accesses in web serving reduces L2 effectiveness). Additionally, we consider the impact of asymmetric read and write latencies, a characteristic of many non-volaltile memories like PCM [83, 84] and STT-RAM [88].

Our intent is to provide guidance on memory hierarchy design for single-chip servers, such as the *Nanostores* [87] proposal by HP Labs. *Nanostores* are single-chip servers with 3D (or 2.5D) stacked logic and dense non-volatile memory. *Nanostores* are further distinguished by their use of low-voltage throughput-oriented many-core processors. Overall, *Nanostores* have been proposed as viable building blocks for scalable data-centric systems.

Since the advent of 3D stacking with through-silicon vias (TSVs), there have been a number of studies to examine 3D stacked memory system design. Loh [63] describes a 3D memory organization that increases "memory level parallelism through a streamlined arrangement of L2 cache banks, MSHRs, memory controllers and the memory arrays themselves". His work improves page-level parallelism by increasing row buffer cache capacity, thereby allowing for a larger set of open memory pages. Loh also uses a data structure called the Vector Bloom Filter to reduce the number of probings required to determine hits and misses in the L2. Loh and Hill [64] explore efficient on-chip DRAM-based caching using conventional cacheline sizes. Their technique schedules tag and data accesses as compound accesses such that the data accesses are always row buffer hits, thereby making hits faster than just storing tags in the DRAM. Their technique also makes misses faster by using MissMaps to reduce stacked-DRAM accesses on misses. Woo et al. [108] propose SMART-3D, a



(a) Main Memory Bandwidth

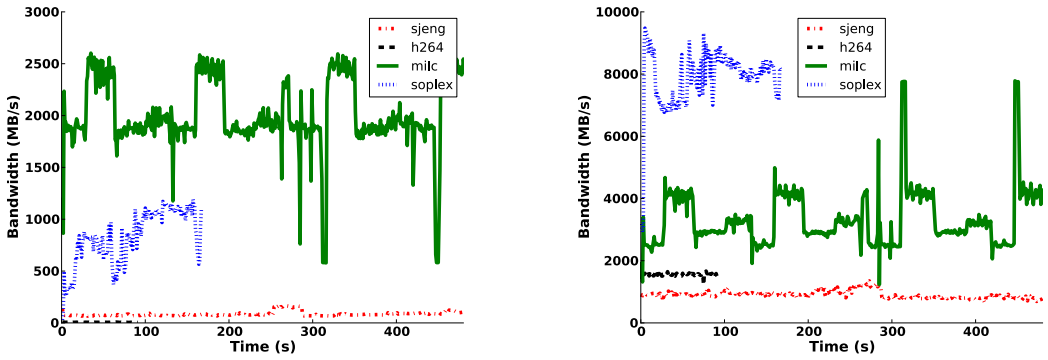
(b) Inter-cache Bandwidth

Figure B.1: Memory and inter-cache bandwidth utilization for non-SPEC benchmarks. (a) Bandwidth between the L2 and main memory for the non-SPEC benchmarks: The highest bandwidth requirement is under 2,500 MB/s. (b) Bandwidth between the L1 and L2 caches for the non-SPEC benchmarks: This is the bandwidth between the L1 and main memory in the absence of an L2. The highest bandwidth requirement is about 29,000 MB/s.

new 3D-stacked memory architecture with a vertical L2 fetch and writeback network using a large array of TSVs. SMART-3D leverages TSV bandwidth to hide latency behind very large data transfers. All of these designs include an intermediate cache level between the L1 and stacked memory hierarchy level, and do not explore flattening the memory hierarchy in the 3D-stacked context.

Several groups have created prototypes of 3D stacked memory systems [41,53,110]. Most academic prototypes have omitted intermediate caches, largely due to area constraints rather than specific design optimization objectives.

Recent work on single-chip servers draws inspiration from earlier work on the *RAMpage* memory hierarchy proposed by Machanick and Salverda [67]. The *RAMpage* memory hierarchy uses DRAM as a paging device, and moves main memory up one level to the lowest level of SRAM. The main motivation for the *RAMpage* hierarchy is to reduce the cost of DRAM references. While RAMpage does not eliminate the L2, it realizes a memory hierarchy that provides similar trade-offs to that proposed in *PicoServer*.



(a) Main Memory Bandwidth

(b) Inter-cache Bandwidth

Figure B.2: Memory and inter-cache bandwidth utilization for SPEC benchmarks. (a) Bandwidth between the L2 and main memory for the SPEC benchmarks: The highest bandwidth requirement is under 2,600 MB/s. (b) Bandwidth between the L1 and L2 caches for the SPEC benchmarks: This is the bandwidth between the L1 and main memory in the absence of an L2. The highest bandwidth requirement is about 9,500 MB/s.

B.3 Methodology

We follow a trace-based methodology to evaluate the profiltability of eliminating intermediate cache levels; this methodology allows us to rapidly study workloads with large footprints. We collect memory traces with *PIN* [65] on a server system with eight 1.9 GHz Intel Xeon cores, 32 kB private L1 caches, an 8 MB 16-way set-associative shared L2 cache, and 16 GB of main memory. The traces contain information on the address, program counter, size, and type (read or write) of each memory access, and the number of non-memory instructions between consecutive memory accesses. We leverage performance counters to measure the peak L2 and memory bandwidth requirements of each workload on this server system using the Linux *Perf* performance analysis tool for use in our bandwidth filtering study.

To assess the impact of various memory subsystem designs, we replay L1 access traces through the *gem5* architectural simulator [20]. The trace replay emulates 1 GHz single-issue in-order cores. We replay one billion memory accesses per core. We collect statistics for the number of accesses to the L2 and stacked main memory; these statistics allow us to calculate the average L1 miss latency and L2/memory

energies under various assumptions. When the L2 is enabled, we simulate an 8 MB 16-way associative shared L2 (i.e., the same L2 cache organization as in our actual x86 server).

