$s2^k$: A Parallel Language for Streaming Text Extraction and Transformations

by

Kenneth S. Herdy

B.Sc. Computing Science, Simon Fraser University, 2006

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Kenneth S. Herdy 2015
SIMON FRASER UNIVERSITY
Summer 2015

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing". Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Kenneth S. Herdy

Degree: Doctor of Philosophy

Title: $s2^k$: A Parallel Language for Streaming Text Extraction and Transformations

Examining Committee: Dr. Kay Weise
Professor
Chair

Dr. Robert D. Cameron
Senior Supervisor
Professor

Dr. Thomas C. Shermer
Senior Supervisor
Professor

Dr. David S. Burggraf
Supervisor
Adjunct Professor

Dr. William N. Sumner
Internal Examiner
Assistant Professor

Dr. José Nelson Amaral
External Examiner
Professor, Computing Science
University of Alberta

Date Defended/Approved: April 14, 2015

Abstract

Computer architectures continue to evolve and expose additional hardware parallelism to software applications. Although programming systems can leverage advances in hardware parallelism for the processing of numerical data, finding ways to exploit additional hardware parallelism for text data is particularly challenging. To address this challenge we define $s2^k$, a global-view parallel programming language for streaming text extraction and transformations that integrates stream programming abstractions and parallel bitstream programming methods.

The $s2^k$ language design involves several aspects. First, the design of domain-specific abstractions that integrate stream programming concepts and parallel bitstream programming methods. Second, the definition of a deterministic parallel programming model with serial semantics. Third, the design of a translation scheme to transform $s2^k$ stream programs into a portable intermediate language suitable for translation to backend representations. Fourth, the definition of $s2^k$ runtime libraries for parallel text processing.

Keywords: parallel programming languages; compilers; streaming text processing; parallel bitstreams; Single Instruction Multiple Data (SIMD);

Contents

Approval ii

Abstract iii

Contents iv

List of Figures viii

1 Introduction 1

1.1 Motivation and Rationale . . . 1

1.2 Basic Assumptions . . . 3

1.3 Evaluation Methods and Past Results . . . 5

1.4 Novelty . . . 7

1.5 Contributions . . . 7

1.6 Organization of this Dissertation . . . 8

2 Background 9

2.1 Trends in Parallel Hardware Design . . . 9

2.2 Text Processing . . . 10

2.3 The Stream Programming Paradigm . . . 11

2.4 The Parallel Bitstream Programming Paradigm . . . 15

2.5 Related Work . . . 18

2.6 Chapter Summary . . . 20

3 The $s2^k$ Programming Language 21

3.1 Overview . . . 21

3.2 Design Principles . . . 21

3.3 Programming Model Structure . . . 22

3.3.1 $s2^k$ Programming Model Versus the SPMD Stream Programming Model. . . . 25

3.4 Programming Model Concepts . . . 26

3.4.1 Parallel Fieldstreams. 26
3.4.2 Fieldstream Operations. 27
3.4.3 Stream Indexing 32
3.4.4 Scalar Promotion 32
3.4.5 Classification of Errors 32
3.5 Language Structure 33
3.5.1 Graph-level Constructs 33
3.5.2 Filter-level Constructs 35
3.6 $s2^k$ Programming Idioms 43
3.6.1 Flow-parallel Programming 43
3.6.2 Unaligned Fieldstream Programming 43
3.6.3 Field-aligned Fieldstream Programming 44
3.6.4 An Illustrative Example: Binary-integer Stream Conversion 44
3.7 8-bit ASCII Stream to 16-bit Binary-integer Stream Conversion 45
3.7.1 Problem Definition. 46
3.7.2 $s2^k$ Implementation. 50
3.8 Discussion 57
3.9 Chapter Summary 58

4 The $b2^k$ Intermediate Language 59
4.1 Overview 59
4.2 Block-at-a-time Processing Model 59
4.2.1 Bitblock Operations. 60
4.2.2 Carry Enumeration 63
4.2.3 Block-at-a-time Management of Carry Values 63
4.3 Intermediate Representation Structure 68
4.3.1 Graph-level Constructs 68
4.3.2 Kernel-level Constructs 71
4.4 Chapter Summary 72

5 Compilation 73
5.1 Overview 73
5.2 $s2^k$ to $b2^k$ Transformations 74
5.2.1 Fieldstream Variable Declaration Statements. 75
5.2.2 Field-parallel Function Call Assignments. 75
5.2.3 Flow-parallel Transformations. 76
5.2.4 Fieldstream-filter Definitions. 79

5.3 $b2^k$ to C++ Transformations . . . 79
5.4 Chapter Summary . . . 81

6 Inverted-index Construction: An $s2^k$ Case Study 82
6.1 Overview . . . 82
6.1.1 Inverted-index Construction . . . 82
6.2 Inverted-index Construction with $s2^k$ . . . 84
6.2.1 Length-partitioning . . . 85
6.2.2 Stream-graph Level Implementation. . . . 90
6.2.3 Filter-level Implementation . . . 93
6.3 Chapter Summary . . . 95

7 Conclusion 97
7.1 Conclusions . . . 97
7.2 Limitations . . . 98
7.3 Future Directions and Work . . . 98

Bibliography 100

Appendix A Glossary of Terms and Notations 106
A.1 General Terms and Notations . . . 106
A.2 $s2^k$ Language Terms and Notations . . . 106

Appendix B $s2^k$ Grammar 109
B.1 EBNF Notations . . . 109
B.2 $s2^k$ Grammar Definition . . . 110

Appendix C $s2^k$ Fieldstream Operations 114
C.1 Flow-parallel Operations . . . 114
C.1.1 Parallel-scanning Operations . . . 114
C.1.2 Spanning Operations . . . 116
C.1.3 LookAhead Operations . . . 117
C.1.4 Masking Operations . . . 117
C.1.5 Positional Operations . . . 118
C.2 Field-parallel Operations. . . . 119
C.2.1 Auxiliary Operations. . . . 119
C.3 Vertical Operations . . . 120
C.3.1 Unary Operations . . . 120
C.3.2 Binary Operations . . . 121

C.3.3 Ternary Operations . . . 125
C.4 Horizontal-packing Operations . . . 125
C.5 Expansion Operations . . . 126
C.6 Field-movement Operations . . . 127
C.7 Casting Operations . . . 128
C.8 Fieldstream-movement Operations . . . 128

Appendix D s2^k Variables and Libraries 130
D.1 s2^k Variables (s2k) . . . 130
D.2 s2^k Libraries . . . 131
D.2.1 Buffer Management Library (io) . . . 131
D.2.2 Standard Library (std) . . . 131
D.2.3 Regular Expression Library (regex) . . . 132
D.3 Flow-parallel Library (flow) . . . 133
D.4 Field-parallel Libraries (field) . . . 133

Appendix E s2^k Case Study Program Listings 134
E.1 8-bit ASCII Stream to 16-bit Binary-integer Conversion . . . 134
E.2 Grep Exact String Search . . . 139
E.3 Inverted-index Construction . . . 143

Appendix F b2^k Grammar 148
F.1 b2^k Grammar Definition . . . 148

Appendix G b2^k Bitblock Operations 152
G.1 Flow-parallel Operations. . . . 152
G.2 Scanning Operations . . . 152
G.3 Spanning Operations . . . 155
G.4 Field-parallel Operations. . . . 155

List of Figures

2.1 A stream graph representation of a stream program. 13
2.2 Procedural language operation expressed on a sequence of data values. 14
2.3 Stream programming language operation expressed on a sequence of data values. 14

3.1 Example $s2^k$ stream-graph. 24
3.2 Flow-parallel operations. 31
3.3 Conditional Fieldstream operations. 38
3.4 Iterative Fieldstream operations. 38
3.5 Unrolled Fieldstream operations of Figure 3.4. 39
3.6 $s2^k$ programming styles to convert an 8-bit ASCII stream data to 16-bit binary integer data. 45
3.7 Sequential 4-bit BCD to 16-bit binary integer conversion. 48
3.8 Parallel 4-bit BCD to 16-bit binary integer conversion. 49

4.1 Straight-line Fieldstream operations. 64
4.2 Equivalent straight-line block-at-a-time operations for the Figure 4.1 Fieldstream operations. 65
4.3 Conditional Fieldstream operations. 66
4.4 Equivalent conditional block-at-a-time operations for the Figure 4.3 Fieldstream operations. 66
4.5 Iterative Fieldstream operations. 67
4.6 Unrolled Fieldstream operations of Figure 4.5. 67
4.7 Iterative block-at-a-time equivalent operations for the Figure 4.6 Fieldstream operations. 68

6.1 The *Keeper* database of single-line documents. 83
6.2 Inverted-index for the *Keeper* database of single-line documents. 84
6.3 Logarithmic length-partitioning of an input text stream. 89
6.4 Stream-graph for length-sorted inverted-index construction. 91

Chapter 1

Introduction

The integration of stream programming concepts with parallel bitstream programming methods in the design of a parallel programming language for streaming text extraction and transformations can simplify programming and maintain application performance. Programmers of parallel bit stream applications are unburdened from many low-level programming details, while compilers can yield applications that achieve the performance of hand-written parallel bitstream applications on existing architectures as well as enhance portability by abstracting over processor features such as Single Instruction Multiple Data (SIMD) register width.

1.1 Motivation and Rationale

Hardware designs continue to evolve and expose additional hardware parallelism to software applications through features such as multicore processors and SIMD vector units [69]. However, finding ways to exploit parallel-hardware features for text processing or other non-numerical computations is particularly challenging [11]. Indeed, a widely cited Berkeley study claims that the Finite State Machine (FSM) algorithms associated with text processing may be the hardest of the thirteen dwarfs[1] to parallelize [6]. This study concludes that, due to inherently sequential processing dependencies, FSM algorithms may be *embarrassingly sequential*, not suitable for parallel implementation, and require fundamental changes in the algorithmic approach.

A new approach to streaming text processing based on the concept of parallel bitstreams has shown considerable promise [14, 17, 68]. However, this approach uses processor information (condition-code flags to indicate when an arithmetic carry or borrow has been generated) that is not directly available to general purpose programming languages such as Python or the C-family languages. To bypass this limitation, initial efforts in programming parallel bitstream operations

[1] The term *dwarf* refers to an algorithmic method that captures a pattern of computation and communication predicted to remain important in the future. [6]

involved the hand-coding of assembly routines. This practice was time-consuming, tedious, and error-prone. A prototype language called Pablo, together with an application framework called Parabix [6], were developed to address these challenges. Pablo is a Domain-Specific Language (DSL) embedded in Python [73] that uses Python unbounded integers to represent arbitrary-length bitstreams, logical and arithmetic operations over unbounded integers to define bitstream operations, and co-opts Python control-flow for bitstream control-flow. The Parabix framework is an application framework for the generation of parallel bitstream text processing applications. This application framework provides: (i) a unifying architectural view of text processing in terms of parallel bitstreams, (ii) a tool chain for automating stages in the generation of parallel bitstream applications, and (iii) a runtime environment that provides portable SIMD programming abstractions [60]. In the Parabix framework, the programmer expresses parallel text processing computations and control flow over arbitrary-length bitstreams in Pablo. A Pablo compiler then translates bitstream programs into equivalent register-at-a-time C++ code fragments. Pablo compiler generated C++ code fragments are then copied into hand-written C++ template files[2], together with runtime environment and post-processing methods, for compilation to executable machine code.

