Incremental Parallel and Distributed Systems

Thesis for obtaining the title of Doctor of Engineering Science of the Faculty of Natural Science and Technology I of Saarland University

From

Pramod Kumar Bhatotia

Saarbrücken
April, 2015
Technical report: MPI-SWS-2015-002

Date of Colloquium:	07/04/2015
Dean of Faculty:	Univ.-Prof. Dr. Markus Bläser
Chair of the Committee:	Prof. Dr. Matteo Maffei
Reporters	
First Reviewer:	Prof. Dr. Rodrigo Rodrigues
Second Reviewer:	Prof. Dr. Peter Druschel
Third Reviewer:	Dr. Rebecca Isaacs
Academic Assistant:	Dr. Rijurekha Sen

ABSTRACT

Incremental computation strives for efficient successive runs of applications by re-executing only those parts of the computation that are affected by a given input change instead of recomputing everything from scratch. To realize the benefits of incremental computation, researchers and practitioners are developing new systems where the application programmer can provide an efficient update mechanism for changing application data. Unfortunately, most of the existing solutions are limiting because they not only depart from existing programming models, but also require programmers to devise an incremental update mechanism (or a *dynamic algorithm*) on a per-application basis.

In this thesis, we present incremental parallel and distributed systems that enable existing real-world applications to automatically benefit from efficient incremental updates. Our approach neither requires departure from current models of programming, nor the design and implementation of dynamic algorithms.

To achieve these goals, we have designed and built the following incremental systems: *(i)* Incoop — a system for incremental MapReduce computation; *(ii)* Shredder — a GPU-accelerated system for incremental storage; *(iii)* Slider — a stream processing platform for incremental sliding window analytics; and *(iv)* iThreads — a threading library for parallel incremental computation. Our experience with these systems shows that significant performance can be achieved for existing applications without requiring any additional effort from programmers.

KURZDARSTELLUNG

Inkrementelle Berechnungen ermöglichen die effizientere Ausführung aufeinanderfolgender Anwendungsaufrufe, indem nur die Teilbereiche der Anwendung erneut ausgefürt werden, die von den Änderungen der Eingabedaten betroffen sind. Dieses Berechnungsverfahren steht dem konventionellen und vollständig neu berechnenden Verfahren gegenüber. Um den Vorteil inkrementeller Berechnungen auszunutzen, entwickeln sowohl Wissenschaft als auch Industrie neue Systeme, bei denen der Anwendungsprogrammierer den effizienten Aktualisierungsmechanismus für die Änderung der Anwendungsdaten bereitstellt. Bedauerlicherweise lassen sich existierende Lösungen meist nur eingeschränkt anwenden, da sie das konventionelle Programmierungsmodel beibehalten und dadurch die erneute Entwicklung vom Programmierer des inkrementellen Aktualisierungsmechanismus (oder einen dynamischen Algorithmus) für jede Anwendung verlangen.

Diese Doktorarbeit stellt inkrementelle Parallele- und Verteiltesysteme vor, die es existierenden Real-World-Anwendungen ermöglichen vom Vorteil der inkrementellen Berechnung automatisch zu profitieren. Unser Ansatz erfordert weder eine Abkehr von gegenwärtigen Programmiermodellen, noch Design und Implementierung von anwendungsspezifischen dynamischen Algorithmen.

Um dieses Ziel zu erreichen, haben wir die folgenden Systeme zur inkrementellen parallelen und verteilten Berechnung entworfen und implementiert: (i) Incoop — ein System für inkrementelle Map-Reduce-Programme; (ii) Shredder — ein GPU-beschleunigtes System zur inkrementellen Speicherung; (iii) Slider — eine Plattform zur Batch-basierten Streamverarbeitung via inkrementeller Sliding-Window-Berechnung; und (iv) iThreads — eine Threading-Bibliothek zur parallelen inkrementellen Berechnung. Unsere Erfahrungen mit diesen Systemen zeigen, dass unsere Methoden sehr gute Performanz liefern können, und dies ohne weiteren Aufwand des Programmierers.

PUBLICATIONS

Parts of the thesis have appeared in the following publications.

- "iThreads: A Threading Library for Parallel Incremental Computation". Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Bjoern Brandenburg, and Rodrigo Rodrigues. In *proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

- "Slider: Incremental Sliding Window Analytics". Pramod Bhatotia, Umut A. Acar, Flavio Junqueira, and Rodrigo Rodrigues. In *proceedings of the 15th Annual ACM/IFIP/USENIX Middleware conference (Middleware)*, 2014. **Best student paper award.**

- "MapReduce for Incremental Computation". Pramod Bhatotia, Alexander Wieder, Umut A. Acar, and Rodrigo Rodrigues. *Invited book chapter: Advances in data processing techniques in the era of Big Data, CRC Press*, 2014.

- "Shredder: GPU-Accelerated Incremental Storage and Computation". Pramod Bhatotia, Rodrigo Rodrigues and Akshat Verma. In *proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

- "Incoop: MapReduce for Incremental Computations". Pramod Bhatotia, Alexander Wieder, Rafael Pasquini, Rodrigo Rodrigues and Umut A. Acar. In *proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.

- "Large Scale Incremental Data Processing with Change Propagation". Pramod Bhatotia, Alexander Wieder, Istemi Ekin Akkus, Rodrigo Rodrigues and Umut A. Acar. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

Additional publications published while at MPI-SWS.

- "Orchestrating the Deployment of Computations in the Cloud with Conductor." Alexander Wieder, Pramod Bhatotia, Ansley Post and Rodrigo Rodrigues. In *proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

- "Performance Evaluation and Optimization of Random Memory Access on Multicores with High Productivity." Vaibhav Saxena, Yogish Sabharwal, Pramod Bhatotia. In *proceedings of ACM/IEEE International Conference on High Performance Computing (HiPC)*, 2010. **Best paper award.**

- "Reliable Data-Center Scale Computations." Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Flavio Junqueira, and Benjamin Reed. In *proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

- "Conductor: Orchestrating the Clouds." Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. *proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

- "Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud." Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. In *proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

Dedicated to my mummy and papa.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 The Promise of Incremental Computation

Many real-world applications are inherently used in an incremental workflow, that is, they are invoked repeatedly with only small changes in input. Common examples span a wide spectrum of applications including scientific simulations, large-scale data analytics, reactive systems, robots, traffic control systems, scheduling systems, etc. All these applications naturally arise in a number of domains, including software systems, graphics, robotics, databases, and networked systems. These applications interact with the physical world observing their input data continuously evolving over time, causing incremental and continuous modifications to the property being computed. Therefore, these applications must respond to such incremental modifications correctly and efficiently.

Such applications, when confronted with localized modifications to the input, often require only small modifications to the output. Therefore, if we have techniques for quickly identifying the parts of the output that are affected by the modifications and updating them while reusing the rest of the output, we will be able to incrementally update the output in a significantly more efficient (and thus faster and cheaper) way than recomputing the entire output from scratch [130].

To realize the benefits of incremental computation, researchers and practitioners are developing new systems where programmers can provide efficient update

mechanism for changing application data. These systems for incremental computation can be significantly more efficient than recomputing from scratch. However, most of the existing approaches have two major limitations: first, these systems depart from existing programming models, which prevents existing, non-incremental programs from taking advantage of these techniques. Second, and more importantly, these systems require programmers to develop efficient incremental update mechanisms (or a *dynamic algorithm*) on a per-application basis.

While dynamic algorithms can be asymptotically more efficient than their conventional non-dynamic versions, they can be difficult to design, implement, and adapt even for simple problems because of their highly specialized nature [55, 60, 70, 71, 78, 83, 118]. Furthermore, dynamic algorithms are overwhelmingly designed for the uniprocessor computing model, and thus cannot take advantage of the parallelism offered in parallel and distributed platforms, which are increasingly important in today's computing world, as motivated next.

1.2 The Inevitability of Parallel & Distributed Systems

Parallel and distributed computing is the most prominent way of computing in the modern environment. Clusters of multicore nodes have become ubiquitous, powering not only some of the most popular consumer applications — Internet services such as web search and social networks — but also a growing number of scientific and enterprise workloads. This computing model is a departure from the uniprocessor computing model where programs run on a single core machine. This shift towards adopting parallel and distributed computing frameworks is driven mainly by a continuous increase in the demand for computing cycles and I/O bandwidth to support these modern applications. The uniprocessor computing model is unable to meet these requirements mainly because increasing a pro-

cessor's speed to get a boost in performance leads to the heat dissipation problem. Also, the I/O bandwidth is limited by available disk and network bandwidth per machine.

To overcome these limitations, the computing platforms are being designed to increase the parallelism by *scaling-them-up* as well as by *scaling-them-out*. In the scaled-up architecture each node consists of a diverse mix of 100s of cores comprising of CPUs and GPUs. These cores operate at a lower frequency to minimize the heat dissipation problem. The scaled-out architecture consists of tens of thousands of those nodes with their corresponding networking and storage subsystems to facilitate I/O parallelization. Thus, the combination of parallel and distributed computing platform provides even more compute cycles and also mitigating the I/O bottlenecks.

The parallel and distributed computing platforms comes with several challenges requiring programmers to manage parallelization, synchronization, load-balancing, fault-tolerance, distributing the data, and communication. To reduce these complexity, programming models such as MapReduce [67], and `pthread` [15]. Due to the growing importance of these computing frameworks, in this work, we focus on building systems to support incremental computation in parallel and distributed systems.

1.3 Thesis Research: Incremental Systems

Thesis statement Incremental systems enable *practical*, *automatic*, and *efficient* incremental computation in real-world parallel and distributed computing.

To this end, our work targets building incremental systems that require neither a radical departure from current models of programming nor complex, application-

specific dynamic algorithms (which, to reiterate, are challenging to design and implement).

The focus of our work is on two common computing paradigms, namely parallel and distributed computing. In particular, we focus on large-scale data processing for distributed incremental computation, and multithreaded programs for parallel incremental computation. To establish the practicality and benefit of the envisioned incremental parallel and distributed frameworks, a crucial aspect of this work is to design and implement incremental systems, and then evaluate the solution with widely applicable case studies to demonstrate the benefits.

1.4 Self-Adjusting Computation

Our approach is to shift the burden of reasoning about how to efficiently process incremental updates from the programmer to the system itself by building on the principles that were developed in the field of self-adjusting computation [24, 25, 27, 59, 87, 88, 103, 104] (a sub-field of programming languages research). The key idea behind self-adjusting computation is to divide a computation into sub-computations, and maintain a dependence graph between sub-computations. Given changes in the input data, a change-propagation algorithm is used to update the output by identifying the parts of the computation that are affected by the changes and rebuilding only those parts. More precisely, self-adjusting computation combines the following three techniques to incrementally update the output: dynamic dependence graph, change propagation, and memoization.

The *dynamic dependence graph* or DDG can be viewed as a representation of the data and control dependences in a computation. The DDG of a program is built as the program is executed by tracking the control and data flow operations, and using it to update the computation and the output when the inputs are modified.

Memoization caches the output of the sub-computations to avoid re-execution. Finally, the *change propagation* mimics a complete re-execution of the program with the modified data, but only re-executes parts of the computation that depend on the modification. Conceptually similar to cell updates in spreadsheets, the change propagation algorithm takes full advantage of previously computed results rather than re-executing everything from scratch on each input change.

Although the work on self-adjusting computation offers a general-purpose framework for developing computations that can perform incremental updates efficiently, it has not been applied to parallel and distributed systems. In this thesis, we extend the work on self-adjusting computation to support parallel and distributed computing.

1.5 Thesis Contributions

In this thesis, we present the design and implementation of the following incremental systems to enable practical, transparent, and efficient incremental computation for real-world parallel and distributed computing.

Incoop is a system for incremental MapReduce computation [51, 53, 52]. Incoop transparently detects changes between two files that are used as inputs to consecutive MapReduce jobs, and efficiently propagates those changes until the new output is produced. The design of Incoop is based on memoizing the results of previously run tasks, and reusing these results whenever possible. Doing this efficiently introduces several technical challenges that are overcome with novel techniques, such as a large-scale storage system called Inc-HDFS that efficiently computes deltas between two inputs, a contraction phase for fine-grained updates, and a memorization-aware scheduling algorithm.

Shredder is a GPU-accelerated system for incremental storage [50]. Shredder was initially designed to improve Inc-HDFS, which has high computational requirements for detecting duplicate content using content-based chunking [114]. To address the computational bottleneck, we designed Shredder, a high performance content-based chunking framework for identifying deltas between two inputs in Inc-HDFS. Shredder exploits the massively parallel processing power of GPUs to overcome the CPU bottlenecks of content-based chunking in a cost-effective manner. Shredder provides several novel optimizations aimed at reducing the cost of transferring data between host (CPU) and GPU, fully utilizing the multicore architecture at the host, and reducing GPU memory access latencies. We used shredder to implement a GPU-accelerated Inc-HDFS for incremental MapReduce computation. In addition, we show that Shredder is a generic system to accelerate incremental storage based on data deduplication

Slider is a batched stream processing platform for incremental sliding window computation [47, 48]. Slider does not require programmers to explicitly manage the intermediate state for overlapping windows, allowing the existing single-pass applications to incrementally update the output every time the computation window slides. The design of Slider incorporates self-adjusting contraction trees, a set of data structures and algorithms for transparently updating the output of data-parallel sliding window computations as the window moves, while reusing, to the extent possible, results from prior computations. Self-adjusting contraction trees organize sub-computations into self balancing trees, with a structure that is better suited to each type of sliding window computation (append-only, fixed-width, or variable-width slides). Furthermore, they enable a split processing mode, where a background processing leverages the predictability of input changes to pave the way for a more efficient foreground processing when the window slides. We

also provide an extension of self-adjusting contraction trees to handle multiple-job workflows such as query processing.

iThreads is a threading library to support parallel incremental computation targeting unmodified C/C++ pthread-based multithreaded programs [49]. iThreads supports shared-memory multithreaded programs: it can be used as a replacement for pthreads by a simple exchange of dynamically linked libraries, without even recompiling the application code. To enable such an interface, we designed algorithms and an implementation to operate at the compiled binary code level by leveraging operating system mechanisms encapsulated in a dynamically linkable shared library. iThreads makes use of *parallel* algorithms for incremental multithreading. The parallel algorithms record the intra- and inter-thread control and data dependencies using a *concurrent* dynamic data dependency graph, and use the graph to incrementally update the output as well as the graph on input changes.

The newly proposed systems for incremental computation are limiting because they not only require substantial programming effort, but also lose backwards-compatibility with widely deployed systems. Thus, there is still room for an improvement that can ignite the widespread adoption of these newly proposed systems. Our incremental systems push the limits of the state-of-the-art by applying the principles and lessons learned in prior algorithms- and programming-language-centric work to parallel and distributed systems, with the goal of building *practical* incremental frameworks that enable existing real-world applications to automatically benefit from efficient incremental updates. Our approach neither requires departure from current models of programming, nor the invention and implementation of application-specific dynamic algorithms for incremental computation. Our experience with these systems shows that our techniques can yield very good per-

formance, both in theory and practice, without requiring programmers to write any special-purpose algorithms for incremental computation.

1.6 Organization

The remainder of the thesis is organized as follows.

In Chapter 2, we present the design and implementation of Incoop.

In Chapter 3, we present the design and implementation of Shredder.

In Chapter 4, we present the design and implementation of Slider.

In Chapter 5, we present the design and implementation of iThreads.

Finally, in Chapter 6, we conclude.

CHAPTER 2

Incoop: Incremental Batch Processing

In this chapter, we describe the design, implementation, and evaluation of Incoop, a MapReduce framework for incremental computation. Incoop detects changes to the input and automatically updates the output by employing an efficient, fine-grained result reuse mechanism.

This chapter is organized as follows. We first motivate the design of Incoop in Section 2.1. We next briefly highlight the contributions of Incoop in Section 2.2. Thereafter, we present a brief background MapReduce in Section 2.3. We next present an overview of Incoop in Section 2.4. The system design is detailed in Sections 2.5, 2.6, and 2.7. We present an experimental evaluation of Incoop in Section 2.9, and case-studies in Section 2.10. We present the related work in Section 2.11. Limitations and conclusion are discussed in Section 2.12 and Section 2.13, respectively.

2.1 Motivation

Distributed processing of large data sets has become an important task in the life of various companies and organizations, for whom data analysis is an important vehicle to improve the way they operate. This area has attracted a lot of attention from both researchers and practitioners over the last few years, particularly after

the introduction of the MapReduce paradigm for large-scale parallel data processing [67].

A usual characteristic of the data sets that are provided as inputs to large-scale data processing jobs is that they do not vary dramatically over time. Instead, the same job is often invoked consecutively with small changes in this input from one run to the next. For instance, researchers have reported that the ratio between old and new data when processing consecutive web crawls may range from 10 to 1000X [110].

Motivated by this observation, there have been several proposals for large-scale *incremental* data processing systems, such as Percolator [121] or CBP [110], to name a few early and prominent examples. In these systems, the programmer is able to devise an incremental update handler (or a dynamic algorithm), which can store state across successive runs, and contains the logic to update the output as the program is notified about input changes. While this approach allows for significant improvements when compared to the "single shot" approach, i.e., reprocessing all the data each time that part of the input changes or that inputs are added and deleted, it also has the downside of requiring programmers to adopt a new programming model and API. This has two negative implications. First, there is the programming effort to port a large set of existing applications to the new programming model. Second, it is often difficult to devise a dynamic algorithm for incrementally updating the output as the input changes.

2.2 Contributions

In this chapter, we present the design and implementation of Incoop, a system for large-scale incremental MapReduce computation [53]. Incoop extends the Hadoop

open source implementation of the MapReduce paradigm to run unmodified MapReduce programs in an incremental way.

The idea behind Incoop is to enable the programmer to automatically incrementalize existing MapReduce programs without the need to make any modifications to the code. To this end, Incoop records information about executed MapReduce tasks so that they can be reused in future MapReduce computations when possible.

The basic approach taken by Incoop consists of (1) splitting the computation into sub-computations, where the natural candidate for a sub-computation is a MapReduce task; (2) memoizing the inputs and outputs of each sub-computation; (3) in an incremental run, checking the inputs to a sub-computation and using the memoized output without rerunning the task when the input remains unchanged. Despite being a good starting point, this basic approach has several shortcomings that motivated us to introduce several technical innovations in Incoop, namely:

- **Incremental HDFS.** We introduce a file system called Inc-HDFS that provides a scalable way of identifying the deltas in the inputs of two consecutive job runs. This reuses an idea from the LBFS local file system [114], which is to avoid splitting the input into fixed-size chunks, and instead split it based on the contents such that small changes to the input keep most chunk boundaries. The new file system is able to achieve a large reuse of input chunks while maintaining compatibility with HDFS, which is the most common interface to provide the input to a job in Hadoop.

- **Contraction phase.** To avoid rerunning a large Reduce task when only a small subset of its input changes, we introduce a new phase in the MapReduce framework called the Contraction phase. This consists of breaking up the Reduce task into smaller sub-computations that form an inverted tree,

such that, when a small portion of the input changes, only the path from the corresponding leaf to the root needs to be recomputed.

- **Memoization-aware scheduler.** We modified the scheduler of Hadoop to take advantage of the locality of memoized results. The new scheduler uses a work stealing strategy to decrease the amount of data movement across machines when reusing memoized outputs, while still allowing tasks to execute on machines that are available.

We implemented Incoop by extending Hadoop and evaluated it using five MapReduce applications. We also employed Incoop to demonstrate two important use cases of incremental processing: *incremental log processing*, where we use Incoop to build a framework to incrementally process logs as more entries are added to them; and *incremental query processing*, where we layer the Pig framework on top of Incoop to enable relational query processing on continuously arriving data.

2.3 Background

We first present a brief background on MapReduce programming model and the associated run-time system.

2.3.1 MapReduce Programming Model

The MapReduce programming model, and a framework that implements it, was first presented by Google [67] to simplify the development and deployment of large-scale data-parallel applications. The framework provides two basic programming constructs: *Map* and *Reduce*. The *Map* function takes a set of input values and maps each value to a set of key-value tuples. The *Reduce* function takes a key and a list of values as input and reduces the list to a final output value. Next, we de-

scribe Hadoop[1], an open-source implementation of the MapReduce programming framework, that forms the basis of our system.

2.3.2 Hadoop Architecture

Hadoop provides a programming and runtime environment for developing and deploying MapReduce programs on large clusters. The framework consists of two main components: the *Hadoop distributed file system* (HDFS) and the *Hadoop MapReduce engine*, which we describe next.

2.3.2.1 Hadoop Distributed File System

Large-scale data-parallel applications often process and generate tremendous amounts of data, and managing data storage at that scale raises its own challenges. To address these, the Hadoop storage component provides the *Hadoop Distributed File System* (HDFS) that is specifically engineered to handle huge amounts of data, such as the data that is used as input, or produced as the output of MapReduce jobs.

In HDFS, the data is distributed across multiple *Data-nodes*, which are typically the same nodes that also execute the MapReduce jobs. To cope with the failure of an individual *Data-node*, data is replicated among a configurable number of different nodes. Files in HDFS are split into smaller *chunks* of fixed size (e.g., 64MB). To locate data blocks in HDFS, a centralized directory service running on the *Name-node* enables clients to look up and access data. To cut the overhead for maintaining data consistency, HDFS does not allow for modifying data once written and provides only append-only interface. This design decision is driven by the fact that MapReduce jobs only write data once and do not modify it afterwards, since intermediate and final results of the jobs are written into new files.

[1]Hadoop: http://hadoop.apache.org/

2.3.2.2 Hadoop MapReduce Engine

The *Hadoop MapReduce engine* implements the logic to coordinate the execution of a MapReduce job on a cluster and distributes tasks to nodes. Users submit jobs to the *Job Tracker* that splits the job into multiple *Tasks*, which can either consist of applying the *Map* function to a specific partition of the input, or applying the *Reduce* function to a key and the associated values generated by the all *Map* functions. The job execution is divided into the *Map phase* where all the Map tasks are executed, and the *Reduce phase*, which starts upon completion of the Map phase, and where the output of all the Map tasks is processed by the *Reduce* tasks.

The *Job Tracker* is responsible for keeping track of cluster utilization and progress of the job, and performs all the scheduling decisions that determine where each task is run. The granularity of *Map* tasks is determined by the fact that each *Task* processes a file *split* (one or more chunk of the input file). On the other hand, the granularity of *Reduce* tasks depends only on the input to the computation, given that each *Reduce* task processes a single key and all corresponding values that were emitted by the union of all Map tasks.

2.4 Overview

We present first a basic design that we use as a starting point, highlight the limitations of this basic design, the challenges in overcoming them, and briefly overview the main ideas behind Incoop, which addresses the limitations of the basic design. Our basic strategy is to adapt the principles of self-adjusting computation to the MapReduce paradigm, and in particular to Hadoop.

2.4.1 Basic design

Our goal is to design a system for large-scale incremental data processing that is able to leverage the performance benefits of incremental computation, while also being transparent, meaning that it does not require changes to existing programs.

To achieve this goal, we apply the principles of self-adjusting computation to the MapReduce paradigm. To remind the reader, self-adjusting computation [24, 26, 25, 87, 59] offers a solution to the incremental computation problem by enabling any computation to respond to changes in its data by efficiently recomputing only the subcomputations that are affected by the changes. To this end, a self-adjusting computation tracks dependencies between the inputs and outputs of subcomputations, and, in incremental runs, only rebuilds subcomputations affected (transitively) by modified inputs. To identify the affected subcomputations, the approach represents a computation as a dependency graph of subcomputations, where two sub-computations are data-dependent if one of them uses the output of the other as input and control-dependent if one takes place within the dynamic scope of another. Subcomputations are also memoized based on their inputs to enable reuse even if they are control-dependent on some affected subcomputation. Given the "delta", the modifications to the input, a *change-propagation algorithm* pushes the modifications through the dependency graph, rebuilding affected subcomputations, which it identifies based on both data and control dependencies. Before rebuilding a subcomputation, change propagation recovers subcomputations that can be re-used, even partially, by using a computation memoization technique that remembers (and re-uses) not just input-output relationships but also the dependency graphs of memoized subcomputations [25].

In order to apply self-adjusting computation techniques to the Map-Reduce paradigm, we first need to decide what forms a sub-computation. The natural

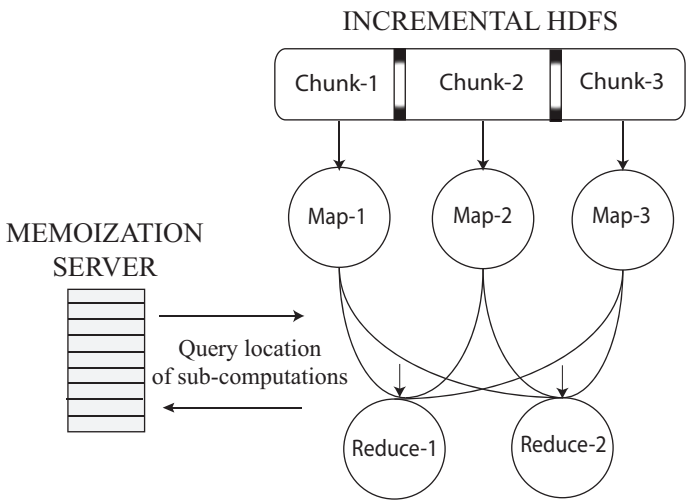


Figure 2.1: Basic design of Incoop.

candidate is to use Map and Reduce tasks as sub-computations; this makes it possible to view the data-flow graph of the MapReduce job as the dependency graph. Since the MapReduce framework implicitly keep track of this graph when implementing the data movement and synchronization between the various tasks, the dependency graph also captures the control dependencies.

This decision leads to our basic design, which is shown in Figure 2.1. In this design, the MapReduce scheduler orchestrates the execution of every MapReduce job normally, by spawning and synchronizing tasks and performing data movement as in a normal MapReduce execution. To record and update the dependency graph implicitly, our design includes a *memoization server* that stores a mapping from the input of a previously run task to the location of the corresponding memoized output. When a task completes, its output is memoized persistently, and a mapping from the input to the location of the output is stored in the memoization server. Then, during an incremental run, when a task is instantiated, the memoization server is queried to check if the inputs to the task match those of a previous run. If so, the system reuses the outputs from the previous run. Otherwise, the

task runs normally and the mapping from its input to the location of the newly produced output is stored in the memoization server.

This basic design raises a series of challenges, which we describe next. In subsequent sections, we describe our key technical contributions that we propose to address these challenges.

2.4.2 Challenge: Transparency

Self-adjusting computation requires knowing the modifications to the input in order to update the output. To this end, it requires a new interface for making changes to the input, so that the edits, which are clearly identified by the interface, can be used to trigger an incremental update. We wish to achieve the efficiency benefits of self-adjusting computation *transparently* without requiring the programmer to change the way they run MapReduce computations. This goal seems to conflict with the fact that HDFS (the system employed to store inputs to MapReduce computations in Hadoop) is an append-only file system, making it impossible to convey input deltas (other than appends). To overcome this challenge, we store the inputs and outputs of consecutive runs in separate HDFS files and compute a delta between two HDFS files in a way that is scalable and performs well.

2.4.3 Challenge: Efficiency

To achieve efficient incremental updates, we must ensure that MapReduce computations remain stable under small changes to their input, meaning that, when executed with similar inputs, many tasks are repeated and their results can be reused. To define stability more precisely, consider performing MapReduce computations with inputs I and I' and consider the respective set of tasks that are executed, denoted T and T'. We say that a task $t \in T'$ is not *matched* if $t \notin T$, i.e., the task that is

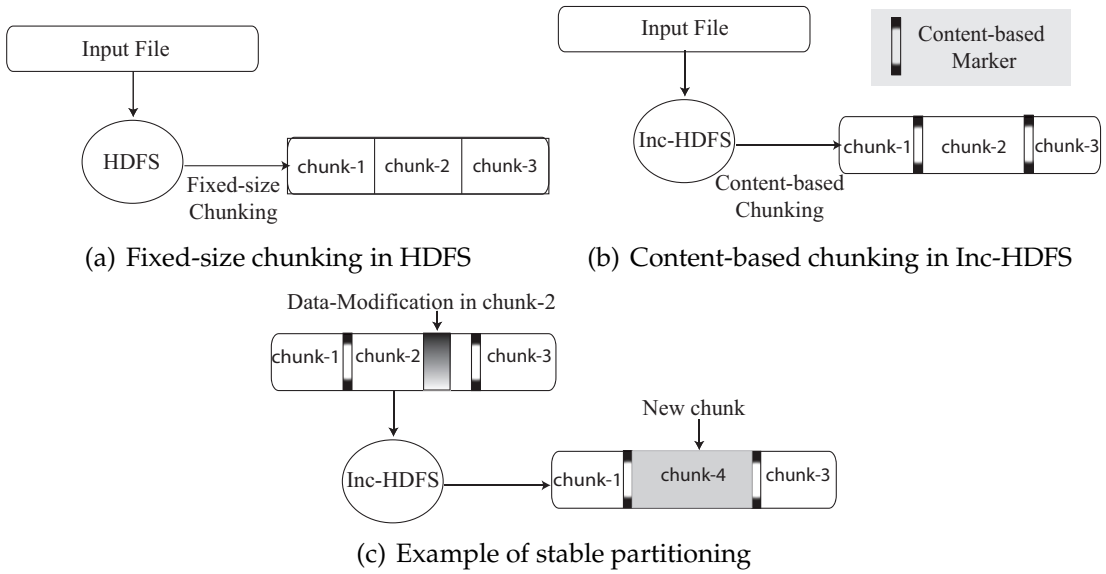
performed with input I' is not performed with the input I. We say that a MapReduce computation is *stable* if the time required to execute the unmatched tasks is small, where small can be more precisely defined as sub-linear in the size of the input.

In the case of MapReduce, stability can be affected by several factors, which we can group into the following two categories: (a) making a small change to the input can change the input to many tasks, causing these tasks to become unmatched; (b) even if a small number of tasks is unmatched, these tasks can take a long time to execute. To address these issues, we introduce techniques for (1) performing a stable input partitioning; (2) controlling the granularity and stability of both Map and Reduce tasks; and (3) finding efficient scheduling mechanisms to avoid unnecessary movement of memoized data.

Stable input partitioning. To see why using HDFS as an input to MapReduce jobs leads to unstable computations, consider inserting a single data item in the middle of an input file. Since HDFS files are partitioned into fixed-sized chunks, this small change will shift each partition point following the input change by a fixed amount. If this amount is not a multiple of the chunk size, all subsequent Map tasks will be unmatched. (On average, a single insert will affect half of all Map tasks.) The problem gets even more challenging when we consider more complex changes, like the order of records being permuted; such changes can be common, for instance, if a crawler uses a depth-first strategy to crawl the web, and a single link change can move the position of an entire subtree in the input file. In this case, using standard algorithms to compute the differences between the two input files is not viable, since this would require running a polynomial-time algorithm (e.g., an edit-distance algorithm). We explain how our new file system called Inc-HDFS leads to stable input partitioning without compromising efficiency in Section 2.5.

Granularity control. A stable partitioning leads directly to the stability of Map tasks. The input to the Reduce tasks, however, is determined only by the outputs of the Map tasks, since each Reduce task processes all values produced in the Map phase and associated with a given key. Consider, for instance, the case when a single key-value pair is added to a Reduce task that processes a large number of values (e.g., linear in the size of the input). This is problematic since it causes the entire task to be re-computed. Furthermore, even if we found a way of dividing large Reduce tasks into multiple smaller tasks, this per se would not solve the problem, since we would still need to aggregate the results of the smaller tasks in a way that avoids a large recomputation. Thus, we need a way to (i) split the Reduce task into smaller tasks and (ii) eliminate potentially long (namely linear-size) dependencies between these smaller tasks. We solve this problem with a new Contraction phase, where Reduce tasks are broken into sub-tasks organized in a tree. This breaks up the Reduce task while ensuring that long dependencies between tasks are not formed, since all paths in the tree will be of logarithmic length. Section 2.6 describes our proposed approach.

Scheduling. To avoid a large movement of memoized data, it is important to schedule a task on the machine that stores the memoized results that are being reused. To ensure this, we introduce a modification to the scheduler used by Hadoop, in order to incorporate a notion of *affinity*. The new scheduler takes into account affinities between machines and tasks by keeping a record of which nodes have executed which tasks in previous runs. This allows for scheduling tasks in a way that decreases the movement of memoized results, but at the cost of a potential degradation of job performance due to stragglers [150]. This is because a strict affinity of tasks results in deterministic scheduling, which prevents a lightly loaded node from performing work when the predetermined node is heavily loaded. Our



(a) Fixed-size chunking in HDFS (b) Content-based chunking in Inc-HDFS

(c) Example of stable partitioning

Figure 2.2: Chunking strategies in HDFS and Inc-HDFS

scheduler therefore needs to strike a balance between work stealing and affinity of memoized results. Section 2.7 describes our modified scheduler.

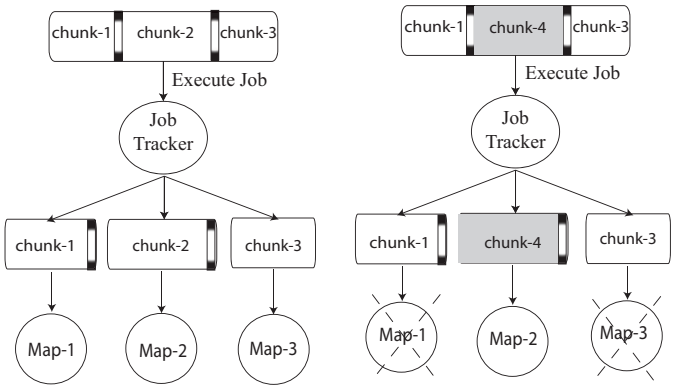
2.5 Incremental HDFS

In this section we present Incremental HDFS (Inc-HDFS), a distributed file system that enables stable incremental computations in Incoop, while keeping the interface provided by HDFS. Inc-HDFS builds on HDFS, but modifies the way that files are partitioned into chunks to use content-based chunking, a technique that was introduced in LBFS [114] for data deduplication. At a high-level, content-based chunking defines chunk boundaries based on finding certain patterns in the input, instead of using fixed-size chunks. As such, insertions and deletions cause small changes to the set of chunks. In the context of MapReduce, this ensures that the input to Map tasks remains mostly unchanged, which translates into a stable recomputation. Figure 2.2 illustrates the differences in the strategies for determining chunk boundaries in HDFS and Inc-HDFS. To perform content-based chunking, we scan the entire file, examining the contents of a fixed-width window whose ini-

tial position is incremented one byte at a time. For each window, we compute its Rabin fingerprint, and if the fingerprint matches a certain pattern (called a *marker*) we place a chunk boundary at that position. (We explain how the marker is selected in § 3.3.) In addition, this approach is extended to avoid creating chunks that are too small or too large, which could affect the overheads and load balancing properties of MapReduce. (Note that all the system designer can tune is the likelihood of finding a marker, but the actual spacing depends on the input.) This is achieved by setting minimum and maximum chunk sizes: after we find a marker m_i at position p_i, we skip a fixed *offset* O and continue to scan the input after position $p_i + O$. In addition, we bound the chunk length by setting a marker after M content bytes even if no marker is found. Despite the possibility of affecting stability in rare cases, e.g., when skipping the offset leads to skipping disjoint sets of markers in two consecutive runs, we found this to be a very limited problem in practice.