As our goal is to identify technology inflection points, we do not select specific latency or static/dynamic energy-per-access values to represent particular memory technologies. Rather, we fix L2 accesses at 10 cycles and 1 nJ per access (which are in the typical range for current technology), and then vary the ratio of stacked memory read latency, write latency, and dynamic energy relative to these values. We also explore a wide range of L2 leakage assumptions.

We focus our study on a suite of four scientific and data-centric server applications. We include *Canneal* and *Fluidanimate* from the PARSEC Benchmark Suite [19] (using the native inputs), *Integer Sort* (*IntSort*) from the NAS Parallel Benchmark Suite, and *Graph500*. For comparison, we also study four SPEC CPU2006 benchmarks with differing (but known) locality characteristics. From SPEC, we include: *h264* and *sjeng*, which have comparitively small working sets, and *milc* and *soplex*, which have large working sets [9]. The SPEC applications are single-threaded and hence are recorded and replayed using only one core.

B.4 Bandwidth Filtering

Nearly three decades ago, Goodman first pointed out that caches can be used as bandwidth filters [43]. We first consider whether application bandwidth needs create a requirement for an intermediate cache between L1 and stacked memory. Using *Perf*, we assess the L2 and memory bandwidth requirements of each of our applications on an 8-core x86 server at one-second granularity. The L2 bandwidth consumption measured on our test system would correspond to the load offered to a stacked main memory if the L2 were elided.

Figure B.1 and Figure B.2 show the bandwidth utilization between the L2 cache and main memory, and between the L1 and L2 caches for the non-SPEC and SPEC benchmarks respectively. The SPEC results reflect the demands of a single Xeon-class core, while the non-SPEC results show the aggregate bandwidth demand of eight cores. As shown in the figures, the bandwidth between the L2 and main memory in the presence of an L2 is under 2,600 MB/s for all benchmarks. Without an L2 cache, the bandwidth required for all benchmarks aside from *IntSort* is under the peak bandwidth of 17 GB/s that one channel of DDR3 can provide [13]. *IntSort* requires a peak bandwidth of approximately 29 GB/s, which can be satisfied by two DDR3 channels. More importantly, this bandwidth requirement is several factors below the projected 192 GB/s available with 3D-stacked memory as projected by Woo et al. [108].

The Xeon cores used in our experiment are much more aggressive than those envisioned for *Nanostores*. Nevertheless, even for aggressive cores, it is clear that 3D-stacked memory provides ample bandwidth, and bandwidth filtering is moot for performance. Hence, we find that *PicoServer's* recommendation to flatten the hierarchy is viable from a bandwidth perspective. Note however, that bandwidth filtering may be valuable when stacking non-DRAM memory technologies like PCM that may suffer from limited endurance [58, 84]. As seen from Figure B.1, eliding the L2 can increase traffic to the stacked memory by up to 20x.

B.5 Latency

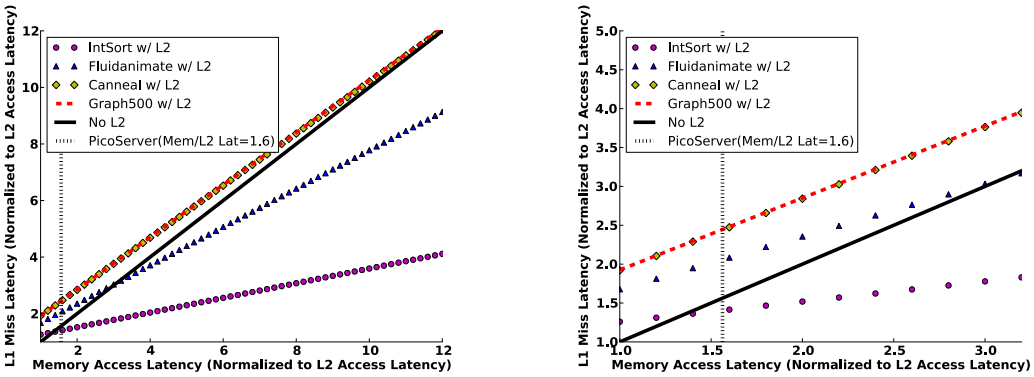
We next consider the latency impact of intermediate caches. We evaluate two scenarios. In the first scenario, we assume that the memory read and write latencies are equal, representing stacked DRAM. In the second scenario, we consider the impact of scaling write latency relative to read latency, as slower writes are common in emerging memory technologies. In each scenario, we contrast systems with and

without an L2. When enabled, we fix the L2 latency at 10 cycles and assume that L2 and memory are accessed in series. We vary the ratio of stacked memory to L2 access latency. We use average L1 miss latency as our evaluation metric.