Despite relative advantages over hand-coding methods in simplifying the programming of parallel bitstream applications, the Pablo language lacks many features necessary to support a broader class of applications. For example, the Pablo stream operations are limited to operations defined over sequences of 1-bit fields, or equivalently, in the Pablo language, arbitrary length unsigned integers. However, a range of applications can benefit from the definition of operations on unbounded sequences of power-of-two bit fields. For example, bioinformatics applications process DNA sequences consisting of the four nucleotide codes A, C, G, and T. Internally, these sequences can be represented as packed sequences of 2-bit fields. Here, stream operations defined on 2-bit field streams simplify streaming bioinformatics programming. Similarly, in text processing, Binary Coded Decimal (BCD) to binary converters operate on streams of 4-bit decimal or hexadecimal digits, and in real-time Global Positioning Systems (GPS), software backends process digitized streams of 2-bit and 4-bit samples. In general, the definition of a new language that extends Pablo and provides bit field operations defined over successive power-of-two bit fields can simplify the expression of a variety of stream processing algorithms such as parallel prefix sum [48, 7, 43] or inductive-doubling algorithms [11]. Here, inductive-doubling refers to a general property of particular algorithms that systematically double or halve bit-field widths or other data attributes with each step. These doubling transitions occur frequently in parallel bit stream programming as well as other applications. In addition to lacking power-of-two field stream representations, Pablo lacks features necessary to

[2]C++ template file here refers to a text file that contains a mixture of C++ code segments and Parabix placeholder elements that are replaced by compiled Pablo code segments in the process of application generation.

support performance scaling on parallel hardware. Most notably, the provision of language-level abstractions to express the concept of data-flow networks found in stream-processing languages such as StreamIt [86] and Brook [10]. The provision of language-level stream programming abstractions simplifies the identification of processing dependencies and communication patterns to the compiler enabling the extraction of combined data, task, or pipeline parallelism [39]. Furthermore, despite relative advantages of the Parabix framework in automating stages of parallel bitstream applications over manual practices, the Parabix framework relies on hand-coded templates for code generation. In general, a distinct application template must be written for each target application, language, and stream buffering strategy. Application template implementation is time-consuming and error-prone. In addition, the manual writing of application templates limits the ability of the programmer to evaluate design alternatives.

1.2 Basic Assumptions

Three assumptions form the basis for the $s2^k$ language.

Global-view programming language abstractions can simplify programming and improve application performance and scalability. A commonly held misconception is that programmer productivity is necessarily at odds with application performance and scalability [23]. That is, by raising the level of abstraction performance or scalability must suffer. However, global-view programming languages express computations or control flow over data aggregates such as arrays or streams. By raising the level of abstraction programmers can focus on the communication of algorithms rather than concentrating on error-prone programming aspects such as task partitioning, data partitioning, or indexing [24]. In addition, whereas fine-grained programming language abstractions can lead to the overspecification of algorithms, global-view programming abstractions enable compiler optimizations, freeing the programmer from writing low-level code [23]. Amongst others, the HPC (High Performance Computing) global-view languages ZPL (Z-level Programming Language) [26] and Chapel [25] demonstrate simultaneous improvements in programmer productivity, application performance, and scalability.

Stream programming languages can simplify programming and improve application performance in the stream-processing domain. Applications in the stream-processing domain share a number of common characteristics [88]. Notably, (i) the processing of large streams of data as a series of independent and self-contained transformation steps, and (ii) stable computation patterns. In essence, stream processing applications apply a series of data transformation operations on one or more input data streams in a regular and predictable order producing one or more output data

streams. Data items enter from some external source, are processed for a limited time, and are then discarded [87]. Bounded per data item processing times distinguishes streaming applications from many scientific applications that process relatively small inputs sets for substantially longer periods. Stream processing applications are commonly found in the domains of digital signal processing, networking, and encryption [88].

Stream programming languages represent applications as a graph of independent actors, commonly referred to as filters, kernels, or nodes, that communicate over data channels. Stream filters are generally independent and self-contained, that is, without references to global variables or other filters [86]. These characteristics simplify programming and result in performance improvements on parallel-hardware architectures [87]. In addition, the abstractions of filters and data channels make explicit data dependencies and communication patterns. This simplifies program analysis and performance scaling on parallel-hardware architectures [29]. Amongst others, the stream programming languages StreamIt [87] and Brook [10] demonstrate simultaneous improvements in programmer productivity, performance, and scalability.

Parallel bitstream processing techniques enable the design of efficient and scalable streaming text processing applications for parallel-hardware architectures. Hardware architectures continue to evolve and expose additional parallelism to software through the widening of SIMD ISAs (Instruction Set Architectures) and the addition of processor cores [33]. Although many streaming text processing applications can be expressed as software pipelines, challenges exist in scaling performance on parallel-hardware architectures [36]. A number of characteristics of streaming text processing present obstacles including, sequential processing dependencies, frequent data-dependent branches, irregular data item lengths, non-aligned memory accesses, and irregular memory access patterns [76].

In spite of these obstacles, several case studies [14, 17, 60, 68] demonstrate that parallel bitstream methods enable the design of scalable text-processing applications. A key insight into scalable application design using parallel bitstream methods is data organization [60]. Parallel bitstream methods operate on a transpose representation of text in which parallel bitstreams are constructed in one-to-one correspondence to the data values of a character stream. Register operations on parallel bitstreams allow the processing of up to the register bit width number of character positions in parallel. For instance, with the 128-bit Intel SSE2 (Streaming SIMD Extensions 2) ISA [50], bitstream methods can process up to 128 character positions in parallel. Likewise, with the 256-bit Intel AVX2 (Advanced Vector Extensions 2) ISA [50] bitstream methods can process up to 256 character positions in parallel [46]. 512-bit Intel AVX is anticipated in the near future [61]. In addition to scaling performance with register bit width, parallel bitstream techniques can scale performance with increases in processor core counts. Parallel bitstream methods expose inter- and intra-stream

processing dependencies[3]. [14] This dependency information permits compilers to automate the partitioning and scheduling of streaming text applications for multi- and many-core hardware [68].

1.3 Evaluation Methods and Past Results

To demonstrate the feasibility of the $s2^k$ language design, we implemented an $s2^k$ language compiler and runtime system. To assess the $s2^k$ language design, we implemented a number of $s2^k$ streaming text processing programs presented throughout this dissertation. $s2^k$ benchmark programs include: exact and regular expression string search, string to numeric conversion, and inverted-index construction. Prior case studies in bitstream programming demonstrate the performance benefits of our approach. Case studies include XML (Extensible Markup Language) well-formedness checking [17, 60] as well as the efficient generation of array sets for the construction of data structures [15].

The $s2^k$ language design is described along a number of axes --- simplification of the programming of streaming text processing programs in comparison to the Pablo programming language, performance, and scalability. However, the simplification of programming streaming text processing programs is the primary focus. Throughout this dissertation we present language design concepts and code segments. Performance is demonstrated through the generation of programs that correspond directly with our prior work. These performance results are referenced but not reproduced in full by this dissertation. A scalable language design is defined by this dissertation. Prior scalability results are referenced but not reproduced.

Simplification of Programming. The Pablo language has limited expressivity. In Pablo, operations defined on sequences of bit fields of width greater than one must be added manually as specialized C-family templates or libraries to produce complete text processig applications. Indeed, case studies [12, 15] emphasize that Pablo programming is a specialized skill and that higher-level text processing constructs are needed to make parallel bitstream technology accessible to the broader programming community. In this dissertation, we present $s2^k$ features that are (i) not expressible in similar languages including the Pablo language or (ii) are not easily expressible in C or C++. Particular emphasis is given to language features that are successful in global-view languages or stream programming languages.

[3]Intra-stream dependencies are exposed during the transformation of unbounded bitstream operations to functionally equivalent block-by-block operations.

Performance. To demonstrate the performance of $s2^k$ we reference our prior work that compares XML parsing benchmark applications with commercial applications as well as optimized hand-coded applications written in C++ [15, 60]. This work establishes that parallel bitstream methods support high-performance streaming text processing on parallel-hardware architectures. To report performance we required a metric. Typically this metric was throughput --- the maximum number of input bytes that can be processed per processor cycle. Additional factors such as branch mis-predictions and cache misses were also considered. Current $s2^k$ implementations generate XML well-formedness applications that correspond directly this prior work. The performance evaluations of this prior work show that parallel bitstream methods are efficient, demonstrating speedups of 2.5 to 4x on document-oriented XML input and 4.5 to 7x on data-oriented XML input.

Scalability. A programming model is an abstract model of computation that is used by the programmer to reason about how a program executes [66]. A processing model is similar but describes the manner in which hardware performs a computation. A programming language implementation translates a computation expressed relative to the programming model into the processing model used by the target hardware. In the ideal situation, the programming model expresses the most important aspects of the processing model and the programming language implementation simply translates the computation onto the target hardware.

Scalability is a frequently-claimed attribute of parallel programming models. Although the basic notion is intuitive, application of the term scalability to programming models has no generally accepted definition. Herein, we define, a scalable programming model as a programming model that (i) allows programmers to reason about computations without reference to specific data or processor distributions, and (ii) is not impacted by the addition of data or processing resources. A scalable programming language is a language that adopts a scalable programming model. Further, a transparently-scalable programming model is a scalable programming model whose concepts support a global-view of data or control. [4] Likewise, a transparently-scalable programming language is a language that adopts a transparently-scalable programming model. Array languages ZPL and Chapel are transparently-scalable.

Given this notion of scalability, a goal of $s2^k$ is the definition of a transparently-scalable programming model and programming language design that (i) enables programmers to focus on the aspects of algorithm design rather than the implementation details of a particular hardware, and (ii) supports the compilation of applications that scale with the available hardware resources with minimal programmer effort.

[4] The term transparently-scalable is used in contrast with programming models such as MPI whose concepts provide a per-processor view of data and control.

Prior performance evaluations of parallel bitstream benchmark applications [60] demonstrate that parallel bitstream methods scale with SIMD register width and demonstrate improvements with advances in SIMD ISAs. In particular, parallel bitstream methods that use 256-bit AVX technology when compared with methods that use earlier 128-bit SSE technology demonstrated a reduction in SIMD instruction count with the use of the AVX SIMD instructions. The observed reduction was attributed to both a doubling in SIMD register width with 256-bit AVX as well as the non-destructive 3-operand instruction format.[5] In addition, a pipeline partitioning of benchmark applications into a series of stages, with each stage assigned to a separate core, resulted in additional performance scaling.

1.4 Novelty

The design of the $s2^k$ language contains many novel aspects. First, $s2^k$ operations express transformations over arbitrary-length sequences of power-of-two bit fields. Second, the $s2^k$ processing model supports the expression of collective operations on sets of parallel streams that are in one-to-one correspondence with the characters of an input data stream. Third, $s2^k$ language design integrates high-level stream-programming abstractions with parallel bitstream techniques for streaming text processing. Fourth, although a variety of programming languages [86, 69, 10] use stream programming abstractions to exploit parallel hardware for numerical computations, equivalent languages do not exist for the streaming processing of text.

1.5 Contributions

The design and implementation of the $s2^k$ language, compiler, and runtime libraries is largely a collaborative effort, with many contributors to the ideas and implementations that support this dissertation. Consequently, this dissertation focuses on the ideas that have not been presented in prior works. However, at times, the presentation of a self-contained view of the original features of the $s2^k$ language features is insufficient and requires assimilating the work of others.

This dissertation makes the following contributions.

1. Specification of the $s2^k$ programming language.
2. A design rationale for the $s2^k$ language, emphasizing language features designed to simplify the programming of parallel streaming text processing programs.
3. Specification of the $b2^k$ intermediate language.

[5] The AVX ISA uses a non-destructive 3-operand instruction format. Previous SSE ISAs use a destructive 2-operand instruction format.

4. Specification of the $s2^k$ processing model.

5. Specification of the operational semantics of the $s2^k$ language constructs and built-in operations.

6. Specification of the operational semantics of the $b2^k$ intermediate language constructs and built-in operations.

7. Specification of a source-to-source translation scheme for the translation of $s2^k$ to $b2^k$.

8. Evaluation of $s2^k$ case study programs.

1.6 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background material reviewing trends in parallel-hardware architecture design, parallel text processing, stream processing, and related programming languages. Chapter 3 describes the $s2^k$ language, and Chapter 4 describes the $b2^k$ intermediate language. Chapter 5 presents source-to-source transformations of the $s2^k$ language to the $b2^k$ intermediate language, as well as the transformations of $b2^k$ to the C++ programming language. Chapter 6 presents an $s2^k$ case study in inverted-index construction. Chapter 7 concludes this dissertation. Appendices define relevant terms and notations, the $s2^k$ and $b2^k$ abstract grammars, the $s2^k$ libraries, as well as present complete program listings for $s2^k$ case study applications.

Chapter 2

Background

This chapter provides related background material in parallel hardware design, text processing, stream programming, and parallel bitstream programming methods. The remainder of this chapter is organized as follows. Section 2.1 describes trends in parallel hardware design. Section 2.2 characterizes the streaming text processing domain. Section 2.3 describes the stream programming paradigm and Section 2.4 describes the parallel bitstream programming paradigm. Section 2.5 surveys related parallel languages. Section 2.6 concludes this chapter.