An important design decision is whether to perform chunking during the creation of the input or when the input is read by the Map task. We chose the former because the cost of chunking can be amortized when chunking and producing the input data are done in parallel. This is relevant in cases where the generation of input data is not limited by the storage throughput.

In order to parallelize the chunking process on multicore machines, our implementation uses multiple threads, each of which starts the search for the marker at a different position. The markers that each thread finds cannot be used immediately to define the chunk boundaries, since some of them might have to be skipped due to the minimum chunk size. Therefore, we collect the markers in a centralized list, and scan the list to determine which markers are skipped; the remaining ones form the chunk boundaries. We next describe, how the MapReduce framework uses these chunks to control the granularity of the Map phase.



(a) Initial run (b) Incremental run

Figure 2.3: Incremental Map tasks

2.6 Incremental MapReduce

This section presents our design for incremental MapReduce computations. We split the presentation by describing the Map and Reduce phases separately.

Incremental Map. For the Map phase, the main challenges have already been addressed by Inc-HDFS, which partitions data in such a way that the input to Map tasks ensures stability and also allows for controlling the average granularity of the input that is provided to these tasks. In particular, this granularity can be adjusted by changing how likely it is to find a marker, and it should be set in a way that strikes a good balance between the following two characteristics: incurring the overhead associated with scheduling many Map tasks when the average chunk size is low, and having to recompute a large Map task if a small subset of its input changes when the average chunk size is large.

Therefore, the main job of Map tasks in Incoop is to implement task-level memoization. To do this, after a Map task runs, we store its results persistently (instead of discarding them after the job execution) and insert a corresponding reference to the result in the memoization server.

During incremental runs, Map tasks query the memoization server to determine if their output has already been computed. If so, they output the location of the memoized result, and conclude. Figure 2.3 illustrates this process: part (a) describes the initial run and part (b) describes the incremental run where chunk 2 is modified (and replaced by chunk 4) and the Map tasks for chunks 1 and 3 can reuse the memoized results.

Incremental Reduce. The Reduce task processes the output of the Map phase: each Reduce task has an associated key k, collects all the key-value pairs generated by all Map tasks for k, and applies the Reduce function. For efficiency, we apply two levels of memoization in this case. First, we memoize the inputs and outputs of the entire Reduce task to try to reuse these results in a single step. Second, we break down the Reduce phase into a Contraction phase followed by a smaller invocation of the Reduce function to address the stability issues we discussed.

The first level of memoization is very similar to that of Map tasks: the memoization server maintains a mapping from a hash of the input to the location of the result of the Reduce task. A minor difference is that a Reduce task receives input from several Map tasks, and as such the key of that mapping is the concatenation of the collision-resistant hashes all these outputs. For the Reduce task to compute this key, instead of immediately copying the output from all Map tasks, it fetches the hashes only to determine if the Reduce task can be skipped entirely. Only if this is not the case the data is transferred from Map to Reduce tasks.

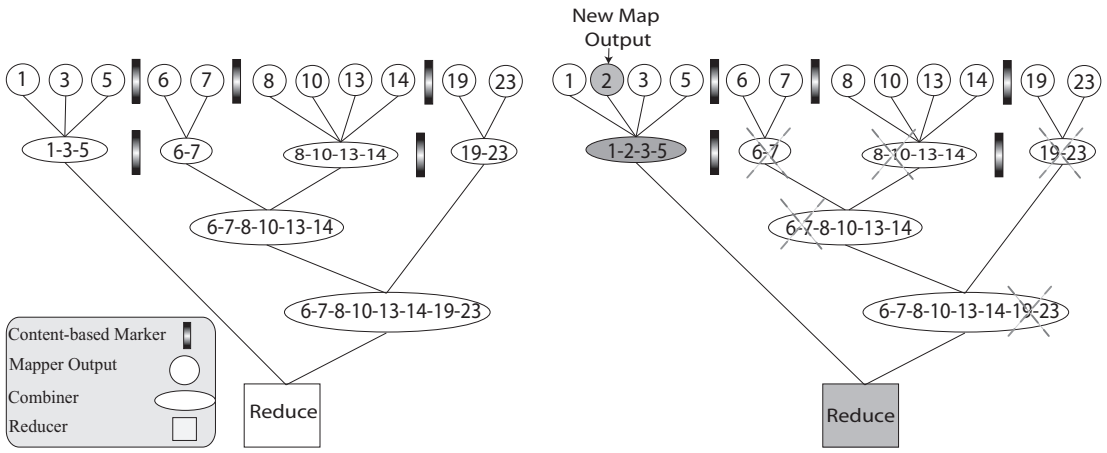
As we mentioned, this first level has the limitation that small changes in the input cause the entire Reduce task to be re-executed, which can result in work that is linear in the size of the original input, even if the delta in the input is small. In fact it may be argued that the larger the Reduce task the more likely it is that a part of its input may change. To prevent this stability problem, we need to find a way to control the granularity of the sub-computations in the Reduce phase, and orga-

nize these sub-computations in way that avoids creating a long dependence chain between sub-computations, otherwise a single newly computed sub-computation could also trigger a large amount of recomputation.

To reduce the granularity of Reduce tasks, we propose a new *Contraction Phase*, which is run by Reduce tasks. This new phase takes advantage of *Combiners*, a feature of the MapReduce frameworks [67], also implemented by Hadoop, which originally aims at saving bandwidth by offloading part of the computation performed by the Reduce task to the Map task. To this end, the programmer specifies a Combiner function, which is invoked by the Map task, and pre-processes a part of the Map output, i.e., a set of ⟨key,value⟩ pairs, merging them into a smaller number of pairs. The signature of the combiner function uses the same input and output type in order to be interposed between the Map and Reduce phase, its inputs and output arguments are a sequence of ⟨key,value⟩ pairs. In all the MapReduce applications we analyzed so far, the Combiners and the Reduce functions perform similar work.

The Contraction phase uses Combiners to break up Reduce tasks into several applications of the Combine function. In particular, we start by splitting the Reduce input into chunks, and apply the Combine function to each chunk. Then we recursively form chunks from the aggregate result of all the Combine invocations and apply the Combine function to these new chunks. The data size gets smaller in each level, and, in the last level, we apply the Reduce function to the output of all the Combiners from the second to last level.

Given the signature of Combiner functions we described before, it is syntactically correct to interpose any number of Combiner invocations between the Map and Reduce functions. However, semantically, Combiners are invoked by the MapReduce or Hadoop frameworks at most once per key/value pair that is output by a Map task, and therefore MapReduce programs are only required to ensure the



(a) First run (b) Incremental run

Figure 2.4: Stability of the Contraction phase

correctness of the MapReduce computation for a single Combiner invocation, that is:

$$R \circ C \circ M = R \circ M$$

where R, C, and M represent the Reduce, Combiner and Map function, respectively. Our new use of Combiner functions introduces a different requirement, namely:

$$R \circ C^n \circ M = R \circ M, \forall n > 0$$

It is conceivable to write a Combiner that meets the original requirement but not the new one. However, we found that, in practice, all of the Combiner functions we have seen obey the new requirement.

Stability of the Contraction phase. When deciding how to partition the input to the Contraction phase, the same issue that was faced by the Map phase arises: if a part of the input to the Contraction phase is removed or a new part is added, then a fixed-size partitioning of the input would not ensure the stability of the dependence graph. This problem is illustrated in Figure 2.4, which shows two consecutive runs of a Reduce task, where a Map task (#2) produces in the second but not in the first run a value associated with the key being processed by this Reduce task. In this case, a partitioning of the input into groups with a fixed number of

input files would cause all groups of files to become different from one run to the next.

To solve this, we again employ content-based chunking, which is applied to every level of the tree of combiners that forms the Contraction phase. The way we perform content-based chunking in the Contraction phase differs slightly from the approach we took in Inc-HDFS, for both efficiency and simplicity reasons. In particular, given that the Hadoop framework splits the input to the Contraction phase into multiple files coming from different Mappers, we require chunk boundaries to be at file boundaries (in other words, the unit of chunking is a sequence of Mapper output files). This way we leverage the existing input partitioning, which not only simplifies the implementation, but also avoids reprocessing this input. This is because we can use the hash of each input file to determine if a marker is present, namely by testing if the hash modulo a pre-determined integer M is equal to a constant $k < M$.

Figure 2.4 also illustrates the importance of content-based chunking. In this example, the marker that delimits the boundaries between groups of input files form a chunk is present only in outputs #5, 7, and 14. Therefore, inserting a new map output will change the first group of inputs but none of the remaining ones. This figure also illustrates how this change propagates to the output: it leads to a new Combiner invocation (labelled 1-2-3-5) and the final Reduce invocation. For all the remaining Combiners we can reuse their memoized outputs without re-executing them.

2.7 Memoization Aware Scheduler

The Hadoop scheduler assigns Map and Reduce tasks to nodes for efficient execution, taking into account machine availability, cluster topology, and the locality

of input data. The Hadoop scheduler, however, is not well-suited for incremental computations because it does not consider the locality of memoized results.

To enable efficient reuse of previously computed intermediate results, Reduce tasks should preferentially be scheduled on the node where some or all of the memoized results they use are stored. This is important in case the Contraction phase needs to run using a combination of newly computed and memoized results, which happens when only a part of its inputs has changed. In addition to this design goal, the scheduler also has to provide some flexibility by allowing tasks to be scheduled on nodes that do not store memoized results, otherwise it can lead to the presence of stragglers, i.e., individual poorly performing nodes that can drastically delay the overall job completion [150].

Based on these requirements, Incoop includes a new memoizationaware scheduler that strikes a balance between exploiting the locality of memoized results and incorporating some flexibility to minimize the straggler effect. The scheduler tries to implement a location-aware policy that prevents the unnecessary movement of data, but at the same time it implements a simple work-stealing algorithm to adapt to varying resource availability. The scheduler works by maintaining a separate task queue for each node in the cluster (instead of a single task queue for all nodes), where each queue contains the tasks that should run on that node in order to maximally exploit the location of memoized results. Whenever a node requests more work, the scheduler dequeues the first task from the corresponding queue and assigns the task to the node for execution. In case the corresponding queue for the requesting node is empty, the scheduler tries to *steal* work from other task queues. The scheduling algorithm searches the task queues of other nodes, and steals a pending task from the task queue with maximum length. If there are multiple queues of maximum length, the scheduler steals the task that has the least amount of memoized intermediate results. Our scheduler thus takes the location

of the memoized results into account, but falls back to a work stealing approach to avoid stragglers and nodes running idle. Our experimental evaluation (Section 2.9.4) shows the effectiveness of this approach.

2.8 Implementation

We built our prototype of Incoop based on Hadoop-0.20.2. We implemented Inc-HDFS by extending HDFS with stable input partitioning, and incremental MapReduce by extending Hadoop with a finer granuality control mechanism and the memoization-aware scheduler.

The Inc-HDFS file system provides the same semantics and interface for accessing all native HDFS calls. It employs a content-based chunking scheme which is computationally more expensive than the fixed-size chunking used by HDFS. As described in §2.5, the implementation minimizes the overhead using two optimizations: (i) we skip parts of the file contents when searching for chunk markers, in order to reduce the number of fingerprint computations and enforce a minimum chunk size; and (ii) we parallelize the search for markers across multiple cores. To implement these optimizations, the data uploader client skips a fixed number of bytes after the last marker is found, and then spawns multiple threads that each compute the Rabin fingerprints over a sliding window on different parts of the content. For our experiments, we set the number of bytes skipped to 40MB unless otherwise stated.

We implemented the memoization server using a wrapper around Memcached v1.4.5, which provides an in-memory key-value store. Memcached runs as a daemon process on the name node machine that acts as a directory server in Hadoop. Intermediate results memoized across runs are stored on Inc-HDFS with the replication factor set to 1, and, in case of data loss, the intermediate results are recom-

puted. A major issue with any implementation of memoization is determining which intermediate results to remember and which intermediate results to purge. As in self-adjusting computation approaches, our approach is to cache the *fresh* results from the "last run", i.e., those results that were generated or used by the last execution, and purge all the other *obsolete* results. This suffices to obtain the efficiency improvements shown in §2.9.5. We implement this strategy using a simple garbage collector that visits all cache entries and purges the obsolete results.

2.9 Evaluation

We evaluate the effectiveness of Incoop for a variety of applications implemented in the traditional MapReduce programming model. In particular, we will answer the following questions:

- What performance benefits does Incoop provide for incremental workloads compared to the unmodified Hadoop implementation? (§2.9.3)
- How effective are the optimizations we propose in improving the overall performance of Incoop? (§2.9.4)
- What overheads does the memoization in Incoop impose when tasks are executed for the first time? (§2.9.5)

2.9.1 Applications and Data Generation

For the experimental evaluation, we use a set of applications in the fields of machine learning, natural language processing, pattern recognition, and document analysis. Table 2.1 lists these applications. We chose these applications to demonstrate Incoop's ability to efficiently execute both data-intensive (`WordCount, Co-Matrix,`

Application	Description
`K-Means`	*K*-means clustering is a method of cluster analysis for partitioning n data points into k clusters, in which each observation belongs to the cluster with the nearest mean.
`Word-Count`	Word count determines the frequency of words in a document.
`KNN`	*K*-nearest neighbors classifies objects based on the closest training examples in a feature space.
`CoMatrix`	Co-occurrence matrix generates an $N \times N$ matrix, where N is the number of unique words in the corpus. A cell m_{ij} contains the number of times word w_i co-occurs with word w_j.
`BiCount`	Bigram count measures the prevalence of each subsequence of two items within a given sequence.

Table 2.1: Applications used in the performance evaluation of Incoop

`BiCount`), and computation-intensive (`KNN` and `K-Means`) jobs. For the data-intensive applications, the computational work done by the Map and the Reduce phases is roughly the same. Whereas, for compute-intensive applications the computational work is mostly done in the Map phase only.

The three data-intensive applications take as input documents written in a natural language. In our benchmarks, we use a publicly available dataset with the contents of Wikipedia.[2] The computation-intensive applications take as input a set of points in a d-dimensional space. We generate this data synthetically by uniformly randomly selecting points from a 50-dimensional unit cube. To ensure reasonable running times, we chose all the input sizes such that the running time of each job would be around one hour. We note that we did not make any changes to the original code for all applications,.

2.9.2 Measurements

Metrics: work and time. For comparing different runs, we consider two types of measures, work and time, which are standard measures for comparing efficiency

[2]Wikipedia data-set: `http://wiki.dbpedia.org/`

in parallel applications. *Work* refers to the total amount of computation performed by all tasks and measured as the total running time of all tasks. *Time* refers to the amount of (end-to-end) time that it takes to complete a parallel computation. Improvements in total work often directly lead to improvements in time but also in the consumption of other resources, e.g., processors, power, etc. As we describe in our experiments, our approach reduces work by avoiding unnecessary computations, which translates to improvements in time (and use of other resources).

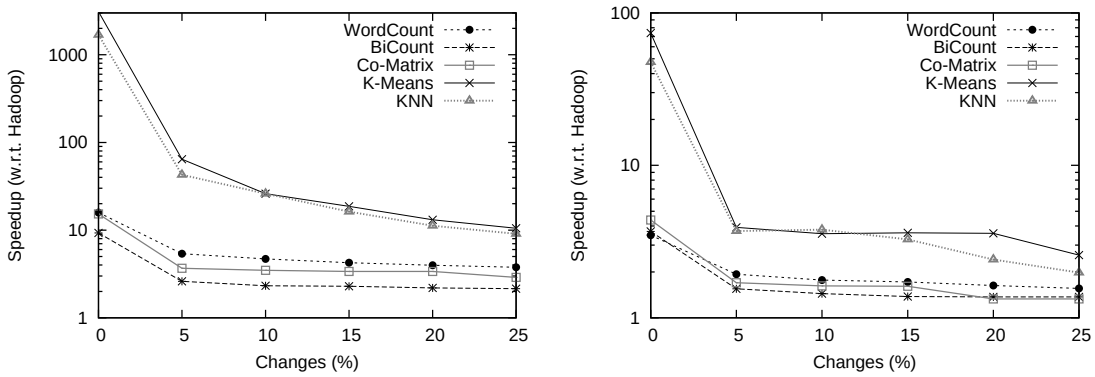
Initial run and dynamic update. The most important measurements we perform involve the comparison of the execution of a MapReduce job with Hadoop vs. with Incoop. For the Incoop measurements, we consider two different runs. The *initial run* refers to a run starting with an empty memoization server that has no memoized results. Such a run executes all tasks and populates the memoization server by storing the performed computations and the location of their results. The *dynamic update* refers to a run of the same job with a modified input, but that happens after the initial run, avoiding re-computation when possible. It also memoizes the intermediate results for newly executed tasks for the next incremental run.

Speedup. To assess the effectiveness of dynamic updates, we measure the work and time after modifying varying percentages of the input data and comparing them to those for performing the same computation with Hadoop. We refer to the ratio of the Hadoop run to the incremental run (Incoop dynamic update) as *speedup* (in work and in time). When modifying p% of the input data, we randomly chose p% of the input chunks and replaced them with new chunks of equal size and newly generated content.

Hardware. Our measurements were gathered using a cluster of 20 machines, running Linux with kernel 2.6.32 in 64-bit mode, connected with gigabit ethernet. The name node and the job tracker ran on a master machine which was equipped with

a 12-core Intel Xeon processor and 12 GB of RAM. The data nodes and task trackers ran on the remaining 19 machines equipped with AMD Opteron-252 processors, 4GB of RAM, and 225GB drives. We configured each task tracker per worker machine to use in total four worker threads: two threads for Map tasks and two threads for Reduce tasks.

2.9.3 Performance Gains



(a) Work speedups versus change size

(b) Time speedups versus change size

Figures 2.5(a) and 2.5(b) show the work and time speedups, which are computed as the ratio between the work and time of a dynamic run using Incoop and those of Hadoop. From these experimental results we can observe the following: (i) Incoop achieves substantial performance gains for all applications when there are incremental changes to the input data. In particular, work and time speedups vary between 3-fold and 1000-fold for incremental modifications ranging from 0% to 25% of data. The speedups with 0% changes mean that the incremental run is run with the same input without any changes. Therefore, speedups peak at 0% changes because we can reuse the entire work from the initial run. (ii) We observe higher speedups for computation-intensive applications (`K-Means`, `KNN`) than for data-intensive applications (`WordCount`, `Co-Matrix`, and `BiCount`). This is because for the data-intensive application, we require large amounts of data

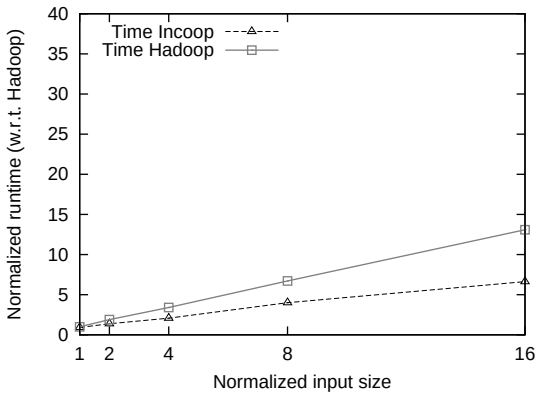


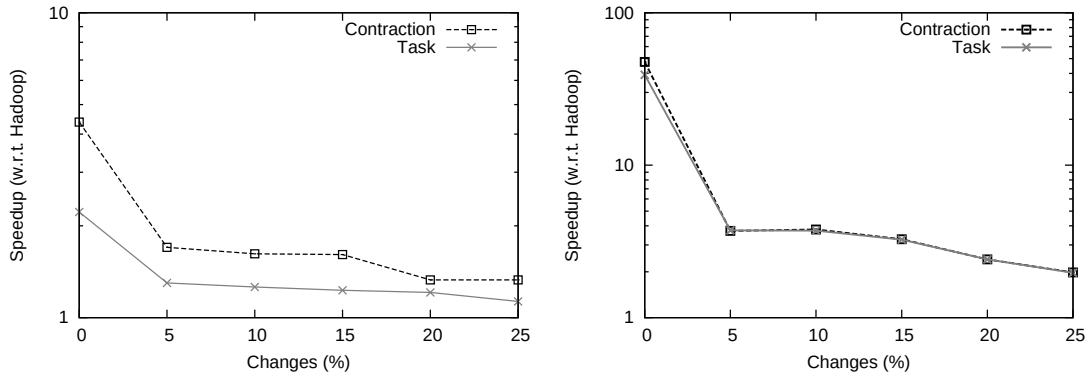
Figure 2.5: Co-Matrix: Time versus input size

movement to restore the memoized intermediate results. (iii) Both work and time speedups decrease as the size of the incremental change increases, because larger changes allow fewer computation results from previous runs to be reused. With very small changes, however, speedups in total work are not fully translated into speedups in parallel time; this is expected because decreasing the total amount of work dramatically (e.g., by a factor 1000) reduces the amount of parallelism, causing the scheduling overheads to be larger. As the size of the incremental change increases, the gap between the work speedup and time speedup closes quickly.

The previous examples all consider fixed-size inputs. We experimented with other input sizes. This is shown in Figure 2.5, which illustrates the time to run Incoop and Hadoop using the Co-Matrix application, and for a modification of a single chunk. This figure shows that the relative improvements hold for various different input sizes.

2.9.4 Effectiveness of Optimizations

We evaluate the effectiveness of the optimizations in improving the overall performance of Incoop by considering (i) the granularity control with the introduction of the Contraction phase; and (ii) the scheduler modifications to minimize unnecessary data movement.



(a) Co-occurrence Matrix (b) k-NN Classifier

Figure 2.6: Performance gains comparison between Contraction and task variants

Granularity control. To evaluate the effectiveness of the Contraction phase, we consider the two different levels of memoization in Incoop: (i) using only the coarse-grained, task-level memoization performed in the implementation, denoted as `Task`, and (ii) adding the fine-grained approach that also includes the Contraction phase in the implementation, denoted as `Contraction`. Figure 2.6 shows our time measurements with `CoMatrix` as a data-intensive application and `KNN` as a computation-intensive application. The effect of the Contraction phase is negligible with `KNN` but significant in `CoMatrix`. The reason for the negligible improvements with `KNN` is that in this application, Reduce tasks perform relatively inexpensive work and thus benefit little from the Contraction phase. Thus, even when not helpful, the Contraction phase does not degrade efficiency.

Scheduler modification. We now evaluate the effectiveness of the scheduler modification in improving the performance of Incoop. The Incoop scheduler avoids unnecessary data movement by scheduling tasks on the nodes where intermediate results from previous runs are stored. Also, the scheduler employs a work stealing algorithm that allows some task scheduling flexibility to prevent nodes from running idle when runnable tasks are waiting. We show the performance comparison between the Hadoop scheduler and the Incoop scheduler in Figure 2.7, where

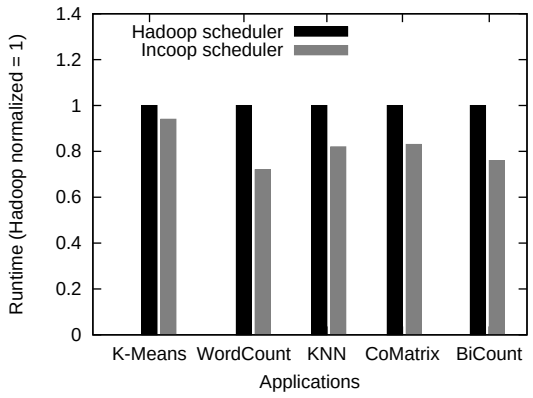


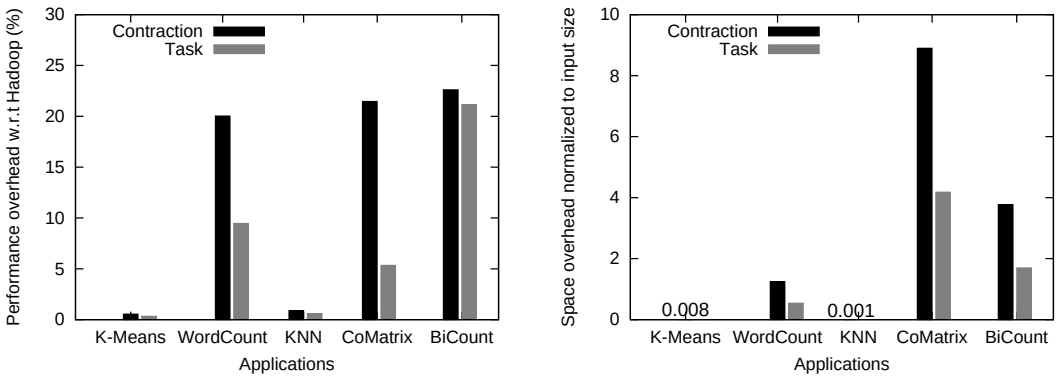
Figure 2.7: Effectiveness of scheduler optimizations.

the Y-axis shows runtime relative to the Hadoop scheduler. The Incoop scheduler saves around 30% of time for data-intensive applications, and almost 15% of time for compute-intensive applications, which supports the necessity and effectiveness of location-aware scheduling for memoization.

2.9.5 Overheads

The memoization performed in Incoop introduces runtime overheads for the initial run when no results from previous runs can be reused. Also, memoizing intermediate task results imposes an additional space usage. We measured both types, performance and space overhead, for each application and present the results in Figure 2.8.

Performance overhead. We measure the worst-case performance overhead by capturing the runtime for the initial run. Figure 2.8(a) depicts the performance penalty for both the `Task` and the `Contraction` memoization based approach. The overhead varies from $5\% - 22\%$, and, as expected, it is lower for computation intensive applications such as `K-Means` and `KNN`, since their run-time is dominated by the actual processing time rather than storing, retrieving and transferring data. For the data intensive applications such as `WordCount, Co-Matrix` and `BiCount`, the



(a) Performance overhead for the first job run

(b) Space overhead

Figure 2.8: Overheads imposed by Incoop in comparison to Hadoop

first run with `Task` level memoization is faster than `Contraction` memoization. This difference in performance can be attributed to the extra processing overheads for all levels of the tree formed in the Contraction phase. Importantly, this performance overhead is a one-time cost and the subsequent runs benefit from a high speedup.

Space overhead. We measure the space overhead by quantifying the space used for remembering the intermediate computation results for the initial run. Figure 2.8(b) illustrates the space overhead as a factor of the input size with Task- and Contraction-level memoization. The results show that the Contraction-level memoization requires more space, which was expected because it stores results for all levels of the Contraction tree. Overall, space overhead varies substantially depending on the application, and can be as high as 9X (`CoMatrix` application) of the input size. However, our approach for garbage collection prevents the storage utilization from growing over time.

2.10 Case Studies

The success of the MapReduce paradigm enables our approach to transparently benefit an enormous variety of bulk data processing workflows. In particular, and aside from the large number of existing MapReduce programs, MapReduce is also being used as an execution engine for other systems. In this case, Incoop will also transparently benefit programs written for these systems.

In this section, we showcase two workflows where we use Incoop to transparently benefit systems from efficient incremental processing in their context, namely *incremental log processing* and *incremental query processing*.

2.10.1 Incremental Log Processing

Log processing is an essential workflow in Internet companies, where various logs are often analyzed in multiple ways on a daily basis [111]. For example, in the area of click log analysis, traces collected from various web server logs are aggregated in a single repository and then processed for various purposes, from simple statistics like counting clicks per user, or more complex analyses like click sessionization.

To perform incremental log processing, we integrated Incoop with Apache Flume [3] – a distributed and reliable service for efficiently collecting, aggregating, and moving large amounts of log data. In our setup, Flume aggregates the data and dumps it into the Inc-HDFS repository. Then, Incoop performs the analytic processing incrementally, leveraging previously computed intermediate results.

We evaluate the performance of using Flume in conjunction with Incoop for incremental log processing by comparing its runtime with the corresponding runtime when using Hadoop. For this experiment, we perform document analysis on an initial set of logs, and then append new log entries to the input, after which

[3] Apache Flume: `https://github.com/cloudera/flume`

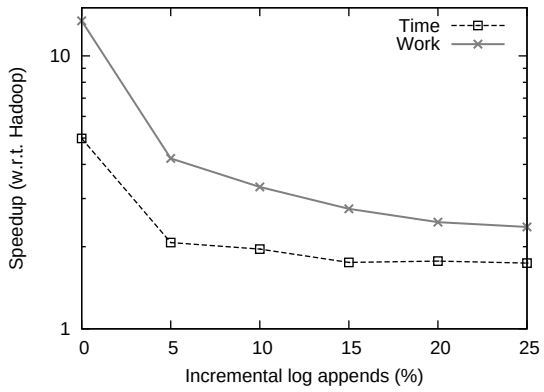


Figure 2.9: Speedup for incremental log processing

we process the resulting larger collection of logs incrementally. In Figure 2.9, we depict the speedup for running Incoop as a function of the size of the new logs that are appended after the first run. Incoop achieves a speedup of a factor of 4 to 2.5 with respect to Hadoop when processing incremental log appends of a size of 5% to 25% of the initial input size, respectively.

2.10.2 Incremental Query Processing

We showcase incremental query processing as another workflow that exemplifies the potential benefits of Incoop. Query processing is an important medium in Internet companies for analyzing large data sets. Query processing frameworks consist of a high-level language, similar to SQL, for easily expressing data analysis programs.

Query processing also follows incremental computing workflow, where the same query is processed frequently for an incrementally changing input data set [115]. We integrated Incoop with Pig to evaluate the feasibility of incremental query processing. Pig [116] is a query processing platform to analyze large data sets built upon Hadoop. Pig provides Pig Latin, an easy-to-use high-level query language similar to SQL. The ease of programming and scalability of Pig made the system

very popular for very large data analysis tasks, which are conducted by major Internet companies today.

Since Pig programs are compiled down to multi-staged MapReduce jobs, the integration of Incoop with Pig was seamless, just by using Incoop as the underlying execution engine for incrementally executing the multi-staged MapReduce jobs. We evaluate two Pig applications, word count and the PigMix[4] scalability benchmark, to measure the effectiveness of Incoop. We observe a runtime overhead of around 15% for first run, and a speedup of a factor of around 3 for an incremental run with unmodified input. The detailed result breakdown is shown in Table 2.2.

Application	**Features**	**M/R stages**	**Overhead**	**Speedup**
Word Count	Group_by, Order_by, Filter	3	15.65 %	2.84
PigMix scalabilty benchmark	Group_by, Filter	1	14.5 %	3.33

Table 2.2: Results for incremental query processing

2.11 Related Work

There are several systems for performing incremental parallel computations with large data sets. We broadly divide them into two categories: non-transparent and transparent approaches. Examples of non-transparent systems include Google's Percolator [121] which requires the programmer to write a program in an event-driven programming model based on observers. Observers are triggered by the system whenever user-specified data is modified. Observers in turn can modify other data forming a dependence chain that implements the incremental data processing. Similarly, continuous bulk processing (CBP) [110] proposes a new data-

[4]Apache PigMix: `http://wiki.apache.org/pig/PigMix`

parallel programming model, which offers primitives to store and reuse prior state for incremental processing. There are two drawbacks to these approaches, both of which are addressed by our proposal. The first is that they depart from the MapReduce programming paradigm and therefore require changes to the large existing base of MapReduce programs. The second, more fundamental problem is that they require that programmer to devise a dynamic algorithm in order to efficiently process data in an incremental manner.

Examples of transparent approaches include DryadInc [122], which extends Dryad to automatically identify redundant computations by caching previously executed tasks. One limitation of this basic approach is that it can only reuse common identical sub-DAGs of the original computation, which can be insufficient to achieve efficient updates. To improve efficiency the paper suggests the programmers specify additional merge functions. Another similar system called Nectar [84] caches prior results at the coarser granularity of entire LINQ sub-expressions. The technique used to achieve this is to automatically rewrite LINQ programs to facilitate caching. Finally, although not fully transparent, Haloop [56] provides task-level memoization techniques for memoization in the context of iterative data processing applications. The major difference between the aforementioned transparent approaches and our proposal is that we use a well-understood set of principles from related work to eliminate the cases where task-level memoization provides poor efficiency. To this end, we provide techniques for increasing the effectiveness of task-level memoization via stable input partitions and by using a more fine-grained memoization strategy than the granularity of Map and Reduce tasks.

Our own short position paper [51] makes the case for applying techniques inspired by self-adjusting computation to large-scale data processing in general, and uses MapReduce as an example. This position paper, however, models MapReduce in a sequential, single-machine implementation of self-adjusting computa-

tion called CEAL [87], and does not offer anything close to a full-scale distributed design and implementation such as we describe here.

The Hadoop online prototype (HOP) [62] extends the Hadoop framework to support pipelining between the map and reduce tasks, so that reducers start processing data as soon as it is produced by mappers. This enables two new features in the framework. First, it can generate an approximate answer before the end of the computation (online aggregation) and second, it can support continuous queries, where jobs run continuously, and process new data as it arrives.

NOVA [115] is a workflow manager recently proposed by Yahoo!, designed for the incremental execution of Pig programs upon continually-arriving data. NOVA introduces a new layer called the workflow manager on top of the Pig/ Hadoop framework. Much like the work on incremental view maintenance, the workflow manager rewrites the computation to identify the parts of the computation affected by incremental changes and produce the necessary update function that runs on top of the existing Pig/Hadoop framework. However, as noted by the authors of NOVA, an alternative, more efficient design would be to modify the underlying Hadoop system to support this functionality. In our work, and particularly with our case study of incremental processing of Pig queries, we explore precisely this alternative design of adding lower-level support for reusing previous results. Furthermore, our work is broader in that it transparently benefits all MapReduce computations, and not only continuous Pig queries.

We provide a detailed comparison of our approach of using contraction with modern streaming systems such as D-streams [149], Naiad [113], etc. in Chapter 4.

2.12 Limitations and Future Work

While Incoop is a significant step towards transparent support for incremental computation, plenty of opportunities remain to further increase its efficiency.

Firstly, Incoop is designed to detect arbitrary changes in the input and update the output accordingly. For that Incoop relies on Inc-HDFS, which uses content-based chunking to detect changes in the input data across successive runs of MapReduce jobs. Content-based chunking is a computationally demanding task because chunking methods need to scan the entire file contents, computing a fingerprint over a sliding window of the data. Therefore, as we scale the system to handle increasing amounts of data, we need to address this computational bottleneck. We address this limitation using Shredder, as described in Chapter 3.

Secondly, since Incoop is designed to handle arbitrary changes to the input, the Contraction tree is not optimized to perform efficient change propagation. In particular, the Contraction tree does not construct the dependency graph of subcomputations explicitly, and resort solely to reuse of memoized results for updating the output. While this approach simplifies the design and the implementation, it can yield asymptotically suboptimal performance, because it requires touching all subcomputations (for the purposes of memoization and reuse) even if they may not be affected by the input modifications. For cases with structured changes in the input, we can further optimize the Contraction tree to perform change propagation. We address this limitation using Slider, as described in Chapter 4.

2.13 Summary

In this chapter, we presented Incoop, a novel MapReduce framework for large-scale incremental computations. Incoop is based on several novel techniques to

maximize the reuse of results from a previous computation. In particular, Incoop incorporates content-based chunking to the file system to detect incremental changes in the input file and to partition the data so as to maximize reuse; it adds a Contraction phase to control the granularity of tasks in the Reduce phase, and a new scheduler that takes the location of previously computed results into account. We implemented Incoop as an extension to Hadoop. Our performance evaluation shows that Incoop can improve efficiency in incremental runs (the common case), at a modest cost in the initial, first run (uncommon case) where no computations can be reused.