B.5.1 Symmetric Read and Write Latencies

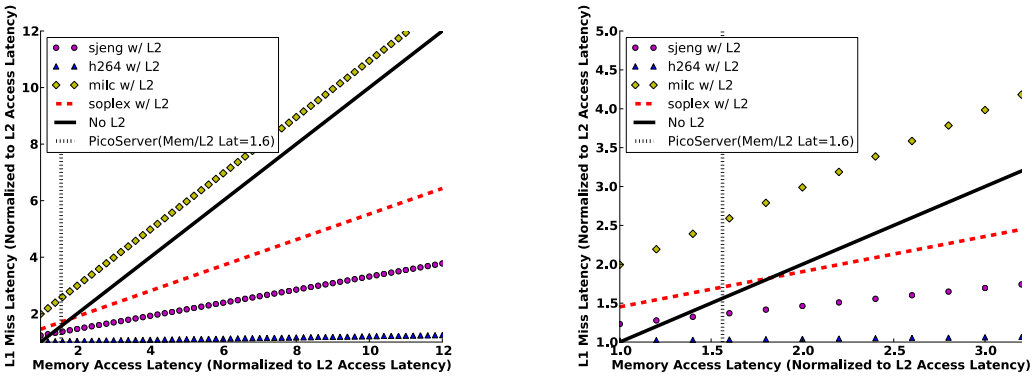
Figure B.3 and Figure B.4 show the average L1 miss latency for non-SPEC and SPEC benchmarks, respectively. The x-axes on the graphs represent the memory access latency normalized to the L2 access latency. The solid black line on each graph with a slope of one displays the average L1 miss latency in the absence of an L2, in which case all L1 misses incur a main memory access (note that latency is therefore the same for all benchmarks). The remaining lines show the L1 miss latency for each benchmark; their origin and slope depends on the hit rate achieved by the 8MB L2. The dotted vertical line indicates the latency ratio assumed in the *PicoServer* study. Of particular interest is the point where each workload's line intersects the "No L2" line; this latency ratio represents the inflection point where an L2 becomes profitable. That is, for stacked memory latencies greater than this inflection point, the time saved on L2 hits outweighs the time lost on misses. This inflection point can be calculated directly given an L2 miss rate using the well-known average memory access latency formulas described by Hennessey and Patterson [45].

Among the non-SPEC benchmarks, only *IntSort* has a sufficient L2 hit rate to warrant an intermediate 8MB L2 for the likely range of stacked DRAM access latency ratios (in the vicinity of the *PicoServer* assumption of 1.6x). It breaks even at a ratio of 1.4x. *Fluidanimate* might derive some benefit if the stacked DRAM is quite distant from the L2, for example, if it is implemented with a slower memory technology. *Canneal* and *Graph500* have exceedingly poor temporal locality in the L2, and hardly benefit even when stacked memory has an access latency similar to off-chip DRAM.



(a) Average L1 Miss Latency (b) Average L1 Miss Latency

Figure B.3: Average L1 miss latency for the non-SPEC benchmarks given symmetric memory read and write latencies. Memory latency is normalized to L2 access latency. As memory latency increases, *IntSort* and *Fluidanimate* benefit from an L2 for relatively memory low latencies. Performance of both *Canneal* and *Graph500* is similar with and without an L2 (a) *IntSort* and *Fluidanimate* benefit from having an L2 beyond a memory latency of 1.4x and 3.1x respectively. *Canneal* and *Graph500* only show improvement beyond a memory latency of 12.7x and 13.4x respectively. (b) Zoomed in graph to show detail.



(a) Average L1 Miss Latency (b) Average L1 Miss Latency

Figure B.4: Average L1 miss latency for the SPEC benchmarks given symmetric memory read and write latencies. Memory latency is normalized to L2 access latency. *sjeng*, *h264*, and *soplex* benefit from an L2. (a) *sjeng* and *soplex* benefit from having an L2 beyond a memory latency of 1.3x and 1.8x respectively. *h264* always benefits from an L2. *milc* does not benefit from an L2. (b) Zoomed in graph to show detail.

Our conclusion is that the *PicoServer* recommendation to flatten the hierarchy will typically hold for this application class.

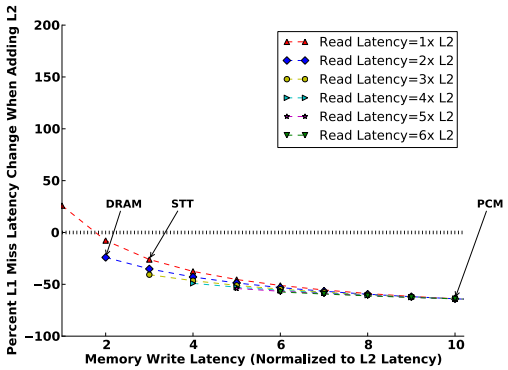
We notice a different behavior with the SPEC applications. As the majority of SPEC applications have much smaller footprints and working sets than our data-

centric workloads, they tend to benefit from a stacked L2 even at relatively low memory latency ratios. Three of our four selected apps (*h264*, *sjeng*, and *soplex*) likely benefit from an L2, only *milc* is better off without. Note that we selected *soplex* because it has one of the largest working sets among the SPEC applications, yet it still rarely spills out of the L2 compared to the data-centric workloads. Because they are designed to stress CPU rather than memory system performance, SPEC applications do not place significant pressure on the caches. A memory system optimized to run SPEC may perform poorly for scientific and data-centric applications.

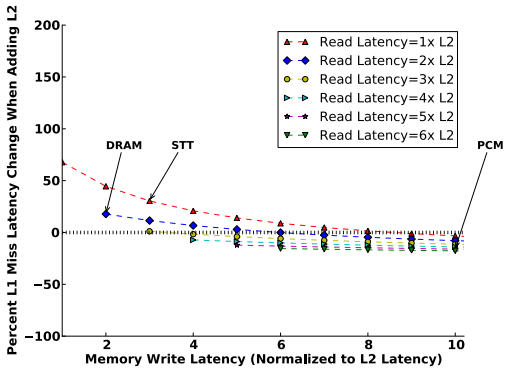
B.5.2 Asymmetric Read and Write Latencies

Figure B.5 and Figure B.6 show the percent change in average L1 miss latency when adding an L2 for the eight benchmarks assuming asymmetric memory read and write latencies (we only consider cases where the write latency is equal to or higher than the read latency). We normalize the memory read and write latencies to the L2 access latency. This experiment models the read-write latency asymmetry that is anticipated in emerging memory technologies, and examines the performance impact of coalescing L1 write-back traffic within the L2. The figures also indicate the approximate write-to-read latency ratio for a few emerging memory technologies [50, 83, 88]. Points above the dotted horizontal line (y=0) on each graph indicate a reduction in average L1 miss latency when eliminating the L2, while points below the dotted line indicate a reduction in average L1 miss latency with the inclusion of an L2. In other words, points above the dotted horizontal line favor exclusion of an L2, while points below the line favor inclusion of an L2.