2.1 Trends in Parallel Hardware Design

Hardware architectures continue to evolve and expose additional parallelism to software applications. Software-visible features include: widened and enhanced SIMD ISAs, multiple processor cores, and mixed CPU-GPU configurations [69]. Whereas enhanced SIMD ISAs and increased core counts have been shown to scale data-intensive numerical applications, finding ways to exploit additional parallel hardware features for streaming text processing is particularly challenging [67]. One factor is a mismatch in the character-at-a-time processing of variable-length text items with hardware features designed to work well with fixed-size data decompositions. For example, traditional text processing applications process a single character at a time, yet general registers are typically 32 or 64 bits and SIMD registers may be 128, 256, or even 512 bits. Beyond this the information computed per byte can be as low as a single bit [17]. A second factor is that text data items are typically not aligned to natural memory boundaries requiring a variety of techniques to avoid unaligned memory accesses such as replication of data in memory, padding of data items, loop peeling, or shift instructions [79]. A third factor is that many text processing applications exhibit irregular memory access patterns. In limited cases, parallel text processing libraries [2] and aggressive auto-parallelizing compilers can re-organize data access patterns or computations to

improve performance. However, re-organization steps typically impose performance penalties that can outweigh the benefits [14, 36].

2.2 Text Processing

Text processing encompasses a broad range of text-oriented applications, techniques, and algorithms, many of which can be described with the finite state machine (FSMs) model of computation. Here, we describe text processing problems that can be modelled with FSMs and present approaches to parallel FSM-based text processing.

Finite-State Machines

An FSM is an abstract model of computation that can be used to model a system whose behaviour is defined in terms of a set of states [6]. At any instance an FSM will be in one of the states, called the current state. The next state of the FSM is determined by the current state and a triggering event or condition, termed a transition [56]. In this manner, FSMs support the concept of a history as a sequence of dependent processing steps [75]. FSMs can be used to model a large number of text processing problems. For example, lexical analysis, as well as a number of text encoding and decoding algorithms, such as encryption and compression algorithms can be modeled with FSMs [75].

Strategies for Parallel Text Processing

A variety of approaches to parallel text processing on parallel hardware are noted in the literature. Most works investigate data parallel schemes that partition input documents into segments and then assign segments to separate cores or processors, such as MapReduce [32]. In general, document partitioning schemes require knowledge of the processing state at segment boundaries [17]. However, in text processing, segment boundary-crossing information dependencies may reach back to the beginning of a text stream. One approach to the identification of independent segments is to perform a low cost pre-parsing phase [17]. A second approach is to employ a speculative partitioning strategy. In speculative partitioning pre-processing costs are avoided but additional post-processing may be incurred. Under specific conditions the influence of computations can be bounded by the problem definition [72]. In such cases, an overlapping window strategy or a small post-processing phase to resolve boundary condition dependencies allows computations to proceed in parallel [75].

Fewer efforts investigate SIMD data parallel approaches. The Intel SSE 4.2 parallel string processing instructions [2] provide string and text processing instructions (STTNI) that use SIMD operations for processing text data on parallel hardware [57]. The SSE 4.2 instructions perform well

when text data are stored appropriately in memory, but degrade quickly when applications must reorganize text prior to computation [80]. Parallel bitstream methods first transpose text data to a transform domain prior to accelerate processing [68, 15]. Here, the cost of transposition is low relative to the overall processing costs. The advantage of this transpose representation is that processor registers can be used to perform up to register-width operations in parallel using bitwise logic, shifting and other operations. A few select studies investigate the application of software pipelining or other forms of mixed parallelism, such as the mixing of data and pipeline parallelism to text processing.

2.3 The Stream Programming Paradigm

Overview

Stream programming is a computer programming paradigm similar to SIMD programming [83]. In SIMD programming, data parallel operations are expressed over short vectors. In contrast, in stream programming operations are expressed on unbounded sequences of data elements. In the stream programming paradigm an application is expressed as a network of independent processing nodes[1]. Each node denotes an independent and self-contained stream processing operation. First-In First-Out (FIFO) channels connect producers to consumers and stream elements flow over producer-consumer channels. Communication is restricted to immediate neighbours and global data accesses are restricted to read-only constants. Taken together, this set of properties simplifies dependency analysis and the scheduling of stream programs on parallel hardware [39, 89]. In contrast to a traditional sequential control flow graph programming model, each node of a stream program can execute in parallel. The order of execution is constrained by the availability of data on input channels [51]. In a typical execution, input data are exhausted and results are appended to an output stream.

Advantages of Stream Programming Languages

Stream programming languages [86, 10] support performance and programmer productivity in the streaming domain through high-level abstractions [29]. Abstractions such as streams and kernels expose data dependencies and communication patterns, and thus enable whole program and stream-specific analysis and optimizations [87]. At the same time, high-level abstractions improve

[1] Although subtle differences exist in the concepts of stream node, kernel, filter, and actor, in our description of the stream programming paradigm, the terms are used interchangeably. Likewise, the terms channel and pipe refer to the same concept.

programmer productivity, and support program reliability and portability [86]. The following subsections describe the advantages of stream programming languages in terms of performance, programmability, portability and scalability.

Performance. Whereas, general-purpose programming languages can obscure dependencies, stream programming languages allow the programmer to structure an application such that dependency information is visible to the compiler [78]. The exposure of dependencies allows stream compilers to perform powerful optimizations. For example, the combining of multiple forms of parallelism (data, task, and pipeline parallelism) in a single application, or the fusing of nodes to increase arithmetic intensity [10]. Modern stream compilers generate device-specific code that is competitive with low-level hand-written code and is suitable for parallel execution [34].

Parallelism. Multiple forms of parallelism are inherent to the structure of a stream graph [39]. In a stream graph, data-parallelism is inherent to *stateless* nodes; nodes that do not have data dependencies. Data-parallelism is well-suited to multicore architectures with SIMD units. Task parallelism is a property of node pairs located on separate parallel branches of a stream graph, such that the output of each kernel does reach the input of the other. Task parallelism denotes a logical parallelism of the underlying algorithm. For example, the nodes N3 and N4 in Figure 2.1 are task parallel. Pipeline parallelism is a property of chains of producers and consumers. For example, the chain of kernels N1, N2, N3, N5 in Figure 2.1 can form a pipeline. Pipeline parallelism enables the parallel execution of chains of producers and consumers. In pipeline parallelism load balancing is critical since the throughput of the stream program is equal to the minimum throughput across each of the nodes involved in a pipelined computation [41].

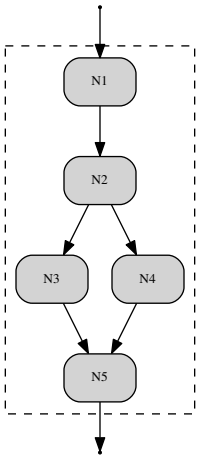

Figure 2.1: A stream graph representation of a stream program.

Arithmetic Intensity. Arithmetic intensity is defined as the ratio of arithmetic operations to memory bandwidth [10]. Applications with high arithmetic intensity are suitable for execution on multicore and manycore architectures in which large volumes of instructions can serve to hide relatively slow memory access times. In contrast, programs with low arithmetic intensity may under utilize functional units or stall while waiting for memory transfers. Although stream programming languages encourage programmers to write express programs as kernel functions with high arithmetic intensity [89, 10], stream graph transformations such as kernel-fission and kernel-fussion can be applied to increase the arithmetic intensity of stream program kernels [36].

Programmability. Programmers commonly use general-purpose programming languages such as Java, C, or C++ to express stream processing computations [86]. However, general-purpose languages lack stream-specific abstractions and thus stream operations are often expressed in a low-level procedural style using assignment statements and structured control flow. For example, in a procedural language such as C, a kernel may be expressed as a block of statements organized as a complicated loop nest. An object-oriented implementation avoids some of the coding complexity of a C program yet still obscures the stream processing logic [86]. The use of general-purpose languages for stream programming has negative impacts on program readability, reliability, and programmer productivity. Performance critical sections often require custom implementations for each target architecture. As well, general-purpose language compilers do not perform stream-specific optimizations [89].

In contrast, stream programming languages expose the concept of a stream in the language design; either explicitly as part of the language type system [10] or implicitly in the design of built-in libraries [86]. The inclusion of a stream abstraction allows programmers to reason about operations expressed over entire collections of data elements. Benefits are realized in program reliability and programmer productivity.[2] For example, Figure 2.2 demonstrates array addition as expressed in a procedural language such as C. In this example, explicit loop indexing and array access operations decrease readability and reliability. Moreover, loop dependence analysis (albeit trivial) is required to automatically parallelize the computation. In contrast, Figure 2.3 shows the equivalent computation expressed in a stream programming language. In this example readability and reliability improve. Furthermore, the meaning of the operation is trivially obvious to the programmer as well as to the compiler.

```
for(i = 0; i < n; i++)
{
  A[i] = A[i] + B[i];
}
```

Figure 2.2: Procedural language operation expressed on a sequence of data values.

```
A = A + B;

or

A += B;
```

Figure 2.3: Stream programming language operation expressed on a sequence of data values.

Portability and Scalability. Stream programming languages abstractly express a stream program in terms of kernels and streams. Kernels and streams describe the high-level structure of a program in which each kernel defines a series of independent transformation operations on a set of input streams, producing a set of output streams. Stream compilers leverage knowledge from the high-level structure of a stream program together with the property of kernel-independence to scale stream programs to the available hardware resources [78]. In stream program compilation, kernel fissioning is the analog of parallelization. Kernel fissioning transformations enable adjustments in the granularity of a stream graph which in turn support improvements in the use of processor cores [40]. Vertical fissioning transformations split kernels into a series of pipeline stages. Each stage performs part of the work of the original kernel. Horizontal fissioning transformations partition stream segments, replicating and distributing a single filter across a number of computational units. In addition to scaling applications through processor cores, stream compilers can scale stream programs through vectorization. The execution of a kernel on a stream performs an implicit loop over

[2] A large variety of programming languages express operations over entire collections of data. The benefits of the expression of operations on collectives applies equally to those languages.

the elements of the stream, often performing an independent operation on each element [10]. Independent per-element operations allow stream compilers to use leverage per-core SIMD processing instructions thus providing an additional parallelism.

2.4 The Parallel Bitstream Programming Paradigm

Overview

The parallel bitstream programming paradigm represents a fundamentally new approach to parallel text processing that employs bitwise data-parallel methods together with the SIMD capabilities of processors to deliver dramatic performance improvements over traditional byte-at-a-time parsing technology [12]. The key insight into the bitstream programming paradigm is data organization [60]. In parallel bitstream programming a first step is the derivation of a set of bitstreams in one-to-one correspondence with the characters of a text stream. This set of bitstreams is called the *basis bitstreams*. In essence, the kth bit of ith character of the source input stream are derived in one-to-one correspondence with the ith bit of the kth basis bitstream. Next, through a sequence of logical bitstream operations, the basis bitstreams are used to derive parallel bitstreams termed property bitstreams. Property streams are bitstreams that mark positions of interest such as occurrences of characters or meta-characters associated with data values of a source input stream. By transitivity, property streams are in one-to-one correspondence with the character code units of a source stream.

In parallel bitstream programming text processing operations are expressed on property streams. The parallel bitstream programming model defines a set of bitwise logic and arithmetic operations on unbounded bitstreams. Bitwise logic and arithmetic operations on unbounded bitstreams correspond directly to bitwise logic and arithmetic operations on unsigned integers. This set of operations serves as the basis for the definition of further text processing operations that support bitwise data parallelism. Each text processing operation is defined independently of the underlying hardware architecture and assumes unlimited data parallelism. Together with text processing operations, the parallel bitstream programming model provides *parallel if* optimizations to conditionally evaluate stream statements, as well as *parallel while* constructs to support bitstream iteration.

A number of studies describe various aspects of parallel bit stream programming in more detail. Fundamental methods including transposition, character class information and sequential bit scans are described in applications involving transcoding [14] and XML parsing [17]. Other notable techniques include parallel scanning with bitstream addition [15] and the inductive-doubling methods for SIMD processing parallel bitstreams [11]. The Pablo language and Parabix tool chain support the expression of programs using unbounded bitstream operations and their translation to block-by-block C++ code [60].