CHAPTER 3

Shredder: Incremental Storage

In this chapter, we describe the design, implementation and evaluation of Shredder, a high performance content-based chunking library for supporting incremental storage. Shredder was initially designed to accelerate content-based chunking in Inc-HDFS for incremental computation. However, our design of Shredder library is generic to accelerate incremental computation as well as incremental storage systems. Shredder exploits the massively parallel processing power of GPUs to overcome the CPU bottlenecks of content-based chunking in a cost-effective manner.

This chapter is organized as follows. We first motivate the design of Shredder in Section 3.1. We next briefly highlight the contributions of Shredder in Section 3.2. In Section 3.3, we provide background on content-based chunking, and discuss specific architectural features of GPUs. An overview of the GPU acceleration framework and its scalability challenges are covered in Section 3.4. Section 3.5 presents present a detailed system design, namely several performance optimizations for increasing Shredder's throughput. We present the implementation and evaluation of Shredder in Section 3.6. We present two case studies for Shredder in Section 3.8. We discuss the related work in Section 3.9. Finally, we present the limitation of Shredder in Section 3.10, and conclude in Section 3.11.

3.1 Motivation

With the growth in popularity of Internet services, online data stored in data centers is increasing at an ever-growing pace. In 2010 alone, mankind is estimated to have produced 1,200 exabytes of data [19]. As a result of this "data deluge," managing storage and computation over this data has become one of the most challenging tasks in data center computing.

A key observation that allows us to address this challenge is that a large fraction of the data that is produced and the computations performed over this data are redundant; hence, *not* storing redundant data or performing redundant computation can lead to significant savings in terms of both storage and computational resources. To make use of redundancy elimination, there exist a series of research and product proposals (detailed in §3.9) for performing *data deduplication* and *incremental computation*, which avoid storing or computing tasks based on redundant data, respectively.

Both data deduplication schemes and incremental computation rely on storage systems to detect duplicate content. In particular, the most effective way to perform this detection is using *content-based chunking*, a technique that was pioneered in the context of the LBFS [114] file system, where chunk boundaries within a file are dictated by the presence of certain content instead of a fixed offset. For instance, Incoop relies on Inc-HDFS, which uses content-based chunking to detect changes in the input data across successive runs of MapReduce jobs.

Even though content-based chunking is useful, it is a computationally demanding task. Chunking methods need to scan the entire file contents, computing a fingerprint over a sliding window of the data. This high computational cost has caused some systems to simplify the fingerprinting scheme by employing sampling techniques, which can lead to missed opportunities for eliminating redun-

dancies [30]. In other cases, systems skip content-based chunking entirely, thus forgoing the opportunity to reuse identical content in similar, but not identical files [85]. Therefore, as we get flooded with increasing amounts of data, addressing this computational bottleneck becomes a pressing issue in the design of storage systems for data center-scale systems.

3.2 Contributions

In this chapter, we present Shredder, a system for performing efficient content-based chunking to support scalable incremental storage and computation. Shredder builds on the observation that neither the exclusive use of multicore CPUs nor the use of specialized hardware accelerators is sufficient to deal with large-scale data in a cost-effective manner: multicore CPUs alone cannot sustain a high throughput, whereas the specialized hardware accelerators lack programmability for other tasks and are costly. As an alternative, we explore employing modern GPUs to meet these high computational requirements (while, as evidenced by prior research [89, 96], also allowing for a low operational cost). The application of GPUs in this setting, however, raises a significant challenge — while GPUs have shown to produce performance improvements for computation intensive applications, where CPU dominates the overall cost envelope [89, 90, 96, 137, 138], it was unclear when we started this work whether GPUs are equally as effective for data intensive applications, which need to perform large data transfers for a significantly smaller amount of processing.

To make the use of GPUs effective in the context of storage systems, we designed several novel techniques, which we apply to two proof-of-concept applications. In particular, Shredder makes the following technical contributions:

GPU acceleration framework. We identified three key challenges in using GPUs for data intensive applications, and addressed them with the following techniques:

- **Asynchronous execution.** To minimize the cost of transferring data between host (CPU) and GPU, we use a double buffering scheme. This enables GPUs to perform computations while simultaneously data is transferred in the background. To support this background data transfer, we also introduce a ring buffer of pinned memory regions.
- **Streaming pipeline.** To fully utilize the availability of a multicore architecture at the host, we use a pipelined execution for the different stages of content-based chunking.
- **Memory coalescing.** Finally, because of the high degree of parallelism, memory latencies in the GPU will be high due to the presence of random access across multiple bank rows of GPU memory, which leads to a higher number of conflicts. We address this problem with a cooperative memory access scheme, which reduces the number of fetch requests and bank conflicts.

We implemented Shredder as a generic C++/CUDA library. We also present two case study applications of Shredder to accelerate storage systems. The first case study is the integration of Shredder with Inc-HDFS to accelerate incremental computation. The second case study is a backup architecture for a cloud environment, where VMs are periodically backed up. We use Shredder on a backup server and use content-based chunking to perform efficient deduplication and significantly improve backup bandwidth.

3.3 Background

In this section, we first present background on content-based chunking, to explain its cost and potential for parallelization. We then provide a brief overview of the

massively parallel compute architecture of GPUs, namely their memory subsystem and its limitations.

3.3.1 Content-based Chunking

Identification of duplicate data blocks has been used for deduplication systems in the context of both storage [114, 128] and incremental computation frameworks [53]. For storage systems, the duplicate data blocks need not to be stored and, in the case of incremental computations, a sub-computation based on the duplicate content may be reused. Duplicate identification essentially consists of:

1. Chunking: This is the process of dividing the data set into chunks in a way that aids in the detection of duplicate data.
2. Hashing: This is the process of computing a collision-resistant hash of the chunk.
3. Matching: This is the process of checking if the hash for a chunk already exists in the index. If it exists then there is a duplicate chunk, else the chunk is new and its hash is added to the index.

In this chapter, we focus on the design of chunking schemes (step 1), since this can be, in practice, one of the main bottlenecks of a system that tries to perform this class of optimizations [30, 85]. Thus we begin by giving some background on how chunking is performed.

One of the most popular approaches for content-based chunking is to compute a Rabin fingerprint [129] over sliding windows of w contiguous bytes. The hash values produced by the fingerprinting scheme are used to create chunk boundaries by starting new chunks whenever the computed hash matches one of a set of markers (e.g., its value $mod\ p$ is lower or equal to a constant). In more detail, given a

w-bit sequence, it is represented as a polynomial of degree $w-1$ over the finite field $GF(2)$:

$$f(x) = m_0 + m_1 x + \cdots + m_{w-1} x^{w-1} \tag{3.1}$$

Given this polynomial, an irreducible polynomial $div(x)$ of degree k is chosen. The fingerprint of the original bit sequence is the remainder $r(x)$ obtained by division of $f(x)$ using $div(x)$. A chunk boundary is defined when the fingerprint takes some pre-defined specific value in a set of values called markers. In addition, practical schemes define a minimum *min* and maximum *max* chunk size, which implies that after finding a marker the fingerprint computation can skip *min* bytes, and that a marker is always set when a total of *max* bytes (including the skipped portion) have been scanned without finding a marker. The minimum size limits the metadata overhead for index management and the maximum size limits the size of the RAM buffers that are required. Throughout the rest of the paper, we will use $min = 0$ and $max = \infty$ unless otherwise noted.

Rabin fingerprinting is computationally very expensive. To minimize the computation cost, there has been work on reducing chunking time by using sampling techniques, where only a subset of bytes are used for chunk identification (e.g., SampleByte [30]). However, such approaches are limiting because they are suited only for small sized chunks, as skipping a large number of bytes leads to missed opportunities for deduplication. Thus, Rabin fingerprinting still remains one of the most popular chunking schemes, and reducing its computational cost presents a fundamental challenge for improving systems that make use of duplicate identification.

When minimum and maximum chunk sizes are not required, chunking can be parallelized in a way that different threads operate on different parts of the data completely independent of each other, with the exception of a small overlap of the size of the sliding window (w bytes) near partition boundaries. Using *min* and

max chunk sizes complicates this task, though schemes exist to achieve efficient parallelization in this setting [105, 107].

3.3.2 General-Purpose Computing on GPUs

GPU architecture. GPUs are highly parallel, multi-threaded, many-core processors with tremendous computational power and very high memory bandwidth. The high computational power is derived from the specialized design of GPUs, where more transistors are devoted to simple data processing units (ALUs) rather than used to integrate sophisticated pre-fetchers, control flows and data caches. Hence, GPUs are well-suited for data-parallel computations with high arithmetic intensity rather than data caching and flow control.

Figure 3.1 illustrates a simplified architecture of a GPU. A GPU can be modeled as a set of Streaming Multiprocessors (SMs), each consisting of a set of scalar processor cores (SPs). An SM works as SIMT (Single Instruction, Multiple Threads), where the SPs of a multiprocessor execute the same instruction simultaneously but on different data elements. The data memory in the GPU is organized as multiple hierarchical spaces for threads in execution. The GPU has a large high-bandwidth device memory with high latency. Each SM also contains a very fast, low latency on-chip shared memory to be shared among its SPs. Also, each thread has access to a private local memory.

Overall, a GPU architecture differs from a traditional processor architecture in the following ways: (*i*) an order of magnitude higher number of arithmetic units; (*ii*) minimal support for prefetching and buffers for outstanding instructions; (*iii*) high memory access latencies and higher memory bandwidth.

Programming model. The CUDA [11] programming model is amongst the most popular programming models to extract parallelism and scale applications on GPUs. In this programming model, a host program runs on the CPU and launches a ker-

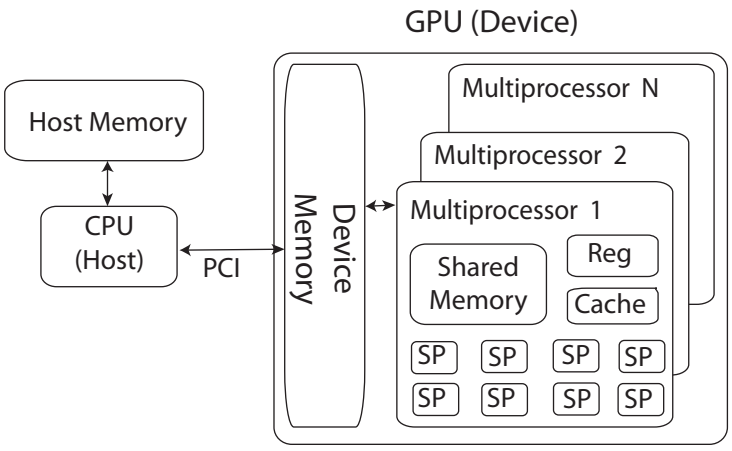


Figure 3.1: A simplified view of the GPU architecture.

nel program to be executed on the GPU device in parallel. The kernel executes as a grid of one or more thread blocks, each of which is dynamically scheduled to be executed on a single SM. Each thread block consists of a group of threads that cooperate with each other by synchronizing their execution and sharing multiprocessor resources such as shared memory and registers. Threads within a thread block get executed on a multiprocessor in scheduling units of 32 threads, called a warp. A half-warp is either the first or second half of a warp.

3.3.3 SDRAM Access Model

Offloading chunking to the GPU requires a large amount of data to be transferred from the host to the GPU memory. Thus, we need to understand the performance of the memory subsystem in the GPU, since it is critical to chunking performance.

The global memory in the Nvidia C2050 GPU is GDDR5, which is based on the DDR3 memory architecture [5]. Memory is arranged into banks and banks are organized into rows. Every bank also has a sense amplifier, into which a row must be loaded before any data from the row can be read by the GPU. Whenever a memory location is accessed, an *ACT* command selects the corresponding bank and brings the row containing the memory location into a sense amplifier. The

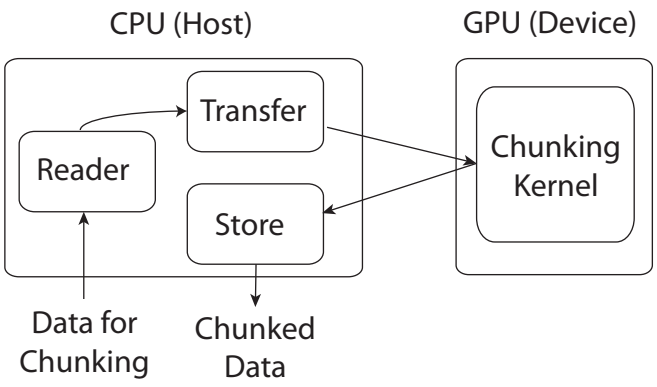


Figure 3.2: Basic workflow of Shredder .

appropriate word is then transferred from the sense amplifier. When an access to a second memory location is performed within the same row, the data is transferred directly from the sense amplifier. On the other hand, if the data is accessed from a different row in the bank, a PRE (pre-charge) command writes the previous data back from the sense amplifier to the memory row. A second ACT command is performed to bring the row into the sense amplifier.

Note that both ACT and PRE commands are high latency operations that contribute significantly to overall memory latency. If multiple threads access data from different rows of the same bank in parallel, that sense amplifier is continually activated (ACT) and pre-charged (PRE) with different rows, leading to a phenomenon called bank conflict. In particular, a high degree of uncoordinated parallel access to the memory subsystem is likely to result in a large number of bank conflicts.

3.4 Overview

In this section, we first present the basic design of Shredder . Next, we explain the main challenges in scaling up our basic design.

3.4.1 Basic GPU-Accelerated Framework

Figure 3.2 depicts the workflow of the basic design for the Shredder chunking service. In this initial design, a multithreaded program running in user mode on the host (i.e., on the CPU) drives the GPU-based computations. The framework is composed of four major modules. First, the *Reader* thread on the host receives the data stream (e.g., from a SAN), and places it in the memory of the host for content-based chunking. After that, the *Transfer* thread allocates global memory on the GPU and uses the DMA controller to transfer input data from the host memory to the allocated GPU (device) memory. Once the data transfer from the CPU to the GPU is complete, the host launches the *Chunking* kernel for parallel sliding window computations on the GPU. Once the chunking kernel finds all resulting chunk boundaries for the input data, the *Store* thread transfers the resulting chunk boundaries from the device memory to the host memory. When minimum and maximum chunk sizes are set, the *Store* thread also adjusts the chunk set accordingly. Thereafter, the *Store* thread uses an upcall to notify the chunk boundaries to the application that is using the Shredder library.

The chunking kernel is responsible for performing parallel content-based chunking of the data present in the global memory of the GPU. Accesses to the data are performed by multiple threads that are created on the GPU by launching the chunking kernel. The data in the GPU memory is divided into equal sized sub-streams, as many as the number of threads. Each thread is responsible for handling one of these sub-streams. For each sub-stream, a thread computes a Rabin fingerprint in a sliding window manner. In particular, each thread examines a 48-byte region from its assigned sub-stream, and computes the Rabin fingerprint for the selected region. The thread compares the resulting low-order 13 bits of the region's fingerprint with a pre-defined marker. This leads to an expected chunk size

Parameter	Value
GPU Processing Capacity	1030 GFlops
Reader (I/O) Bandwidth	2 GBps
Host-to-Device Bandwidth	5.406 GBps
Device-to-Host Bandwidth	5.129 GBps
Device Memory Latency	400 - 600 cycles
Device Memory Bandwidth	144 GBps
Shared Memory Latency	L1 latency (a few cycles)

Table 3.1: Performance characteristics of the GPU (NVidia Tesla C2050)

of 4 KB. If the fingerprint matches the marker then the thread defines that particular region as the end of a chunk boundary. The thread continues to compute the Rabin fingerprint in a sliding window manner in search of new chunk boundaries by shifting a byte forward in the sub-stream, and repeating this process.

3.4.2 Scalability Challenges

The basic design for Shredder that we presented in the previous section corresponds to the traditional way in which GPU-assisted applications are implemented. This design has proven to be sufficient for computation-intensive applications, where the computation costs can dwarf the cost of transferring the data to the GPU memory and accessing that memory from the GPU's cores. However, it results in only modest performance gains for data intensive applications that perform single-pass processing over large amounts of data, with a computational cost that is significantly lower than traditional GPU-assisted applications.

To understand why this is the case, we present in Table 3.1 some key performance characteristics of a specific GPU architecture (NVidia Tesla C2050), which helps us explain some important bottlenecks for GPU-accelerated applications. In particular, and as we will demonstrate in subsequent sections, we identified the following bottlenecks in the basic design of Shredder.

GPU device memory bottleneck. The fact that data needs to be transferred to the GPU memory before being processed by the GPU represents a serial dependency: such processing only starts to execute after the corresponding transfer concludes.

Host bottleneck. The host machine performs three serialized steps (performed by the Reader, Transfer, and Store threads) in each iteration. Since these three steps are inherently dependent on each other for a given input buffer, this serial execution becomes a bottleneck at host. Also, given the availability of multicore architecture at the host, this serialized execution leads to an underutilization of resources at host.

High memory latencies and bank conflicts. The global device memory on the GPU has a high latency, the order of 400 to 600 cycles. This works well for HPC algorithms, which are quadratic $O(N^2)$ or a higher degree polynomial in the input size N, since the computation time hides the memory access latencies. Chunking is also compute intensive, but it is only linear in the input size ($O(N)$, though the constants are high). Hence, even though the problem is compute intensive on traditional CPUs, on a GPU with an order of magnitude larger number of scalar cores, the problem becomes memory-intensive. In particular, the less sophisticated memory subsystem of the GPU (without prefetching or data caching support) is stressed by frequent memory access by a massive number of threads in parallel. Furthermore, a higher degree of parallelism causes memory to be accessed randomly across multiple bank rows, and leads to a very high number of bank conflicts. As a result, it becomes difficult to hide the latencies of accesses to the device memory.

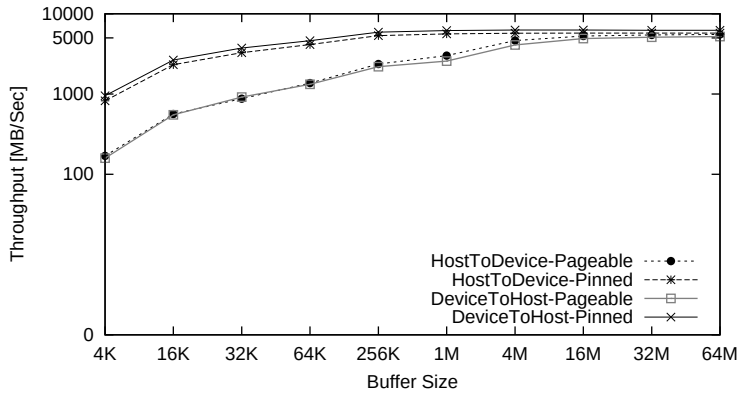


Figure 3.3: Bandwidth test between host and device.

3.5 Optimizations

In this section, we describe several optimizations that extend the basic design to overcome the challenges we highlighted in the previous section.

3.5.1 Device Memory Bottlenecks

3.5.1.1 Concurrent Copy and Execution

The main challenge we need to overcome is the fact that traditional GPU-assisted applications were designed for a scenario where the cost of transferring data to the GPU is significantly outweighed by the actual computation cost. In particular, the basic design serializes the execution of copying data to the GPU memory and consuming the data from that memory by the Kernel thread. This serialized execution may not suit the needs of data intensive applications, where the cost of the data transfer step becomes a more significant fraction of the overall computation time.

To understand the magnitude of this problem, we measured the overhead of a DMA transfer of data between the host and the device memory over the PCIe link connected to GPU. Figure 3.3 summarizes the effective bandwidth between host memory and device memory for different buffer sizes. We measured the band-

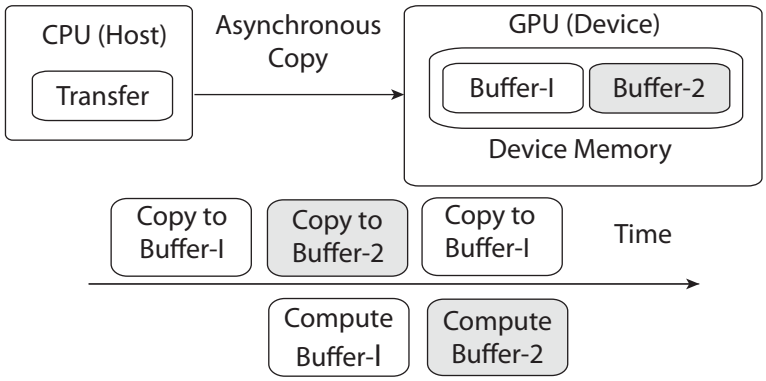


Figure 3.4: Concurrent copy and execution.

width both ways between the host and the device to gauge the DMA overhead for the Transfer and the Store thread. Note that the effective bandwidth is a property of the DMA controller and the PCI bus, and it is independent of the number of threads launched in the GPU. In this experiment, we also varied the buffer type allocated for the host memory region, which is allocated either as pageable or pinned memory regions. (The need for pinned memory will become apparent shortly.)

Highlights. Our measurements demonstrate the following: (i) small sized buffer transfers are more expensive than those using large sized buffers; (ii) the throughput saturates for buffer sizes larger than 32 MB (for pageable memory region) and 256 KB (for pinned memory region); (iii) for large sized buffers (greater than 32 MB), the throughput difference between pageable and pinned memory regions is not significant; and (iv) the effective bandwidth of the PCIe bus for data transfer is on the order of 5 GB/sec, whereas the global device memory access time by scalar processors in GPUs is on the order of 144 GB/sec, an order of magnitude higher.

Implications. The time spent to chunk a given buffer is split between the memory transfer and the kernel computation. For a non-optimized implementation of the chunking computation, we spend approximately 25% of the time performing the transfer. Once we optimize the processing in the GPU, the host to GPU memory transfer may become an even greater burden on the overall performance.

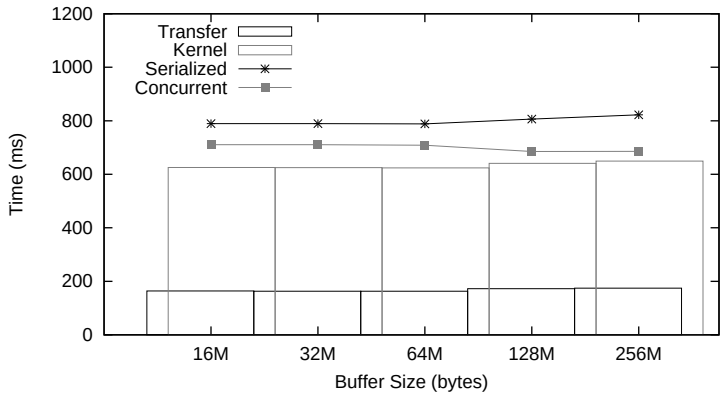


Figure 3.5: Normalized overlap time of communication with computation with varied buffer sizes for 1GB data.

Optimization. In order to avoid the serialized execution of the copy and data consumption steps, we propose to overlap the copy and the execution phases, thus allowing for the concurrent execution of data communication and the chunking kernel computations. To enable this, we designed a double buffering technique as shown in Figure 3.4, where we partition the device memory into twin buffers. These twin buffers will be alternatively used for communication and computation. In this scheme, the host asynchronously copies the data into the first buffer and, in the background, the device works on the previously filled second buffer. To be able to support asynchronous communication, the host buffer is allocated as a pinned memory region, which prevents the region from being swapped out by the pager.

Effectiveness. Figure 3.5 shows the effectiveness of the double buffering approach, where the histogram for transfer and kernel execution shows a 30% time overlap between the concurrent copy and computation. Even though the total time taken for concurrent copy and execution (`Concurrent`) is reduced by only 15% as compared to the serialized execution (`Serialized`), it is important to note that the total time is now dictated solely by the compute time. Hence, double buffering is able to remove the data copying time from the critical path, allowing us to focus only on optimizing the computation time in the GPU (which we address in § 3.5.3).

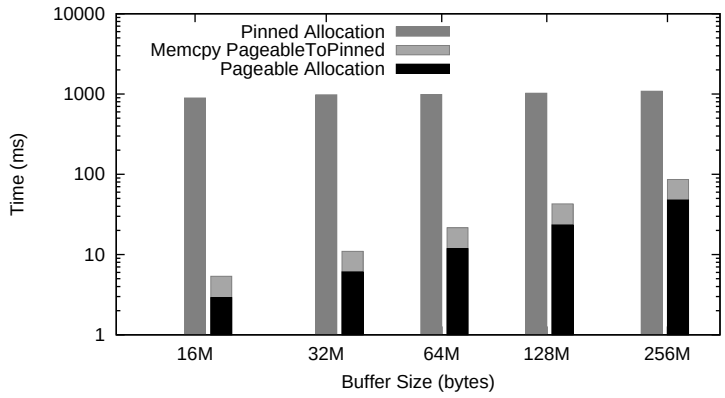


Figure 3.6: Comparison of allocation overhead of pageable with pinned memory region.

Supporting, the concurrent copy and execution, however, requires us to pin memory at the host, which reduces the memory allocation performance at the host. We next present an optimization to handle this side effect and ensure that double buffering leads to an end-to-end increase in chunking bandwidth.

3.5.1.2 Circular Ring Pinned Memory Buffers

As explained above, the double buffering requires an asynchronous copy between host memory and device memory. To support this asynchronous data transfer, the host side buffer should be allocated as a pinned memory region. This locks the corresponding page so that accessing that region does not result in a page fault until the region is subsequently unpinned.

To quantify the allocation overheads of using a pinned memory region, we compared the time required for dynamic memory allocation (using `malloc`) and pinned memory allocation (using the CUDA memory allocator wrapper). Since Linux follows an optimistic memory allocation strategy, where the actual allocation is deferred until memory initialization, in our measurements we initialized the memory region (using `bzero`) to force the kernel to allocate the desired buffer

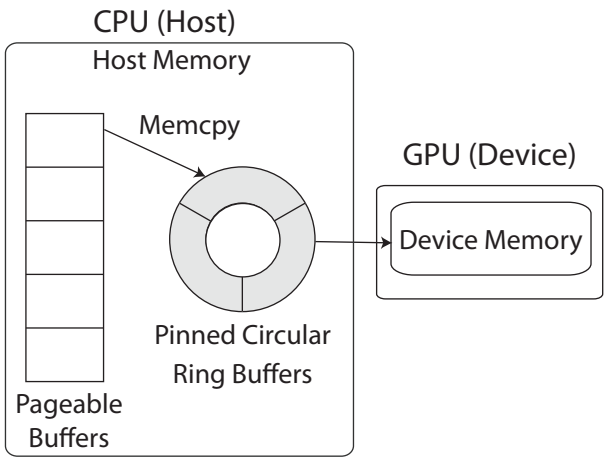


Figure 3.7: Ring buffer for the pinned memory region.

size. Figure 3.6 compares the allocation overhead of pageable and pinned memory for different buffer sizes.

Highlights. The important take away points are the following: (i) pinned memory allocation is more expensive than the normal dynamic memory allocation; and (ii) an adverse side effect of having too many pinned memory pages is that it can increase paging activity for unpinned pages, which degrades performance.

Implications. The main implication for our system design is that we need to minimize the allocation of pinned memory region buffers, to avoid increased paging activity or even thrashing.

Optimization. To minimize the allocation of pinned memory region while restricting ourselves to using the CUDA architecture, we designed a circular ring buffer built from a pinned memory region, as shown in Figure 3.7, with the property that the number of buffers can be kept low (namely as low as the number of stages in the streaming pipeline, as described in §3.5.2). The pinned regions in the circular buffer are allocated only once during the system initialization, and thereafter are reused in a round-robin fashion after the transfer between the host and the device

Buffer size (bytes)	16M	32M	64M	128M	256M
Device execution time (ms)	11.39	22.74	42.85	85.7	171.4
Host kernel launch time (ms)	0.03	0.03	0.03	0.08	0.09
Total execution time (ms)	11.42	22.77	42.88	85.78	171.49
Host `RDTSC` ticks @ 2.67 GHz	3.0e7	6.1e7	1.1e8	2.7e8	5.3e8

Table 3.2: Host spare cycles per core due to asynchronous data-transfer and kernel launch.

memory is complete. This allows us to keep the overhead of costly memory allocation negligible and have sufficient memory pages for other tasks.

Effectiveness. Figure 3.6 shows the effectiveness of our approach, where we compare the time for allocating pageable and pinned memory regions. Since we incur the additional cost of copying the data from pageable memory to the pinned memory region, we add this cost to the total cost of using pageable buffers. Overall, our approach is faster by an order of magnitude, which highlights the importance of this optimization.

3.5.2 Host Bottleneck

The previously stated optimizations alleviate the device memory bottleneck for DMA transfers, and allow the device to focus on performing the actual computation. However, the host side modules can still become a bottleneck due to the serialized execution of the following stages (`Reader`→`Transfer`→`Kernel`→`Store`). In this case, the fact that all four modules are serially executed leads to an underutilization of resources at the host side.

To quantify this underutilization at the host, we measured the number of idle spare cycles per core after the launch of the asynchronous execution of the kernel. Table 3.2 shows the number of RDTSC tick cycles for different buffer sizes. The RDTSC [21] (Read-Time Stamp Counter) instruction keeps an accurate count of every cycle that occurs on the processor for monitoring the performance. The

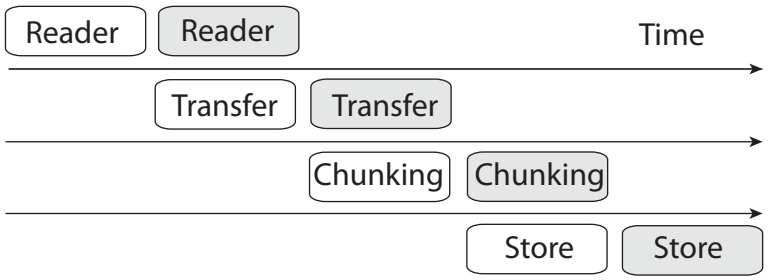


Figure 3.8: Multi-staged streaming pipeline.

device execution time captures the asynchronous copy and execution of the kernel, and the host kernel launch time measures the time for the host to launch the asynchronous copy and the chunking kernel.

Highlights. These measurements highlight the following: (i) the kernel launch time is negligible compared to the total execution time for the kernel; (ii) the host is idle during the device execution time; and (ii) the host has a large number of spare cycles per core, even with a small sized buffer.

Implications. Given the prevalence of host systems running on multicore architectures, the sequential execution of the various components leads to the underutilization of the host resources, and therefore these resources should be used to perform other operations.

Optimization. To utilize these spare cycles at the host, Shredder makes use of a multi-stage streaming pipeline as shown in Figure 3.8. The goal of this design is that once the Reader thread finishes writing the data in the host main memory, it immediately proceeds to handling a new window of data in the stream. Similarly, the other threads follow this pipelined execution without waiting for the next stage to finish.

To handle the specific characteristics of our pipeline stages, we use different design strategies for different modules. Since the Reader and Store modules deal with I/O, they are implemented as Asynchronous I/O (as described in §3.6.2.1),

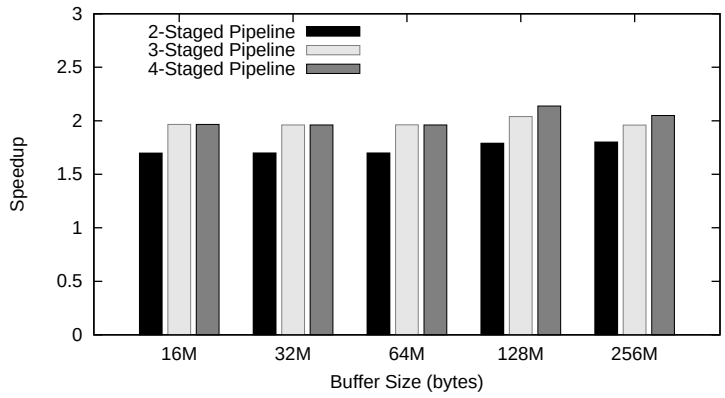


Figure 3.9: Speedup for streaming pipelined execution.

whereas the transfer and kernel threads are implemented using multi-buffering (a generalization of the double buffering scheme described in §3.5.1.1).

Effectiveness. Figure 3.9 shows the average speedup from using our streaming pipeline, measured as the ratio of time taken by a sequential execution to the time taken by our multi-stage pipeline. We varied the number of pipeline stages that can be executed simultaneously (by restricting the number of buffers that are admitted to the pipeline) from 2 to 4. The results show that a full pipeline with all four stages being executed simultaneously achieves a speedup of 2; the reason why this is below the theoretical maximum of a 4X gain is that the various stages do not have equal cost.

3.5.3 Device Memory Conflicts

We have observed (in Figure 3.5) that the chunking kernel dominates the overall time spent by the GPU. In this context, it is crucial to try to minimize the contribution of the device memory access latency to the overall cost.

Highlights. The very high access latencies of the device memory (on the order of 400-600 cycles @ 1.15 GHz) and the lack of support for data caching and prefetch-

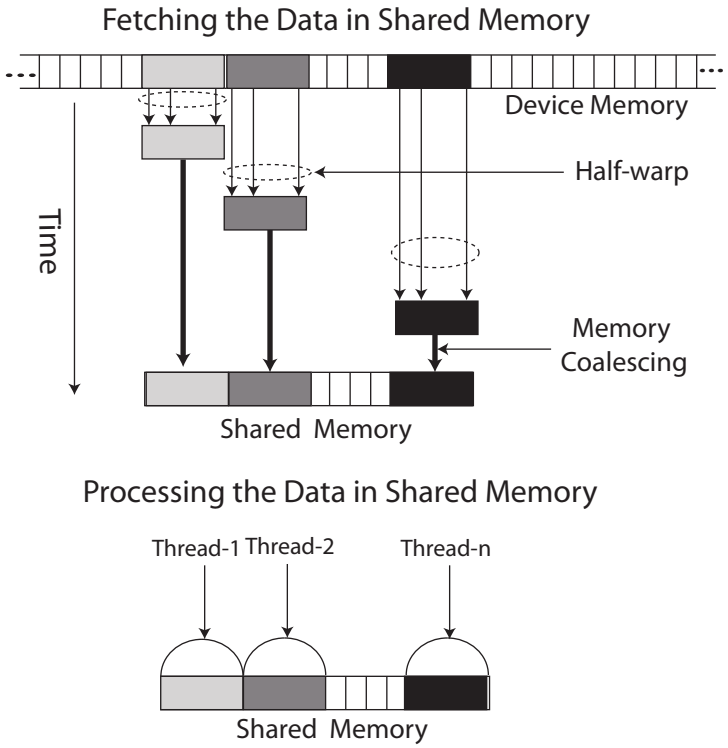


Figure 3.10: Memory coalescing to fetch data from global device memory to the shared memory.

ing can imply a significant overhead in the overall execution time of the chunking kernel.

Implications. The hierarchical memory of GPUs provides us an opportunity to hide the latencies of the global device memory by instead making careful use of the low latency shared memory. (Recall from § 3.3.2 that the shared memory is a fast and low latency on-chip memory which is shared among a subset of the GPU's scalar processors.) However, fetching data from global to the shared memory requires us to be careful to avoid bank conflicts, which can negatively impact the performance of the GPU memory subsystem. This implies that we should try to improve the inter-thread coordination in fetching data from the device global memory to avoid these bank conflicts.