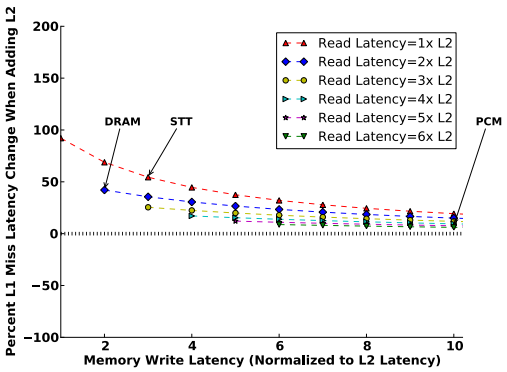
In all but one case of the SPEC workloads, slower writes degrade performance when the L2 is elided (*milc* sees no difference because the L2 is ineffective for this workload). However, among the non-SPEC applications, with the exception of *IntSort*, the performance gap is small. Hence, we conclude that for data-centric workloads, an



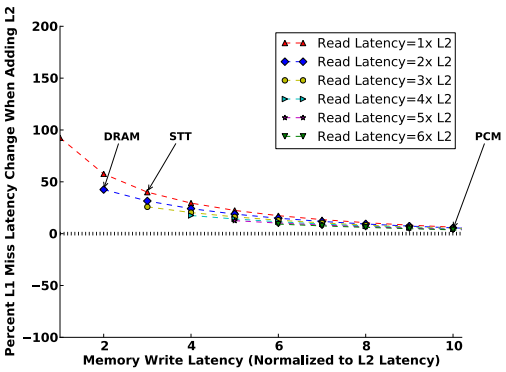
(a) Average L1 Miss Latency: *IntSort*



(b) Average L1 Miss Latency: *Fluidanimate*

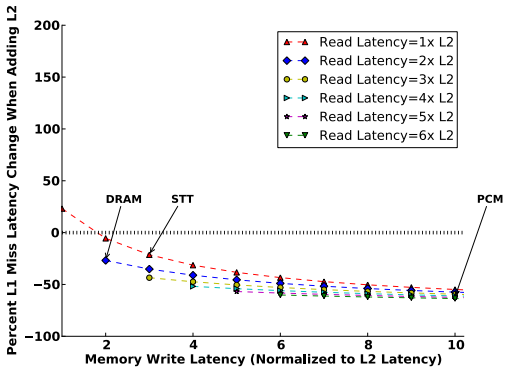


(c) Average L1 Miss Latency: *Canneal*

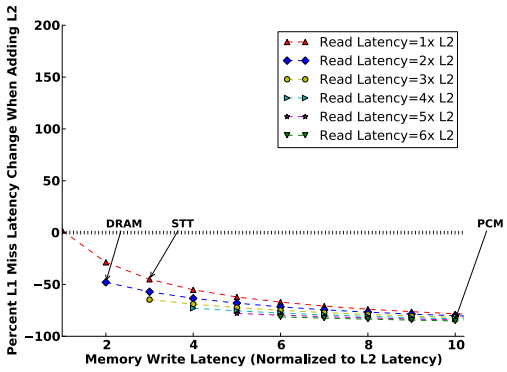


(d) Average L1 Miss Latency: *Graph500*

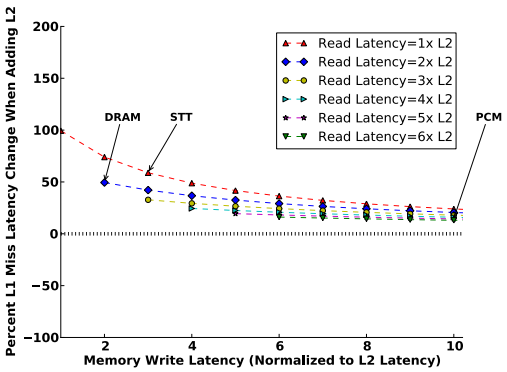
Figure B.5: Percent change in average L1 miss latency for the non-SPEC benchmarks under conditions of asymmetric memory read and write latencies. Memory read and write latency are normalized to L2 the access latency. Points below the dotted line (y=0) favor inclusion of an L2, while points above the dotted line favor exclusion of the L2. In general, an L2 is not beneficial except for *IntSort*. *Fluidanimate* finds an L2 beneficial only when read and write latencies are over 4x the L2 access latency.



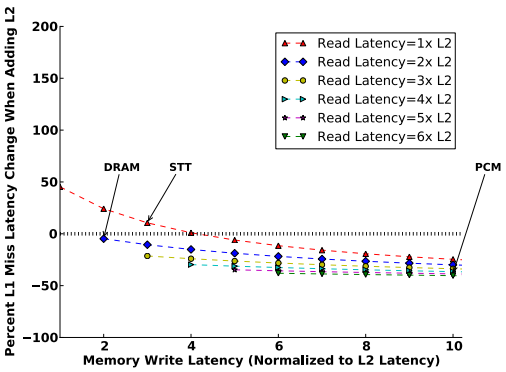
(a) Average L1 Miss Latency: *sjeng*



(b) Average L1 Miss Latency: *h264*



(c) Average L1 Miss Latency: *milc*



(d) Average L1 Miss Latency: *soplex*

Figure B.6: Percent change in average L1 miss latency for the SPEC benchmarks for asymmetric memory read and write latencies. Points below the dotted line (y=0) favor inclusion of an L2, while points above the dotted line favor exclusion of the L2. When memory writes are significantly slower than memory reads, an L2 improves performance for all benchmarks except *milc*.

intermediate cache is not warranted even if stacked memory writes are substantially slower than reads. Again, we see a markedly different behavior in SPEC apps, which have substantial L1 writeback traffic, and benefit from write coalescing in L2.