Advantages of the Parallel bitstream Processing Paradigm

Performance. A number of case studies have investigated the use of parallel bitstream methods to accelerate streaming text processing applications. Amongst others, benchmarked applications include: UTF-8 to UTF-16 character set transcoding, XML well-formedness checking, and XML-to-XML document transformation. Applied to the problem of UTF-8 to UTF-16 character set transcoding parallel bitstream techniques achieved speed ups of 3 to 25 times over optimized byte-at-a-time code [14]. Initial studies in XML well-formedness checking investigated the replacement of byte-at-a-time scanning loops with built-in *bit scan* instructions. Using bit scan instructions, bitstream techniques for sequential scanning can advance markers by the bit width of a general purpose register (32 or 64 bits) in a single instruction. This sequential scanning approach yielded considerable performance improvements over traditional byte-at-a-time parsing technology [17, 45]. Further studies in XML well-formedness checking presented a new parallel scanning method using the concept of bitstream addition. On processors supporting w-bit addition operations, this method can perform up to w finite-state transitions per instruction. This parallel scanning approach yielded speed-ups beyond previous sequential approaches [15]. In [60] data parallel bitstream operations were combined with software pipeline parallelism. This hybrid approach to parallel text processing yielded speed ups beyond those of parallel bitstream methods alone. Moving beyond research prototypes, [68] investigated the integration of parallel bitstream methods into a commercial XML parser --- Xerces-C++ parser of the Apache Software Foundation. This work represented the first case study documenting the performance benefits that may be realized through the integration of parallel bitstream technology into an existing and widely-used software library. Case studies in XML-to-XML transformation investigated the use of parallel bitstream techniques to accelerate the transformation of Geography Markup Language (GML) document format to the Scalable Vector Graphics (SVG) format. In this study parallel bitstream methods were compared against a variety of XML processing technologies with speed up ranging from 1.2 to 4.5 times [17].

Parallelism. The parallel bitstream programming model uses multiple forms of parallelism in combination ranging from bit-level data parallelism to more coarsely grained data, task, and pipeline parallelism. For instance, in parallel bitstream programming, bitwise logic, arithmetic, and shift operations simultaneously process multiple bit positions in parallel. On processors supporting w-bit operations, w positions are processed in parallel. Moreover, with the introduction of superscalar CPU architectures, multiple w-bit operations can be dispatched during a single clock cycle. Beyond this traditional application of bitwise data parallelism, the parallel bitstream programming model supports an additional form of bit-level parallelism. We term this form of parallelism *bitwise flow-parallelism* or simply *flow parallelism*. In parallel scanning using bitstreams, bitwise flow-parallel operations are defined that leverage the carry mechanism of bitwise addition to simultaneously

scan multiple variable-length data items. For instance, given a span stream T together with a marker stream M that marks the start position of each span, the addition of M to T advances each marker bit m in M independently through each individual span t of T. We term operations defined in this manner *parallel scanning* operations. Likewise, given a starts positions stream S and a follow positions streams E, the subtraction of S from E creates a span stream that marks all bit positions between pairwise first and follow positions, including first positions and excluding follow positions. We term operations defined in this manner *parallel spanning* operations. As described in Section 2.3, the organization of an application as a stream graph exposes program dependencies and enables the use of multiple forms of parallelism.

Programmability. The parallel bitstream programming model benefits programmability in a number of ways. First, parallel bitstream operations are global-view operations expressed on unbounded streams. In general, these operations are expressive, clear, and concise [26]. Second, the parallel bitstream programming model allows programmers to reason in terms of transformative operations applied to entire collections. Third, the parallel bitstream programming model is based on a structured form of data parallelism that by construction avoids common sources of error such as race conditions and deadlock. The result of bitstream computations is consistent with a single, deterministic, serial ordering. Determinism together with sequential consistency simplifies maintenance, testing, and debugging. Fourth, parallel bitstream programs are shorter and closer to a mathematical problem specification. In the parallel bitstream programming model, compiler implementations are responsible for the selection of SIMD instructions specific to particular ISAs as well as the partitioning of computations for differences in multicore hardware.

Despite the benefits, a global-view programming model to the programmer is not without additional complexity in compiler implementations. In general, compilers and runtime systems for global-view languages must be more sophisticated [25]. bitstream operation translations must ensure the propagation of carry-bit and borrow-bit information at register, buffer, and file boundaries. Moreover, the propagation of information is context dependent. For instance, differences exist in the translation of bitstream operations within if statements and while loops.

Portability and Scalability. The parallel bitstream programming model does not expose processor specific features and is thus portable by design. As previously described, case studies in text processing demonstrate that parallel bitstream techniques scale with increased SIMD register width and processor core counts [60, 68].

2.5 Related Work

A wide variety of languages have provided variations on stream programming abstractions, including many elegant ways of expressing streaming computations. Notably, Stephens provides a comprehensive survey of stream programming languages [82]. This survey, however, does not address the latest trends in hardware architecture. A more recent trend in stream programming language design is the development of languages for special purpose-stream processing hardware. For example, the Brook [10] language was designed to program Stanford's Merrimac streaming supercomputer [30], whereas the StreamIt [86] language was initially designed to program MIT's RAW architecture [84]. An emerging trend is the design of stream programming languages for parallel-hardware architectures. In particular, languages designed to leverage the multicore parallelism of general-purpose processors (GPPs) [51, 65, 42] or the manycore parallelism of general-purpose graphics processing units (GPGPUs) [69, 22]. Such languages commonly express data parallel operations over collections [67]. Data-parallel languages can be imperative or functional with collective operations made available through external or built-in libraries. Given the number and variety of parallel programming languages the following sections non-exhaustively describe languages that relate to the design of $s2^k$.

Functional Languages

In lazy functional languages, such as Scheme [35], streams represent *infinite* sequences, implemented as delayed lists, and processed with the same set of operations as lists (map, filter and accumulate). The strategy of lazy evaluation delays the evaluation of an expression until its value is needed and requires support for incremental evaluation of streams [4]. Differences in terminology in functional languages and stream programming languages primarily indicate a difference of community [87]. Functional languages are not surveyed further herein.

Stream Programming Languages

Brook. Brook [10] is a stream programming language for programming modern graphics hardware. Brook provides two primary performance benefits to streaming numerical applications, (i) data-parallelism, and (ii) arithmetic intensity. A Brook program consists of legal ANSI C code together with language extensions to define streams and kernels. In Brook, a stream is an abstract data type that enables the representation of an unbounded sequences of elements that can be operated on in parallel. To accommodate variety in hardware architectures, the layout of stream elements within memory is hidden from the programmer. Kernels are special parallel functions that

operate on every element of an input stream. In Brook, calling a kernel function on a set of input streams performs an implicit loop over the elements of the streams and hence Brook may be considered a collective language.

StreamIt. StreamIt [86] is a programming language and compiler infrastructure designed to facilitate the programming of data-intensive numerical streaming applications. StreamIt targets a variety of architectures, which include: single-core and multicore architectures, as well as workstation clusters. In StreamIt, a stream program is expressed as a stream graph of filters connected via stream data channels. Each StreamIt filter defines an initialization and steady state work function, as well as declares input and output data channel rates. StreamIt filter operations are expressed in Java, together with built-in push, pop and peek operations to express the data transfers to neighbours.

Data Parallel Languages

There is an extensive literature investigating the many design alternatives for data parallel languages. A common approach has been the provision of a collective data type with inherently parallel semantics, *e.g.*, an array or list type. The following subsections highlight data parallel languages relevant to the design of $s2^k$.

NESL. NESL [8, 9] is a high-level language designed for the expression of nested data-parallel programs. In NESL, implicitly-parallel operations are applied to sequences using a set-like notation. NESL pioneered the field of data parallel languages and demonstrated that complex algorithms on nested data structures can be expressed at a high-level of abstraction.

ZPL. ZPL [81] is an array programming language for science and engineering computations. ZPL provides a set of parallel abstractions that includes: array language semantics, regions, flooding, reductions and scans. In ZPL, programmers conceptualize and apply parallel operations to entire arrays of values. ZPL is an implicitly-parallel global-view programming language with instructions to realize data parallelism inserted by a ZPL compiler.

Copperhead. Copperhead [22] is a data-parallel language embedded in Python. Copperhead provides language constructs with implicitly parallel semantics, such as map, zip, reduce, and permute to manipulate 1-dimensional data arrays. Copperhead uses Python annotations to mark Copperhead procedures. Procedures are just-in-time compiled to generate data-parallel C++ code for execution by the Python runtime. Non-Copperhead procedures are executed as normal Python programs.

Intel® Array Building Blocks. Intel® The Array Building Blocks [69] (ArBB) language is based on the research related to the Sh project [33, 38], ArBB is a data-parallel language embedded in C++ design to leverage multicore processors, GPUs and Intel Many Integrated Core Architecture processors whose syntax is implemented as an application programming interface (API). ArBB defines new parallel types, such as collections, that support the simultaneous extraction of data and task parallelism. Unlike NESL-like languages ArBB does not support operations on arbitrarily nested vectors.

Pablo Prototype Language. As described earlier, Pablo is a prototype language for the expression of text processing applications using parallel bitstreams [17, 15]. Pablo programs are translated to efficient block-at-a-time C++ code for execution on parallel hardware using the Parabix application framework [60].

2.6 Chapter Summary

In designing $s2^k$ we have studied numerous successful and unsuccessful language and library designs. We have examined selected features that we believe have the potential to work well together. $s2^k$'s primary influences include: C, C++, Brook, StreamIt, ZPL, Chapel, ArBB, and Pablo. To a lesser extent, languages that support collective operations such as APL, MatLab, HPF, as well as well-crafted programming languages such as Ada and Scheme influence the design of $s2^k$. Intermediate language designs that influence the design of the $s2^k$ intermediate language $b2^k$ described in Chapter 4 include C-- and LLVM in which the type system are deliberately designed to reflect hardware constraints, *i.e.*, the representation of values that can be stored as register values.

Chapter 3

The s2^k Programming Language

This chapter describes the s2^k programming language. The remainder of this chapter is organized as follows. Section 3.1 introduces the s2^k language. Section 3.2 presents the language design principles. Section 3.3 describes the structure of the s2^k programming model and Section 3.4 presents key programming model concepts. Section 3.5 goes on to describe s2^k language features such as the type system, expressions, control-flow statements, and built-in s2^k operations. Section 3.6 describes common s2^k programming styles. Section 3.7 introduces fundamental s2^k programming concepts with a programming example, 8-bit ASCII string to 16-bit binary integer conversion. Section 3.8 goes on to discuss advantages of the s2^k language design and Section 3.9 concludes the chapter.

3.1 Overview

s2^k is a global-view parallel language for streaming text extraction and transformations. Similar to other global-view languages, s2^k language abstractions support a global view of computation and a global view of control. Global-view constructs unburden the s2^k programmer from many low-level details of programming such as writing stream processing loops, managing buffer-boundary conditions, and specifying index calculations. In general, the s2^k programmer is concerned with the organization of stream data, the expression of stream transformations, and the gathering of results. The s2^k compiler and runtime system manages the details of block-by-block stream processing, parallelism, and memory management.

3.2 Design Principles

In this section, we present the fundamental s2^k language design principles. This list should be considered an exploration of themes rather than a specification of requirements.

Clarity. Programming languages communicate algorithms to a compiler. The more clearly a language describes programmer intentions, the more semantic information the compiler has available to optimize programs [24].

A Global-View of Computation and Control. Global-view languages express computations and control-flow on collections of data such as arrays, lists, or streams. In contrast to local-view languages, in which the programmer must explicitly handle the low-level programming aspects of indexing and partitioning data for parallel execution, global-view languages raise the level of abstraction, simplifying programming and facilitating compiler optimizations [86, 26, 25].

A Deterministic Parallelism Programming Model with Serial Semantics. Adoption of a deterministic parallel programming model with serial semantics has significant advantages. Notably, programs can be analyzed without concern for race conditions, deadlocks, or other complications inherent to shared memory programming models [67, 69, 22].

Support for Mixed Forms of Parallelism. Hardware architectures make multiple forms of hardware parallelism available to software applications including the provision of multiple processing elements, SIMD units, and GPUs. As such, an objective of $s2^k$ is providing a multi-resolution language design that permits the use of multiple forms of parallelism.

Avoidance of Unnecessary Sequential Language Abstractions. Inherently sequential-programming language abstractions can introduce programming complexity and lead to sequential implementations that perform poorly [67]. For example, priority scatter operations that resolve collisions deterministically adhere to rules that may be confusing, error-prone, and that require sequential implementations.