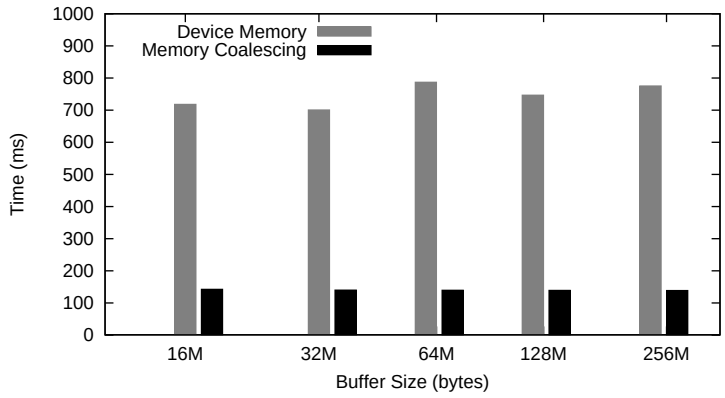


Figure 3.11: Normalized chunking kernel time with varied buffer-sizes for 1 GB data.

Optimization. We designed a thread cooperation mechanism to optimize the process of fetching data from the global memory to the shared memory, as shown in Figure 3.10. In this scheme, a single block that is needed by a given thread is fetched at a time, but each block is fetched with the cooperation of all the threads, and their coordination to avoid bank conflicts. The idea is to iterate over all data blocks for all threads in a thread block, fetch one data block at a time in a way that different threads request consecutive but non-conflicting parts of the data block, and then, after all data blocks are fetched, let each thread work on its respective blocks independently. This is feasible since threads in a warp (or half-warp) execute the same stream of instructions (SIMT). Figure 3.10 depicts how threads in a half-warp cooperate with each other to fetch different blocks sequentially in time.

In order to ensure that the requests made by different threads when fetching different parts of the same data block do not conflict, we followed the best practices suggested by the device manufacturer to ensure these requests correspond to a single access to one row in a bank [11, 12, 135]. In particular, Shredder lets multiple threads of a half-warp read a contiguous memory interval simultaneously, under following conditions: (i) the size of the memory element accessed by each thread is either 4, 8, or 16 bytes; (ii) the elements form a contiguous block of memory; i.e, the

Nth element is accessed by the Nth thread in the half-warp; and (iii) the address of the first element is aligned at a boundary of a multiple of 16 bytes.

Effectiveness. Figure 3.11 shows the effectiveness of the memory coalescing optimization, where we compare the execution time for the chunking kernel using the normal device memory access and the optimized version. The results show that we improve performance by a factor of 8 by reducing bank conflicts. Since the granularity of memory coalescing is 48 KB (which is the size for the shared memory per thread block), we do not see any impact from varying buffer sizes (16 MB to 512 MB), and the benefits are consistent across different buffer sizes.

3.6 Implementation

We implemented Shredder in CUDA [11], and for an experimental comparison, we also implemented an optimized parallel `pthreads`–based host-only version of content-based chunking. This section describes these implementations and evaluates them.

3.6.1 Host-Only Chunking using pthreads

We implemented a library for parallel content-based chunking on SMPs using POSIX pthreads. We derived parallelism by creating pthreads that operate in different data regions using a Single Program Multiple Data (SPMD) strategy and communicate using a shared memory data structure. At a high level, the implementation works as follows: (1) divide the input data equally in fixed-size regions among N threads; (2) invoke the Rabin fingerprint-based chunking algorithm in parallel on N different regions; (3) synchronize neighboring threads in the end to merge the resulting chunk boundaries.

An issue that arises is that dynamic memory allocation can become a bottleneck due to the serialization required to avoid race conditions. To address this, we used the Hoard memory allocator [43] instead of `malloc`.

3.6.2 Shredder Implementation

The Shredder library implementation comprises two main modules, the *host driver* and the *GPU kernel*. The host driver runs the control part of the system as a multi-threaded process on the host CPU running Linux. The GPU kernel uses one or more GPUs as co-processors for accelerating the SIMT code, and is implemented using the CUDA programming model from the NVidia GP-GPU toolkit [11]. Next we explain some key implementation details for both modules.

3.6.2.1 Host Driver

The host driver module is responsible for reading the input data either from the network or the disk and transferring the data to the GPU memory. Once the data is transferred then the host process dispatches the GPU kernel code in the form of RPCs supported by the CUDA toolkit. The host driver has two types of functionality: (1) the Reader/Store threads deal with reading and writing data from and to I/O channels; and (2) the Transfer thread is responsible for moving data between the host and the GPU memory. We implemented the Reader/Store threads using Asynchronous I/O and the Transfer thread using CUDA RPCs and page-pinned memory.

Asynchronous I/O (AIO). With asynchronous non-blocking I/O, it is possible to overlap processing and I/O by initiating multiple transfers at the same time. In AIO, the read request returns immediately, indicating that the read was successfully initiated. The application can then perform other processing while the background read operation completes. When the read response arrives, a signal regis-

tered with the read request is triggered to signal the completion of the I/O transaction.

Since the Reader/Store threads operate at the granularity of buffers, a single input file I/O may lead to issuing multiple `aio-read` system calls. To minimize the overhead of multiple context switches per buffer, we used `lio-listio` to initiate multiple transfers at the same time in the context of a single system call (meaning one kernel context switch).

3.6.2.2 GPU Kernel

The GPU kernel can be trivially derived from the C equivalent code by implementing a collection of functions in equivalent CUDA C with some assembly annotations, plus different access mechanisms for data layout in the GPU memory. However, an efficient implementation of the GPU kernel requires a bit more understanding of vector computations and the GPU architecture. We briefly describe some of these considerations.

Kernel optimizations. We have implemented minor kernel optimizations to exploit vector computation in GPUs. In particular, we used loop unrolling and instruction-level optimizations for the core Rabin fingerprint block. These changes are important because of the simplified GPU architecture, which lacks out-of-order execution, pipeline stalling in register usage, or instruction reordering to eliminate Read-after-Write (RAW) dependencies.

Warp divergence. Since the GPU architecture is Single Instruction Multiple Threads (SIMT), if threads in a warp diverge on a data-dependent conditional branch, then the warp is serially executed until all threads in it converge to the same execution path. To avoid a performance dip due to this divergence in warp execution, we carefully restructured the algorithm to have little code divergence within a warp, by minimizing the code path under data-dependent conditional branches.

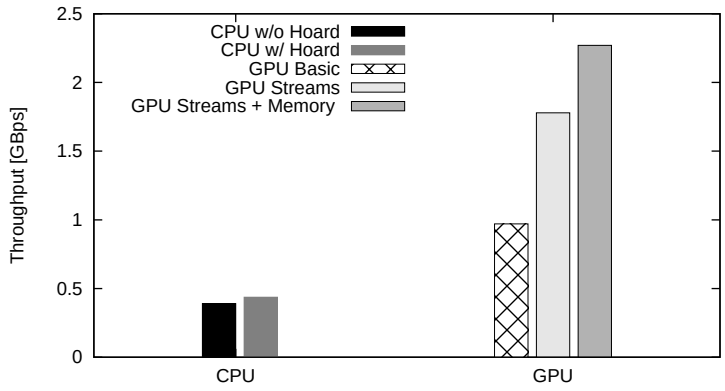


Figure 3.12: Throughput comparison of content-based chunking between CPU and GPU versions.

3.7 Evaluation

We now present our experimental evaluation of the performance of Shredder.

Experimental setup. We used a fermi-based GPU architecture, namely the Tesla C2050 GPU consisting of 448 processor cores (SPs). It is organized as a set of 14 SMs each consisting of 32 SPs running at 1.15 GHz. It has 2.6 GB of off-chip global GPU memory providing a peak memory bandwidth of 144 GB/s. Each SM has 32768 registers and 48 KB of local on-chip shared memory, shared between its scalar cores.

We also used an Intel Xeon processor based system as the host CPU machine. The host system consists of 12 Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The host machine is running Linux with kernel 2.6.38 in 64-bit mode, additionally patched with GPU direct technology [7] (for SAN devices). The GCC 4.3.2 compiler (with -O3) was used to compile the source code of the host library. The GPU code is compiled using the CUDA toolkit 4.0 with NVidia driver version 270.41.03. The posix implementation is run with 12 threads.

Results. We measure the effectiveness of GPU-accelerated content-based chunking by comparing the performance of different versions of the host-only and GPU

based implementation, as shown in Figure 3.12. We compare the chunking throughput for the pthreads implementation with and without using the Hoard memory allocator. For the GPU implementation, we compared the performance of the system with different optimizations turned on, to gauge their effectiveness. In particular, GPU Basic represents a basic implementation without any optimizations. The GPU Streams version includes the optimization to remove host and device bottlenecks using double buffering and a 4-stage pipeline. Lastly GPU Streams + Memory represents a version with all optimizations, including memory coalescing.

Our results show that a naive GPU implementation can lead to a 2X improvement over a host-only optimized implementation. The observation clearly highlights the potential of GPUs to alleviate computational bottlenecks. However, this implementation does not remove chunking as a bottleneck since SAN bandwidths on typical data servers exceed 10 Gbps. Incorporating the optimizations lead to Shredder outperforming the host-only implementation by a factor of over 5X.

3.8 Case Studies

In this section, we apply Shredder to two case study applications: (1) incremental HDFS, and (2) incremental cloud backup solution.

3.8.1 GPU-accelerated Incremental HDFS

This section presents a case study of applying Shredder in the context of incremental computation by integrating Shredder with Incoop.

GPU-Accelerated Incremental HDFS. We use Shredder to support Incoop by designing a GPU-accelerated version of Inc-HDFS, which is integrated with Incoop as shown in Figure 3.13. In particular, Inc-HDFS leverages Shredder to perform

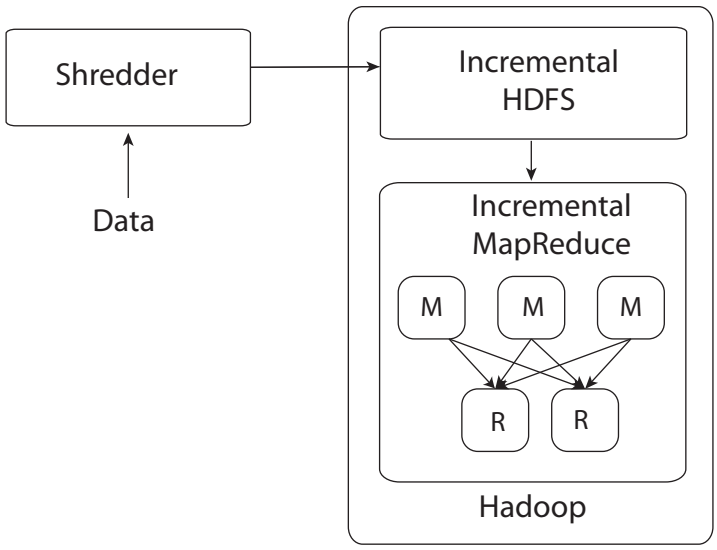


Figure 3.13: Incremental computation using Shredder.

content-based chunking instead of using fixed-size chunking as in the original HDFS, thus ensuring that small changes to the input lead to small changes in the set of chunks that are provided as input to Map tasks. This enables the results of the computations performed by most Map tasks to be reused.

Implementation. We built our prototype GPU-accelerated Inc-HDFS on Hadoop-0.20.2. It is implemented as an extension to Inc-HDFS, where the computationally expensive chunking is offloaded to the Shredder-enabled HDFS client (as shown in Figure 3.14), before uploading chunks to the respective data nodes that will be storing them.

Inc-HDFS client. We integrated the Shredder library with Inc-HDFS client using a JAVA-CUDA interface. Once the data upload function is invoked, the Shredder library notifies the chunk boundaries to the Store thread, which in turn pushes the chunks from the memory of the client to the data nodes of HDFS.

Semantic chunking framework. The default behavior of the Shredder library is to split the input file into variable-length chunks based on the contents. However, since chunking is oblivious to the semantics of the input data, this could cause

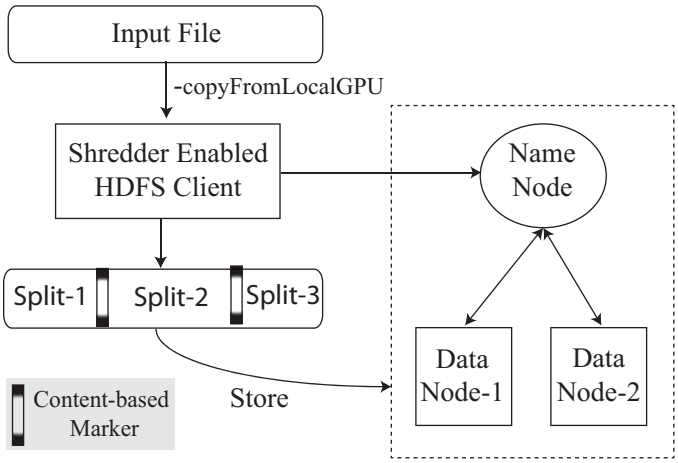


Figure 3.14: Shredder enabled chunking in Inc-HDFS.

chunk boundaries to be placed anywhere, including, for instance, in the middle of a record that should not be broken. To address this, we leverage the fact that the MapReduce framework relies on the `InputFormat` class of the job to split up the input file(s) into logical InputSplits, each of which is then assigned to an individual Map task. We reuse this class to ensure that we respect the record boundaries in the chunking process.

3.8.2 GPU-accelerated Incremental Cloud Backup

In this section, we present our second case study where we use Shredder in the context of a consolidated incremental backup system.

Background: Cloud Backup. Figure 3.15 describes our target architecture, which is typical of cloud back-ends. Applications are deployed on virtual machines hosted on physical servers. The file system images of the virtual machines are hosted in a virtual machine image repository stored in a SAN volume. In this scenario, the backup process works in the following manner. Periodically, full image snapshots are taken for all the VM images that need to be backed up. The core of the backup process is a backup server and a backup agent running inside the backup server. The image snapshots are mounted by the backup agent. The

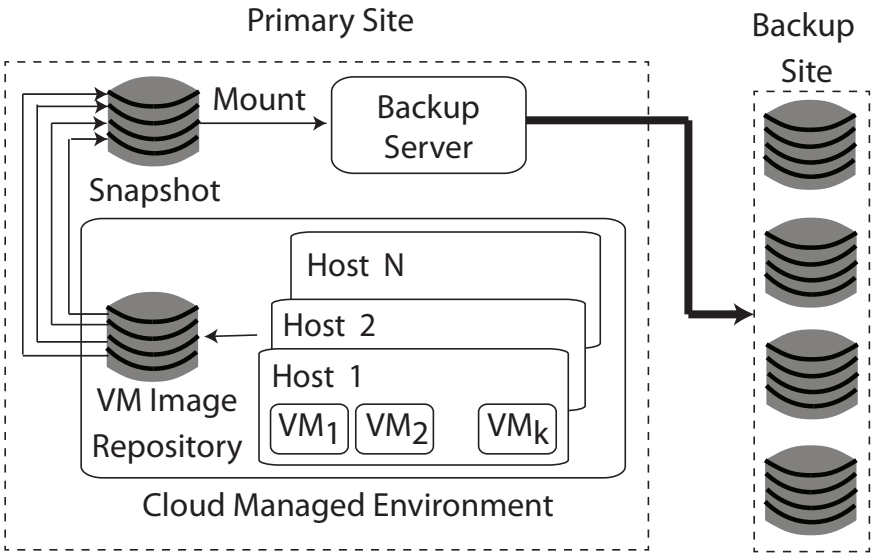


Figure 3.15: A typical cloud backup architecture

backup server performs the actual backup of the image snapshots onto disks or tapes. The consolidated or centralized data backup process ensures compliance of all virtual machines with the agreed upon backup policy. Backup servers typically have very high I/O bandwidth since, in enterprise environments, all operations are typically performed on a SAN [99]. Furthermore, the use of physical servers allows multiple dedicated ports to be employed solely for the backup process.

GPU-Accelerated Data Deduplication. The centralized backup process is eminently suitable for deduplication via content-based chunking, as most images in a data-center environment are standardized. Hence, virtual machines share a large number of files and a typical backup process would unnecessarily copy the same content multiple times. To exploit this fact, we integrate Shredder with the backup server, thus enabling data to be pushed to the backup site at a high rate while simultaneously exploiting opportunities for savings.

The Reader thread on the backup server reads the incoming data and pushes that into Shredder to form chunks. Once the chunks are formed, the Store thread computes a hash for the overall chunk, and pushes the chunks in the backup setup

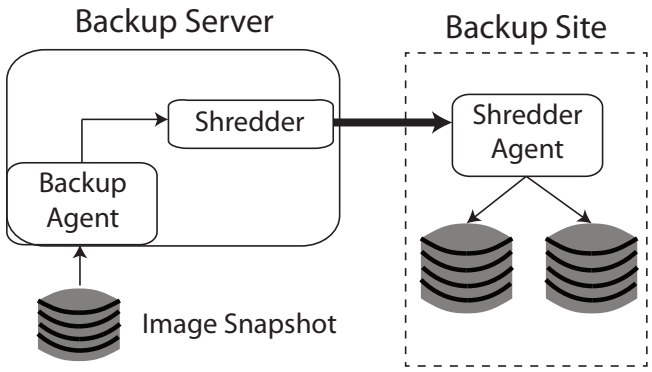


Figure 3.16: GPU-accelerated consolidated backup setup

as a separate pipeline stage. Thereafter, these hashes collected for the chunks are batched together to enqueue in an index lookup queue. Finally, a lookup thread picks up the enqueued chunk fingerprints and looks up in the index whether a particular chunk needs to be backed up or is already present in the backup site. If a chunk already exists, a pointer to the original chunk is transferred instead of the chunk data. We deploy an additional Shredder agent residing on the backup site, which receives all the chunks and pointers and recreates the original uncompressed data. The overall architecture for integrating Shredder in a cloud backup system is described in Figure 3.16.

Implementation and Evaluation. Since high bandwidth fiber channel adapters are fairly expensive, we could not recreate the high I/O rate of modern backup servers in our testbed. Hence, we used a memory-driven emulation environment to experimentally validate the performance of Shredder. On our backup agent, we keep a master image in memory using memcached [9]. The backup agent creates new file system images from the master image by replacing part of the content from the master image using a predefined similarity table. The master image is divided into segments. The image similarity table contains a probability of each segment being replaced by a different content. The agent uses these probabilities to decide which segments in the master image will be replaced. The image gener-

ation rate is kept at 10 Gbps to closely simulate the I/O processing rate of modern X-series employed for I/O processing applications [99].

In this experiment, we also enable the requirement of a minimum and maximum chunk size, as used in practice by many commercial backup systems. As mentioned in Section 3.4, our current implementation of Shredder is not optimized for including a minimum and maximum chunk size, since the data that is skipped after a chunk boundary is still scanned for computing a Rabin fingerprint on the GPU, and only after all the chunk boundaries are collected will the Store thread discard all chunk boundaries within the minimum chunk size limit. As future work, we intend to address this limitation using more efficient techniques that were proposed in the literature [105, 107].

As a result of this limitation, we observe in Figure 3.17 that we are able to achieve a speedup of only 2.5*X* in backup bandwidth compared to the pthread implementation, but still we manage to keep the backup bandwidth close to the target 10 Gbps. The results also show that even though the chunking process operates independently of the degree of similarity in input data, the backup bandwidth decreases when the similarity between the data decreases. This is not a limitation of our chunking scheme but of the unoptimized index lookup and network access, which reduces the backup bandwidth. Combined with optimized index maintenance (e.g., [68]), Shredder is likely to achieve the target backup bandwidth for the entire spectrum of content similarity.

3.9 Related Work

Our work builds on contributions from several different areas, which we survey.

GPU-accelerated systems. GPUs were initially designed for graphics rendering, but, because of their cost-effectiveness, they were quickly adopted by the HPC

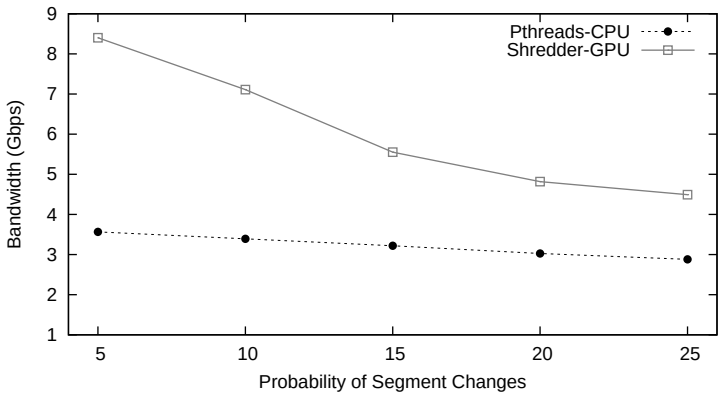


Figure 3.17: Backup bandwidth improvement due to Shredder with varying image similarity ratios

community for scientific computations [6, 119]. Recently, the systems research community has leveraged GPUs for building other systems. In particular, PacketShader [89] is a software router for general packet processing, and SSLShader [96] uses GPUs in web servers to efficiently perform cryptographic operations. GPUs have also been used to accelerate functions such as pattern matching [138], network coding [137], and complex cryptographic operations [90]. In our work, we explored the potential of GPUs for large scale data, which raises challenges due to the overheads of data transfer. Recently, GPUs were used in software-based RAID controllers [66] for performing high-performance calculations of error correcting codes. However, this work does not propose optimizations for efficient data transfer.

The most closely related work to Shredder proposes a framework for using GPUs to accelerate computationally expensive MD-based hashing primitives in storage systems [31, 82]. Our work focuses on large-scale data systems where the relative weight of data transfer can be even more significant. In particular, our chunking service uses Rabin fingerprinting, which is less computationally demanding than MD5, and is impacted more significantly by the serialization and memory latency issues. In addition, while the two papers address similar

bottlenecks in GPU-based systems, our techniques go beyond the ones proposed in [31, 82]. In particular, this prior work proposes memory management optimizations to avoid memory bank conflicts when accessing the shared memory, whereas in contrast we address the issue of bank conflicts in the global device memory of the GPU, which required us to introduce a thread co-operation mechanism using memory coalescing. In comparison to the remaining two optimizations we propose in Shredder, this prior work does not mention an execution pipeline that uses multicores at the host, and support for asynchronous data transfer is mentioned but not described in detail in their paper. We also present two real life end-to-end case studies of incremental MapReduce and cloud backup that benefit from Shredder.

Incremental Storage. Data deduplication is commonly used in storage systems. In particular, there is a large body of research on efficient index management [46, 68, 106, 146, 151]. In this paper, we focus on the complementary problem of content-based chunking [79, 98, 114]. High throughput content-based chunking is particularly relevant in environments that use SANs, where chunking can become a bottleneck. To overcome this bottleneck, systems have compromised the deduplication efficiency with sampling techniques or fixed-size chunking, or they have tried to scale chunking by deploying multi-node systems [61, 75, 76, 142]. A recent proposal shows that multi-node systems not only incur a high cost but also increase the reference management burden [85]. As a result, building a high throughput, cost-effective, single node systems becomes more important. Our system can be seen as an important step in this direction.

Network Redundancy Elimination. Content-based chunking has also been proposed in the context of redundancy elimination for content distribution networks (CDNs), to reduce the bandwidth consumption of ISPs [30, 35, 36, 124]. Also, many commercial vendors (such as Riverbed, Juniper, Cisco) offer middleboxes

to improve bandwidth usage in multi-site enterprises, data centers and ISP links. Our proposal is complementary to this work, since it can be used to improve the throughput of redundancy elimination in such solutions.

3.10 Limitations and Future Work

While Shredder is a significant step towards general support for incremental storage, there are several interesting avenues for future work. First, we would like to incorporate into the library several optimizations for parallel content-based chunking [105, 107]. Second, our proposed techniques need to continuously adapt to changes in the technologies that are used by GPUs, such as the use of high-speed InfiniBand networking, which enables further optimizations in the packet I/O engine using GPU-direct [7]. Third, we would like explore new applications like middleboxes for bandwidth reduction using network redundancy elimination [35]. Finally, we would like to incorporate Shredder as an extension to recent proposals to devise new operating system abstractions to manage GPUs [134].

3.11 Summary

In this chapter, we have presented Shredder, a novel framework for content-based chunking using GPU acceleration. We have proposed several optimizations to improve the performance of content-based chunking on GPUs. We have applied Shredder to two incremental storage and computation applications, and our experimental results show the effectiveness of the optimizations that are included in the design of Shredder. We believe that Shredder can be a useful building block in the construction of efficient solutions for redundancy elimination in areas such as data backup, incremental computation, and CDNs.

CHAPTER 4

Slider: Incremental Stream Processing

In this chapter, we describe the design, implementation, and evaluation of Slider, a batched stream processing framework for incremental sliding window analytics. The design of Slider incorporates self-adjusting contraction trees, a set of data structures and algorithms for transparently updating the output of data-parallel sliding window computations as the window moves, while reusing, to the extent possible, results from prior computations.

This chapter is organized as follows. We first motivate the design of Slider in Section 4.1. We next briefly highlight the contributions of Slider in Section 4.2. Thereafter, we present present an overview of the basic approach in Section 4.3. Next, we present the detailed design of self-adjusting contraction trees in Sections 4.4, 4.5 and 4.6. The architecture of Slider is described in Section 4.7. Section 4.8 presents an experimental evaluation of Slider, and our experience with the case studies is reported in Section 4.9. We present the related work in Section 4.10. Finally, limitations and conclusion are presented in Section 4.11 and Section 4.12, respectively.

4.1 Motivation

"Big data" systems (as mentioned in Chapter 2) are very often used for analyzing data that is collected over very long periods of time. Either due to the nature of the

analysis, or in order to bound the computational complexity of analyzing a monotonically growing data set, applications often resort to a *sliding window* analysis. In this type of processing, the scope of the data analysis is limited to an interval over the entire set of collected data, and, periodically, newly produced inputs are appended to the window and older inputs are discarded from it as they become less relevant to the analysis.

Stream processing is the state-of-the-art distributed computing medium for large-scale sliding window analytics. Broadly speaking, stream processing platforms can be classified based on the programming model as trigger-based streaming systems and batch-based streaming systems. Trigger-based systems provide an event-driven programming model; whereas, batch-based systems provide a simply but powerful data-parallel programming model for the application developers. (We provide a detailed comparison of the trade-offs involved between the two types of streaming systems in Section 4.10.)

At a high-level, trigger-based streaming systems provide a mechanism for incrementally updating the output for sliding window analytics. However, these systems rely on the application programmers to devise the incremental update mechanism by designing and implementing application-specific *dynamic algorithms*. While these systems can be efficient, they require programmers to design dynamic algorithms. Such algorithms, as explained previously, are often difficult to design, analyze, and implement even for simple problems.

On the other hand, batch-based streaming systems provide a "one-shot" computing mechanism for sliding window analytics, where the entire window is recomputed from scratch whenever the window slides. Consequently, even old, unchanged data items that remain in the window are reprocessed, thus consuming unnecessary computational resources and limiting the timeliness of results.

In this chapter, we answer the following question: Is it possible to achieve the benefits of incremental sliding window analytics without requiring dynamic algorithms in batch-based streaming systems? We showed in the case of Incoop that it is possible to obtain performance gains for incremental computation in batch-processing systems in a transparent way, i.e., without changing the original (single pass) data analysis code. However, Incoop did not leverage the particular characteristics of sliding windows and resorted solely to the memoization of sub-computations, which still requires time proportional to the size of the whole data rather (albeit with a small constant) than the change itself.

4.2 Contributions

In this chapter, we present *self-adjusting contraction trees*, a set of data structures for incremental sliding window analytics, where the work performed by incremental updates is proportional to the size of the changes in the window rather than the whole data. Using these data structures only requires the programmer to devise a non-incremental version of the application code expressed using a conventional data-parallel programming model. We then guarantee an automatic and efficient update of the output as the window slides. Moreover, we make no restrictions on how the window slides, allowing it to shrink on one end and to grow on the other end arbitrarily. However, as we show, more restricted changes lead to simpler algorithms and more efficient updates. Overall, our contributions include:

- **Self-adjusting contraction trees:** A set of self-adjusting data structures that are designed specifically for structuring different variants of sliding window computation as a (shallow) balanced dependence graph. These balanced graphs ensure that the work performed for incremental updates is propor-

tional to the size of the changes in the window (the "delta") incurring only a logarithmic—rather than linear—dependency on the size of window (§4.4).

- **Split processing algorithms:** We introduce a *split processing model*, where the incremental computation is divided into a background pre-processing phase and a foreground processing phase. The background processing takes advantage of the predictability of input changes in sliding window analytics to pave the way for a more efficient foreground processing when the window slides (§4.5).

- **Query processing—multi-level trees:** We present an extension of the proposed data structures for multi-level workflows to support incremental dataflow query processing (§4.6).

We implemented self-adjusting contraction trees in a system called Slider, which extends Hadoop [8], and evaluated the effectiveness of the new data structures by applying Slider to a variety of micro-benchmarks and applications. Furthermore, we report on three real world use cases: *(i)* building an information propagation tree [132] for Twitter; *(ii)* monitoring Glasnost [74] measurement servers for detecting traffic differentiation by ISPs; and *(iii)* providing peer accountability in Akamai NetSession [28], a hybrid CDN architecture.

4.3 Overview

Our primary goal is to design data structures for incremental sliding window analytics, so that the output is efficiently updated when the window slides. In addition, we want to do so transparently, without requiring the programmer to change any of the existing application code, which is written assuming non-incremental (batch) processing. In our prototype system called Slider, non-incremental com-

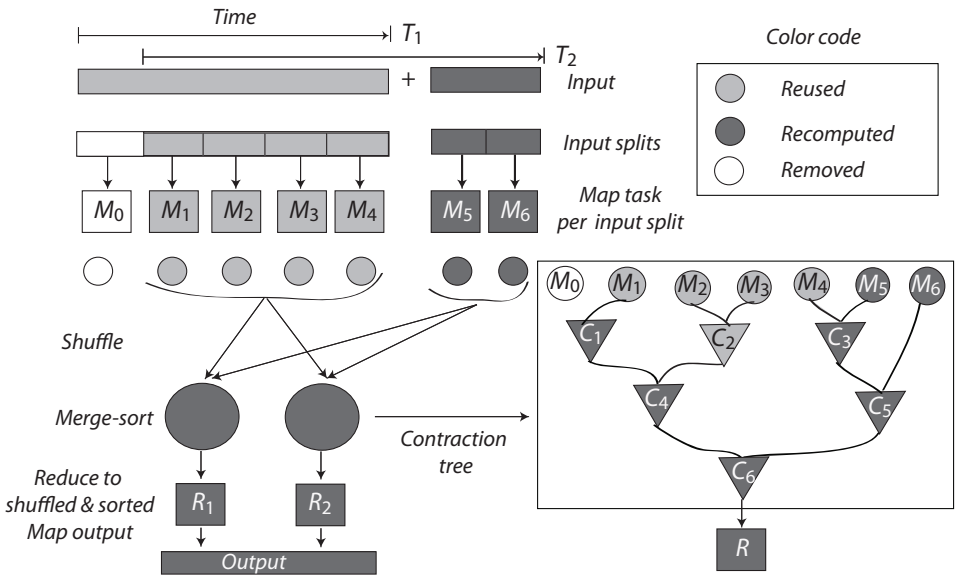


Figure 4.1: Strawman design and contraction phase

putations are expressed under the MapReduce [67] programming model (or alternatively as Pig [116] programs), but the data structures can be plugged into other data-parallel programming models that allow for decomposing computations into associative sub-computations, such as Dryad [95] and Spark [148].

4.3.1 Strawman Design

The design of self-adjusting contraction trees is based on self-adjusting computation [24] (summarized in Chapter 2), where the basic idea is to create a graph of data dependent sub-computations and propagate changes through this graph. Just like in Incoop, when applying the principles of self-adjusting computation to MapReduce, vertices in the dependence graph correspond to Map and Reduce tasks, and edges represent data transferred between tasks (as depicted in Figure 4.1).

When applying the strawman design for sliding window analytics, new data items are appended at the end of the previous window and old data items are dropped from the beginning. To update the output incrementally, we launch a Map task for each new "split" (a partition of the input that is handled by a single

Map task) and reuse the results of Map tasks operating on old but live data. We then feed the newly computed results together with the reused results to Reduce tasks to compute the final output.

This initial strawman design highlights an important limitation: even a small change to the input can preclude the reuse of all the work after the first level of nodes in the graph. This is because the second level nodes (i.e., the Reduce tasks) take as input all values for a given key ($< k_i >, < v_1, v_2, .., v_n >$). In the example shown in Figure 4.1, as the computation window slides from time T_1 to T_2, it invalidates the input of all Reduce tasks because of the removal of the M_0 output and the addition of new Map outputs (M_5 & M_6) to the window. To address this limitation, we refine the strawman design by organizing the second level nodes (corresponding to the Reduce phase) into a *contraction phase*, as proposed in Incoop (Chapter 2). The contraction phase is interposed between the Map and the Reduce phase, we briefly explain the contraction phase next.

4.3.2 Adding the Contraction Phase

As explained in Chapter 2, the idea behind the contraction phase is to break each Reduce task into smaller sub-computations, which are structured in a *contraction tree*, and then propagate changes through this tree. We construct the contraction tree by breaking up the work done by the (potentially large) Reduce task into many applications of the Combiner function. Combiner functions [67] were originally designed to run at the Map task for saving bandwidth by doing a local reduction of the output of Map, but instead we use Combiners at the Reduce task to form the contraction tree. More specifically, we split the Reduce input into small partitions (as depicted in Figure 4.1), and apply the Combiner to pairs of partitions recursively in the form of a binary tree until we have a single partition left. Finally, we apply the Reduce function to the last Combiner, to get the final output. This re-

Algorithm 1: Basic algorithm for sliding windows

```
1  Require changes: △ ← (−δ1, +δ2)
2  /* Process new input +δ2 by running Map tasks*/
3  for i = T_end to T_end + (+δ2) do
4      M_i({k}) ← run_maptask(i);
5  /* Propagate △ using contraction tree*/
6  forall the keys k do
7      /* Delete Map outputs for −δ1*/
8      for i = T_start to T_start − (−δ1) do
9          contraction_tree·delete(M_i(k));
10     /* Insert Map outputs for +δ2*/
11     for i = T_end to T_end + (+δ2) do
12         contraction_tree·insert(M_i(k)));
13     /* Perform change propagation*/
14     contraction_tree·update(k);
15 /*Adjust the window for the next incremental run*/
16 T_start ← T_start − (−δ1);
17 T_end ← T_end + (+δ2);
```

quires Combiner functions to be associative, an assumption that is met by every Combiner function we have come across.

The final strawman design we obtain after adding the contraction phase is shown in Algorithm 1. As a starting point, changes ($\bigtriangleup$) in the input are specified by the user as the union of old items ($-\delta$) that are dropped and new items ($+\delta$) that are added to the window. Subsequently,

1. the items that are added to the window ($+\delta$) are handled by breaking them up into fixed-sized chunks called "splits", and launching a new Map task to handle each split (*line 3-4*);

2. the outputs from these new Map tasks along with the old splits that fall out from the sliding window ($-\delta$) are then fed to the contraction phase instead of the Reduce task *for each emitted key k* (*line 6-14*);

3. finally, the computation time window is adjusted for the next incremental run (*line 16-17*).