Benchmark	Latency
IntSort	1.4x
Fluidanimate	3.1x
Canneal	12.7x
Graph500	13.4x
sjeng	1.3x
h264	1.0x
milc	178.5x
soplex	1.8x

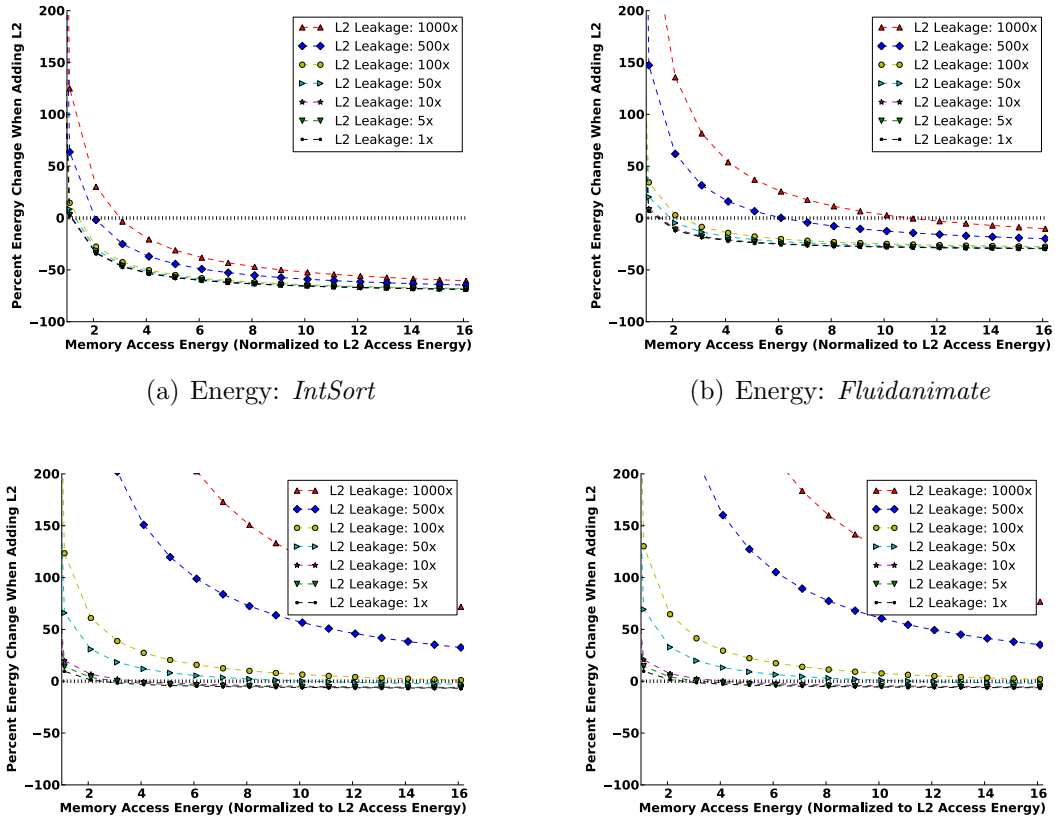
Table B.1: Memory access latency inflection points.

B.6 Energy Filtering

Finally, we consider the impact of an intermediate cache as an energy filter. To simplify our analysis, we select the stacked memory access latency on a per workload basis. The memory latency value is normalized to the L2 access latency such that the application runtime is equal with and without an L2. At this point, due to the equal runtimes, the leakage energy of the stacked memory is the same with and without an L2. Hence, we can neglect the stacked memory leakage in our analysis. Table B.1 lists the memory latency values used for each workload.

We hold the L2 dynamic energy fixed at 1 nJ per access (a typical value for an 8MB L2 in current technology), and vary the stacked memory energy-per-access relative to this fixed value. The ratio of main memory to L2 dynamic energy is reflected on the x-axis in Figure B.7 and Figure B.8. We also vary the L2 static energy over a wide range, from 1 mW (1x leakage) to 1W (1000x leakage). We sweep leakage power over this large range because L2 leakage can vary by orders of magnitude due to two effects. First, leakage can vary relative to dynamic energy by a factor of ten based on the activity factor of the L2 cache. Second, leakage can vary by nearly 100x based on

the particular circuit implementation selected for the cache (e.g., low-operating-power vs. high performance cells).



(a) Energy: *IntSort*

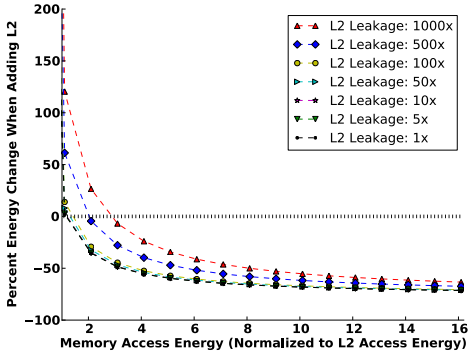
(b) Energy: *Fluidanimate*

(c) Energy: *Canneal*

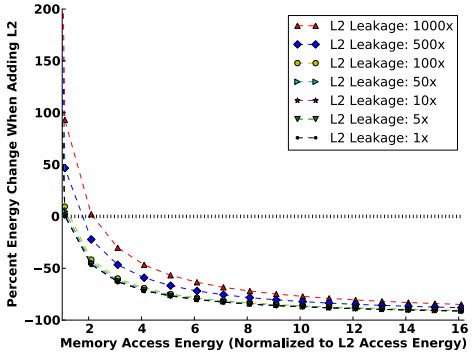
(d) Energy: *Graph500*

Figure B.7: Percent energy change when adding an L2 for the non-SPEC benchmarks. Memory access energy is normalized to L2 access energy. L2 leakage is swept from a baseline of 1x (1 mW) up to 1000x (1W). Points below the dotted line (y=0) favor inclusion of an L2, while points above the dotted line favor exclusion of the L2. (a) *IntSort* benefits from having an L2 beyond a memory access latency of 4x over the baseline, regardless of leakage. (b) *Fluidanimate* benefits from having an L2 beyond a memory access latency of 12x over the baseline, regardless of leakage. For lower memory access energy, and higher leakage, *Fluidanimate* does not require an L2. (c) and (d) *Canneal* and *Graph500* do not require an L2 in general.