Interoperability and Separation of Concerns. Although domain-specific computations are, by design, expressible in a straightforward manner with a domain-specific language, DSLs are, by definition, of limited expressiveness [51]. Furthermore, due to the large volume of existing code it is unrealistic to believe that many existing programs will be rewritten using a new programming language [25]. As such, language features that allow domain-specific programs to interact with general-purpose languages in a straightforward manner are desirable.

3.3 Programming Model Structure

The $s2^k$ programming model adopts a two-level structure, with the $s2^k$ *stream-graph* level modelling coarser-grained computations and data-flow between filters, and the $s2^k$ *filter-level* defining

finer-grained sequences of stream transformation statements. In s2^k the fundamental stream data type is the 2^k-bit Fieldstream data type --- a parameterized type that denotes an ordered sequence of 2^k-bit values for some non-negative integer k as described in Section 3.5.2. Overall, this two-level organization is simple yet sufficiently expressive to support a variety of text processing applications.

Stream-Graph Level.

The stream-graph level of an s2^k program models the structure of a program as a network of filters that transform input streams into output streams. The vertices of the stream-graph denote filters and the edges denote one or more 2^k-bit Fieldstreams. Each filter defines one or more output Fieldstreams through the execution of sequences of Fieldstream operations on one or more inputs. Edges connect filter outputs to neighbouring filter inputs. This organization captures the high-level data flow of an s2^k program, making data access patterns explicit, and providing opportunities for the modelling of pipeline parallelism.

Ignoring the implementation details for now, Figure 3.1 shows the stream-graph of an example s2^k program. The starting point of the program is a stream read operation. As shown in Figure 3.1, the *Read* filter transfers a data stream from the host language space to the s2^k embedded language space. Next, the *Transpose* filter transposes the input stream producing a set of basis bitstreams. The *Classify* filter then consumes the set of basis bitstreams and produces a set character-class streams. Next, the *Convert* filter consumes both the character-class stream set and the input stream, and performs a Fieldstream transformation operation, 8-bit ASCII to 16-bit binary decimal integer conversion. Finally, the *Write* filters writes the results stream to the host language space.

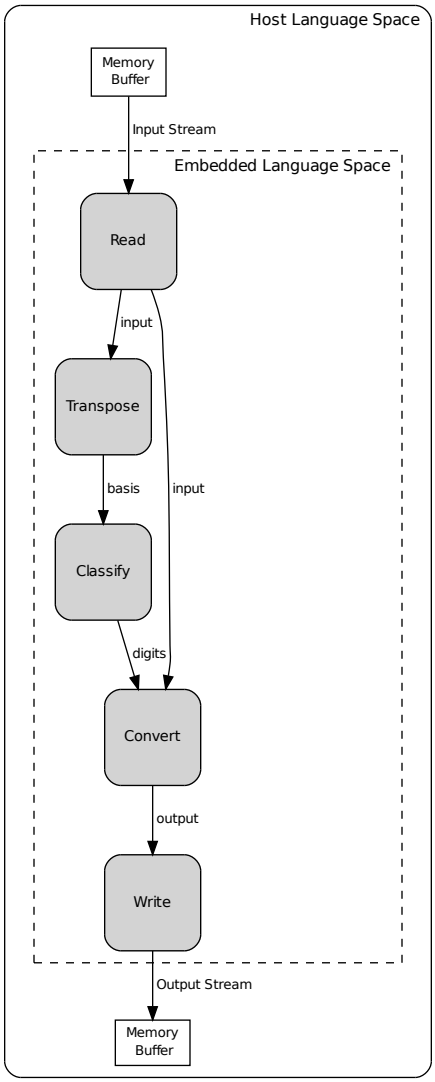

Figure 3.1: Example $s2^k$ stream-graph.

Stream-Filter Level.

s2^k stream filters define independent and self-contained Fieldstream transformations. The statements of a filter resemble the statements of an imperative programming language. However, unlike imperative languages, that update scalar values, s2^k operations are, in general, collective operations over n-bit Fieldstreams that produce n-bit Fieldstream values without side effects.[1] Scalar operations are permitted in s2^k but are restricted to constant expressions.

s2^k operations have value semantics. Function arguments are never modified by function calls and assignments have copy-by-value semantics. For example, the s2^k statement `t = Add<8>(t,t)`, denotes the 8-bit element-wise sum of two distinct copies of an 8-bit Fieldstream, `t`, which are copied and assigned to a copy of `t`. Value semantics simplify programmer reasoning and yield program statements which produce the same results regardless of the execution environment. Furthermore, unlike data-parallel programming models that allow the mixing of scalar and vector operations and control-flow, the s2^k programming model restricts non-constant expressions and control-flow to Fieldstream types. Global accesses are restricted to read-only constants.

Overall, the properties of the filter-level model eliminates complexities associated with many imperative languages such as pointers and aliases as well as non-deterministic program behaviours due to language features such as threads. Finally, the filter-level model provides a clear separation of scalar and vector programming concepts that simplifies streaming text programming.

3.3.1 s2^k Programming Model Versus the SPMD Stream Programming Model.

The s2^k programming model resembles the SPMD stream programming model of languages such as Brook [10] and Intel's Array Building Blocks [69]. In the SPMD programming model a program is organized into a stream graph of SPMD kernels. In contrast to SIMD kernels, which execute fixed instruction sequences and use data-flow techniques for conditional execution, SPMD kernels resemble general-purpose programs which include control-flow and iteration statements. This allows SPMD kernels to select different execution paths based on computed values in order to short circuit computations and to iterate when necessary. In this way, the s2^k programming model is similar to the SPMD stream programming model.

The s2^k programming model diverges from the SPMD stream programming model in the translation of kernels for parallel execution. In the SPMD stream programming model, no assumption is made on the order in which kernels are executed, and data-partitioning schemes are typically straightforward [66]. However, in s2^k, stream filters are serially dependent and thus cannot be

[1] As a programmer convenience, the s2^k language definition provides logical AND, OR, and NOT operators defined on 1-bit Fieldstreams. These operators are semantically equivalent to the corresponding s2^k operations.

executed in an arbitrary order. As such, in s2^k, the term *stream programming* refers to stream programming systems that exhibit both intra- and inter-stream dependences, and that use methods such as pipeline parallelism for parallel execution.

3.4 Programming Model Concepts

This section introduces the s2^k programming model. A programming model is an abstract model of computation used by the programmer to reason about program execution [66]. At the highest level of abstraction, the s2^k programming model is a data-parallel stream programming model with a global-view of computation and control. In s2^k, sequences of Fieldstream operations, together with conditional and iteration statements, transform sets of input streams into sets of output streams. The s2^k programming model differs from other data-parallel models through the provision of both bitwise data-parallel *flow-Parallel* operations defined over 1-bit Fieldstreams (bitstreams), and data-parallel *field-parallel* operations defined over 2^k-bit Fieldstreams. Together these operations allow the programmer to express parallel text processing computations on variable-length text data items at arbitrary stream offsets. We next describe the Fieldstream programming model concepts and operation categories.

3.4.1 Parallel Fieldstreams.

To define parallel Fieldstreams we first define the 2^k-bit Fieldstreams as well as basic properties of 2^k-bit Fieldstreams. A 2^k-bit Fieldstream, denoted T_{2^k}, is a homogeneous sequence of 2^k-bit fields for some non-negative integer k. The prefix of 2^k-bit may optionally be omitted if brevity improves clarity. A bitstream, denoted T_1, is a 1-bit Fieldstream. In s2^k, bitstreams are semantically equivalent to arbitrary length unsigned integers. A Fieldstream field, denoted t_{2^k}, is an element of a Fieldstream. A 2^k-bit field is an ordered sequence of 2^k 0s and 1s for some non-negative integer k. The prefix of 2^k-bit field may also be omitted. The length of a Fieldstream T_{2^k}, denoted $Length(T_{2^k})$, is the number of fields in T_{2^k}. The width of each field of a Fieldstream T_{2^k}, denoted $Width(T_{2^k})$, is the number of bits in each field of T_{2^k}. The size of a 2^k-bit Fieldstream T_{2^k}, denoted $Size(T_{2^k})$, is defined as $Length(T_{2^k}) * Width(T_{2^k})$. Since Fieldstreams are homogeneous sequences of fields, the width of all fields of a Fieldstream are equal. Two or more Fieldstreams with fields in one-to-one correspondence are termed parallel Fieldstreams. For example, Fieldstreams S_{2^k} and T_{2^j} are parallel if and only if at each field position i $S_{2^k}[i]$ is associated with $T_{2^j}[i]$ by a particular property of the data values represented, $T_{2^j}[i]$ is associated with $S_{2^k}[i]$ by a particular property of the data values represented, and $Length(S_{2^k})$ equals $Length(T_{2^j})$ for some non-negative integers k and j. Here, the syntax $S_{2^k}[i]$ indexes the ith field of S_{2^k}.

3.4.2 Fieldstream Operations.

This section describes the s2^k Fieldstream operations. We introduce a simple model and notation for the operation categories and define operation semantics.

Field-parallel Operations. The field-parallel operations are a set of collective operations designed to support the simultaneous calculation of n-bit Fieldstream fields, for $n = 2^k$ where $0 \leq k \leq K$. The field-parallel operations can be implemented using SIMD instruction registers. Typical values for K include: $K = 6$ for the 64-bit registers of Intel MMX, $K = 7$ for the 128-bit registers of Intel SSE, $K = 8$ for Intel AVX, and $K = 9$ for Intel AVX2 technology. SIMD registers of bit width N can be partitioned into N/n fields of width $n = 2^k$ bits for values of k, where $0 \leq k \leq K$. Typical values of k include $k = 3$ for operations on 8-bit fields (bytes), $k = 4$ for operations on 16-bit fields and $k = 5$ for operations on 32-bit fields, however, field-parallel operations support all values of k. Similar values of k and K can be defined for additional (non-Intel) SIMD ISAs.

A goal of the field-parallel operations is to support the expression of inductive-doubling algorithms on the n-bit fields of Fieldstreams in accordance with the idealized inductive-doubling architecture defined on SIMD registers of size $N = 2^K$ [11]. This idealized architecture consists of four key elements. First, the definition of a set of binary functions on n-bit fields for all field widths $n = 2^k$, where $0 \leq k \leq K$. Second, the definition of a set of bit field processing modifiers that allow the inductive processing of fields of size $2n$ in terms of combinations of n-bit values selected from the high-half and the low-half of $2n$-bit fields, where $0 \leq k < K$. Third, the definition of packing operations that compress two consecutive registers of n-bit values into a single register of $n/2$-bit values, where $1 \leq k \leq K$. Fourth, the definition of merging operations that produce a set of $2n$-bit fields by concatenating corresponding n-bit fields from a pair of consecutive registers, where $0 \leq k < K$.

The s2^k field-parallel *vertical*, *horizontal packing*, and *expansion* operation categories support the elements of the idealized inductive-doubling model defined over n-bit Fieldstreams. A fourth category of operations called *field-movement* operations, support m-bit subfield operations on n-bit Fieldstream elements, for $m = 2^j$, where $0 \leq j \leq k \leq K$. Appendix C.2 specifies the s2^k field-parallel operations.

Vertical Operations. Vertical operations combine corresponding n-bit fields of Fieldstreams of length L, producing an n-bit Fieldstream of length L. Vertical operations, defined over n-bit Fieldstreams of length L and denoted $R_n = F_n(S_n), \ldots, R_n = F_n(S_n, T_n, U_n)$, are defined by the

element-wise calculation of the n-bit fields of Fieldstream arguments S_n, T_n, or U_n. Vertical operations produce an n-bit Fieldstream result R_n of length L in accordance with a specific conversion function $f_n(s_n) \ldots f_n(s_n, t_n, u_n)$ defined on n-bit values s_n, t_n, or u_n for $n = 2^k$ where $0 \leq k \leq K$.

$$R_n[i] = f_n(S_n[i]), \forall i < L. \tag{3.1}$$

$$R_n[i] = f_n(S_n[i], T_n[i]), \forall i < L. \tag{3.2}$$

$$R_n[i] = f_n(S_n[i], T_n[i], U_n[i]), \forall i < L. \tag{3.3}$$

For example, given n-bit input values s_n and t_n, n-bit binary addition operation is defined as follows.

$$r_n = add_n(s_n, t_n) = (s_n + t_n) \mod 2^n. \tag{3.4}$$

Correspondingly, given n-bit Fieldstreams S_n and T_n of length L, n-bit Fieldstream addition is defined as follows.

$$R_n[i] = add_n(S_n[i], T_n[i]) = (S_n[i] + T_n[i]) \mod 2^n, \forall i < L. \tag{3.5}$$

A subcategory of the Vertical operations defines the inductive processing of n-bit fields in terms of binary functions defined on the high and the low $n/2$-bits of an n-bit fields of a Fieldstream. For example, given an n-bit Fieldstream S_n of length L, n-bit *high/low addition* is defined as follows.

$$R_n[i] = addHiLo_n(S_n[i], S_n[i]) = hi_n(S_n[i]) + lo_n(S_n[i]), \forall i < L. \tag{3.6}$$

Here, given an n-bit value s_n, auxiliary functions $hi_n(s_n) = s_n/(n/2)$ and $lo_n(s_n) = s_n \mod (n/2)$ mask the high or low $n/2$-bits respectively of an n-bit field respectively and return an $n/2$-bit field.