4.3.3 Efficiency of the Contraction Tree

The efficiency of self-adjusting computation in responding to an input modification is determined by the stability of the computation. For change propagation to be efficient, it is important that *(a)* the computation is divided into small enough sub-computations; and *(b)* no long chain of dependencies exists between sub-computations. We call such computations ***stable*** because few of their sub-computations change when the overall input is modified by a small amount.

The Contraction tree proposed in Incoop provides stability for arbitrary changes in the input. However, the (plain) Contraction tree does not construct the dependency graph of subcomputations explicitly, and thus does not perform change propagation on the dependency graph. Instead, the graph is recorded implicitly by memoizing subcomputations—MapReduce tasks—and change propagation is performed by re-visiting all subcomputations and reusing those that can be reused via memoization. While this approach simplifies the design and the implementation, it can yield asymptotically suboptimal performance, because it requires touching all subcomputations (for the purposes of memoization and reuse) even if they may not be affected by the input modifications. In contrast, Slider proposes a series of data structures that perform change propagation by taking advantage of the fact that in sliding window computations the input changes happen only at the end points of the computation window. Furthermore, Slider takes advantage of the predictability of changes in sliding window computations to improve the timeliness of the results by enabling a split processing mode, where a background processing leverages the predictability of input changes to pave the way for a more efficient foreground processing when the window slides.

We next present a set of novel data structures, called *self-adjusting contraction trees*, that replace the simple binary tree in the strawman design of the contraction phase. The goal of these data structures is to ensure that the path from the newly added and dropped inputs to the root has a low depth, and that as many unaffected sub-computations as possible are outside that path. Furthermore, these data structures must perform a form of rebalancing after each run, i.e., a change in the sliding window not only triggers an update to the output but also to the structure of the contraction tree, to ensure that the desirable properties of our data structures hold for subsequent runs.

4.4 Self-Adjusting Contraction Trees

In this section, we present the general case data structures for incremental sliding window analytics. When describing our algorithms, we distinguish between two modes of running: an *initial run* and an *incremental run*. The *initial run* assumes all input data items are new and constructs the self-adjusting contraction tree from scratch. The *incremental run* takes advantage of the constructed tree to incrementally update the output.

4.4.1 Folding Contraction Tree

Our first data structure, called a *self-adjusting folding tree*, permits shrinking and extending the data window arbitrarily, i.e., supports *variable-width window* slides. The goal of this data structure is to maintain a small height for the tree, since this height determines the minimum number of Combiner functions that need to be recomputed when a single input changes.

Initial run. Given the outputs of the Map phase consisting of M tasks, we construct a folding tree of height $\lceil \log_2 M \rceil$ and pair each leaf with the output of a Map

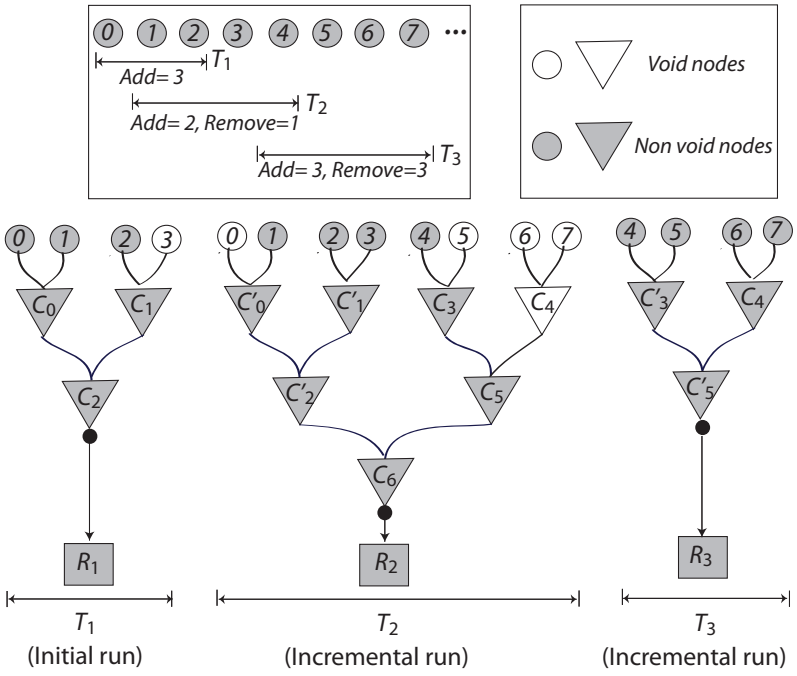


Figure 4.2: Example of folding contraction tree

task, such that all Map tasks are mapped to a contiguous subset of the leaves with no unpaired leaves in between. The leaf nodes that cannot be filled in the complete binary tree (adding up to $2^{height} - M$ nodes) are marked as void nodes; these nodes will be occupied by future Map tasks. To compute the output, we apply Reduce to the root of the folding tree.

Figure 4.2 illustrates an example. At time T_1, we construct a complete binary tree of height two, where the leaves are the Map outputs of $\{0, 1, 2\}$, and with an additional void node to make the number of leaves a power of two. We then apply combiners $\{C_0, C_1, C_2\}$ to pairs of nodes to form a binary tree.

Incremental run. When the window slides, we want to keep the folding tree balanced, meaning that the height of the tree should be roughly be equal to logarithmic to the current window size ($H = \lceil \log_2 M \rceil$), which is the minimum possible height. The basic idea is to assign the outputs of new Map invocations to the void leaf nodes on the right hand side of the tree, and mark the leaf nodes on the left

hand side corresponding to Map tasks that were dropped from the window as void. Adding new nodes on the right hand side may require a change in the tree size, when the new Map outputs exceed the number of void nodes. Conversely, dropping nodes from the left hand side can cause the entire left half of the tree to contain void leaves. These are the two cases that lead to a change in the height of the tree, and they are handled by folding and unfolding units of *complete (sub-)trees*, i.e., increasing or decreasing the tree height by one, while maintaining a complete tree.

In particular, when inserting items, we first try to fill up the void nodes to the right of the non-void leaves that still have not been used in the previous run. If all void nodes are used, a new complete contraction tree is created, whose size is equal to the current tree, and we merge the two trees. This increases the height of the tree by one. When removing items, we always attempt to reduce the tree height in order to make incremental processing more efficient by checking if the entire left half of the leave nodes are void. If so, we discard half of the tree by promoting the right hand child of the root node to become the new root.

Figure 4.2 shows a set of example incremental runs for this algorithm. At time T_2, two Map outputs (nodes 3 & 4) are inserted, causing the tree to expand to accommodate node 4 by constructing another subtree of height two and joining the new subtree with the previous tree, which increases the height to three. Conversely, at time T_3 the removal of three Map outputs (nodes 1, 2, & 3) causes the tree height to decrease from three to two because all leaf nodes in the left half of the tree are void.

4.4.2 Randomized Folding Tree

The general case algorithm performs quite well in the normal case when the size of the window does not change drastically. However, the fact that tree expansion and

contraction is done by doubling or halving the tree size can lead to some corner cases where the tree becomes imbalanced, meaning that its height is no longer $H = \lceil \log_2 M \rceil$. For example, if the window suddenly shrinks from a large value of M elements to $M' = 2$ elements, and the two remaining elements happen to be on different sides with respect to the root of the tree, then the algorithm ends up operating on a folding tree with height $\lceil log_2(2M+1) \rceil$ when the window is of size $M' << M$.

One way to address this problem is to perform an initial run whenever the size of the window is more than some desired constant factor (e.g., 8, 16) smaller than the number of leaves of the folding tree. On rebalancing, all void nodes are garbage collected and a freshly balanced folding tree is constructed ($H = \lceil log_2(M') \rceil$) similar to the initial run. This strategy is attractive for workloads where large variations in the window size are rare. Otherwise, frequently performing the initial run for rebalancing can be inefficient.

For the case with frequent changes in the window size, we designed a randomized algorithm for rebalancing the folding tree. This algorithm is very similar to the one adopted in the design of *skip lists* [126], and therefore inherits its analytical properties. The idea is to group nodes at each level probabilistically instead of folding/unfolding complete binary trees. In particular, each node forms a group boundary with a probability $p = 1/2$. In the expected case, and by analogy to the skip list data structure, the average height of the tree is $H = \lceil log_2(current\_window\_size) \rceil$.

Figure 4.3 shows an example of a randomized folding tree with 4 levels for 16 input leaf nodes. The tree is constructed by combining nodes into groups, starting from left to right, where for each node a coin toss decides whether to form a group boundary: with probability $p = 1/2$, a node either joins the previous group or creates a new group. In the example, leaf nodes $0, 1, 2$ join the same group C_0, and leaf node 3 creates a new group C_1 which is joined by nodes $4, 5, 6$. This process

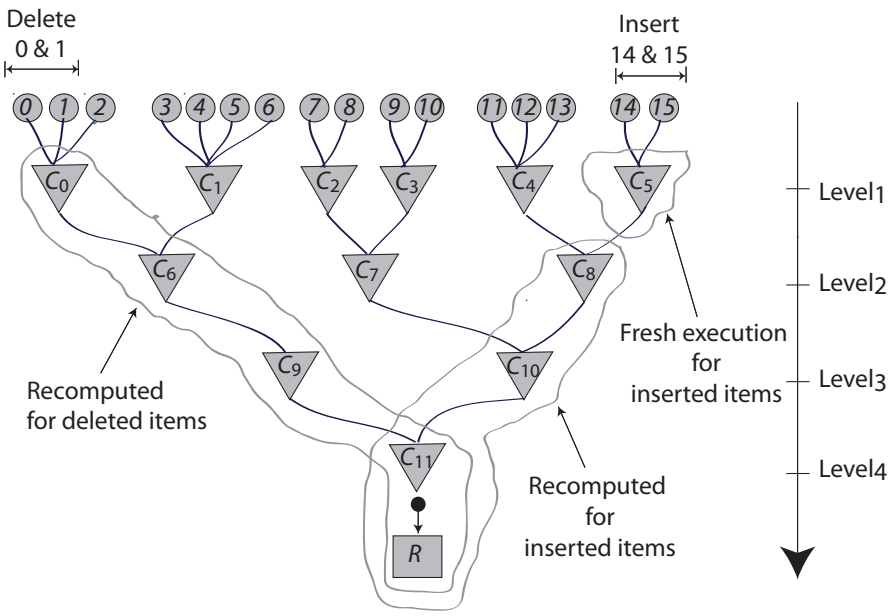


Figure 4.3: Example of randomized folding tree

is repeated at all the levels. When nodes are deleted, all nodes on paths from the deleted nodes to the root are recomputed. In the example, after nodes 0 and 1 are deleted, nodes C_0, C_6, C_9, C_{11} are recomputed. Similarly, the newly inserted items are grouped probabilistically at all the levels, and then the merged nodes (combination of new and old nodes) are re-computed.

4.5 Split Processing Algorithms

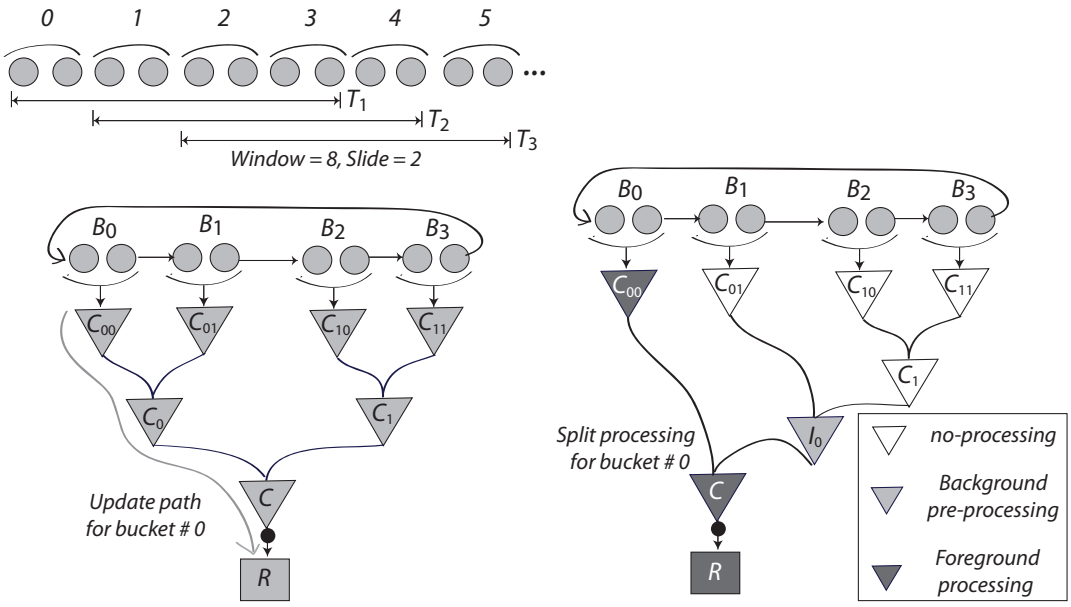
We now consider two special cases of sliding windows, where, in addition to offering specialized data structures, we also introduce a split processing optimization. In the first special case, the window can be extended on one end and reduced on the other, as long as the size of the window remains the same (§4.5.1). (This is also known as *fixed-width window processing*.) In the second case, the window is only extended monotonically on one end by append operations (§4.5.2). (This is also known as *bulk-appended data processing*.)

In both these cases, since we know more in advance about the type of change that is going to take place in the next run, we leverage this fact for improving the responsiveness of incremental updates by preparing for the incremental run before it starts. More precisely, we split the change propagation algorithm into two parts: a *foreground processing* and a *background pre-processing*. The foreground processing takes place right after the update to the computation window, and minimizes the processing time by combining new data with a pre-computed intermediate result. The background pre-processing takes place after the result is produced and returned, paving the way for an efficient foreground processing by pre-computing an intermediate result that will be used in the next incremental update. The background pre-processing step is optional and we envision performing it on a best-effort basis and bypassing it if there are no spare cycles in the cluster.

4.5.1 Rotating Contraction Trees

In *fixed-width* sliding window computations, new data is appended at the end, while the same amount of old data is dropped from the beginning of the window, i.e., w new splits (each processed by a new Map task) are appended and w old splits are removed. To perform such computations efficiently, we use *rotating contraction trees* (depicted in Figure 4.4). Here, w splits are grouped using the combiner function to form what we call a bucket. Then, we form a balanced binary contraction tree where the leaves are the buckets. Since the number of buckets remains constant when the window slides, we just need to rotate over the leaves in a round-robin fashion, replacing the oldest bucket with the newly produced one.

Initial run. In this case, the steady state of incremental runs is only reached when the window fills up. As such, we need to consider the sequence of initial runs during which no buckets are dropped. At each of these runs, we combine the w newly produced Map outputs to produce a new bucket. By the time the first



(a) foreground only mode (b) w/ background mode

Figure 4.4: Example of rotating contraction trees

window fills, we construct the contraction tree by combining all bucket outputs in pairs hierarchically, to form a balanced binary tree of height $\lceil log_2(N) \rceil$, where N is the total number of buckets in a window. Figure 4.4(a) shows an example with $w = 2$ and $N = 4$. At T_1, the first level of the tree (C_{00}, C_{01}, C_{10}, C_{11}) is constructed by invoking combiners on the N buckets of size $w = 2$, whose results are then recursively combined to form a balanced binary tree. The output of the combiner at the root of the tree is then used as input to the Reduce task.

Incremental run. We organize the leaf nodes of the contraction tree as a circular list. When w new splits arrive and w old splits are removed from the data set, we replace the oldest bucket with the new bucket and update the output by recomputing the path affected by the new bucket. Figure 4.4(a) shows an example. At T_2 the new bucket 4 replaces the oldest bucket 0. This triggers a propagation of this change all the way to the root, where each step combines a memoized combiner output with a newly produced combiner output. In this example, we reuse the

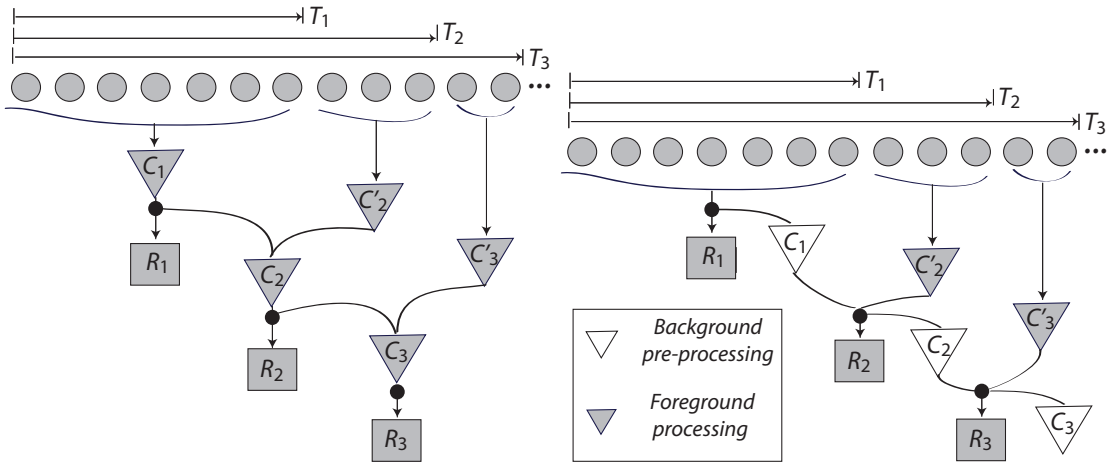
memoized outputs of combiners C_{01}, and C_1. In total, this requires recomputing a number of combiners that is equal to $log(N)$. The rotation of buckets requires commutativity in addition to the associativity of the combiner invocations. Both properties were held by Combiner functions in the applications we analyzed.

Background pre-processing. As explained before, this background step anticipates part of the processing since in this case we can predict the window change that will take place. In particular, in this case we know exactly what are the subtrees of the next incremental run whose outputs will be reused – these are the subtrees that fall outside the path from the next bucket to be replaced to the root. We take advantage of this by pre-combining all the combiner outputs that are at the root of those subtrees. For example, in Figure 4.4(b), we can pre-compute the combiner output I_0 by combining C_{01} and C_1 along the update path of bucket 0 in the background. This way, the incremental run only needs to invoke the Reduce task with the output of this pre-computed Combiner invocation (I_0) and the outputs of the newly run Map tasks.

4.5.2 Coalescing Contraction Trees

In the *append only* variant, the window grows monotonically as the new inputs are appended at the end of the current window, i.e., old data is never dropped. For this kind of workflow we designed a data structure called a *coalescing contraction tree* (depicted in Figure 4.5).

Initial run. The first time input data is added, a single-level contraction tree is constructed by executing the Combiner function (C_1 in Figure 4.5(a)) for all Map outputs. The output of this combiner is then used as input to the Reduce task, which produces the final output (R_1 in Figure 4.5(a)).



(a) foreground only mode (b) w/ background mode

Figure 4.5: Example of coalescing contraction trees

Incremental run. The outputs of the new Map tasks (C_2' in Figure 4.5(a)) are combined, and the result is combined with the output of the contraction tree from the previous run to form a new contraction tree (C_2 combines the outputs of C_1 and C_2'). The output of the root of this new tree is then provided to a Reduce task (R_2 in the example), which produces the new output.

Background pre-processing. In the foreground processing step (see Figure 4.5(b)) the new output is computed directly by invoking the Reduce task on the root of the old contraction tree and on the output of a Combiner invocation on the new Map inputs. In the background pre-processing phase, we prepare for the next incremental run by forming a new root of the contraction tree to be used with the next new input data. This is done by combining the root of the old tree with the output of the previous Combiner invocation on the new Map inputs. Figure 4.5(b) depicts an example for background pre-processing. We perform the final reduction (R_2) directly on the union of the outputs of the combiner invocation from the previous run (C_1), and the combiner invocation, which aggregates the outputs of the newly run Map tasks (C_2'). In the background, we run the new combiner that

will be used in the next incremental run (C_2), using the same inputs as the Reduce task, to anticipate the processing that will be necessary in the next run.

4.6 Query Processing: Multi-Level Trees

We next present an extension of self-adjusting contraction trees to integrate them with tools that support declarative data-flow query languages, such as Pig [116] or DryadLINQ [95]. These languages have gained popularity in the context of large-scale data analysis due to the ease of programming using their high-level primitives. To support these systems, we leverage the observation that programs written in these query languages are compiled to a series of pipelined stages where each stage corresponds to a program in a traditional data-parallel model (such as MapReduce or Dryad), for which we already have incremental processing support.

In particular, our query processing interface is based on Pig [116]. Pig consists of a high-level language (called Pig-Latin) similar to SQL, and a compiler that translates Pig programs to a workflow of multiple pipelined MapReduce jobs. Since our approach handles MapReduce programs transparently, each stage resulting from this compilation can run incrementally by leveraging contraction trees. A challenge, however, is that not all the stages in this pipeline are amenable to a sliding window incremental computation. In particular, after the first stage MapReduce job that processes the input from the sliding window, changes to the input of subsequent stages could be at arbitrary positions instead of the window ends. Thus, we adapt the strategy we employ at different stages as follows: (1) in the first stage, we use the appropriate self-adjusting contraction tree that corresponds to the desired type of window change; and, (2) from the second stage onwards in the pipeline, we use the strawman contraction tree (§4.3) to detect and propagate changes.

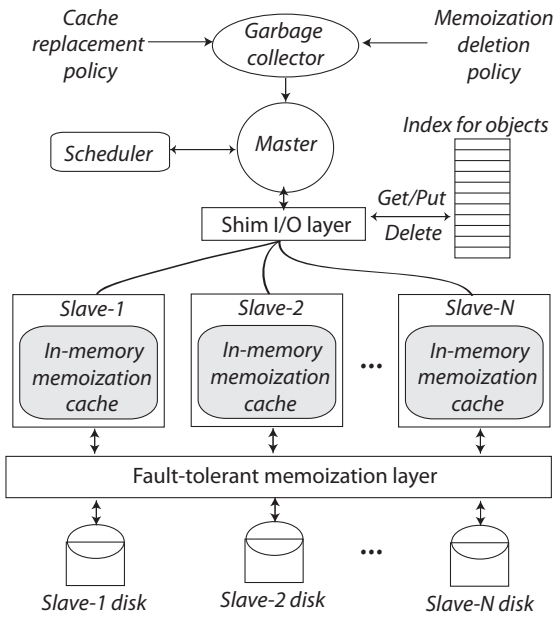


Figure 4.6: Slider architecture

4.7 Implementation

We implemented self-adjusting contraction trees in a system called Slider, as an extension to Incoop. Here, we highlight only the additional components we built on the top of Incoop. An overview of the implementation is depicted in Figure 4.6.

Self-adjusting contraction trees. Our data structures are implemented by inserting an additional Contraction phase between the shuffle stage and the sort stage. To prevent unnecessary data movement in the cluster, the new Contraction phase runs on the same machine as the Reduce task that will subsequently process the data.

In-memory distributed cache. The implementation includes an in-memory distributed data caching layer to provide fast access to memoized results. The use of in-memory caching is motivated by two observations: first, the number of subcomputations that need to be memoized is limited by the size of the sliding window; second, main memory is generally underutilized in data-centric computing,

thus creating an opportunity for reusing this resource [37]. We designed a simple distributed caching service that memoizes the outputs of sub-computations. The distributed cache is coordinated by a master node (in our case, the namenode of Hadoop), which maintains an index to locate the data items.

Fault-tolerance. Storing memoized results in the in-memory data cache is beneficial for performance, but it can lead to reduced memoization effectiveness when machines fail, as the loss of memoized results will trigger otherwise unnecessary recomputations. To avoid this situation, we built a fault-tolerant memoization layer, which, in addition to storing memoized data in the in-memory cache, creates two replicas of this data in persistent storage. The replication is transparently handled by a shim I/O layer that provides low-latency access to the in-memory cache when possible and falls back to the persistent copies when necessary.

Garbage collection. To ensure that the storage requirements remain bounded, we developed a garbage collector (implemented at the master node) that manages the space used by the memoization layer. The garbage collector can either automatically free the storage occupied by data items that fall out of the current window, or have a more aggressive user-defined policy.

Memoization-aware scheduling. Slider makes use of the memorization aware scheduler of Incoop (described in Section 2.7) to schedule Reduce tasks where the previously run objects are memoized.

4.8 Evaluation

Our evaluation answers the following questions:

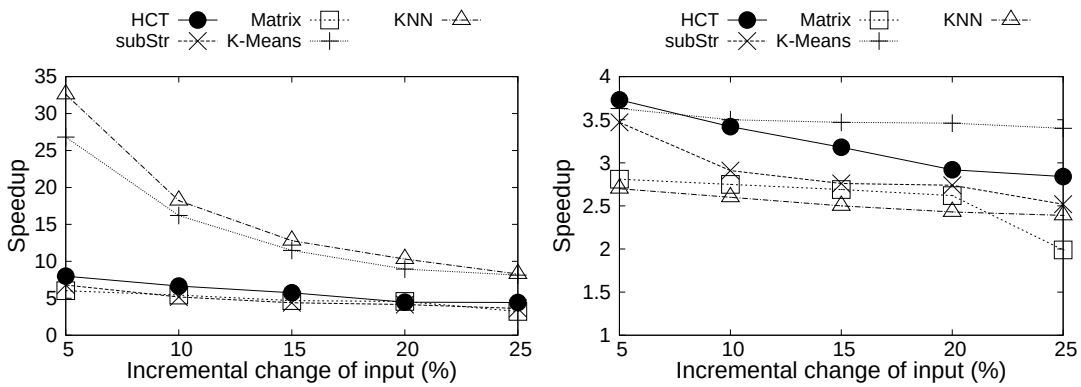
- How does the perfomance of Slider compare to recomputing over the entire window of data and with the memoization-based strawman approach? (§ 4.8.2)
- How effective are the optimizations we propose in improving the performance of Slider? (§ 4.8.3)
- What are the overheads imposed during a fresh run of an application? (§ 4.8.4)

4.8.1 Experimental Setup

Applications and dataset. Our micro-benchmarks span five MapReduce applications that implement typical data analysis tasks. Two are compute-intensive applications: K-means clustering **(K-Means)**, and *K*-nearest neighbors **(KNN)**. As input to these tasks we use synthetically generated data by randomly selecting points from a 50-dimensional unit cube. The remaining three are data-intensive applications: a histogram-based computation **(HCT)**, a co-occurrence matrix computation **(Matrix)**, and a string computation extracting frequently occurring sub-strings **(subStr)**. As input we use a publicly available dataset of Wikipedia [22].

Cluster setup. Our experiments run on a cluster of 25 machines. We configured Hadoop to run the namenode and the job tracker on a master machine, which was equipped with a 12-core Intel Xeon processor and 48 GB of RAM. The data nodes and task trackers ran on the remaining 24 machines equipped with AMD Opteron-252 processors, 4 GB of RAM, and 225 GB drives.

Measurements. We consider two types of measures: *work* and *time*. Work refers to the total amount of computation performed by all tasks (Map, contraction, and Reduce) and is measured as the sum of the active time for all the tasks. Time refers to the (end-to-end) total amount of running time to complete the job.



(a) Work – Append-only (A)

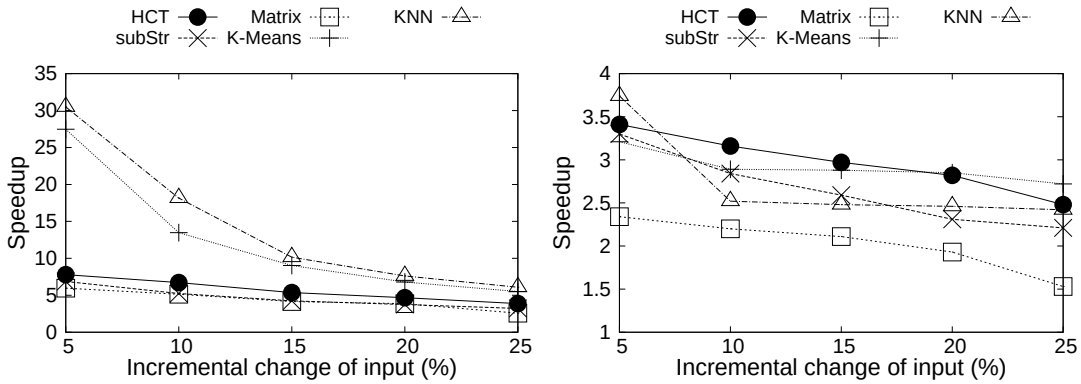
(b) Time – Append-only (A)

Figure 4.7: Performance gains of Slider compared to recomputing from scratch for the append-only windowing mode

Methodology. To assess the effectiveness of Slider, we measured the work and run-time of each micro-benchmark for different dynamic update scenarios, i.e., with different amounts of modified inputs, ranging from 5% to 25% of input data change. For the append-only case, a p% incremental change of the input data means that p% more data was appended to the existing data. For the fixed-width and variable-width sliding window cases, the window is moved such that p% of the input buckets are dropped from the window's beginning, and replaced with the same number of new buckets containing new content appended to the window's end.

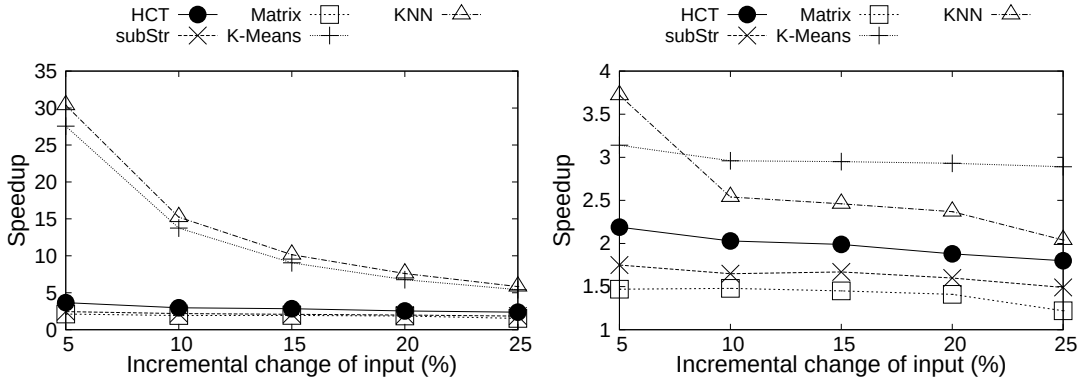
4.8.2 Performance Gains

Speedups w.r.t. recomputing from scratch. We first present the performance gains of Slider in comparison with recomputing from scratch. For the comparison, we compared the work and time of Slider to an unmodified Hadoop implementation. Figures 4.7, 4.8, and 4.9 show the performance gains for append-only, fixed-width, and variable-width windowing modes, respectively. These results show that the gains for compute-intensive applications (K-Means and KNN) are the most substantial, with time and work speedups between 1.5 and 35-fold. As expected, the



(a) Work – Fixed-width (F) (b) Time – Fixed-width (F)

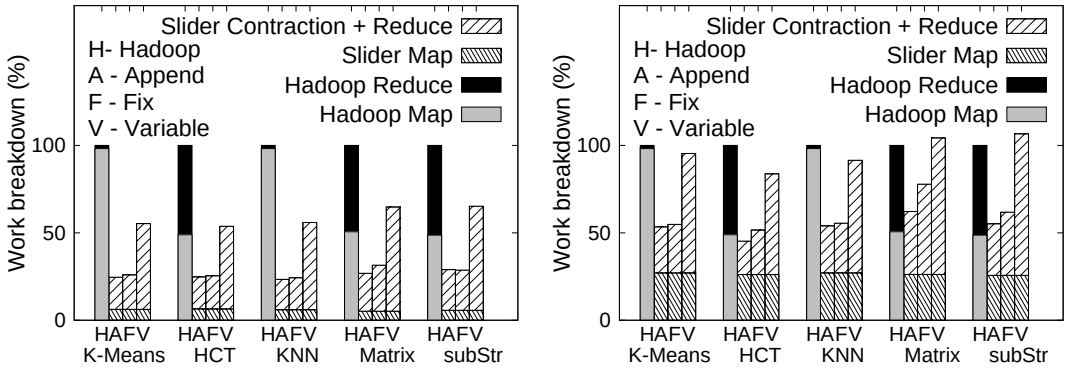
Figure 4.8: Performance gains of Slider compared to recomputing from scratch for the fixed-width windowing mode



(a) Work – Variable-width (V) (b) Time – Variable-width (V)

Figure 4.9: Performance gains of Slider compared to recomputing from scratch for the variable-width windowing mode

speedup decreases as the overlap between the old and the new window becomes smaller. Nonetheless, for these two benchmarks, even for a 25% input change, the speedup is still between 1.5 and 8-fold depending on the application. Speedups for data-intensive applications (HCT, Matrix, and subStr) are between 1.5-fold and 8-fold. Despite these also being positive results, the speedup figures are lower than in the case of applications with a higher ratio of computation to I/O. This is because the basic approach of memoizing the outputs of previously run sub-computations is effective at avoiding the CPU overheads but still requires some data movement to transfer the outputs of sub-computations, even if they were



(a) For 5% change in the input

(b) For 25% change in the input

Figure 4.10: Performance breakdown for work

memoized. The performance gains for variable-width sliding windows are lower than for the append-only and fixed-width window cases because updates require rebalancing the tree, and thus incur a higher overhead.

Performance breakdown. Figure 4.10(a) and Figure 4.10(b) show the normalized execution time breakdown in the incremental run with 5% and 25% changes in the input, respectively. The Map and Reduce contributions to the total time for the baseline vanilla Hadoop are shown in the bar labelled "H". The "H" bar breakdown shows that the compute-intensive applications (Kmeans and KNN) perform around 98% of the work in the Map phase, whereas the other applications (HCT, Matrix, and SubStr) perform roughly the same amount of work in each phase.

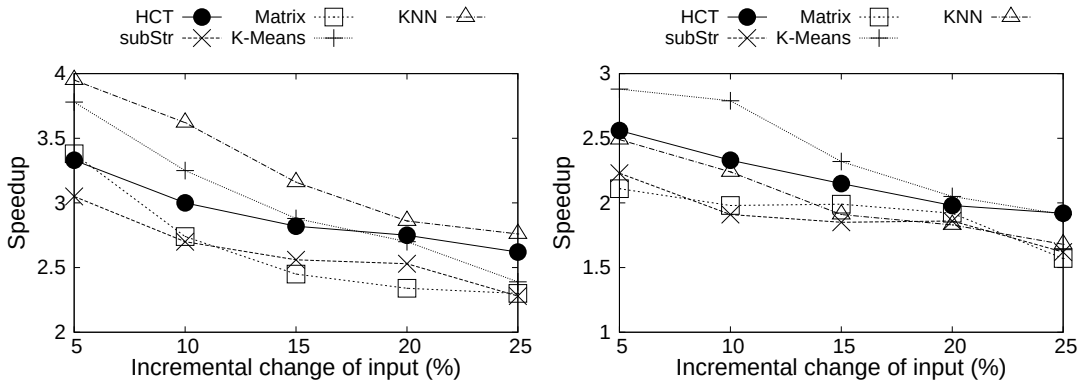
The same figures also show the breakdown for all three modes of operation ("A" for Append, "F" for Fixed-width, and "V" for Variable-width windowing), where the Slider-Map and Slider-contraction + Reduce portions in these bars represent the execution time computed as a percentage of the baseline Hadoop-Map and Hadoop-Reduce (H) times, respectively. In other words, the percentage execution time for Slider-Map is normalized to Hadoop-Map, while the percentage execution time for Slider-contraction + Reduce is normalized to Hadoop-Reduce.