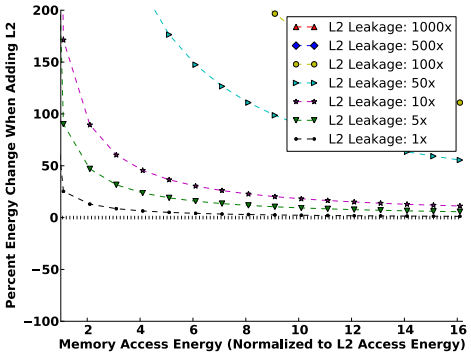
Figure B.7 and Figure B.8 show the percent energy change when adding an L2 for the non-SPEC and SPEC benchmarks, respectively. Points above the dotted horizontal line (y=0) on each graph indicate energy savings when eliminating the L2, while points below the dotted line indicate lower energy with inclusion of an L2.



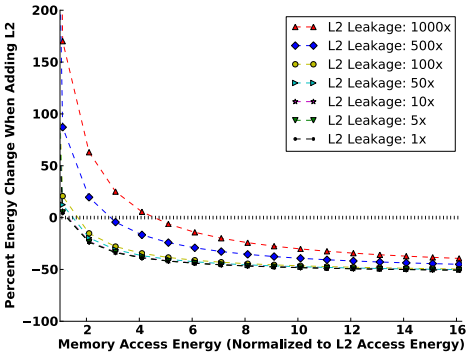
(a) Energy: *sjeng*



(b) Energy: *h264*



(c) Energy: *milc*



(d) Energy: *soplex*

Figure B.8: Percent energy change when adding an L2 for the SPEC benchmarks. Memory access energy is normalized to L2 access energy. L2 leakage is swept from a baseline of 1x (1 mW) up to 1000x (1W). Points below the dotted line (y=0) favor inclusion of an L2, while points above the dotted line favor exclusion of the L2. (a) and (b) *sjeng* and *h264* benefit from having an L2 beyond a memory access latency of 3x over the baseline, regardless of leakage. (c) *milc* never benefits from the inclusion of an L2. (d) *soplex* benefits from having an L2 when the memory access latency exceeds the baseline by over 5x.

Because DRAM accesses require relatively little dynamic energy, it is likely that the memory-to-L2 access energy ratio will be quite low (below 1.5) for stacked DRAM (stacked DRAM accesses are projected to require 1 to 1.5 nJ, similar to the dynamic energy of an access to an 8MB SRAM). At this ratio, *none* of the applications benefit from an L2 from an energy filtering perspective. Even if stacked memory dynamic energy-per-access turns out to be substantially worse than expected (e.g., greater than 4x the L2 dynamic energy-per-access), L2 will still only be profitable if L2 leakage energy is low. For example, for the data centric applications, L2 leakage must be below 100 mW to provide a substantial energy benefit for any application except *IntSort*. Hence, we conclude that, from an energy perspective, an intermediate cache is unlikely to provide an energy advantage; the intermediate cache conserves energy only if (1) the L2 hit rate is high, (2) the stacked memory requires at least 4x the energy per access of the L2, and (3) the L2 leakage energy is low (well below 100mW). Given the good energy characteristics projected for stacked DRAM, we find it unlikely that all three of these conditions will typically hold.

B.7 Conclusion

Recent years have witnessed the emergence of 3D die stacking which has made single-chip server designs feasible. Moreover, memory technology is poised to undergo a transformation with the advent of non-volatile memories. 3D stacking alters key assumptions of conventional memory hierarchy design by drastically increasing cross-layer bandwidth and reducing the latency ratio between successive layers. In light of these technology trends, the *PicoServer* study has suggested flattening the memory hierarchy and eliminating caches that lengthen the critical path between the L1 and stacked memory.

In this chapter, we revisit the conclusions made by *PicoServer*. By considering various design factors like the latency and energy ratios between the L2 and main

memory, along with various L2 leakage values, we identify inflection points where an L2 becomes profitable. We guide our investigation by analyzing intermediate caches from three perspectives: *bandwidth*, *latency*, and *energy filtering*. We focus our study on data-centric and scientific workloads and draw contrasts with the SPEC applications that are frequently used in computer architecture studies.

We conclude that bandwidth filtering is moot given the plentiful bandwidth provided by 3D stacking. From a performance (latency) perspective, although a flat hierarchy is optimal for access patterns with poor temporal locality, some workloads benefit from the access latency reduction afforded by an L2. This effect is more prominent when memory write and read latencies are asymmetric, a characteristic of memory technologies like PCM and STT-RAM. Finally, from an energy perspective, we find that for scientific and data-centric workloads with large footprints, an L2 cache is unlikely to be beneficial from an energy perspective. In particular, we find that low L2 leakage power is critical. SPEC applications tend to have good hit rates in an 8MB cache and hence favor inclusion of an L2; we conclude that these benchmarks should be used with caution when designing memory hierarchies for data-centric applications.

In summary, we find that the conclusions drawn in the *PicoServer* study hold true for the data-centric and scientific workloads. For these benchmarks, having an L2 is only profitable under the following situations: (1) when the L2 leakage is small, (2) when main memory is relatively distant compared to the L2, and (3) for write-intensive workloads when memory write latency exceeds memory read latency.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apache Flume.
http://flume.apache.org.

[2] Apache Hadoop.
http://hadoop.apache.org.

[3] Deloitte University Press: In-Memory Revolution.
http://dupress.com/articles/2014-tech-trends-in-memory-revolution/.

[4] English Letter Frequency.
http://www.math.cornell.edu/ mec/2003-2004/cryptography/subs/frequencies.html.