Horizontal-packing Operations. The Horizontal-packing operations are operations which reduce n-bit Fieldstreams of length L, producing an $n/2$-bit Fieldstream of length L. The Horizontal-packing operations, defined over n-bit Fieldstreams of length L and denoted $R_{n/2} = F_n(S_n)$, are defined by the element-wise calculation of $n/2$-bit fields given an n-bit Fieldstream argument S_n. The Horizontal-packing operations produce an $n/2$-bit Fieldstream result, $R_{n/2}$ of length L, in accordance with a specific function f_n defined on n-bit values that produce an $n/2$-bit Fieldstream for $n = 2^k$ where $1 \leq k \leq K$.

$$R_{n/2}[i] = f_n(S_n[i]), \forall i < L. \tag{3.7}$$

For example, given an n-bit Fieldstream value S_n, the $PackHi_n$ Fieldstream operation is defined as follows.

$$R_{n/2}[i] = hi_n(S_n[i]), \forall i < L. \tag{3.8}$$

Here, given an n-bit value s_n and $n/2$-bit value $r_{n/2}$, the auxiliary function $hi_n(s_n)$ returns the high $n/2$-bits of s_n such that $r_{n/2} = s_n/(n/2)$.

Similarly, the $PackLo_n$ Fieldstream operations is defined as follows.

$$R_{n/2}[i] = lo_n(S_n[i]), \forall i < L. \tag{3.9}$$

Here, given an n-bit value s_n and $n/2$-bit value $r_{n/2}$, the auxiliary function $lo_n(s_n)$ returns the low $n/2$-bits of s_n such that $r_{n/2} = s_n \mod (n/2)$.

Expansion Operations. The Expansion operations are operations which expand n-bit Fieldstreams of length L, producing an $2n$-bit Fieldstream of length L. The Expansion operations, defined over n-bit Fieldstreams of length L and denoted $R_{2n} = F_n(S_n, T_n)$, are defined by the element-wise calculation of the $2n$-bit fields of n-bit Fieldstream arguments S_n and T_n. The Expansion operations produce an $2n$-bit Fieldstream result, R_{2n} of length L, in accordance with a function f_n defined on n-bit values that produces $2n$-bit values for $n = 2^k$ where $0 \leq k \leq K$.

$$R_{2n}[i] = f_n(S_n[i], T_n[i]), \forall i < L. \tag{3.10}$$

For example, given n-bit input values s_n and t_n, the n-bit merging operation is defined as follows.

$$r_{2n} = merge_n(s_n, t_n) = (s_n * 2^n) + t_n. \tag{3.11}$$

Correspondingly, given n-bit Fieldstreams S_n and T_n, the n-bit Fieldstream merging operation is defined as follows.

$$R_{2n}[i] = merge_n(S_n[i], T_n[i]), \forall i < L. \tag{3.12}$$

Field-movement Operations. Field-movement operations permute, broadcast, and set the fields of n-bit Fieldstream of length L, producing an n-bit Fieldstreams of length L. Field-movement operations, defined over n-bit Fieldstreams of length L and denoted $R_n = F_{n,m}(S_n, T_n)$ or $R_n =$

$F_{n,m}(S_n, c)$, are defined by the simultaneous calculation of the individual n-bit fields of n-bit Fieldstream arguments S_n,T_n, or constant integer c. Field-movement operations produce an n-bit Fieldstream result, R_n of length L, in accordance with a function f_n defined over the m-bit subfields of n-bit values of the following forms, for constant integer c, $n = 2^k$, and $m = 2^j$, where $0 \leq j \leq k \leq K$.

$$R_n[i] = f_n(S_n[i], T_n[i], L), \forall i < L. \tag{3.13}$$

$$R_n[i] = f_n(S_n[i], c, L), \forall i < L. \tag{3.14}$$

Casting Operations. The s2^k language defines a single explicit Fieldstream casting operation. The casting operation $R_c = BitCast_n(S_n, c)$ re-interprets a 2^k-bit Fieldstream, S_n, as a c-bit Fieldstream, R_c, for $n = 2^k$ and constant integer $c = 2^j k$, where $0 \leq k \leq K$ and $0 \leq j \leq K$.

Fieldstream-movement Operations. Fieldstream-movement operations are operations which restructure n-bit Fieldstreams, producing an n-bit Fieldstream whose length is determined at runtime. Fieldstream-movement operations, denoted $R_n = F_{n,m}(S_n, I)$, define the parallel movement of the fields of an n-bit Fieldstream R_n of length L in accordance with an index stream I for $n = 2^k$, $m = 2^j$, where $0 \leq k \leq j \leq K$.

Flow-parallel Operations. The flow-parallel operations form a set of bitstream operations for parallel scanning that leverage the carry-generation properties of bitwise addition and subtraction to propagate information across field boundaries [15]. The s2^k flow-parallel bitstream operations, denoted $R = F(S)$, $R = F(S, T)$, or $R = F(S, c)$ are defined using bitwise logic and arithmetic operations on bitstreams S, T of length L or an integer constant c. As previously mentioned, in s2^k, bitstreams are semantically equivalent to arbitrary length unsigned integers. Take the flow-parallel operation $R = Advance(S)$ as an example. This operation is defined as $R = Advance(S) = (S + S)$ mod 2^L, where $L = Length(S)$. [2] Appendix C.1 specifies the flow-parallel Fieldstream operations.

Figure 3.2 shows the basic bitwise data-parallel behaviour of the parallel scanning operations. A left triangle $\lhd$ indicates that Fieldstreams are read from right to left. That is, we use little-endian notation and follow the convention that the bit order is the same endianness as the byte order. The left column shows flow-parallel operations together with initialization operations and bitwise logic operations on bitstreams. The right column shows the effects of the statements. Line numbers

[2] In practical applications a carry bit generated at the final bitstream position can be used to indicate error conditions such as an expectation to match a character in lexical analysis with bitstreams. In addition, a sentinel value may be appended to each input stream. The addition of a sentinel value guarantees termination of bitstream scanning operations, indicates failure to terminate syntactic constructs, marks follows positions of matches in token scanning, or indicates positions that that can be reached in regular expression matching using the $MatchStar$ approach to regular expression matching [18].

are shown at the left hand side of each line. The figure heading shows a data stream that contains several spans of digits as well as non-digit characters indicated with hyphens. Line 1 marks with 1s the positions at which decimal digits occur. Line 2 shows the calculation of an initial set of marker bits that mark two initial scanning positions. Line 3 shows the result of the *ScanThru* operation which accepts bitstream arguments m_0 and d as input. The *ScanThru* operation is defined as $R = ScanThru(M, C) = ((M+C) \wedge (\neg C)) \mod 2^L$, where $L = Length(M) = Length(C)$. This operation moves each marker bit of m_0 through the corresponding spans of digits of d to the positions immediately following each span. The result of bitstream addition advances marker bits through variable-length digit spans of d. Additional digits spans, *e.g.* 123, that are not involved in the operation are removed by masking off any bits from the digit bitstream using the bitwise and complement of d. These positions can never be marker positions resulting from a *ScanThru* operation.

	8-bit ASCII data $\lhd$	-123----------1456---2493711----
1	$d = [0-9]$	.111.....1111...1111111....
2	$m_0 = d \wedge Advance(\neg d)$	.....1.....1....
3	$m_1 = ScanThru(m_0, d)$	.....1.....1.....

Figure 3.2: Flow-parallel operations.

For a more complete discussion of parallel bitstream techniques we refer the reader to [15] and [18].

Length-preserving Operations. A Fieldstream is a sequence of n-bit values of length L. A *length-preserving* operation is a Fieldstream operation that produces a Fieldstream of length equal to the length of each Fieldstream argument. For example, the operation $R_{n/2} = PackHi_n(S_n)$ is length-preserving. This operation generates a $n/2$-bit Fieldstream with elements in one-to-one correspondence to the n-bit Fieldstream argument, S_n. Although the size of $R_{n/2}$ does not equal the size of S_n, the length of $R_{n/2}$ equals the length of S_n where the size of an n-bit Fieldstream of length L is $L * n$.

Non Length-preserving Operations. A *non length-preserving* operation is a Fieldstream operation that does not guarantee the generation of a Fieldstream of length equal to the lengths of each Fieldstream argument. Non length-preserving operations are restricted to a small set of Fieldstream operations that include s2^k parallel gather and casting operations.

3.4.3 Stream Indexing

Collective array languages can be characterized by the mechanisms they provide to reference array elements [70]. In the first array language, APL, all array operations are collective operations applied to an entire array, *i.e.*, array elements cannot be referenced. In Fortran 90, absolute integer indices are used to define array slices. In C*, indices relative to stream elements can be defined. In general, indexing schemes that force the programmer to consider low-level index calculations are tedious and error-prone. Further, low-level indexing can introduce sequential dependencies and additional complexity in determining overlapping regions.

In $s2^k$, Fieldstream elements are collectively indexed with *parallel index-streams*. A parallel index-stream is a 1-bit Fieldstream generated by $s2^k$ operations that is used to reference the elements of a Fieldstream. Parallel index stream provide a number of benefits. First, by construction, parallel index-streams avoid programmer errors due to index calculations that are inherent to programming languages which use explicit integer-indexing. Second, parallel index-streams can eliminate runtime bounds-checking in the indexing of Fieldstreams. Third, as arguments to $s2^k$ Fieldstream-movement operations, index streams guarantee deterministic and conflict-free behaviour since the elements of parallel index and source streams are in one-to-one correspondence. Finally, parallel index-streams raise the level of abstraction and simplify indexing. $s2^k$ provides no other mechanism to index Fieldstream elements.

3.4.4 Scalar Promotion

Scalars can be promoted to Fieldstreams in $s2^k$ taking the form (field width and Fieldstream length) required by the context in which they are used.

3.4.5 Classification of Errors

The $s2^k$ language definition specifies rules that must not be violated by $s2^k$ programs. Errors are classified as follows.

1. Static Errors. Errors that must be detected statically by an $s2^k$ compiler.

2. Dynamic Errors. Errors that must be detected dynamically by an $s2^k$ runtime environment.

3. Erroneous execution. An $s2^k$ program execution is erroneous when it generates an error that is not required to be detected statically or dynamically. The effect of erroneous execution is unpredictable. Each $s2^k$ language description statement is understood to have ``,unless execution is erroneous'' after it [28].

3.5 Language Structure

In this section we outline s2^k language features and present simple s2^k programming examples. The syntax of s2^k was designed to resemble the syntax of the C-family of languages since C and C++ are targets of the s2^k compiler framework. Like C-family languages, s2^k statements are delimited with semicolons and compound statements are defined using curly brackets. One departure from C-family language syntax can be seen in the use of a left-to-right keyword-based approach to identifying program statements. For example, variable declarations are introduced with s2^k keywords. Similarly, function declarations are indicated by the keyword `function`. This design decision makes s2^k programs easier to read and simplifies parsing.

We organize the following description of s2^k into two parts, graph-level constructs and filter-level constructs. While this format would not be appropriate for a programming language reference manual it serves to organize the description of language features.

3.5.1 Graph-level Constructs

s2^k graph-level constructs include: Fieldstream structures, functions, filters, and graphs. The following sections describe each feature in detail.