For the Map phase of Slider, the breakdown shows that the percentage of Map work (compared to the non-incremental baseline) in the incremental run is proportional to the input change, as expected. In contrast, the work done by the Reduce phase is less affected by the amount of input change. In particular, the contraction + Reduce phase execution averages 31% of the baseline Reduce execution time (min: 18.39%, max: 59.52%) for 5% change, and averaged 43% (min: 26.39%, max: 80.95%) for 25% change across all three modes of operation.

Speedups w.r.t. memoization (strawman approach). Figures 4.11, 4.12, and 4.13 present the work and time speedup of Slider w.r.t. the memoization-based strawman approach (as presented in Section 4.3) for append-only, fixed-width, and variable-width windowing modes, respectively. The processing performed in the Map phase is the same in both approaches, so the difference lies only in the use of self-adjusting contraction trees instead of the strawman contraction tree. The work gains range from 2X to 4X and time gains range from 1.3X to 3.7X for different modes of operation with changes ranging from 25% to 5% of the input size. The work speedups for the compute-intensive applications (Kmeans and KNN) decrease faster than other applications as the input change increases because most performance gains were due to savings in the Map phase. Overall, although less pronounced than in the comparison to recomputing from scratch, the results show considerable speedups due to the data structures that are specific to each type of sliding window processing, when compared to the strawman approach.

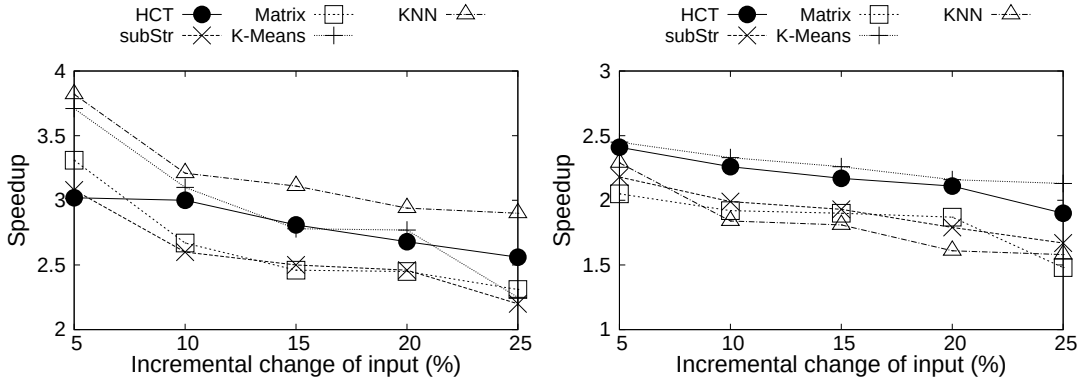
4.8.3 Effectiveness of Optimizations

We now evaluate the effectiveness of the individual optimizations in improving the overall performance.



(a) Work – Append-only (A) (b) Time – Append-only (A)

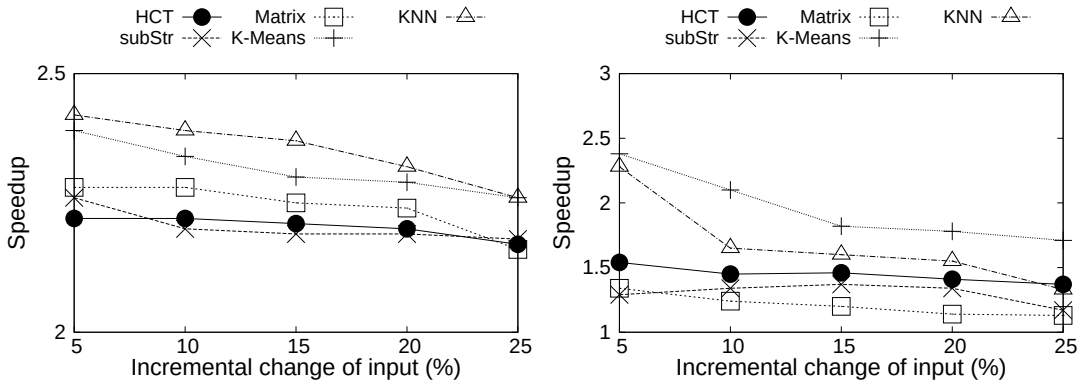
Figure 4.11: Performance gains of Slider compared to the memoization based approach (the strawman design) for the append-only windowing mode



(a) Work – Fixed-width (F) (b) Time – Fixed-width (F)

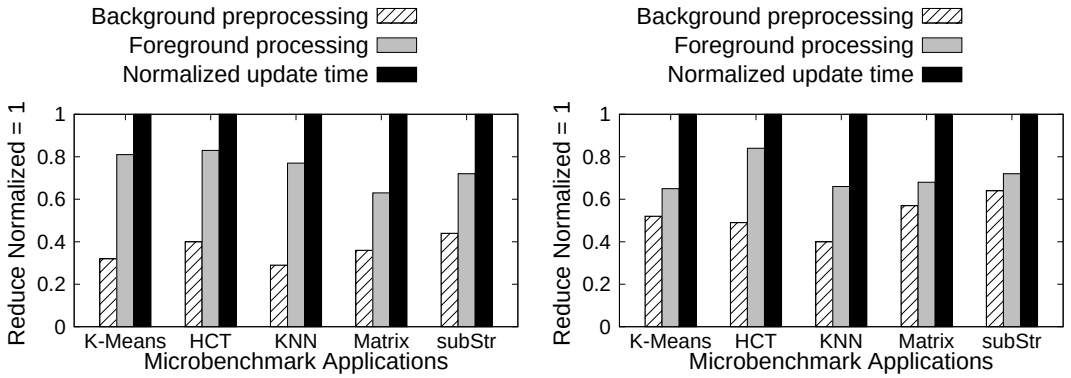
Figure 4.12: Performance gains of Slider compared to the memoization based approach (the strawman design) for the fixed-width windowing mode

Split processing. Slider is designed to take advantage of the predictability of future updates by splitting the work between background and foreground processing. To evaluate the effectiveness in terms of latency savings from splitting the execution, we compared the cost of executing with and without it, for both the append-only and the fixed-width window categories. Figures 4.14(a) and 4.14(b) show the time required for background preprocessing and foreground processing, normalized to the total time (total update time = 1) for processing the update without any split processing. Figure 4.14(a) shows this cost when a new input with 5% of the original input size is appended, for different benchmarking applications,



(a) Work – Variable-width (V) (b) Time – Variable-width (V)

Figure 4.13: Performance gains of Slider compared to the memoization based approach (the strawman design) for the variable-width windowing mode



(a) Append-only case (b) Fixed-width case

Figure 4.14: Effectiveness of Split processing

whereas Figure 4.14(b) shows the same cost for a 5% input change in the fixed-width window model. The results show that with the split processing model, we are on average able to perform foreground updates up to 25%-40% faster, while offloading around 36%-60% of the work to background pre-processing.

The results also show that the sum of the cost of background pre-processing and foreground processing exceeds the normal update time (total update time = 1) because of the extra merge operation performed in the split processing model. Our results show that the additional CPU usage for the append-only case is in the range of 1% to 23%, and 6% to 36% for the fixed-window processing.

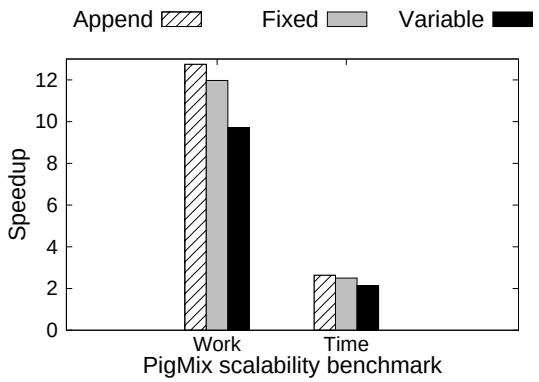


Figure 4.15: Query processing speedups using Slider

Data-flow query interface. To demonstrate the potential of incremental query-based sliding-window computations, we evaluate Slider using the PigMix [3] benchmark, which generates a long pipeline of MapReduce jobs derived from Pig Latin query scripts. We ran the benchmark in our three modes of operation with changes to 5% of its input. Figure 4.15 shows the resulting run-time and work speedups. As expected, the results are in line with the previous evaluation, since ultimately the queries are compiled to a set of MapReduce analyses. We observe an average speedup of 2.5X and 11X for time and work, respectively.

In-memory distributed memoization caching. For evaluating the effectiveness of performing in-memory data caching, we compared our performance gains with and without this caching support. In particular, we disabled the in-memory caching support from the shim I/O layer, and instead used the fault-tolerant memoization layer for storing the memoized results. Therefore, when accessing the fault-tolerant memoization layer, we incur an additional cost of fetching the data from the disk or network. Table 4.1 shows reduction in the time for reading the memoized state with in-memory caching for fixed-width windowing. This shows that we can achieve 50% to 68% savings in the read time by using the in-memory caching.

K-Means	HCT	KNN	Matrix	subStr
48.68%	56.87%	53.19%	67.56%	66.2%

Table 4.1: Read time reduction with memory caching

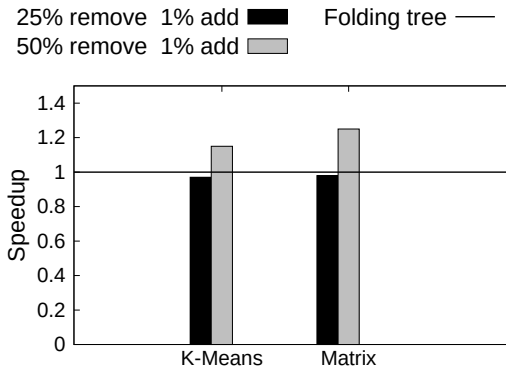
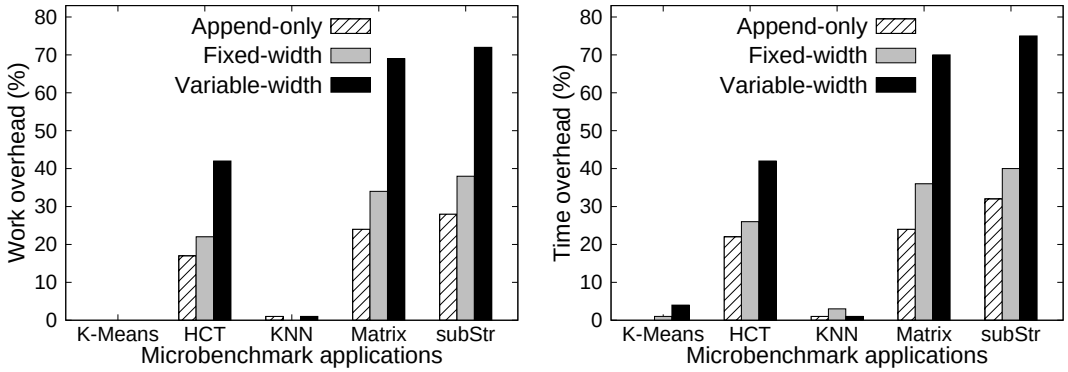


Figure 4.16: Randomized folding tree

Randomized folding tree. To evaluate the effectiveness of the randomized folding tree, we compared the gains of the randomized version with the normal folding tree (see Figure 4.16). We compare the performance for two update scenarios: reducing the window size by two different amounts (25% and 50%) and, in both cases, performing a small update adding 1% of new items to the window. We report work speedups for two applications (K-Means and Matrix), representing both compute and data-intensive applications.

The experiments show that a large imbalance (of 50% removals to 1% additions) is required for the randomized data structure to be beneficial. In this case, the randomized version leads to a performance improvement ranging from 15% to 22%. This is due to the fact that decreasing the window size by half also reduces the height of the randomized folding tree by one when compared to the original version (leading to more efficient updates). In contrast, with 25% removals to the same 1% additions the standard folding tree still operates at the same height as the randomized folding tree, which leads to a similar, but slightly better performance compared to the randomized structure.



(a) Work overhead (b) Time overhead

Figure 4.17: Performance overheads of Slider for the initial run

4.8.4 Overheads

Slider adds two types of overhead. First, the *performance overhead* for the initial run (a one time cost only). Second, the *space overhead* for memoizing intermediate results.

Performance overheads. Figure 4.17(a) and Figure 4.17(b) show the work and time overheads for the initial run, respectively. Compute-intensive applications (`K-means` & `KNN`) show low overhead as their run-time is dominated by the actual processing time and are less affected by the overhead of storing intermediate nodes of the tree. For data-intensive applications, the run-time overhead is higher because of the I/O costs for memoizing the intermediate results.

The overheads for the variable-width variant are higher than those for fixed-width computations, and significantly higher than the append case. This additional overhead comes from having more levels in the corresponding self-adjusting contraction tree.

Space overhead. Figure 4.18 plots the space overhead normalized by the input size. Again, the variable-width sliding-window computation shows the highest overhead, requiring more space than the other computations, for the same reasons

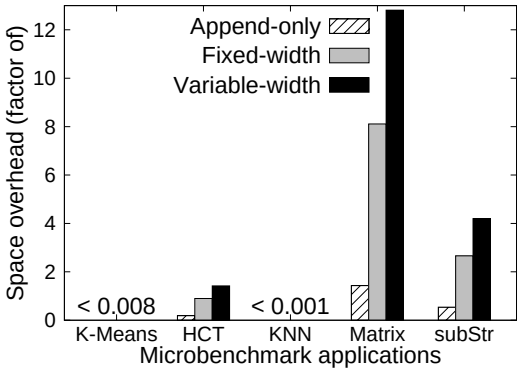


Figure 4.18: Space overheads of Slider

that were mentioned in the case of the performance overheads. Space overhead highly depends on the application. Matrix has the highest space overhead of 12X, while K-Means and KNN have almost no space overhead.

The results show that the overheads, despite being visible, are a one-time cost likely to be worth paying in workloads where the initial run is followed by many incremental runs that reap the benefit of incremental computations. Furthermore, the garbage collection policy can further limit the space overheads.

4.8.5 Analytical Comparison with memoization based approach and batch-based stream processing

We next analytically compare the performance gains of Slider with batch-based streaming processing (D-Streams), memoization-based approach (Incoop), and re-computation from scratch. Our comparison is based on a simulation that analyzes how many tasks have to be revisited in the incremental run for both Spark and Slider. We simulated a fixed configuration of a single MapReduce job with a single insertion and deletion as a function of window size. Figure 4.19 depicts number of items processed for Slider compared with Incoop [53] (a memoization-based system), D-Streams [149] (a state-of-the-art batch-based streaming platform), and recomputation from scratch. As expected, recomputing requires an amount of ef-

fort that is linear in the size of the window. Furthermore, Incoop is also showing a linear growth, due to the fact that all previously run tasks are revisited to check if their input has changed (even if this is a short-lived effort because in many cases the memoized results are still valid). In contrast, Slider has an effort that is logarithmic in the size of the window, because it propagates changes through the contraction tree.

When comparing our approach with D-Streams, we consider the following two cases for the analytical comparison: re-computation from scratch and inverse function. (We note that we did not perform empirically comparison between D-Streams and Slider because the underlying platforms have different characteristics for data management.) Our two comparisons with D-streams build on the observation that D-Streams, by default, re-computes over the entire window from scratch, even if there is overlap between two consecutive windows. For incremental updates to the output, D-Streams require an inverse function to exist [149], which may not be trivial to devise for complex computations. As shown in Figure 4.19, D-Streams with an inverse function requires processing of constant number of elements (equal to the window displacement size), and therefore, it is much better than Slider. However, for the default option of re-computation from scratch option, D-Streams require work linearly proportional to the size of the window. In our work, we address these limitations in batch-based streaming systems by proposing a transparent approach for incremental sliding window computations without requiring programmers to devise an application-specific inverse function.

4.9 Case Studies

We used Slider to evaluate three real-world case studies covering all three operation modes for sliding windows. Our case studies include: *(i)* building an informa-

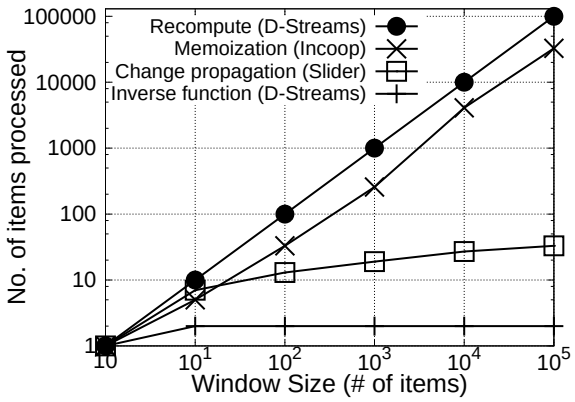


Figure 4.19: Analytical comparison of Slider with batch-based stream processing, memoization-based approach, and re-computation from scratch

tion propagation tree [132] for Twitter for append-only windowing; *(ii)* monitoring Glasnost [74] measurement servers for detecting ISPs traffic differentiation for fixed-width windowing; and *(iii)* providing peer accountability in Akamai NetSession [28], a hybrid CDN for variable-width windowing.

4.9.1 Information Propagation in Twitter

Analyzing information propagation in online social networks is an active area of research. We used Slider to analyze how web links are spread in Twitter, repeating an analysis done in [132].

Implementation. The URL propagation in Twitter is tracked by building an information propagation tree for posted URL based on Krackhardt's hierarchical model. This tree tracks URL propagation by maintaining a directed edge between a spreader of a URL and a receiver, i.e., a user "following" the account that posted the link.

Dataset. We used the complete Twitter snapshot data from [132], which comprises 54 million users, 1.9 billion follow-relations, and all 1.7 billion tweets posted by Twitter users between March 2006 and September 2009. To create a workload

Time interval	Mar'06 Jun'09	Jul'09 1-7	Jul'09 8-14	Jul'09 15-21	Jul'09 22-28
Tweets (M)	1464.3	74.2	81.5	79.4	85.6
Change	-	5.1%	5.3%	4.9 %	5.0%
Time speedup	-	8.9	9.2	9.42	9.25
Work speedup	-	14.22	13.67	14.22	14.34

Table 4.2: Summary of the Twitter data analysis

where data is gradually appended to the input, we partitioned the dataset into five non-overlapping time intervals as listed in Table 4.2. The first time interval captures all tweets from the inception of Twitter up to June 2009. We then add one week worth of tweets for each of the four remaining time intervals. For each of these intervals, an average cumulative change of 5% was performed with every new append.

Performance gains. We present the performance gains of incrementally building the information propagation tree using Slider in Table 4.2. The speedups are almost constant for the four time intervals, at about $8X$ for run-time and about $14X$ for work. The run-time overhead for computing over the initial interval is 22%.

4.9.2 Monitoring of a Networked System

Year 2011	Jan-Mar	Feb-Apr	Mar-May	Apr-Jun	May-Jul	Jun-Aug	Jul-Sep	Aug-Oct	Sep-Nov
#files	4033	4862	5627	5358	4715	4325	4384	4777	6536
Change size	4033	1976	1941	1441	1333	1551	1500	1726	3310
% change size	100 %	40.65 %	34.50 %	26.89 %	28.27 %	35.86 %	34.22 %	36.13 %	50.64 %
Time speedup	-	2.07	2.8	3.79	3.32	2.44	2.56	2.43	1.9
Work speedup	-	2.13	2.9	4.12	3.37	3.15	2.93	2.46	1.91

Table 4.3: Summary of the Glasnost network monitoring data analysis

Glasnost [74] is a system that enables users to detect whether their broadband traffic is shaped by their ISP. The Glasnost system tries to direct users to a nearby measurement server. Slider enabled us to evaluate the effectiveness of this server selection.

Implementation. For each Glasnost test run, a packet trace of the measurement traffic between the measurement server and the user's host is stored. We used this trace to compute the minimum round-trip time (RTT) between the server and the user's host, which represents the distance between the two. Taking all minimum RTT measurements of a specific measurement server, we computed the median across all users that were directed to this server.

Dataset. For this analysis, we used the data collected by one Glasnost server between January and November 2011 (see Table 4.3). We started with the data collected from January to March 2011. Then, we added the data of one subsequent month at a time and computed the mean distance between users and the measurement server for a window of the most recent 3 months. This particular measurement server had between 4,033 and 6,536 test runs per 3-month interval, which translate to 7.8 GB to 18 GB of data per interval.

Performance gains. We measured both work and time speedups as shown in Table 4.3. The results show that we get an average speedup on the order of 2.5X, with small overheads of less than 5%.

4.9.3 Accountability in Hybrid CDNs

Content distribution networks (CDN) operators like Akamai recently started deploying hybrid CDNs, which employ P2P technology to add end user nodes to the distribution network, thereby cutting costs as fewer servers need to be deployed.

% clients online to upload logs	100%	95%	90%	85%	80%	75%
Time speedup	1.72	1.85	1.89	2.01	2.1	2.24
Work speedup	2.07	2.21	2.29	2.44	2.58	2.74

Table 4.4: Akamai NetSession data analysis summary

However, this also raises questions about the integrity of the answers that are provided by these untrusted clients [28].

Implementation. Aditya et al. [28] presented a design of a hybrid CDN that employs a tamper-evident log to provide client accountability. This log is uploaded to a set of servers that need to audit the log periodically using techniques based on PeerReview. Using Slider, we implemented these audits as a variable-sized sliding-window computation, where the amount of data in a window varies depending on the availability of the clients to upload their logs to the central infrastructure in the hybrid CDN.

Dataset. To evaluate the effectiveness of Slider, we used a synthetic dataset generated using trace parameters available from Akamai's NetSession system, a peer-assisted CDN operated by Akamai (which currently has 24 million clients). From this data set, we selected the data collected in December 2010. However, due to the limited compute capacity of our experimental setup, we scaled down the data logs to 100,000 clients. In addition to this input, we also generated logs corresponding to one week of activity with a varying percentage of clients (from 100% to 75%) uploading their logs to the central infrastructure, so that the input size varies across weeks. This allows us to create an analysis with a variable-width sliding window by using a window corresponding to one month of data and sliding it by one week in each run.

Performance gains. Table 4.4 plots the performance gains for log audits for a different percentage of client log uploads for the 5th week. We observe a speedup of

2X to 2.5X for a fraction of clients uploading the log in the final week that varied from 75% to 100%. Similarly, the run-time speedups are between 1.5X and 2X.

4.10 Related Work

We compare our work to two major classes of systems that are suitable for sliding window analytics: trigger-based windowing systems, and batch-based windowing systems. We conclude with a broader comparison to incremental computation mechanisms.

Trigger-based windowing systems. These systems follow *record-at-a-time* processing model, where every new data entry triggers a state change and possibly produces new results, while the application logic, known as a *standing query*, may run indefinitely. This query can be translated into a network with stateless and/or stateful nodes. A stateful node updates its internal state when processing incoming records, and emits new records based on that state. Examples of such systems include Storm [18], S4 [4], StreamInsight [32], Naiad [113], Percolator [121], Photon [38], and streaming databases [40]. Despite achieving low latency, these systems also raise challenges [149]:

(1) Fault-tolerance: To handle faults, these systems either rely on *replication with synchronization* protocols such as Flux [136] or Borealis's DPC [40], which have a high overhead, or on checkpointing *upstream backup* mechanisms, which have a high recovery time. In addition, neither fault tolerance approach handles stragglers.

(2) Semantics: In a trigger-based system, it can be difficult to reason about global state, as different nodes might be processing different updates at different times. This fact, coupled with faults, can lead to weaker semantics. For instance, S4 provides at most once semantics, and Storm [18] provides at-least-once semantics. Naiad [113] is an exception in this category, which provides strong semantics. Na-

iad makes use of timely dataflow using which the global state of a computation's progress is always known.

(3) Programming model: The *record-at-a-time* programming model in trigger-based systems requires the users to manage the intermediate state and wire the query network topology manually. Furthermore, programmers need to understand how to update the output of each node in the query network as its input evolves. The design of the update logic is further complicated by the weak semantics provided by the underlying platform. While supporting incremental computation for aggregate operations is straightforward, this can be very challenging for non-trivial computations like matrix operations or temporal joins [127, 20].

Batch-based windowing systems. These systems model sliding window analytics as a series of deterministic batch computations on small time intervals. Such systems have been implemented both on top of trigger-based systems (e.g., Trident [20] built over Storm [18] or TimeStream [127] built over StreamInsight [32]) and systems originally designed for batch processing (e.g., D-Streams [149] built over Spark [148] or MapReduce online [62] and NOVA [115] built over MapReduce [67]). These systems divide each application into a graph of short, deterministic tasks. This enables simple yet efficient fault recovery using recomputation and speculative execution to handle stragglers [150]. In terms of consistency, these systems trivially provide "exactly-once" semantics, as they yield the same output regardless of failures. Finally, the programming model is the same as the one used by traditional batch processing systems.

We build on this line of research, but we observe that these systems are not geared towards incremental sliding window computation. Most systems recompute over the entire window from scratch, even if there is overlap between two consecutive windows. The systems that allow for an incremental approach require an inverse function to exist [149], which may not be trivial to devise for complex

computations. In this work, we address these limitations in batch-based windowing systems by proposing a transparent solution for incremental sliding window analytics.

Incremental Computation. While there exists some prior work on enabling incremental computations in batch processing systems, this work did not leverage the particular characteristics of sliding windows, among other important differences. In more detail, incremental computation in batched processing systems such as Incoop (Chapter 2), Haloop [56], Nectar [84], DryadInc [122] requires linear time in the size of the input data, even to process a small slide in the window. The reasons for this are twofold: Firstly, these systems assume that inputs of consecutive runs are stored in separate files and simply compute their `diffs` to identify the input changes. The change detection mechanism relies on techniques such as content-based chunking (as in Incoop using IncHDFS [53]), which requires performing linear work in the size of the input [50]. In contrast, sliding window computation provides diffs naturally, which can be leveraged to overcome the bottleneck of identifying changes. Second, and more importantly, these systems do not perform change propagation, relying instead on memoization to recover previously computed results. Consequently, they require visiting all tasks in a computation even if the task is not affected by the modified data, i.e. the delta, thus requiring an overall linear time. In contrast, this chapter proposes an approach that only requires time that is linear in the delta, and not the entire window, and we propose new techniques that are specific to sliding window computation.

4.11 Limitations and Future Work

Both Incoop and Slider rely on Combiners for aggregating the intermediate data by performing local reduction, and henceforth, they reduce the volume of the data

that needs to be processed by subsequent stages. Although, this property holds for almost all Combiners we analyzed in practice (Apache Hama [1] and Mahout [2]), there might be cases where a Combiner lacks the ability to perform aggregation. For such cases, our approach of building Contraction trees may not yield any performance benefits.

Secondly, rotation contraction trees require commutativity in addition to the associativity of the combiner invocations to be able to rotate the buckets. Both properties were held by Combiner functions in most applications we analyzed in Apache Hama [1] and Mahout [2]. Nonetheless, this restriction can be lifted using the most general solution for a variable-width window.

Furthermore, while Slider is a significant step towards supporting incremental sliding window analytics, plenty of opportunities remain to further increase the range of supported workloads.

First, we would like to incorporate the notion of assigning weights to the data elements, which can change as the window moves. This would also be amenable to incremental computations, but would require a change in the framework that would enable a new class of computations for weighted sliding window computations.

Secondly, as we know that a fundamental challenge for data analytics is to be able to efficiently tune and debug multi-step dataflows. We would like to extend the infrastructure of Slider to provide time travel debugging using incremental computation. By extending the notion of lineage in the dependence graph, we should be able to discover record-level data lineage for debugging errors in analytics.

4.12 Summary

In this chapter, we presented self-adjusting contraction trees for incremental sliding window analytics. The idea behind our approach is to structure distributed data-parallel computations in the form of balanced trees that can perform updates in asymptotically sublinear time, thus much more efficiently than recomputing from scratch. We present several algorithms and data structures for supporting this type of computations, describe the design and implementation of Slider, a system that uses our algorithms, and present an extensive evaluation showing that Slider is effective on a broad range of applications. This shows that our approach provide significant benefit to sliding window analytics, without requiring the programmer to write the logic for handling updates.

CHAPTER 5

iThreads: Incremental Multithreading

In this chapter, we describe the design, implementation, and evaluation of iThreads, a threading library for parallel incremental computation. iThreads supports unmodified shared-memory multithreaded programs: it can be used as a replacement for pthreads by a simple exchange of libraries linked, without even recompiling the application code.

This chapter is organized as follows. We first motivate the design of iThreads in Section 5.1. We next briefly highlight the contributions of iThreads in Section 5.2. Thereafter, we present an overview of our approach 5.3. We next detail the memory consistency and synchronization model for iThreads in Section 5.4. We then present our algorithms for incremental multithreading in Section 5.5. Thereafter, we present an implementation of the algorithms in Section 5.6. Next, we present an empirical evaluation of iThreads in Section 5.7 and case-studies in Section 5.8. We present the related work in Section 5.9. Finally, we present the limitations of iThreads in Section 5.10, and conclude in Section 5.11.

5.1 Motivation

The advent of multicores has made parallel programs ubiquitous. Parallel programs are being used in a variety of domains, from scientific computing to computer-aided design and engineering. To take advantage of incremental computation in

multithreaded programs automatically, only recently researchers in the programming-languages community have proposed two compiler- and language-based approaches for incremental multithreading [57, 86].

An important lesson from these recent proposals is that by leveraging a language-based approach, these two prior proposals [57, 86] have achieved substantial speedups, thus establishing that the promise of incremental computation can be realized in parallel programs. Specifically, these systems enable efficient and correct incremental updates to the output through the use of new programming languages with special data types (§5.9), and by requiring a *strict* fork-join programming model, where threads communicate *only* at end points (i.e., when forking/joining).

These choices reflect a difficult design tradeoff: they provide the compiler and runtime system with the information required to maximize reuse, but they also impose a cost on the programmer, who has to provide appropriate type annotations and, in some cases, also application-specific functions to safely implement the new type system. Furthermore, due to the restricted programming model, they also preclude support for many existing shared-memory multithreaded programs and synchronization primitives (such as *R/W locks*, *mutexes*, *semaphores*, *barriers*, and *conditional wait/signal*).

In this chapter, we instead target increased generality, and to this end, we propose an operating systems-based approach to parallel incremental computation. More specifically, we present iThreads, a threading library for parallel incremental computation, which achieves the following goals.

- Practicality: iThreads supports the shared-memory multi-threaded programming model with the full range of synchronization primitives in the `POSIX` API.
- Transparency: iThreads supports unmodified programs written in C/C++, without requiring the use of a new language with special data types.

- Efficiency: iThreads achieves efficiency, without limiting the available application parallelism, as its underlying algorithms for incremental computation are *parallel* as well.

5.2 Contributions

In this chapter, we present the design of iThreads. iThreads relies on recording the data and control dependencies in a computation during the initial run by constructing a *Concurrent Dynamic Dependence Graph (CDDG)*. The CDDG tracks the input data to a program, all sub-computations (a *sub-computation* is a unit of the computation that is either reused or recomputed), the data flow between them, and the final output. For the incremental run, a (parallel) *change propagation* algorithm updates the output and the CDDG by identifying sub-computations that are affected by the input changes and recomputing only those sub-computations. Overall, we make the following contributions:

- We present the design of our *parallel* algorithms for incremental multithreading. Our algorithms record the intra- and inter-thread control and data dependencies using a concurrent *dynamic* data dependency graph, and use the graph to incrementally update the output as well as the graph on input changes.
- We have implemented our algorithms by leveraging operating system mechanisms encapsulated in a dynamically linkable shared library using the Dthreads infrastructure [109], which we call iThreads.
- We empirically demonstrate the effectiveness of iThreads by applying it to applications of multithreaded benchmark suites (PARSEC [54] and Phoenix [131]) and case-studies.

5.3 Overview

In this section, we present an overview of the basic design, and design choices for iThreads.

5.3.1 iThreads Overview

We base our design on `POSIX` threads, or `pthreads`, a widely used threading library for shared-memory multithreading with a rich set of synchronization primitives. This choice has several advantages, namely that the `POSIX` interface is portable across different architectures and OSes, and also that `pthreads` is used as the underlying threading library for many higher level abstractions for parallel programming (e.g., `OpenMP`). Therefore, our design choice to be compatible with `pthreads` targets a large class of existing parallel programs.

The iThreads library is easy to use (see Figure 5.1 for the workflow): the user just needs to preload iThreads to replace `pthreads` by using the environment variable `LD_PRELOAD`. The dynamically linkable shared library interface allows existing executables to benefit from iThreads.

For the first run of a program (or the *initial run*), iThreads computes the output from scratch and records an execution trace. All subsequent runs for the program are *incremental runs*. For an incremental run, the user modifies the input and specify the changes; e.g., assuming that the program reads the input from a file, the user specifies the `offset` and `len` for the changed parts of the file. Thereafter, iThreads incrementally updates the output based on the specified input changes and the recorded trace from the previous run.

```
$ LD_PRELOAD=iThreads.so                    // preload iThreads
$./<program_executable> <input-file>        // initial run
$ emacs <input-file>                        // input modified
$ echo "<off> <len>" >> changes.txt          // specify changes
$./<program_executable> <input-file>        // incremental run
```

Figure 5.1: How to run an executable using iThreads

5.3.2 The Basic Approach

Our design adapts the principles of self-adjusting computation [24] for shared-memory multithreading, and also makes use of techniques from record-replay systems employed for reliable multithreading (§5.9). At a high level, the basic approach proceeds in the following three steps:

1. Divide a computation into a set of sub-computations N.

2. During the initial run, record an execution trace to construct a Concurrent Dynamic Dependence Graph (or CDDG). The CDDG captures a partial order $O = (N, \rightarrow)$ among sub-computations with the following property: given a sub-computation n (where $n \in N$) and the subset of sub-computations M that precede it according to $\rightarrow$, i.e., $M = \{M \subset N \mid \forall m \in M, m \rightarrow n\}$, if the inputs to all $m \in M$ are unchanged and the incremental run follows the partial order $\rightarrow$, then n's input is also unchanged and we can reuse n's memoized effect without recomputing n.

3. During the incremental run, propagate the changes through the CDDG. That is, the incremental run follows an order that is consistent with the recorded partial order $\rightarrow$, reusing the sub-computations whose input is unchanged and re-computing the ones whose input has changed.