[5] IBM: Inside the Linux 2.6 Completely Fair Scheduler.
http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler.

[6] Intel Corp.: 3D XPoint Unveiled-The Next Breakthrough in Memory Technology.
http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html.

[7] Intel W5590 Processor Specifications.
http://ark.intel.com/products/41643/Intel-Xeon-Processor-W5590.

[8] Red Hat Corporation: Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel.
http://www.redhat.com/magazine/008jun05/features/schedulers.

[9] SPEC CPU2006 Memory Characterization.
http://www.jaleels.org/ajaleel/workload.

[10] Splunk.
http://http://www.splunk.com.

[11] Use Aho-Corasick Algorithm to Search Multiple Fixed Words.
https://lists.gnu.org/archive/html/bug-grep/2005-08/msg00006.html.

[12] Why GNU grep is Fast.
https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html.

[13] Samsung Electronics: Samsung High-performance SDRAM for Main Memory. 2007.

[14] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[15] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6), June 1975.

[16] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proceedings of the European Conference on Computer Systems*, 2011.

[17] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-Locality in Datacenter Computing Considered Irrelevant. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2011.

[18] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, 2007.

[19] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

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

[21] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, et al. Die Stacking (3D) Microarchitecture. In *Proc. 39th Annual Intnl. Symp. on Microarchitecture*, 2006.

[22] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[23] D. Borthakur. *The Hadoop Distributed File System*. Apache Software Foundation, 2007.

[24] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Commun. ACM*, 1977.

[25] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. In *INFOCOM, 2010 Proceedings IEEE*, 2010.

[26] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic Clustering of the Web. *Comput. Netw. ISDN Syst.*, 29(8-13), 1997.

[27] D. Bryant. *Disrupting the Data Center to Create the Digital Services Economy.* Intel Corporation, 2014.

[28] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time Search at Twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, 2012.

[29] C.-C. Chen and S.-D. Wang. An Efficient Multicharacter Transition Stringmatching Engine Based on the Aho-corasick Algorithm. *ACM Transactions on Architecture and Code Optimization*, 2013.

[30] T.-W. Chen and S.-Y. Chien. Flexible Hardware Architecture of Hierarchical K-Means Clustering for Large Cluster Number. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8), 2011.

[31] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding Replicated Web Collections. *SIGMOD Rec.*, 29(2):355–366, May 2000.

[32] B. Commentz-Walter. A String Matching Algorithm Fast on the Average. *Intl. Coll. on Automata, Languages, and Programming*, 1979.

[33] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.

[34] B. Ding and A. C. König. Fast Set Intersection in Memory. *Proc. VLDB Endow.*, 4(4):255–266, Jan. 2011.

[35] N. Doshi. *Using File Contents as Input for Search.* Splunk Blogs, 2009.

[36] J. V. L. (ed.). *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* MIT Press/Elsevier, 1990.

[37] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: Flexible Data Placement and its Exploitation in Hadoop. In *Proceedings of the VLDB Endowment*, 2011.

[38] G. Erkan and D. R. Radev. LexRank: Graph-Based Lexical Centrality as Salience in Text Summarization. *J. Artif. Int. Res.*, 22(1), Dec. 2004.

[39] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual Intl. Symposium on*, 2011.

[40] A. D. Ferguson and R. Fonseca. Understanding Filesystem Imbalance in Hadoop. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[41] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, et al. Centip3De: A 3930DMIPS/W Configurable Near-threshold 3D Stacked System with 64 ARM Cortex-M3 Cores. In *Intnl. Solid-State Circuits Conf.*, 2012.

[42] S. Fushimi and M. Kitsuregawa. GREO: A Commercial Database Processor Based on a Pipelined Hardware Sorter. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 449–452, 1993.

[43] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. 10th Ann. Intnl. Symp. on Computer Architecture*, 1983.

[44] J. S. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Workshop on Hot Topics in Operating Systems*, 1995.

[45] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[46] J. H. Howard. An Overview of the Andrew File System. In *USENIX Winter Technical Conference*, 1988.

[47] N. Hua, H. Song, and T. Lakshman. Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection. In *INFOCOM 2009, IEEE*, 2009.

[48] IBM Corporation. *IBM PureData System for Analytics Architecture: A Platform for High Performance Data Warehousing and Analytics*. IBM Corporation, 2010.

[49] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[50] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers. Improving Write Operations in MLC Phase Change Memory. In *Proc. 18th Intnl. Symp. on High Performance Computer Architecture*, 2012.

[51] S. Kandula, J. Padhye, and P. Bahl. Flyways To De-Congest Data Center Networks. In *Proceedings of the Workshop on Hot Topics in Networks*, 2009.

[52] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. PicoServer: Using 3D Stacking Technology To Enable A Compact Energy Efficient Chip Multiprocessor. In *Proc. 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[53] D. H. Kim, K. Athikulwongse, M. Healy, M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y.-J. Lee, D. Lewis, T.-W. Lin, et al. 3D-MAPS: 3D Massively Parallel Processor with Stacked Memory. In *Proc. Intnl. Solid-State Circuits Conf.*, 2012.

[54] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proc. 30th Ann. Intnl. Symp. on Microarchitecture*, 1997.

[55] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[56] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.*, 2012.

[57] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. In *Proceedings of the VLDB Endowment*, 2010.

[58] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *Proc. 36th Ann. Intnl. Symp. on Computer Architecture*, 2009.

[59] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *SIGOPS Operating Systems Review*, 44(1), 2010.

[60] D. Li, J. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *Proc. Intnl. Parallel and Distributed Processing Symp.*, 2012.

[61] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[62] C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the Processor-Memory Performance Gap with 3D IC Technology. *IEEE Design Test of Computers*, 22(6), 2005.

[63] G. H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proc. 35th Ann. Intnl. Symp. on Computer Architecture*, 2008.