Fieldstream Structures. An s2^k Fieldstream structure is a non-nestable record type. The following example defines and declares an s2^k Fieldstream structure. The identifier of the structure is `Lex`. `Lex` contains four `1`-bit Fieldstream members, `n`, `e`, `d`, and `l`. The declared instance of type `struct Lex` is named `lex`.

```
struct Lex
{
  s<1> n;
  s<1> e;
  s<1> d;
  s<1> l;
}
struct Lex lex;
```

Fieldstream Functions. An s2^k Fieldstream function is a side-effect free method that computes a single Fieldstream value. Unlike Fieldstream filters which require the specification of parameter modes, function parameters are, by default, *in* parameters that do not require mode specification. The following example shows the definition of a Fieldstream function that returns the bit counts of the 16-bit fields of a Fieldstream. As described in Section 3.5.2, the declaration of Fieldstreams of

symbolic length are shown using an symbolic identifier, `L`,`M`,`N` and `O` which follows the Fieldstream width parameter.

```
function s<16> PopCount(s<1> s)
{
  s<2,L>  t0  = AddHiLo<2>(BitCast<1>(s,2));
  s<4,M>  t1  = AddHiLo<4>(BitCast<2>(t0,4));
  s<8,N>  t2  = AddHiLo<8>(BitCast<4>(t1,8));
  s<16,O> t3  = AddHiLo<16>(BitCast<8>(t2,16));
  return t3;
}
```

Fieldstream Filters. Fieldstream filters transform sets of input Fieldstreams into sets of output Fieldstreams. As described in Section 3.3, s2^k filters define independent and self-contained stream transformation operations. Filters cannot be nested nor contain calls to other filters.

Fieldstream filter definitions specify input parameters and output parameters using parameter modes. Parameters modes are limited to *in* and *out*, where `in` indicates read-only parameters and `out` indicates write-only parameters. The parameters of a filter are available to all statements within a filter. Section 3.2 shows example s2^k filter definitions.

Fieldstream Graph. The s2^k Fieldstream graph is the main program unit of an s2^k program. The vertices of a Fieldstream graph are Fieldstream filters and the edges are Fieldstream structures. Fieldstream filters describe data transformation operations on Fieldstreams. Fieldstream filter arguments define inter-filter connections. Fieldstream graphs are not composable and must not contain cycles.

s2^k provides a syntax for defining Fieldstream graphs with the s2^k `graph` construct. When defining a graph the programmer first defines Fieldstream structures and then connects individual Fieldstream filters. Filter connections are defined through the assignment of Fieldstream structures variables to filter instances.

The simplest graph is a pipeline. An s2^k pipeline contains one or more Fieldstream filters connected through Fieldstream structures. The source filter reads a set of input Fieldstreams and the sink filter writes a set of output Fieldstreams. The following example shows a simple two-stage pipeline with an initial filter, `src`, and a terminal filter, `snk`, together with a Fieldstream structure named `lex` that connects the `src` filter to the `snk` filter.

```
struct Lex { ... }
filter Src(out: struct Lex lex) { ... }
filter Snk(in: struct Lex lex) { ... }

graph
{
  struct Lex lex;
  Src(lex);
  Snk(lex);
}
```

3.5.2 Filter-level Constructs

Type System. s2^k is statically typed. s2^k scalar types include: integer types, floating point types, and string literals. Aggregate types consist of the n-bit Fieldstream types. s2^k does not provide pointer or reference types, nor a Boolean type for relational operations.

Scalar Types. s2^k scalar[3] types are compile-time value-types. The role of scalar types in s2^k is limited. The s2^k programmer uses scalar types to initialize configuration variables, as arguments to s2^{k}perations, as parameters to Fieldstream types, or as iteration bounds on s2^k count-controlled loops.

Aggregate Types. The primary s2^k data type is the n-bit Fieldstream type. The n-bit Fieldstream type is a parametric polymorphic type that represents an ordered sequences of n-bit values. The field width of each Fieldstream type declaration is specified using an integer constant following the keyword `stream` or `s`, and delimited by left- and right-angle brackets. This syntax is similar to C++ template specialization syntax. The following s2^k example show a series of n-bit Fieldstream variable declaration assignments.

```
s<4> s_7654     = PackHi<8>(s_76543210);
s<2> s_76       = PackHi<4>(s_7654);
s<1> s_7        = PackHi<2>(s_76);
```

Since the length of a source data stream is commonly unknown to the programmer s2^k supports the declaration of Fieldstreams of symbolic length. The symbolic length of a Fieldstream type is specified with with an identifier that follows the Fieldstream field width parameter. In the following

[3]The term scalar type was intentionally chosen over the term primitive type. Scalar types are commonly contrasted with aggregates such as arrays, lists, or streams, whereas primitives are typically contrasted with reference types.

example, 8-bit fields are packed to produce 4-bit fields. Next, 4-bit fields are cast to 2-bit fields which are then cast to 1-bit fields. Here, each non length-preserving operation produces a Fieldstream of length increases by a factor of two. Symbolic lengths capture this information.

```
s<4,L> s4_7654    = PackHi<8>(s_76543210);
s<2,M> s2_7654    = BitCast<4>(s4_7654,2);
s<1,N> s1_7654    = BitCast<2>(s2_7654,1);
```

Symbolic length parameters can be omitted for Fieldstreams derived by length-preserving operations since the derived Fieldstreams are of equal length. Non length-preserving operations are required to introduce symbolic length parameters.

Type Conversions and Casts. With the exception of the promotion of scalar constants to Fieldstreams, implicit type conversions are not performed in s2^k. The s2^k operations $ZExt_n$ and $SExt_n$, provide Fieldstream type conversion support. The $ZExt_n$ and $SExt_n$ operations widen Fieldstream fields by a factor of two through zero-extension and sign-bit extension respectively. In addition, the $PachHi_n$ and $PackLo_n$ operations narrow Fieldstream fields by a factor of two. The $BitCast_n$ operation reinterprets the field width of a Fieldstream. Storage requirements change as a result of conversion operations but do not change with casting operations.

Expressions. s2^k expressions include scalar expressions and Fieldstream expressions.

Scalar Expressions. s2^k scalar expressions are statically evaluated constant expressions. Constant expressions are useful in a number of scenarios such as in the specification of countable loops and in the evaluation of function arguments.

Fieldstream Expressions. s2^k expressions are collective operations on Fieldstreams. With the exception of the bitwise operators, provided as a programmer convenience, the s2^k Fieldstream expressions are restricted to the s2^k operations defined in Appendix C.

Assignment Statements. The semantics of an s2^k assignment statement is that the right-hand side expression of the statement is evaluated and then assigned to the left-hand side variable. Assignment statements have copy-on-assignment semantics. As in imperative languages, assignment statements are sequentially executed with each s2^k statement completing prior to the execution of the next.

Control-Flow Statements. s2^k control-flow statements include: parallel *if*, parallel *while*, and sequential *foreach* statements. In each case, control-flow conditions are restricted to 1-bit Fieldstream (bitstream) expressions such that, if the condition is non-zero then the statements of the body are executed, otherwise execution of the control-flow statement is complete. We next describe the s2^k control-flow statements. We defer our description of the translation of s2^k control-flow statements to block-at-a-time equivalents to Chapter 4.

Parallel If. The parallel if statement has the syntax *if*$(E)\{S^*\}$, where E is a bitstream expression and S^* is a sequence of s2^k statements. The purpose of the parallel if statement is to conditionally execute S^* by computing the consequences of marker bits (1 bits) in E. If there are no marker bits in E, the execution of the statements S^* is skipped and control continues to the statement following the parallel if statement.

In general, S^* may be an expensive computation that can be avoided for a region R of E that consists of a run of consecutive 0 bits. In such a case, there are no field-parallel consequences of 1 bits in R since there are none in E. If the compiler or run-time system can ensure that there are no flow-parallel consequences of operations in S^* that extend from positions prior to R into positions within R itself, then execution of S^* may be skipped for all of region R. In such a case, region R is termed skippable.

Whether a compiler or run-time system skips execution for a particular skippable region R is implementation-dependent. A block-oriented implementation may choose to skip only regions that correspond to full input blocks. In the event that the execution of S^* for such a skippable region R produces a different result than if execution is skipped for the region R, then program execution is erroneous.

Figure 3.3 presents an example of conditional scanning in s2^k, the scanning of decimal digit spans. Following the conventions of Figure 3.2, Line 1 marks with 1s the positions at which decimal digits occur. Next, lines 2-4 show the calculation of marker bits that mark initial scanning positions in m_0. Line 5 shows a parallel if statement with m_0 as the value of the if condition. Line 6 shows the conditional scanning of digits. In this simple example, Figure 3.3 shows that a single statement may be avoided. In general, zero or more operations may be skipped. In addition, as described in Chapter 4, parallel if statements are translated to block-at-a-time if statements. This limits the number of stream positions simultaneously evaluated by an if condition to block-width regions. This increases the effectiveness of the parallel if statement as an optimization technique.

8-bit ASCII data $\lhd$	...	`--------------1456---2493711----`

1	$d = [0-9]$	...	`.....1111...1111111....`
2	$t_0 = \neg d$	...	`11111111111111....111.....1111`
3	$t_1 = Advance(t_0)$	...	`1111111111111....111.....1111.`
4	$m_0 = d \wedge t_1$	...	`.....1.....1....`
5	$if(m_0)$	...	`.....1.....1....`
6	$\quad m_1 = ScanThru(m_0, d)$	...	`.....1.....1.....`

Figure 3.3: Conditional Fieldstream operations.

Parallel While. The parallel while statement has the syntax $while(E)\{S^*\}$, where E is a bitstream expression and S^* is a sequence of s2^k statements. The purpose of a parallel while statement is to repeatedly execute the statements S^* until there are no marker bits in E. For each iteration of the parallel-while statement, if there are no marker bits in E, then execution of the statements S^* is skipped and control continues to the statement following the while statement.

As is the case for parallel if statements, S^* may be an expensive computation that can be avoided for a region R of E that consists of a run of consecutive 0 bits. Again, if there are no field-parallel consequences of 1 bits in R there are none in E. If the compiler or run-time system can ensure that there are no flow-parallel consequences of operations in S^* that extend from positions prior to R into positions within R itself, then execution of S^* may be skipped for all of region R.

Figure 3.4 shows iterative logic for the sequential scanning of digits using the parallel while statement. Line 1 marks with 1s the positions of decimal digits. Line 2 marks the beginning of the first digit span with the marker stream, m_0. Lines 3-5 iteratively move m_0 marker bits through the remaining digit spans using the bitstream operations $ScanThru$ and $ScanTo$. The loop terminates when all bit positions in m_0 are zero.

```
1  d = [0 − 9]
2  m0 = ScanToFirst(d)
3  while(m0)
4    m1 = ScanThru(m0, d)
5    m0 = ScanTo(m1, d)
```

Figure 3.4: Iterative Fieldstream operations.

To illustrate the movement of marker bits, Figure 3.5 shows unrolled iterations of the statements of Figure 3.4. An underscored value indicates loop iteration values. For example, m_{0_1} indicates the first loop iteration value of m_0 and m_{0_2} indicates the second.

	8-bit ASCII data $\lhd$	...	----------1456---2493711----
1	$d = [0-9]$	...	.....1111...1111111....
2	$m_0 = ScanToFirst(d)$	...	.....1....
3	$m_{1_1} = ScanThru(m_0, d)$	...	.....1.....
4	$m_{0_1} = ScanTo(m_{1_1}, d)$	...	.....1.....
5	$m_{1_2} = ScanThru(m_{0_1}, d)$	...	.....1.....
6	$m_{0_2} = ScanTo(m_{1_2}, d)$	...	.....

Figure 3.5: Unrolled Fieldstream operations of Figure 3.4.

Sequential Foreach. The sequential-foreach statement has the syntax *foreach* I *in* $E\{S^*\}$, where I is a bitstream, E is a bitstream expression and S^* is a sequence of s2^k statements. In contrast to the parallel-while statement that repeatedly executes S^* for multiple markers in E, the purpose of the sequential-foreach statement is to sequentially execute S^* once for each marker in E.

The sequential-foreach statement is evaluated as follows.

1. The foreach condition expression E is evaluated.

2. If E evaluates to zero the sequence of statements S^* are not executed and control continues to the statement following the foreach statement.

3. If E evaluates to non-zero, the value of I is assigned $ScanToFirst(E)$, the value of E is assigned $E \wedge \neg I$, the statements S^* are executed, and control continues to the evaluation of the foreach condition E. The conditions for optimized and erroneous executions of sequential-foreach statements are analogous to those described for the parallel-if and parallel-while statements.