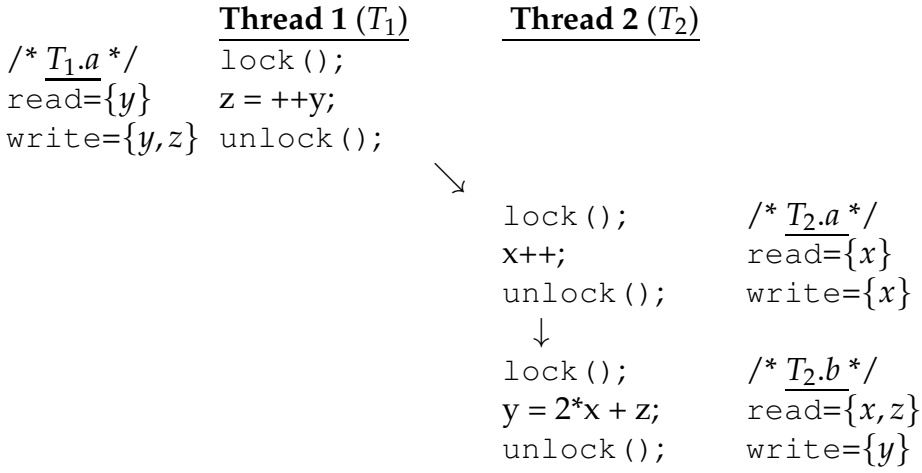


Figure 5.2: A simple example of shared-memory multithreading

Case	Input	Thread schedule	Sub-computations	
			Reused	Recomputed
A	x, y^*, z	$T_1.a \rightarrow T_2.a \rightarrow T_2.b$	$T_2.a$	$T_1.a$, $T_2.b$
B	x, y, z	$(T_2.a \rightarrow T_2.b \rightarrow T_1.a)^*$	$T_2.a$	$T_1.a$, $T_2.b$
C	x, y, z	$T_1.a \rightarrow T_2.a \rightarrow T_2.b$	$T_1.a$, $T_1.b$, $T_2.a$	—

Figure 5.3: For the incremental run, some cases with changed input or thread schedule (changes are marked with *)

5.3.3 Example

To provide more information on how our approach works, we present a simple example, shown in Figure 5.2. The example considers a multi-threaded execution with two threads (T_1 & T_2) modifying three shared variables (x, y, & z) using a lock.

Step #1: Identifying sub-computations. We divide a thread execution into sub-computations at the boundaries of `lock()/unlock()`. (We explain this design choice in §5.4.) We identify these sub-computations as $T_1.a$ for thread T_1, and $T_2.a$ & $T_2.b$ for thread T_2. For the initial run, let us assume that thread T_2 acquired the lock after execution of sub-computation $T_1.a$. This resulted in the following thread schedule for sub-computations: $T_1.a \rightarrow T_2.a \rightarrow T_2.b$.

Step #2: Construct the CDDG. To understand the dependencies that need to be recorded to build the CDDG, we consider incremental runs with changes either in the input data or the thread schedule (shown in Figure 5.3).

We first consider the case of change in the input data. An important function of the CDDG is to propagate the changes through the graph by determining whether a sub-computation is transitively affected by the input change. For example consider **case *A*** in Figure 5.3, when the value of variable y is changed—in this case, we need to recompute $T_1.a$ because it reads the modified value of y. In contrast, we can still reuse $T_2.a$ because it is independent of y and also not affected by the writes made by $T_1.a$. However, we need to recompute $T_2.b$ even though it does not directly depends on y, since it is still transitively affected (via modified z) by the writes made by $T_1.a$. Therefore, the CDDG needs to record *data dependencies* (meaning the modifications in an incremental run to a value that is read by a sub-computation) to determine whether a sub-computation can be reused or if it has to be recomputed.

We next consider the case of change in the thread schedule. In the general case, multi-threaded programs are non-deterministic because the OS scheduler is free to interleave sub-computations in different ways. As a result, a problem can arise if the initial and the incremental runs follow different schedules. This might unnecessarily alter the shared state, and therefore cause unnecessary re-computations even without any changes to the input. For example consider **case *B*** in Figure 5.3: if thread T_1 acquires the lock after the execution of $T_2.b$ (i.e., a changed thread schedule of $T_2.a \rightarrow T_2.b \rightarrow T_1.a$) then sub-computations $T_1.a$ and $T_2.b$ need to be recomputed because the value of variable y has changed. Therefore, (and as observed by prior work on deterministic multithreading (§5.9)) the CDDG should also record all *happens-before* order among synchronization events to ensure that, given unchanged input and that all threads acquire locks in the same order as

Algorithm 2: Basic algorithm for the incremental run

```
1  dirty-set ← {changed input};
2  executeThread()
3  forall the sub-computations in the thread do
4      // Check a sub-computation's validity in happens-before order
5      if (read-set ∩ dirty-set) then
6          – recompute the sub-computation
7          – add the write-set to the dirty-set
8      else
9          – skip execution of the sub-computation
10         – write memoized value of the write-set to the address space
```

dictated by $\rightarrow$, all sub-computations are efficiently reused (as shown in **case *C*** in Figure 5.3).

Step #3: Change propagation. The previous observations allow us to reach a refined explanation of our basic algorithm (see Algorithm 2). The starting point is the CDDG that records the happens-before order ($\rightarrow$) between sub-computations, according to the synchronization events. Furthermore, data dependencies are recorded implicitly in the CDDG by storing the read and write sets: if we know what data is read and written by each sub-computation, we can determine whether a data dependency exists, i.e., if a sub-computation is reading data that was modified by another sub-computation. Therefore, the incremental run visits sub-computations in an order that is compatible with $\rightarrow$, and, for each sub-computation, uses the read and write sets to determine whether part of its input was modified during the incremental run. If the read-set is modified then the sub-computation is recomputed, otherwise we skip the execution of the sub-computation, and directly write the memoized value of the write-set to the address space.

5.4 System Model

Memory consistency model. As we explained in the previous section, the happens-before order $\rightarrow$ implicitly records read-after-write data dependencies between sub-computations using the read and write sets. The efficiency of the mechanism that

records $\rightarrow$ depends on the memory model we use, and consequently on the granularity of sub-computations. As a design choice, our approach relies on the use of the Release Consistency [81] (RC) memory model to define the granularity of sub-computations. To understand why this is the case, consider a possible option of using a strict memory model such as Sequential Consistency [101] (SC). Under the SC model, one would have to define a single instruction as the granularity of a sub-computation. This is because any write made by a thread can potentially affect the execution of another thread. Intercepting this inter-thread communication would be prohibitively expensive, essentially requiring tracking the order in which threads access the shared memory at the granularity of individual instructions.

To efficiently record the communication between threads of the program, our approach relies on using a memory model that restricts the points for inter-thread communication (i.e., the points at which updates performed by one thread to the shared memory is visible to all other threads). In particular, we weaken the memory model by implementing RC instead of SC. The RC memory model requires writes made by one thread to become visible to another thread only at synchronization points, thus restricting inter-thread communication to such points. Therefore, this allows us to define the granularity of a sub-computation at the boundaries of synchronization points, which is essential to achieving feasible runtime overheads.

Note that the RC memory model still guarantees correctness and liveness for applications that are data-race-free [29]. In fact, the semantics provided by `iThreads` is no more restrictive than `pthreads` semantics [16], which mandates that all accesses to shared data structures must be properly synchronized using `pthreads` synchronization primitives, and which guarantees only that any updates become visible to other threads when invoking a `pthreads` synchronization primitive.

Synchronization model. We support the full range of synchronization primitives in the `pthreads` API. However, due to the weakly consistent RC memory model,

our approach does not support *ad-hoc synchronization mechanisms* [147] such as user-defined spin locks. While ad-hoc synchronization mechanisms have been shown to be error-prone in practice, introducing bugs or performance issues [147], they are nonetheless used for either flexibility or performance reasons in some applications. In many such cases, it may be possible to replace these mechanisms with equivalent `pthreads` API calls; for instance, user-defined spin locks can be replaced by `pthreads` spin lock API calls [17]. An interesting direction for future work would be to extend `iThreads` with an interface for specifying ad-hoc synchronization primitives (e.g., at the level of `gcc`'s built-in atomic primitives) to identify thunk boundaries, thereby paving the way for transparent support for lock-free and wait-free data structures in `iThreads`.

5.5 Algorithms

We next formally present two parallel algorithms for incremental multithreading. The first algorithm is for the initial run that executes the program from scratch and constructs the CDDG. The second algorithm is for the incremental run that performs change propagation through the CDDG. The core of our approach is the CDDG, which we explain first.

5.5.1 Concurrent Dynamic Dependence Graph (CDDG)

The CDDG is a directed acyclic graph with vertices representing sub-computations (or thunks), and two types of edges to record dependencies between thunks: happens-before edges and data-dependence edges. We next explain how to derive vertices and edges.

Thunks (or sub-computations). We define a *thunk* as the sequence of instructions executed by a thread between two `pthreads` synchronization API calls. We model

an execution of thread t as a sequence of thunks (L_t). Thunks in a thread are totally ordered based on their execution order using a monotonically increasing thunk counter (α). We refer to a thunk of thread t using the counter α as an index in the thread execution sequence (L_t), i.e., $L_t[\alpha]$.

Happens-before edges. We derive happens-before edges by modeling synchronization primitives as *acquire* and *release* operations. During synchronization, a synchronization object S is *released* by one set of threads and subsequently *acquired* by a corresponding set of threads blocked on the synchronizing object. For example, an `unlock(S)` operation releases S and a corresponding `lock(S)` operation acquires it. Similarly, all other synchronization primitives can also be modeled as acquire and release operations [80, 123].

Under the acquire-release relation, a release operation happens-before the corresponding acquire operation. Given that a thunk's boundaries are defined based on synchronization primitives, the acquire and release operations also establish the happens-before ordering between thunks of different threads. In addition, thunks of the same thread are ordered by their execution order.

More specifically, there are two types of happens-before edges: control edges, which record the intra-thread execution order, and synchronization edges, which record explicit inter-thread synchronization events. Formally, two thunks $L_{(t_1)}[\alpha]$ and $L_{(t_2)}[\beta]$ are connected by a

- *control edge* iff they belong to the same thread ($t_1 = t_2$) and $L_{(t_1)}[\alpha]$ was executed immediately before $L_{(t_2)}[\beta]$; and by a
- *synchronization edge* iff $L_{(t_1)}[\alpha]$ releases a synchronization object S and $L_{(t_2)}[\beta]$ is the thunk that acquires S.

Data-dependence edges. Data dependencies are tracked to establish the *update-use relationship* between thunks. Intuitively, such a relationship exists between two

Algorithm 3: The initial run algorithm

```
1  ∀S, ∀i ∈ {1, ..., T} : C_S[i] ← 0; // All sync clocks set to zero executeThread(t)
2  begin
3      initThread(t);
4      while t has not terminated do
5          startThunk(); // Start new thunk
6          repeat
7              Execute instruction of t;
8              if (instruction is load or store) then
9                  onMemoryAccess();
10         until t invokes synchronization primitive;
11         endThunk(); // Memoize the end state of thunk
12         α ← α + 1; // Increment thunk counter
13         // Let S denote invoked synchronization primitive
14         onSynchronization(S);
```

thunks if one reads data written by the other and they can be ordered based on the happens-before order. More formally, for a thunk $L_t[\alpha]$, the *read-set* $L_t[\alpha].R$ and the *write-set* $L_t[\alpha].W$ are the set of addresses that were read-from and written-to, respectively, by the thread t while executing the thunk. Two thunks $L_{(t_1)}[\alpha]$ and $L_{(t_2)}[\beta]$ are then connected by a

- *data-dependence edge* iff $L_{(t_2)}[\beta]$ is reachable from $L_{(t_1)}[\alpha]$ via happens-before edges and $L_{(t_1)}[\alpha].W \cap L_{(t_2)}[\beta].R \neq \emptyset$.

5.5.2 Algorithm for the Initial Run

During the initial run, we record the execution of the program to construct the CDDG. Algorithm 3 presents the high-level overview of the initial run algorithm, and details of subroutines used in the algorithm are presented in Algorithm 4. The algorithm is executed by multiple threads in parallel. The algorithm employs run-time techniques to derive the information needed for the CDDG. In particular, during a thread execution, the thread traces memory accesses on `load/store` instructions (using routine `onMemoryAccess()`), and adds them to the read and the write set of the executing thunk. (In our implementation, described in §5.6, we actually derive the read and write sets at the granularity of memory pages

Algorithm 4: Subroutines for the initial run algorithm

```
1  initThread(t)
2  begin
3      α ← 0; // Initializes thunk counter (α) to zero
4      ∀i ∈ {1, ..., T} : C_t[i] ← 0; // t's clock set to zero
5  startThunk()
6  begin
7      C_t[t] ← α; // Update thread clock with thunk counter (α) value
8      ∀(i ∈ {1, ..., T}) : L_t[α].C[i] ← C_t[i]; // Update thunk's clock
9      L_t[α].R/W ← ∅; // Initialize read/write sets to empty set
10 onMemoryAccess()
11 begin
12     if load then
13         L_t[α].R ← L_t[α].R ∪ {memory-address}; // Read access
14     else
15         L_t[α].W ← L_t[α].W ∪ {memory-address}; // Write access
16 endThunk()
17 begin
18     memo (L_t[α].W) ← content(L_t[α].W); //(globals and heap)
19     memo (L_t[α].Stack) ←content (Stack);
20     memo (L_t[α].Reg) ←content (CPU_Registers);
21 onSynchronization(S)
22 begin
23     switch Syncronization type do
24         case release(S):
25             // Update S's clock to hold max of its and t's clocks
26             ∀i ∈ {1, ..., T} : C_S[i] ← max(C_S[i], C_t[i]);
27             sync(S); // Perform the synchronization
28         case acquire(S):
29             sync(S); // Perform the synchronization
30             // Update t's clock to hold max of its and S's clocks
31             ∀i ∈ {1, ..., T} : C_t[i] ← max(C_S[i], C_t[i]);
```

using the OS memory protection mechanism.) The thread continues to execute instructions and perform memory tracing until a synchronization call is made to the `pthreads` library. At the synchronization point, we define the end point for the executing thunk and memoize its end state (using routine `endThunk()`). Thereafter, we let the thread perform the synchronization. Next, we start a new thunk and repeat the process until the thread terminates.

To infer the CDDG, control and synchronization edges are derived by ordering thunks based on the happens-before order. To do so, we use vector clocks (C) [112] to record a partial order that defines the happens-before relationship between thunks during the initial run, and in the incremental run we follow this

partial order to propagate the input changes. Our use of vector clocks is motivated by its efficiency for recording a partial order in a decentralized manner, rather than having to serialize all synchronization events in a total order.

Our algorithm maintains one vector clock for each threads, thunks, and synchronization objects. (A vector clock is an array of size T, where T denotes the number of threads in the system, which are numbered from 1 to T.) Each thread t has a vector clock, called its *thread clock* C_t, to track its local logical time, which is updated at the start of each thunk (using routine `startThunk()`) by setting $C_t[t]$ to the thunk index α. Further, each thunk $L_t[\alpha]$ has a *thunk clock* $L_t[\alpha].C$ that stores a snapshot of $C_t[t]$ to record the thunk's position in the CDDG.

Finally, each synchronization object S has a *synchronization clock* C_S that is used to order release and acquire operations (see `onSynchronization()`). More precisely, if a thread t invokes a release operation on S, then t updates C_S to the component-wise maximum of its own thread clock C_t and C_S. Alternatively, if t invokes an acquire operation on S, it updates its own thread clock C_t to the component-wise maximum of C_t and S's synchronization clock C_S. This ensures that a thunk acquiring S is thus always ordered after the last thunk to release S.

At the end of the initial run algorithm, the CDDG is defined by the read/write sets and the thunk clock values of all thunks.

5.5.3 Algorithm for the Incremental Run

The incremental run algorithm takes as input the CDDG ($\forall t : L_t$) and the modified input (named the dirty set M), and performs change propagation to update the output as well as the CDDG for the next run. As explained in the basic change propagation algorithm (Algorithm 2), each thread transitions through its list of thunks by following the recorded happens-before order to either reuse or recom-

pute thunks. To make this algorithm work in practice, however, we need to address the following three limitations.

(1) Missing writes. When a thunk is recomputed during the incremental run, it may happen that the executing thread no longer writes to a previously written location because of a data-dependent branch. For such cases, our algorithm should update the dirty set with the new write-set of the thunk as well as the *missing writes*. The missing writes include the set of memory locations that were part of the thunk's write-set in the previous run, but are missing in the current write-set.

(2) Stack dependencies. As briefly mentioned previously, we transparently derive read and write sets by tracking the global memory region (heap/globals) using the OS memory protection mechanism (detailed in §5.6). Unfortunately, this mechanism is inefficient for tracking the per-thread stack region (which usually resides in a single page storing local variables) because the stack follows a *push/pop* model, where the stack is written (or gets dirty) when a call frame is pushed or popped, even without a local variable being modified. To avoid the overheads of tracking these accesses to the stack, we do not track the stack. Instead, we follow a conservative strategy to capture the intra-thread data dependencies. In our design, once a thunk is recomputed (or invalidated) in a thread, all remaining thunks of the thread are also invalidated in order to capture a possible change propagation via local variables.

(3) Control flow divergence. During the incremental run, it may happen that the control flow diverges from the previously recorded execution. As a result of the divergence, new thunks may be created or existing ones may be deleted. The algorithm we propose addresses the stack problem and, more generally, the control flow divergence by reusing a prefix of each thread and striving to make this prefix as large as possible using a simple state machine approach, as explained next.

Algorithm 5: The incremental run algorithm

```
   Data: Shared dirty set M ← { modified pages } and L_t
1  ∀S, ∀i ∈ {1, ..., T} : C_S[i] ← 0; // All sync clocks set to 0
2  executeThread(t)
3  begin
4      initThread(t); // Same as initial run algorithm
5      while (t has not terminated and isValid(L_t[α]) ) do
6          // Thread t is valid
7          await (isEnabled(L_t[α]) or ! isValid(L_t[α]) );
8          if (isEnabled(L_t[α]) then
9              resolveValid(L_t[α]);
10             C_t[t] ← α; // Update thread clock
11             α ← α + 1; // Increment thunk counter
12     // The thread has terminated or a thunk has been invalidated
13     L'_t ← L_t; // Make a temp copy for missing writes
14     while (t has not terminated or α < |L'_t|) do
15         // Thread t is invalid
16         if (α < |L'_t|) then
17             M ← M ∪ L'_t[α].W; // Add missing writes
18             C_t[t] ← α; // Update thread clock
19         if (t has not terminated) then
20             resolveInvalid(L_t[α]);
21         α ← α + 1; // Increment thunk counter
22     // The thread has terminated
```

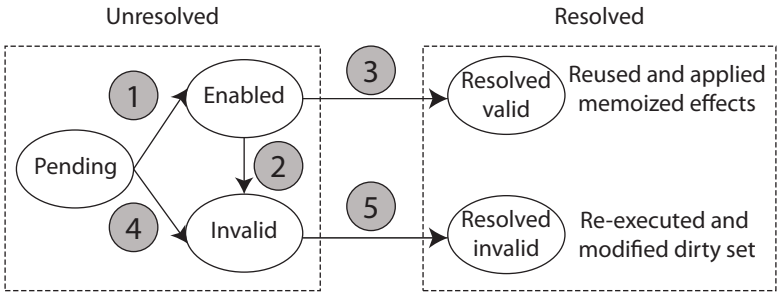


Figure 5.4: State transition for thunks during incremental run

Details. Algorithm 5 presents the overview of the incremental run algorithm, and details of subroutines used in the algorithm are presented in Algorithm 6. The incremental run algorithm allows all threads to proceed in parallel, and associates a state with each thunk of every thread. The state of each thunk follows a state machine (shown in Figure 5.4) that enforces that each thread waits until all thunks that happened-before its next thunk to be executed are resolved (i.e., either recomputed or reused), and only when it is certain that reusing memoized results is not possible will it start to re-execute its next thunk. In particular, the state of a thunk

Algorithm 6: Subroutines for the incremental run algorithm

```
1  isEnabled(L_t[α])
2  begin
3      if (∀i ∈ {1, ..., T} \ {t} : (C_i[i] > L_t[α].C[i])) then
4          // All thunks happened-before are resolved
5          return (isValid(L_t[α])); // check if it's valid
6      return false;
7  isValid(L_t[α])
8  begin
9      if ((L_t[α].R ∩ M) = ∅) then
10         return true; // Read set does not intersects with dirty set
11     return false;
12 resolveInvalid(L_t[α])
13 begin
14     startThunk(); // Same as initial run algorithm
15     repeat
16         Execute instruction of t;
17         if (instruction is load or store) then
18             onMemoryAccess(); // Same as initial run algorithm
19     until t invokes synchronization primitive;
20     M ← M ∪ L_t[α].W; // Add the new writes
21     endThunk(); // Same as initial run algorithm
22     onSynchronization(S); // Same as initial run algorithm
23 resolveValid(L_t[α])
24 begin
25     address space ← memo(L_t[α].W); // Globals and heap
26     stack ← memo(L_t[α].Stack);
27     CPU registers ← memo(L_t[α].Reg); // Also adjusts PC
28     onSynchronization(S); // Same as initial run algorithm
```

is either resolved or unresolved. The state of a thunk is resolved when the thunk has either been reused (`resolved-valid`) or re-executed (`resolved-invalid`). Otherwise, the thunk is still unresolved. An unresolved thunk is in one of the following states: `pending`, `enabled` or `invalid`.

Initially, the state of all thunks is `pending`, except for the initial thunk, which is already `enabled`. A pending thunk is not "ready" to be looked at yet. A `pending` thunk of a thread is `enabled` (state transition ①) when all thunks (of any thread) that happened-before are resolved (either `resolved-valid` or `resolved-invalid`). To check for this condition (using routine `isEnabled()`), we make use of the strong clock consistency condition [112] (if $C(a) < C(b)$ then $a \rightarrow b$) provided by vector clocks to detect causality. In particular, we compare the recorded clock

value of the thunk against the current clock value of all threads to check that all threads have passed the time recorded in the thunk's clock.

An `enabled` thunk transitions to `invalid` (state transition ②) if the read set of the thunk intersects with the dirty set. Otherwise, the `enabled` thunk transitions to `resolved-valid` (state transition ③), where we skip the execution of the thunk and directly apply the memoized write-set to the address space including performing the synchronization operation (using the `resolveValid()` routine).

A `pending` thunk transitions to `invalid` (state transition ④) if any earlier thunk of the same thread is `invalid` or `resolved-invalid`. The `invalid` thunk transitions to `resolved-invalid` (state transition ⑤) when the thread re-executes the thunk and adds the write set to the dirty set (including any missing writes). The executing thread continues to resolve all the remaining `invalid` thunks to `resolved-invalid` until the thread terminates. To do so, we re-initialize the read/write sets of the new thunk to the empty set and start the re-execution similar to the initial run algorithm (using the `resolveInvalid()` routine). While re-executing, the thread updates the CDDG, and also records the state of the newly formed thunks for the next incremental run.

5.6 Implementation

We implemented iThreads as a 32-bit dynamically linkable shared library for the GNU/Linux OS (Figure 5.5). iThreads reuses two mechanisms of the `Dthreads` implementation [109]: the memory subsystem (§5.6.1) and a custom memory allocator (§5.6.4). Additionally, our implementation also includes the iThreads memoizer, which is a stand-alone application. We next describe the iThreads implementation in detail.

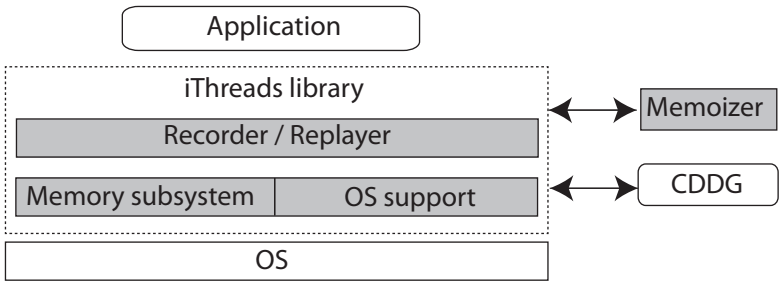


Figure 5.5: iThreads architecture (components are in grey)

5.6.1 iThreads Library: Memory Subsystem

The iThreads memory subsystem implements the RC memory model and derives per-thunk read/write sets.

Release consistency memory model. To implement the RC memory model, iThreads converts threads into separate processes using a previously proposed mechanism [44]. This "thread-as-a-process" approach provides each thread with its own private address space, and thus allows iThreads to restrict inter-thread communication. In practice, iThreads forks a new process on `pthread_create()` and includes a *shared memory commit* mechanism [97, 58] that enables communication between processes at the synchronization points, as required by the RC memory model.

At a high level, throughout the application execution, iThreads maintains a copy of the address space contents in a (shared) reference buffer, and it is through this buffer, with instrumentation provided by iThreads at the synchronization points, that the processes transparently communicate (Figure 5.6). Communication between processes is implemented by determining the thunk write-set, as explained next, which is then used to calculate a byte-level delta [109].

To compute the byte-level delta for each dirty page (which are located by deriving the process write set, as explained later), Slider performs a byte-level comparison between the dirty page and the corresponding page in the reference buffer, and then applies atomically the deltas to the reference buffer. In case there are over-

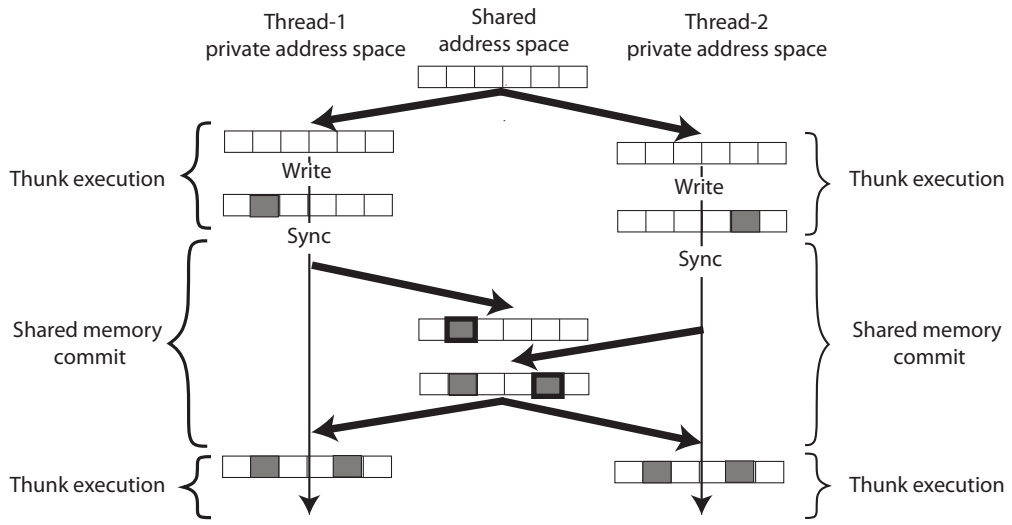


Figure 5.6: Overview of the recorder

lapping writes to the same memory location, made by different processes, Slider resolves the conflict by using a last-writer wins policy.

Furthermore, for efficiency reasons, the implementation of the communication mechanism relies on private memory mapped files – this allows different processes to share physical pages until processes actually write to the pages, and still keeps performance overheads low by virtue of the OS copy-on-write mechanism.

Read and write set. Besides serving as the foundation for the RC memory model, the adopted thread-as-a-process mechanism is also essential for tracking memory references: by splitting the original multi-threaded process into multiple single-threaded processes, iThreads can easily derive *per-thread* read and write sets.

Since each thread is implemented as a separate process, iThreads uses the OS memory protection mechanism to track the read and write sets. In particular, iThreads renders the address space inaccessible by invoking `mprotect(PROT_NONE)` at the beginning of each thunk, which ensures that a signal is triggered the first time a page is read or written by the thunk. Hence, within the respective signal handler, iThreads is able to record the locations of the accesses made to memory at the granularity of pages. Immediately after recording a memory access, the

iThreads library proceeds to reset the page protection bits, allowing the thunk to resume the read/write operation as soon as the signal handler returns. In addition, resetting the memory permissions also ensures that subsequent accesses proceed without further page faults. In this way, iThreads incurs at most two page faults (one for reads and one for writes) for each accessed page.

5.6.2 iThreads Library: Recorder and Replayer

The iThreads library executes the application in either *recording* or *replaying* mode. We next describe the two sub-components, recorder and replayer, that realize these modes of execution by implementing the algorithms described in §5.5.

Recorder. Since iThreads reuses the `Dthreads` memory subsystem, which serializes memory commit operations from different threads, the implementation of the recording algorithm is greatly simplified. Due to the resulting implicit serialization of thunk boundaries, the employed thread, thunk, and synchronization vector clocks effectively reduce to scalar sequence numbers, which allows the recorder to simply encode the thread schedule using thunk sequence numbers.

The recorder is further responsible for memoizing the state of the process at the end of each thunk. To this end, using an assembly routine, iThreads stores the register values in the stack, takes a snapshot of the dirty pages in the address space, and stores the snapshot in the memoizer (§5.6.4). In addition, the recorder also stores the CDDG, consisting of thunk identifiers (thread number and thunk sequence number) and their corresponding read/write sets, to an external file.

Replayer. Similarly to the recorder, the replayer relies on thunk sequence numbers to enforce the recorded schedule order. The replayer first reads the file with the input changes and the CDDG to initialize the replay algorithm. During an incremental run, whenever memoized thunks can be reused, the replayer retrieves

the appropriate state from the memoizer, patches the address space and restores the state of registers.

5.6.3 iThreads Library: OS Support

As practical applications depend on OS services, there are two important aspects related to the OS that iThreads needs to address. First, system calls are used by the application to communicate with the rest of the system, so the effects of system calls (on the rest of the system and on the application itself) need to be addressed; in particular, input changes made by the user need to be handled. Second, there are OS mechanisms that can unnecessarily change the memory layout of the application across runs, preventing the reuse of memoized thunks.

System calls and input changes. Since iThreads is a user-space library running on top of an unmodified Linux kernel, it has no access to kernel data structures. The effects of system calls cannot thus be memoized or replayed. To support system calls, iThreads instead considers system calls to be thunk delimiters (in addition to synchronization calls). Hence, immediately before a system call takes place, iThreads memoizes the thunk state, and immediately after the system call returns, iThreads determines whether it still can reuse the subsequent thunks according to the replayer algorithm.

To guarantee that system calls *take effect* (externally and internally), iThreads invokes system calls in all executions, even during replay runs. To guarantee that effects of system calls on the application (i.e., the return values and writes made to the process address space) are *accounted for* by the thunk invalidation rules, iThreads infers the write-set of the system calls and checks whether the return values match previous runs by leveraging knowledge of their semantics (e.g., some system call parameters represent pointers where data is written).

Importantly, to infer the write-set of system calls that may read large volumes of data (e.g., `mmap`), iThreads allows the user to specify input changes (which potentially modify the write-set of these system calls) explicitly using an external file that lists byte offset ranges as specified by the file `changes.txt` in Figure 5.1.

In practice, our implementation intercepts system calls through wrappers at the level of `glibc` library calls.

Memory layout stability. To avoid causing unnecessary data dependencies between threads, iThreads reuses the custom memory allocator of `Dthreads`, which is based on HeapLayer [45]. The allocator isolates allocation and deallocation requests on a per-thread basis by dividing the application heap into a fixed number of per-thread sub-heaps. This ensures that the sequence of allocations in one thread does not impact the layout of allocations in another thread, which otherwise might trigger unnecessary re-computations.

In addition, iThreads ensures that the Address Space Layout Randomization (ASLR) [13], an OS feature that deliberately randomizes the application memory layout, is disabled.

5.6.4 iThreads Memoizer

The memoizer is responsible for storing the end state of each thunk so that its effects can be replayed in subsequent incremental runs. The memoizer is implemented as a separate program that stores the memoized state in a shared memory segment, which serves as the substrate to implement a key-value storage that is accessible by the recorder/replayer. The memoized state is stored in-memory for fast access, and asynchronously replicated to disk for persistence across reboots.

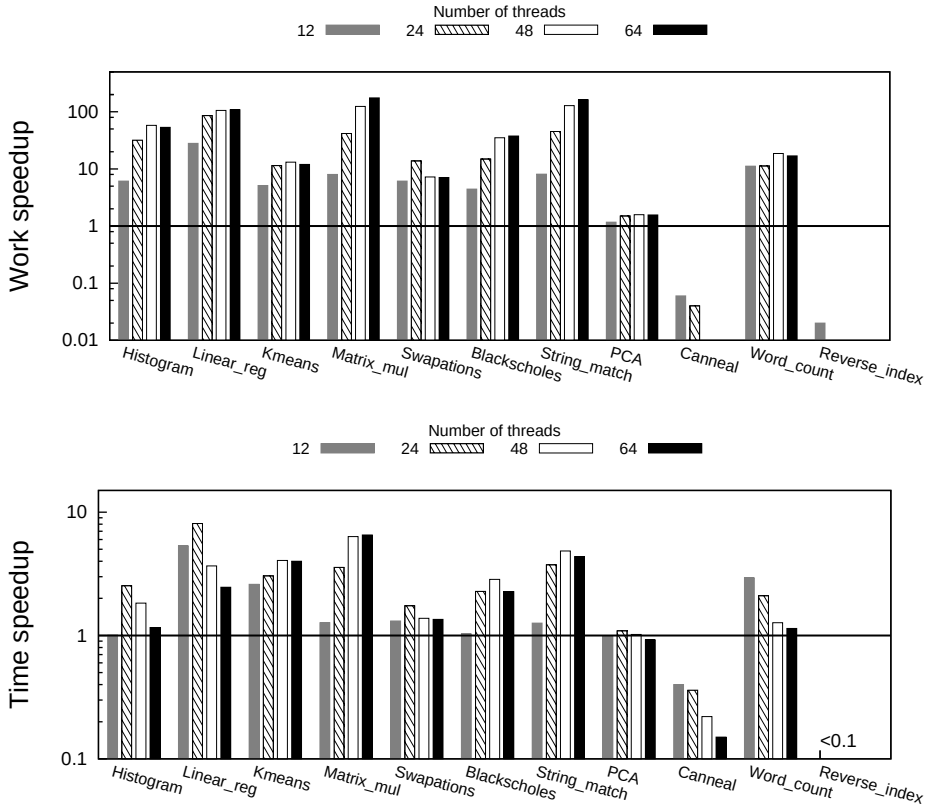


Figure 5.7: Performance gains of iThreads with respect to pthreads for the incremental run

5.7 Evaluation

Our evaluation answers the following three main questions:

- What performance gains does iThreads provide for the incremental run? (§ 5.7.1)
- How do these gains scale with increases in the size of the input, the computation (work), and the input change? (§ 5.7.2)
- What overheads does iThreads impose for memoization and performance for the initial run? (§ 5.7.3)

Experimental setup. We evaluated iThreads on a six-core Intel(R) Xeon(R) CPU X5650 platform with 12 hardware threads running at 2.67 GHz and 32 GB of main memory.