[64] G. H. Loh and M. D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proc. 44th Ann. Intnl. Symp. on Microarchitecture*, 2011.

[65] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, 2005.

[66] J. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012.

[67] P. Machanick and P. Salverda. Preliminary Investigation of the RAMpage Memory Hierarchy. *South African Computer Journal*, (21):16 –25, August 1998.

[68] S. Manegold, M. L. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. *Proceedings of the VLDB Endowment*, 2009.

[69] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2010.

[70] D. Meisner and T. F. Wenisch. Stochastic Queuing Simulation for Data Center Workloads. In *Proceedings of the Workshop on Exascale Evaluation and Research Techniques*, 2010.

[71] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, 2012.

[72] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010.

[73] J. Moscola, Y. Cho, and J. Lockwood. Hardware-Accelerated Parser for Extraction of Metadata in Semantic Network Content. In *Aerospace Conference, 2007 IEEE*, pages 1–8, March 2007.

[74] S. Mullender(ed.). *Distributed Systems*. ACM Press, 1989.

[75] R. Müller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

[76] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *International Conf. on Data Mining Workshops*, 2010.

[77] D. Pao, W. Lin, and B. Liu. A Memory-efficient Pipelined Implementation of the Aho-corasick String-matching Algorithm. *ACM Transactions on Architecture and Code Optimization*, 2010.

[78] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[79] D. Perera and K. F. Li. On-Chip Hardware Support for Similarity Measures. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 354–358, 2007.

[80] D. Perera and K. F. Li. Hardware Acceleration for Similarity Computations of Feature Vectors. *Electrical and Computer Engineering, Canadian Journal of*, 33(1):21–30, 2008.

[81] K. Puttaswamy and G. Loh. 3D-Integrated SRAM Components for High-Performance Microprocessors. *IEEE Trans. Computers*, 58(10), 2009.

[82] V. Qazvinian and D. R. Radev. Scientific Paper Summarization Using Citation Summary Networks. In *Proceedings of the 22nd International Conference on Computational Linguistics*, COLING '08, 2008.

[83] M. Qureshi, M. Franceschini, and L. Lastras-Montano. Improving Read Performance of Phase Change Memories Via Write Cancellation and Write Pausing. In *Proc. 16th Intnl. Symp. on High Performance Computer Architecture*, 2010.

[84] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proc. 36th Ann. Intnl. Symp. on Computer Architecture*, 2009.

[85] S. Radicati. *Email Statistics Report, 2014-2018*. Sara Radicati, 2014.

[86] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. Wenisch, and M. Martin. Computational Sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

[87] P. Ranganathan. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer*, 44(1):39 –48, January 2011.

[88] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili. An Energy Efficient Cache Design Using Spin Torque Transfer (STT) RAM. In *Intnl. Symp. on Low-Power Electronics and Design*, 2010.

[89] M. E. Richard L. Villars, Carl W. Olofson. *Big Data: What It Is and Why You Should Care*. IDC, 2011.

[90] P. Roy, J. Teubner, and G. Alonso. Efficient Frequent Item Counting in Multi-Core Hardware. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge discovery and data mining*, KDD '12, pages 1451–1459, 2012.

[91] M. Sahami and T. D. Heilman. A Web-Based Kernel Function for Measuring the Similarity of Short Text Snippets. In *Proceedings of the 15th International conference on World Wide Web*, WWW '06, 2006.

[92] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.*, 39(4), Apr. 1990.

[93] B. Schlegel, T. Willhalm, and W. Lehner. Fast Sorted-Set Intersection using SIMD Instructions. *ADMS Workshop 2011*, 2011.

[94] J. Shafer, S. Rixner, and A. Cox. The Hadoop Distributed Filesystem: Balancing Portability and Performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, 2010.

[95] V. Sikka, F. Färber, A. K. Goel, and W. Lehner. SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform. *PVLDB*, 6(11):1184–1185, 2013.

[96] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 678–684, 2005.

[97] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *ACM SIGMOD Record*, 2005.

[98] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2013.

[99] Synopsys. *DesignWare Building Blocks*. Synopsys Inc., 2011.

[100] D. Tam. *Facebook Processes More Than 500 TB of Data Daily*. CNET, 2012.

[101] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, 2005.

[102] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2002.

[103] W. Tang, R. Gupta, and A. Nicolau. Design of a Predictive Filter Cache for Energy Savings in High Performance Processor Architectures. In *Intnl. Conf. on Computer Design*, 2001.

[104] M. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, 2012.

[105] D. Terdiman. CNET: Twitter Hits Half a Billion Tweets a Day. http://news.cnet.com/8301-1023_3-57541566-93/report-twitter-hits-half-a-billion-tweets-a-day/.

[106] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 229–240, 2012.

[107] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proc. 9th USENIX Conf. on File and Storage Technologies*, 2011.

[108] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. An Optimized 3D-stacked Memory Architecture by Exploiting Excessive, High-density TSV Bandwidth. In *16th Intnl. Symp. on High Performance Computer Architecture*, 2010.

[109] L. Woods, J. Teubner, and G. Alonso. Complex Event Detection at Wire Speed with FPGAs. *PVLDB*, 3(1):660–669, 2010.

[110] M. Wordeman, J. Silberman, G. Maier, and M. Scheuermann. A 3D System Prototype of an eDRAM Cache Stacked over Processor-like Logic Using Through-Silicon Vias. In *Intnl. Solid-State Circuits Conf.*, 2012.

[111] D. Wu, F. Zhang, N. Ao, F. Wang, J. Liu, and G. Wang. A Batched GPU Algorithm for Set Intersection. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, 2009.

[112] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[113] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving MapReduce Performance Through Data Placement in Heterogeneous Hadoop Clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010.

[114] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the European Conference on Computer Systems*, 2010.

[115] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[116] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2008.

[117] X. Zha and S. Sahni. GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU. *Computers, IEEE Transactions on*, 2013.