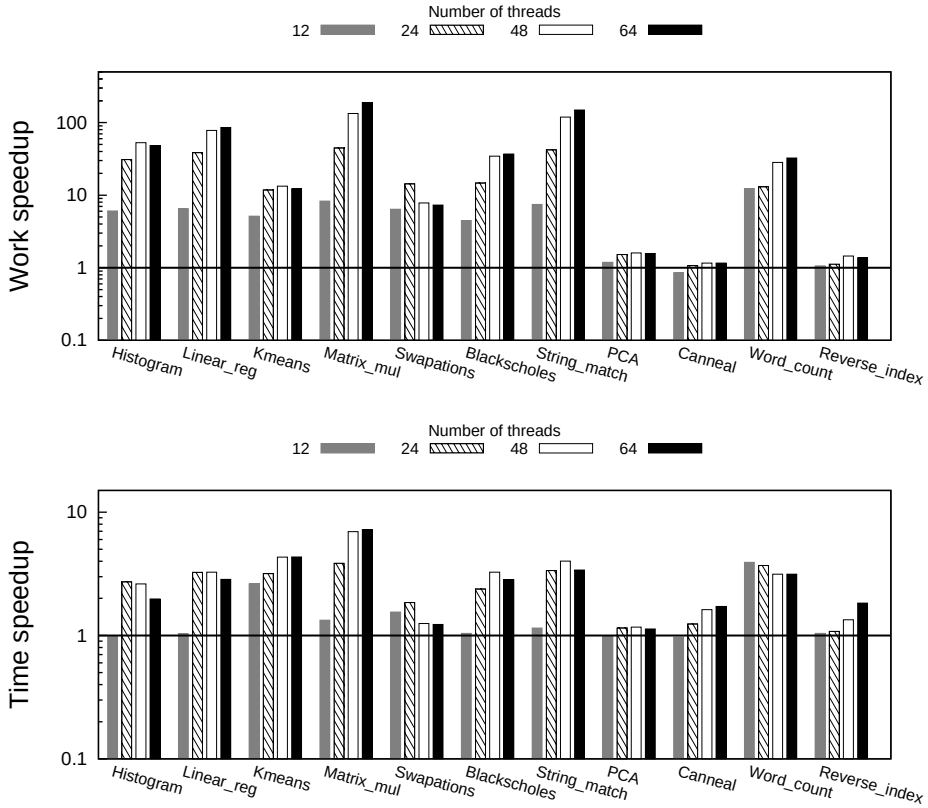


Figure 5.8: Performance gains of iThreads with respect to Dthreads for the incremental run

Applications and datasets. We evaluated iThreads with applications from two benchmark suites: PARSEC [54] and Phoenix [131]. Table 5.1 lists the applications evaluated and their respective used input sizes in terms of 4KB pages. In addition, we also report the gains with two case studies (§5.8).

Metrics: work and time. We consider two types of measures, *work* and *time*. Work refers to the total amount of computation performed by all threads and is measured as the sum of the total runtime of all threads. Time refers to the end-to-end runtime that it takes to complete the parallel computation. Time savings reflect reduced end user perceived latency, whereas work savings reflect improved overall resource utilization.

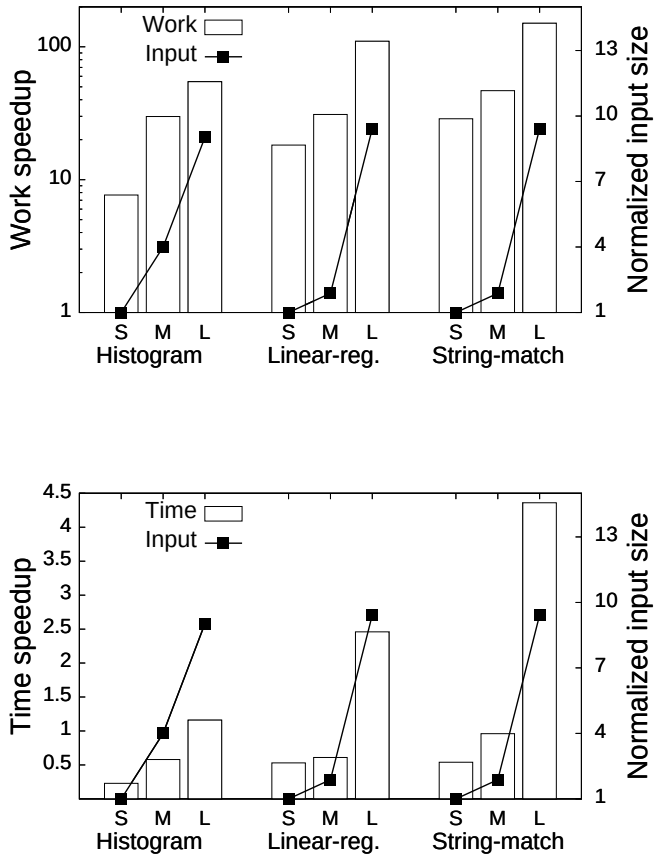


Figure 5.9: Scalability with data (work and time speedups)

Measurements. For all measurements, each application was executed 12 times. We exclude the lowest and highest measurements, and report the average over the 10 remaining runs.

5.7.1 Performance Gains

We first present a comparison of iThreads 's incremental run with pthreads and `Dthreads`, as shown in Figure 5.7 and Figure 5.8 respectively. In this experiment, we modified one randomly chosen page of the input file prior to the incremental run. We then measured the work and time required by iThreads 's incremental run, as well as by pthreads and `Dthreads`, which re-compute everything from scratch.

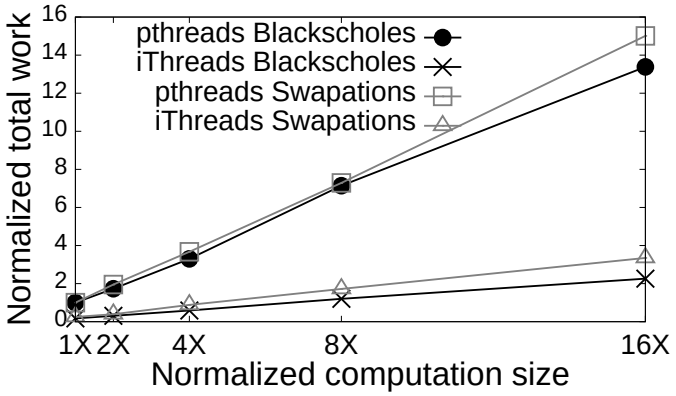


Figure 5.10: Scalability with work

We report the work and time speedups (i.e., iThreads 's performance normalized by the performance of pthreads/`Dthreads`) for a varying number of threads ranging from 12 to 64 threads. When comparing the performance, we use the same number of threads in iThreads and pthreads/`Dthreads`.

The experiment shows that the benefits of using iThreads vary significantly across applications. In over half of the evaluated benchmarks (7 out of 11), iThreads was able to achieve at least 2*X* time speedups. In contrast, for applications such as `canneal` and `reverseindex`, iThreads can be very inefficient, by a factor of more than 15*X*, an effect that we explain in further detail in §5.7.3. Overall, the results show that iThreads is effective across a wide range of benchmark tasks, but also that the library is not a one-size-fits-all solution.

As expected, we observed that increasing the number of threads tended to yield higher speedups. This is because, for a fixed input size, a larger number of threads translates to less work per thread. As a result, iThreads is forced to recompute fewer thunks when a single input page is modified.

Note that work speedups do not directly translate into time speedups. This is because even if just a single thread is affected by input changes, the end-to-end runtime is still dominated by the (slowest) invalidated thread's execution time.

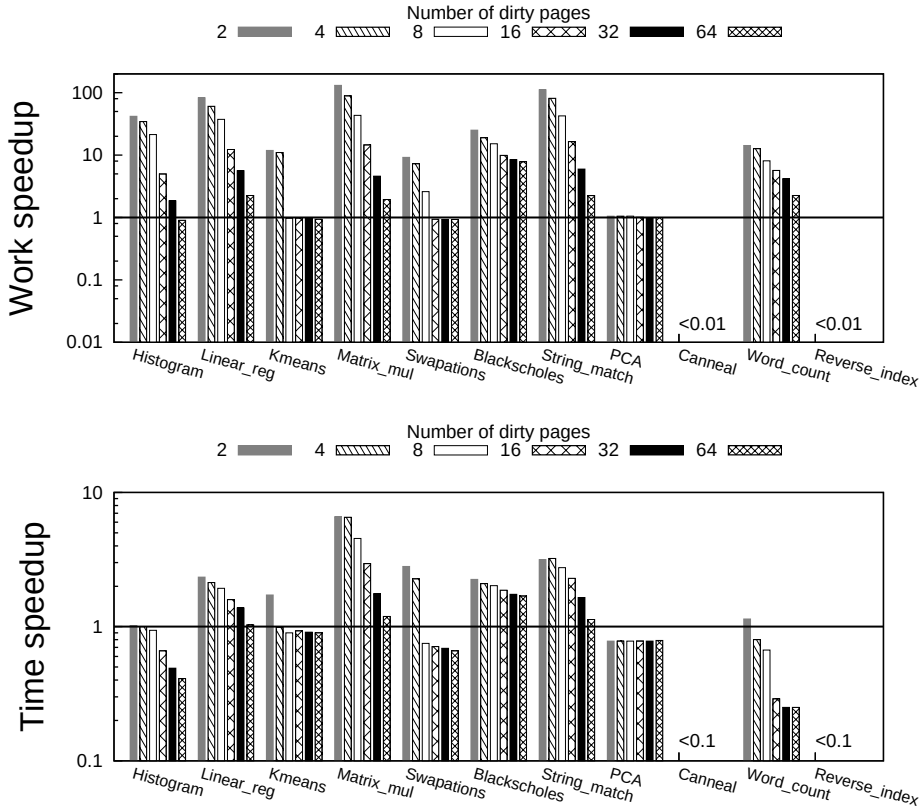


Figure 5.11: Scalability with input change compared to pthreads for 64 threads

5.7.2 iThreads Scalability

In a second experiment, we investigated the scalability of iThreads w.r.t. increases in the size of the input, the amount of computation, and the size of the input change.

Input size. We first present the performance of iThreads as we increase the input data size for the three application benchmarks (`histogram`, `linear regression`, and `string match`) that are available in three input sizes: small (*S*), medium (*M*), and large (*L*). (We used the large size in §5.7.1.) Figure 5.9 shows a bar plot of the work and time speedups w.r.t. pthreads for different input sizes (S, M, L) with a single modified page for 64 threads. For reference, the normalized input size is also

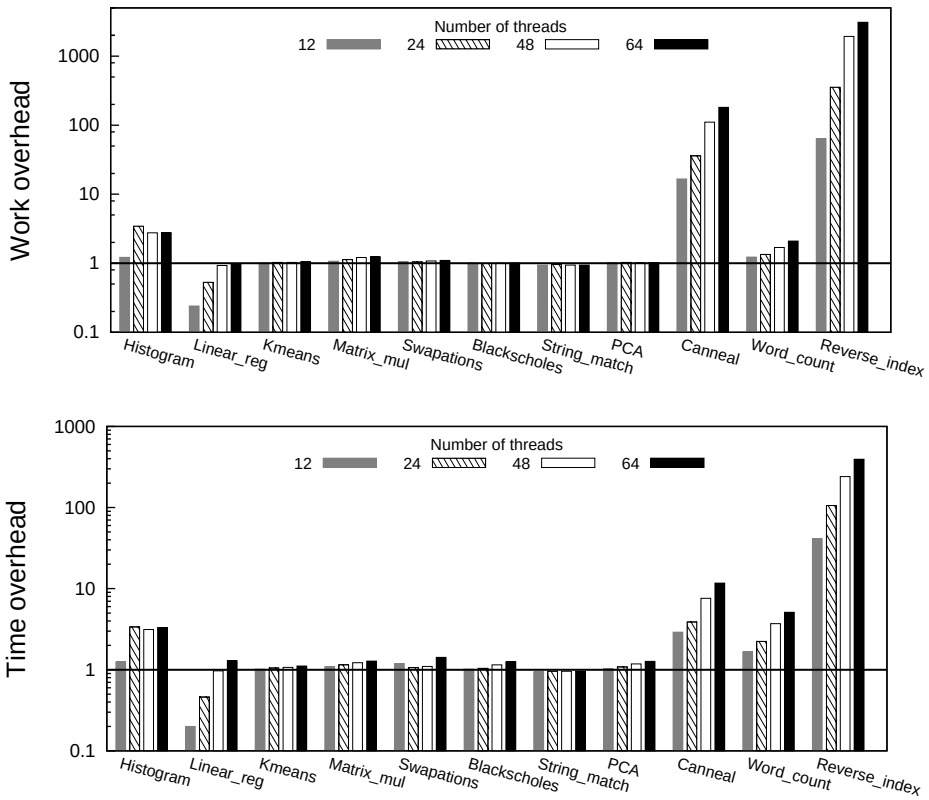


Figure 5.12: Performance overheads of iThreads with respect to pthreads for the initial run

shown by the line plot in the same figure. In summary, this result shows that the speedups increase as expected with the input size due to increased work savings.

Computation (work). We next present iThreads's incremental run performance for two applications (`swapations` and `blackscholes`) that allow the amount of work required to be tuned with a parameter. Figure 5.10 reports the normalized total work as the required work is increased (from $1X$ to $16X$) for a single modified page and 64 threads. The result shows the gap between pthreads and iThreads widens as the total work increases, which directly translates to higher speedups.

Input change. Finally, we present iThreads's incremental run performance in the case of multiple modified input pages. To avoid localization of changes to a single thread, we modified multiple non-contiguous pages of the input that are read by

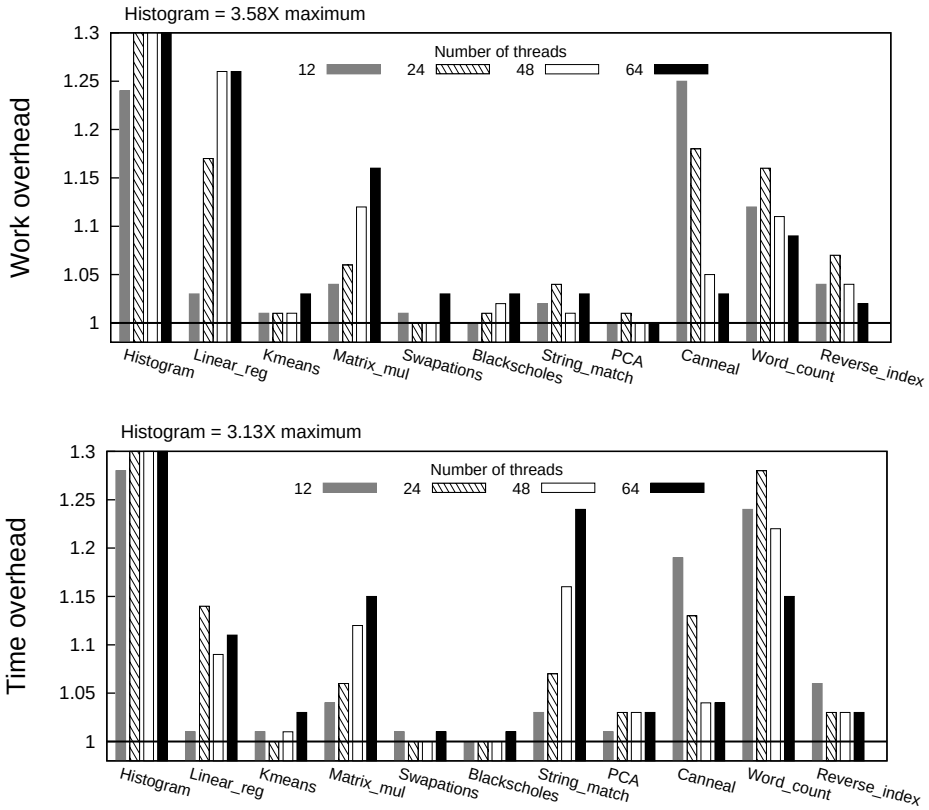


Figure 5.13: Performance overheads of iThreads with respect to Dthreads for the initial run

different threads. Figure 5.11 shows speedups w.r.t pthreads with different change sizes (ranging for 2 to 64 dirty pages) when the application is running with 64 threads. As expected, the result confirms that speedups decrease as larger portions of the input are changed because more threads are invalidated.

5.7.3 Overheads

iThreads imposes two types of overheads: (1) space overheads; and (2) performance overheads during the initial run.

Space overheads. Table 5.1 shows the space overheads for memoizing the end state of the thunks and storing the CDDG. We report the overheads in terms of 4KB pages for 64 threads (space overhead grows with the number of threads). To

Application	Input size	Memoized state		CDDG	
Histogram	230400	347	(0.15%)	57	(0.02%)
Linear-reg.	132436	192	(0.14%)	33	(0.02%)
Kmeans	586	1145	(195.39%)	27	(4.61%)
Matrix-mul.	41609	4162	(10.00%)	64	(0.15%)
Swapations	143	1473	(1030.07%)	1	(0.70%)
Blackscholes	155	201	(129.68%)	1	(0.65%)
String match	132436	128	(0.10%)	33	(0.02%)
PCA	140625	3777	(2.69%)	43	(0.03%)
Canneal	9	15381	(170900.00%)	4	(44.44%)
Word count	12811	10191	(79.55%)	24	(0.19%)
Rev-index	359	260679	(72612.53%)	64	(17.83%)

Table 5.1: Space overheads in 4*KB* pages and input percentage

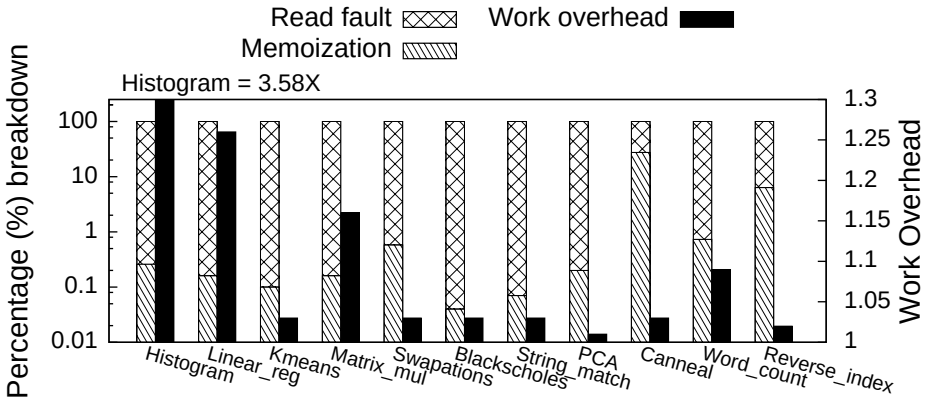


Figure 5.14: Work overheads breakdown w.r.t. Dthreads

put the overheads into perspective, we also report overheads as a percentage of the input size. The space overheads varied significantly across applications. We found that three applications (`canneal`, `swapations` and `reverse-index`) incur in very high overheads (exceeding 1000% of the input size), but, interestingly, nearly half of the applications (5/11) have a very low overhead (ranging from 0.1% to 10% of the input size).

Performance overheads. We measured iThreads 's performance overheads during the initial run (in terms of work and time) by comparing it against both pthreads and `Dthreads` (Figures 5.12 and 5.13). Our results show that most of the applications (7/11) incur modest overheads when compared with either pthreads (i.e.,

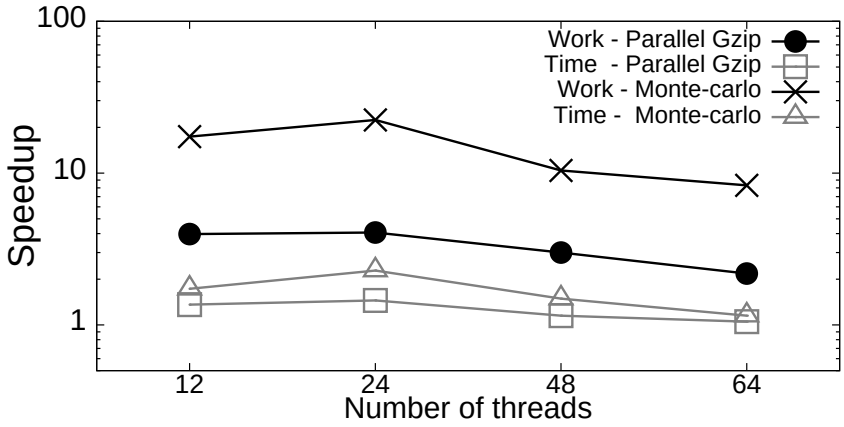


Figure 5.15: Work & time speedups for case-studies

lower than 50%) or `Dthreads` (i.e., lower than 25%). In fact, `linear-reg` and `string-match` even performed better during the initial run of iThreads than with pthreads, which is explained by the fact that private address space mechanism avoid false sharing, as previously noted by Sheriff [108]. At the other end of the spectrum, applications such as `canneal` and `reverse-index` incur in high overheads mainly due to the high number of memory pages written by these applications (as shown in Table 5.1), because each dirty page incurs a write page fault.

When compared to `Dthreads` as the baseline, iThreads incurs work overheads of up to 3.58*X* and time overheads of up to 3.13*X*. The iThreads implementation incurs the additional overheads on top of `Dthreads` mainly from two sources: memoization of intermediate state and read page faults to derive the read set (`Dthreads` incurs write faults only). We show the work overheads along with a breakdown of these two sources of overheads with respect to `Dthreads` for 64 threads in Figure 5.14. The overheads are dominated by read page faults (around 98%) for most applications. For instance, `histogram` incurs overheads of roughly 3.5X due the large number of page faults while reading a large input file (as shown in Table 5.1). In contrast, some application such as `canneal` and `reverse-index`

suffer a significant overhead for memoization (around 24%) due to a large number of dirtied pages, which lead to high number of write page faults.

5.8 Case Studies

In addition to the benchmark applications, we report the performance gains for two case-study applications: (1) Pigz [14], a parallel gzip compression library compressing a 50MB file, and (2) a scientific monte-carlo simulation [10]. To compute speedups, we modified a random input block and compared the performance of the iThreads incremental run with the pthreads run. Figure 5.15 shows the work and time speedups with a varying number of threads (from 12 to 64). The performance gains peak at 24 threads for both applications, while increasing the number of threads beyond 24 lead to diminishing gains. In particular, iThreads achieves a time speedup of 1.45X and a work speedup of 4X for gzip compression, and a time speedup of 2.28X and a work speedup of 22.5X for the monte-carlo simulation with 24 threads.

To conclude, while there exist specific workloads for which our OS-based approach is not suitable due to inherent memoization and read/write set tracking costs, our evaluation is overall positive: iThreads is able to achieve significant time and work speedups both for many of the benchmark applications and also for the two considered case-studies.

5.9 Related Work

Incremental computation is a well-studied area in the programming languages community; see [130] for a classic survey. Earlier work on incremental computation was primarily based on dependence graphs [69, 93] and memoization [23, 91, 125]. In the past decade, with the development of self-adjusting computa-

tion [24, 25, 27, 59, 87, 88, 103, 104], the efficiency of incremental computation has much improved. In contrast to iThreads, however, most prior work in this area targets sequential programs only. Nonetheless, iThreads's central data structure, the CDDG, is inspired and informed by these foundations.

For supporting parallel incremental computation, the existing proposals [57, 86] require a strict fork-join programming model without any support for synchronization primitives. Furthermore, these proposals rely on the use of a new programming language with special data types (e.g., *isolation, versioned, cumulative, merge function* [57] and *read, write, mod, and letpar* [86]). In contrast, our approach targets unmodified multithreaded programs supporting the full range of synchronization primitives in the POSIX API.

In very recent work, Tseng and Tullsen [141] proposed compiler-based whole-program transformations to eliminate redundant computation, which can be leveraged for faster incremental computation. The transformed programs rely on underlying hardware [139] and software [140] support to dynamically identify redundant code that can be skipped. In contrast, our approach directly operates at the binary level without requiring access to source code. A further design difference is that iThreads realizes incremental computation based on explicit change propagation, and that iThreads memoizes and reuses intermediate results of previous runs.

In the context of increased reliability, prior work has yielded a large range of hardware and software solutions to eliminate *non-determinism* from multithreaded programs. Most relevant to iThreads are the wide range record and replay techniques (e.g., [34, 77, 92, 100, 102, 120, 133, 143, 144, 145]) and deterministic multithreading approaches (e.g., [39, 41, 42, 44, 63, 64, 65, 72, 73, 94, 109, 117]). As described throughout the paper, these proven techniques are leveraged by iThreads,

which applies them in a novel context, namely *transparent* parallel incremental computation.

5.10 Limitations and Future Work

While iThreads is a significant step towards general and practical support for parallel incremental computation, plenty of opportunities remain to further increase the range of supported workloads.

For one, iThreads's memory model currently lacks support for ad-hoc synchronization mechanisms [147]. While such mechanisms have been shown to be error-prone in practice, introducing bugs or performance issues [147], they are nonetheless used for either flexibility or performance reasons in some applications. In many such cases, it may be possible to replace these mechanisms with equivalent pthreads API calls; for instance, user-defined spin locks can be replaced by pthreads spin lock API calls [17]. An interesting direction for future work would be to extend iThreads with an interface for specifying ad-hoc synchronization primitives (e.g., at the level of `gcc`'s built-in atomic primitives) to identify thunk boundaries, thereby paving the way for transparent support for lock-free and wait-free data structures in iThreads.

Another interesting research challenge is improving support for small, localized insertions and deletions in the input data. Whereas iThreads is currently tuned for in-place modifications of the input data, insertions and deletions lead to the displacement of otherwise unchanged data, which causes an excessively large dirty-set. Prior work has solved the displacement problem in the context of data-deduplication by replacing fixed-size input chunking with variable-size, content-based chunking [114]. We plan to explore similar approaches in the context of iThreads.

Lastly, our current implementation assumes the number of threads in the system remains the same. However, our approach can be extended to handle dynamically varying number of threads by considering newly forked threads or deleted threads as invalidated threads, where the writes of deleted threads are handled as "missing writes". The happens-before relationship for dynamically varying number of threads can be detected using interval tree clocks [33].

5.11 Summary

In this chapter, we have explored a new dimension for supporting parallel incremental computations. Our approach targets shared-memory multi-threaded programs supporting the full range of synchronization primitives in the POSIX API. Our ambitious goals to be transparent, practical, and efficient pushed us to investigate a new set of challenges. In the process of the investigation, we have made a set of assumptions and design choices. The end result of our efforts is iThreads, a straightforward solution to use: it simply replaces the pthreads library, allowing existing C/C++ applications to run in incremental fashion by a simple exchange of libraries linked, without even recompiling the code.

CHAPTER 6

Conclusions

How should we design systems to support incremental computation for parallel and distributed computing? This dissertation shows that, in many cases, a simple abstraction of self-adjusting computation can enable *practical*, *automatic*, and *efficient* incremental computation in real-world parallel and distributed computing. Our approach neither requires departure from current models of programming, nor the invention and implementation of application-specific dynamic algorithms for incremental computation.

To illustrate our approach, this dissertation presents the design and implementation of the following four systems for incremental parallel and distributed computation: *(i)* Incoop — a system for incremental MapReduce computation; *(ii)* Shredder — a GPU-accelerated system for incremental storage; *(iii)* Slider — a batched stream processing platform for incremental sliding window computation; and *(iv)* iThreads — a threading library to support parallel incremental computation for unmodified C/C++ `pthread`-based multithreaded programs. Our experience with these systems shows that our techniques can yield very good performance, both in theory and practice, without requiring programmers to write any special-purpose algorithms for incremental computation.

While cluster infrastructure will continue to evolve, we hope that our design choices will provide a useful reference point for system designers of parallel and distributed computing frameworks.

BIBLIOGRAPHY

[1] Apache hama (http://hama.apache.org/).

[2] Apache mahout: http://mahout.apache.org/.

[3] Apache PigMix. http://wiki.apache.org/pig/PigMix.

[4] Apache s4. http://incubator.apache.org/s4/.

[5] Calculating Memory System Power for DDR3. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3 %20Power.pdf.

[6] General Purpose computation on GPUs. http://www.gpgpu.org.

[7] GPUDirect. http://developer.nvidia.com/gpudirect.

[8] Hadoop. http://hadoop.apache.org/.

[9] Memcached. http://memcached.org/.

[10] Monte-carlo method. http://cdac.in/index.aspx?id =ev_hpc_pthread_benchmarks_kernels.

[11] NVidia CUDA. http://developer.nvidia.com/cuda-downloads.

[12] NVidia CUDA Tutorial. http://people.maths.ox.ac.uk/g̃ilesm/-hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf.

[13] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[14] Pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines. http://zlib.net/pigz/.

[15] POSIX threads. http://gcc.gnu.org/.

[16] Pthreads memory model. http://pubs.opengroup.org /onlinepubs/9699919799/basedefs/v1_chap04.html.

[17] Pthreads spin lock. http://pubs.opengroup.org /onlinepubs/009695399/functions/pthread_spin_lock.html.

[18] Storm. http://storm.apache.org/.

[19] The data deluge. http://www.economist.com/node/15579717, Feb.

[20] Trident. http://storm.apache.org/documentation/trident-api-overview.html.

[21] Using the RDTSC Instruction for Performance Monitoring - Intel Developers Application Notes . http://www.ccsl.carleton.ca/ jamuir/rdtscpm1.pdf.

[22] Wikipedia datasets. http://wiki.dbpedia.org/.

[23] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1996.

[24] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[25] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[26] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.

[27] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *proceedings of the Twenty-sixth Annual Symposium on Computational Geometry (SoCG)*, 2010.

[28] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable Client Accounting for P2P-Infrastructure Hybrids. In *proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[29] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communication of ACM (CACM)*, 2010.

[30] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: an end-system redundancy elimination service for enterprises. In *proceedings of the 7th USENIX conference on networked systems design and implementation (NSDI)*, 2010.

[31] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *proceedings of the 17th international symposium on High performance distributed computing (HPDC)*, 2008.

[32] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP Server and Online Behavioral Targeting. In *proceedings of VLDB Endowment (VLDB)*, 2009.

[33] P. S. Almeida, C. Baquero, and V. Fonte. Interval tree clocks. In *proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, 2008.

[34] G. Altekar and I. Stoica. ODR: Output-deterministic Replay for Multicore Debugging. In *proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[35] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *proceedings of the ACM SIGCOMM 2008 conference on Data communication (SIGCOMM)*, 2008.

[36] A. Anand, V. Sekar, and A. Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *proceedings of the ACM SIGCOMM 2009 conference on Data communication (SIGCOMM)*, 2009.

[37] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[38] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In *proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[39] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-enforced Deterministic Parallelism. In *proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[40] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2005.

[41] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, 2010.

[42] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[43] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[44] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*, 2009.

[45] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, 2001.

[46] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *proceedings*

of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2009.

[47] P. Bhatotia, U. A. Acar, F. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *proceedings of the 15th Annual ACM/IFIP/USENIX Middleware conference (Middleware)*, 2014.

[48] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. In *Technical Report: MPI-SWS-2012-004*, 2012.

[49] P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[50] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[51] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[52] P. Bhatotia, A. Wieder, R. Rodrigues, and U. A. Acar. Incremental MapReduce Computations. In *proceedings of advances in data processing techniques in the era of Big Data. CRC Press*, 2014.

[53] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.

[54] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, 2008.

[55] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[56] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *proceedings of VLDB Endowment (VLDB)*, 2010.

[57] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, 2011.

[58] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[59] Y. Chen, J. Dunfield, and U. A. Acar. Type-Directed Automatic Incrementalization. In *proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

[60] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *proceedings of the IEEE*, 1992.

[61] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *proceedings of the 2009 conference on USENIX Annual technical conference (USENIX ATC)*, 2009.

[62] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[63] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[64] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[65] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[66] M. L. Curry, H. L. Ward, A. Skjellum, and R. Brightwell. A Lightweight, GPU-Based Software RAID System. In *proceedings of the 39th International Conference on Parallel Processing (ICPP)*, 2010.

[67] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[68] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIX ATC)*, 2010.

[69] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *proceedings of the 8th*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1981.

[70] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 35: Dynamic Trees. Dinesh Mehta and Sartaj Sahni (eds.), CRC Press Series, in Computer and Information Science, 2005.

[71] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. Dinesh Mehta and Sartaj Sahni (eds.), CRC Press Series, in Computer and Information Science, 2005.

[72] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2009.

[73] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. *proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[74] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[75] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *proceedings of the 9th USENIX conference on File and Storage technologies (FAST)*, 2011.

[76] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable

Secondary Storage. In *Proccedings of the 7th conference on File and storage technologies*, 2009.

[77] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. *proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002.

[78] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[79] K. Eshghi and H. K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Technical Report HPL-2005-30R1, HP Technical Report, 2005.

[80] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2009.

[81] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, 1990.

[82] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU accelerated storage system. In *proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[83] L. J. Guibas. Kinetic data structures: a state of the art report. In *proceedings of the third workshop on the algorithmic foundations of robotics (WAFR)*, 1998.

[84] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[85] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC)*, 2011.

[86] M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *proceedings of the 2007 workshop on Declarative aspects of multicore programming (DAMP)*, 2007.

[87] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2009.

[88] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[89] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM)*, 2010.

[90] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *proceedings of the 17th conference on Security symposium (USENIX Security)*, 2008.

[91] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI)*, 2000.

[92] N. Honarmand and J. Torrellas. RelaxReplay: Record and Replay for Relaxed-consistency Multiprocessors. In *proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[93] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, May 1987.

[94] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[95] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[96] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI)*, 2011.

[97] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (USENIX)*, 1994.

[98] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *proceedings of the 8th USENIX conference on File and Storage technologies (FAST)*, 2010.

[99] V. Kulkarni. Delivering on the i/o bandwidth promise: over 10gb/s large sequential bandwidth on ibm x3850/x3950 x5. Technical report, IBM, 2010.

[100] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS)*, 2010.

[101] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 1997.

[102] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, 2010.

[103] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2009.

[104] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP)*, 2008.

[105] M. Lillibridge. Parallel processing of input data to locate landmarks for chunks, 16 August 2011. U.S. Patent No. 8,001,273.

[106] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies (FAST)*, 2009.

[107] M. Lillibridge, K. Eshghi, and G. Perry. Producing chunks from input data using a plurality of processing elements, 12 July 2011. U.S. Patent No. 7,979,491.

[108] T. Liu and E. D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. In *proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[109] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[110] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[111] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC)*, 2011.

[112] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, 1989.

[113] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[114] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[115] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.

[116] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2008.

[117] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2009.

[118] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 1981.

[119] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 2007.

[120] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.

[121] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[122] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: reusing work in large-scale computations. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[123] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2003.

[124] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *proceedings of the 4th USENIX conference on Networked systems design and implementation (NSDI)*, 2007.

[125] W. Pugh. *Incremental Computation via Function Caching*. PhD thesis, Department of Computer Science, Cornell University, Aug. 1988.

[126] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Communications of ACM (CACM)*, 1990.

[127] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. TimeStream: reliable stream computation in the cloud. In *proceedings of the 8th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2013.

[128] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[129] M. O. Rabin. Fingerprinting by random polynomials. Technical report, 1981.

[130] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1993.

[131] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[132] T. Rodrigues, F. Benevenuto, M. Cha, K. Gummadi, and V. Almeida. On Word-of-Mouth Based Discovery of the Web. In *proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, 2011.

[133] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems (TOCS)*, 1999.

[134] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[135] V. Saxena, Y. Sabharwal, and P. Bhatotia. Performance evaluation and optimization of random memory access on multicores with high productivity. In *International Conference on High Performance Computing (HiPC)*. IEEE, 2010.

[136] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2004.

[137] H. Shojania, B. Li, and X. Wang. Nuclei: GPU-accelerated many-core network coding. In *proceedings of IEEE Infocom (INFOCOM)*, 2009.

[138] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[139] H.-W. Tseng and D. M. Tullsen. Data-Triggered Threads: Eliminating Redundant Computation. In *proceedings of 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[140] H.-W. Tseng and D. M. Tullsen. Software Data-triggered Threads. In *proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[141] H.-W. Tseng and D. M. Tullsen. CDTT: Compiler-generated data-triggered threads. In *proceedings of 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[142] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system. In *proceedings of the 8th USENIX conference on File and storage technologies (FAST)*, 2010.

[143] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and Surviving Data Races Using Complementary Schedules. In *proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[144] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *proceedings of the sixteenth international conference on Architectural support for programming languages and operating system (ASPLOS)*, 2011.

[145] N. Viennot, S. Nair, and J. Nieh. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[146] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: a similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC)*, 2011.

[147] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[148] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[149] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[150] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[151] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.