Field-Width Metaprogramming Many streaming text processing algorithms can be expressed with inductive-doubling operations applied to n-bit Fieldstream fields at successive power-of-two field widths. As previously described, inductive-doubling refers to a general property of stream processing algorithms that systematically doubles or halves bit-field widths at each step. Commonly, the number of ouput Fieldstreams also doubles at each step, or equivalently, the number of input Fieldstream halves. As defined in s2^k, inductive-doubling algorithms produce a final set of output Fieldstreams in a standard order called the for..in order. This order corresponds to a depth-first and left-to-right traversal of a recursion tree of inductive-doubling operations.

For instance, determining the number of one bits contained within the 16-bit fields of an input Fieldstream can be expressed by following an inductive-doubling approach. At each step, n-bit Fieldstream field counts are determined by adding the high $n/2$-bits of each n-bit field to the low

$n/2$- bits. Beginning at $n = 2$ and doubling the field width at each step, the final 16-bit Fieldstream population counts can be determined in four steps. Here, the number of Fieldstreams produced by each step remains constant. However, as described in Section 3.7.2, an inductive-doubling approach to 8-bit Fieldstream transposition begins with a single 8-bit Fieldstreams input and produces a set of eight 1-bit Fieldstreams. First, a single 8-bit Fieldstream produces two 4-bit Fieldstreams. Next, a pair of 4-bit Fieldstreams produce four 2-bit Fieldstreams, and finally, four 2-bit Fieldstreams produce eight 1-bit fieldstreams. 8-bit inverse Fieldstream transposition inverts the process, beginning with eight 1-bit Fieldstreams, and producing a single 8-bit Fieldstream.

For..In. The purpose of the for..in expression is to provide a concise means to express inductive-doubling algorithms. The for..in expression has the syntax *for* P *in* R *by*S $\{E^*\}$. The for..in syntax consists of four primary components: P, R, S, and E^*. Together these components define the set of Fieldstreams that the for..in expression represents, as follows. P is the for..in index parameter. P is a parameter rather than a variable since for..in expressions are statically unrolled by the compiler. R is a field-width range expression of the form $l..u$. A range expression represents a sequence of non-negative integer values beginning with the value of l and ending with the value of u. S is a stride expression, a constant integer expression that defines the distance between any two adjacent members of a range sequence. E^* is the body expression of the for..in expression defined as a sequence of recursively evaluated s2^k expressions.

Evaluation of E^* begin with $P = u$. The for..in expression body is recursively unrolled for each value in the range expression and for each expression in the for..in body. Evaluation of E^* terminates on evaluation of E^* with $P = u$. In general, for..in expressions are restricted to iteration over range expressions with an optional stride expression where the range expression and stride are constant integer expressions. The for..in expression returns a Fieldstream or Fieldstream structure result with members assigned in standard order called the for..in order.

The following s2^k code sequence shows an the for..in expression for the population count problem on 2^k-bit fields, *i.e.*, determining the number of one bits within 2^k-bit Fieldstream elements for $k = 32$. The for..in implementation proceeds in five inductive-doubling steps, evaluating the body expression for k at 2,4,8,16, and 32, At each step, the Fieldstream operation `AddHiLo<k>(t)` adds the high $k/2$-bits to the low $k/2$-bits, producing k-bit Fieldstream population counts. The final value of $k = 32$ is returned as a 32-bit Fieldstream of population counts. Intermediate inductive results are not visible to the s2^k programmer.

```
s<2>  t;
s<32> r = for k in 2..32 by 2**k { AddHiLo<k>(t); };
```

The following s2^k code sequence shows the equivalent unrolled sequence of operations.

```
s<16> r;
s<2>  t0 = AddHiLo<2>(BitCast<1>(t,2));
s<4>  t1 = AddHiLo<4>(BitCast<2>(t0,4));
s<8>  t2 = AddHiLo<8>(BitCast<4>(t1,8));
s<16> t3 = AddHiLo<16>(BitCast<8>(t2,16));
r = t3;
```

The following $s2^k$ code sequence shows the for..in expression for 8-bit Fieldstream transposition. This operation returns the result set as a Fieldstream structure of 1-bit Fieldstreams. Program 3.4 shows the equivalent expanded inductive-halving operations in the context of an $s2^k$ program.

```
struct Basis {
  s<1> b_7;
  s<1> b_6;
  s<1> b_5;
  s<1> b_4;
  s<1> b_3;
  s<1> b_2;
  s<1> b_1;
  s<1> b_0;
}
s<8>  t;
struct Basis b = for k in 8..1 by 2**k { PackHi<k>(t); PackLo<k>(t); };
```

Built-in Libraries. In this section we describe the $s2^k$ library categories. In addition to the field-parallel and flow-parallel abstract operations described in Section 3.4.2, $s2^k$ provides libraries for buffer management as well as standard $s2^k$ operations such as transposition and the definition of character classes. We describe the individual operations in Appendix D.

1. Buffer Management Library (io). A collection of $s2^k$ operations for the expression of data access operations between an $s2^k$-program and a host-program.

2. Standard Library (std). A collection of operations which express standard algorithms for streaming text processing using parallel bitstreams such as Fieldstream transposition and inverse-transposition [14].

3. Regular Expression Library (regex) A collection of regular expression operations for streaming text search. Regular expression library operations support a subset of the Unicode Level 1 requirements set out in Unicode Technical Standard #18 [31]. At present, $s2^k$ does not

support back references or capture groups. In general, regular expression search returns a parallel bitstream that marks the longest-leftmost follow positions of each match in a given input stream. Similarly, character classes operations return a parallel bitstream that marks the positions of character classes elements in a given input stream.

4. Flow-parallel Library (flow). A collection of flow-parallel operations for the expression of flow-parallel operations. The Flow-parallel Library functions implement the operations described previously.

5. Field-parallel Libraries (field). A collection of libraries for the expression of inductive-doubling algorithm on parallel Fieldstreams. The field-parallel Library functions implement the operations categories described previously. The field-parallel libraries consist of four sub-libraries.

   (a) Vertical Fieldstream Library (vstrm). The Vertical library defines n-bit collective operations on vertically aligned Fieldstream elements. Vertical library operations accept n-bit Fieldstreams and return n-bit Fieldstreams.

   (b) Horizontal-packing Fieldstream Library (hstrm). The Horizontal-packing library defines n-bit collective operations which operate on the high or low $n/2$-bits of n-bit Fieldstream elements. Horizontal library operations accept n-bit Fieldstreams and return $n/2$-bit Fieldstreams.

   (c) Expansion Fieldstream Library (estrm). The Expansion Fieldstream library defines n-bit collective operations which expand n-bit Fieldstreams. Expansion library operations accept n-bit Fieldstreams and return $2n$-bit Fieldstreams.

   (d) Field-movement Fieldstream Library (mstrm). The Field-movement library defines collective operations on m-bit subfields of n-bit Fieldstreams elements which permute, broadcast, and set subfield element values. Field-movement library operations accept n-bit Fieldstreams and return n-bit Fieldstreams.

A complete listing of the s2^k intrinsics is provided in Appendix D.2.

6. Fieldstream-movement Library (fstrm). A collection of s2^k operations for the expression of parallel Fieldstream restructuring operations, which include parallel scatter and parallel gather operations. Parallel scatter operations perform Fieldstream indexed reads from a n-bit Fieldstream producing an m-bit Fieldstream. Conversely, parallel gather operations perform indexed writes from a m-bit Fieldstream producing an n-bit Fieldstream.

3.6 s2^k Programming Idioms

This section introduces common s2^k programming idioms. We describe each idiom and then illustrate these idioms together with a simple programming example.

s2^k programming styles include: *flow-parallel programming* methods, *unaligned Fieldstream programming* methods, and *field-aligned Fieldstream programming* methods. These styles can be used in isolation, but they are more often used in combination. Typically, the s2^k programmer uses flow-parallel programming methods to mark data item positions or extents, unaligned Fieldstream programming methods to transform Fieldstreams data independently of individual data item alignments, and the field-aligned programming methods to organize data for data-parallel inductive-doubling computations.

3.6.1 Flow-parallel Programming

Parallel Bitstreams. A core concept in the s2^k programming model is the generation of parallel bitstreams, *i.e.*, bitstreams in one-to-one correspondence with the data elements of an input stream. Typically, an input stream is a byte stream encoded with a particular character encoding such as an 8-bit ASCII encoding or UTF-8, although wider encodings such as UTF-16 and UTF-32 are supported equivalently. In the case of an 8-bit encoding, a byte stream is transposed into eight parallel bitstreams, such that bitstream i comprises the ith bit of each byte. Analogously, the transposition of 16-bit and 32-bit encoded data streams produces sixteen and thirty-two parallel bitstreams respectively.

Character-class Streams. Given a set of basis bitstreams, many additional parallel bitstreams can be calculated, such as *character-class* streams, *i.e.*, bitstreams such that each bit i of the stream specifies whether character i of the input stream is a member of the character class or not [68]. The derivation of character-class streams is a key step in many text processing methods such as lexical analysis.

Bitwise Data-Parallel Scanning and Matching. Given a set of character-class streams, flow-parallel operations can be used to mark the positions of variable-length data items. The marking of data items involves the concept of marker streams, *i.e.*, 1-bit Fieldstreams that mark the positions of matches throughout the parallel matching process.

3.6.2 Unaligned Fieldstream Programming

Unaligned Fieldstream programming methods use Fieldstream operations to transform Fieldstream elements in parallel independently of individual element alignments. Examples include,

masking operations to mark stream regions, and restructuring operations to organize stream data prior to additional processing.

3.6.3 Field-aligned Fieldstream Programming

Field-aligned Fieldstream programming methods require that stream elements are aligned to known offsets. A number of parallel streaming text processing algorithms such as parallel reductions and conversions require aligned stream elements [14, 11, 48]. For example, Section 3.7 describes 8-bit ASCII stream to 16-bit binary integer stream conversion in which ASCII decimal digit sequences are first gathered and aligned prior to parallel conversion.

3.6.4 An Illustrative Example: Binary-integer Stream Conversion

Without describing the details for now, Figure 3.6 illustrates $s2^k$ programming styles applied to the problem of 8-bit ASCII-integer stream to 16-bit binary-integer stream conversion, more specifically, conversion of XML tag-delimited and comma-separated ASCII digits to binary integers. Figure 3.6 follows our previously described conventions. As before, Fieldstreams are shown right-to-left and ellipses indicate elided streams. Label subscripts indicate field width. Stream subscripts indicate bit offsets, *e.g.*, the rightmost angle bracket character '>' is presented at bit offset 80. With the exception of the initial XML data stream, data values are expressed in hexadecimal.

Row 1 of Figure 3.6 shows an 8-bit XML input stream. Row 2 shows the corresponding XML values expressed in hexadecimal. Rows 3 to 5 illustrate flow-parallel methods that mark XML character data regions. Row 3 shows parallel marker positions, $M1_1$, at an intermediate point in the parallel scanning process, *i.e.*, markers at the final position of XML closing tags, <. $M2_1$ shows marker bits at the start positions of corresponding XML tag opening angle brackets obtained from $M1_1$ by scanning to the next opening angle bracket, >. Finally, $M3_1$ marks positions inclusively between $M1_1$ and $M2_1$ using bitwise subtraction, *i.e.*, $M3_1 = (M2_1 - M1_1) \wedge \neg M1_1$. Rows 6 to 8 illustrate unaligned Fieldstream methods, *i.e.*, Fieldstream operations applied independently of the stream offsets of individual stream elements. Row 6 produces the character-class stream $[0-9]$ and then masks this value with M_3 to produce $D1_1$, where $D1_1 = [0-9] \wedge M_3$. Row 7 produces an 8-bit mask stream, $D2_8$ from $D1_1$. Row 8 applies the 8-bit mask value to the 8-bit XML input stream producing $D3_8$. Row 9 forms the 4-bit Fieldstream from $D4_4$ by packing the low 4-bits of each field of $D3_8$ into $D4_4$. Following the approach described in Section 3.7.1, rows 10 to 12 illustrate the use of field-aligned Fieldstream methods to perform parallel 8-bit ASCII stream to 16-bit binary-integer conversion. Here, the conversion algorithm uses a pair-wise inductive-doubling approach. The computation involves three stages. In each stage, pairs of stream values are combined using collective shifting, multiplication, and addition operations to produce a stream of intermediate results.

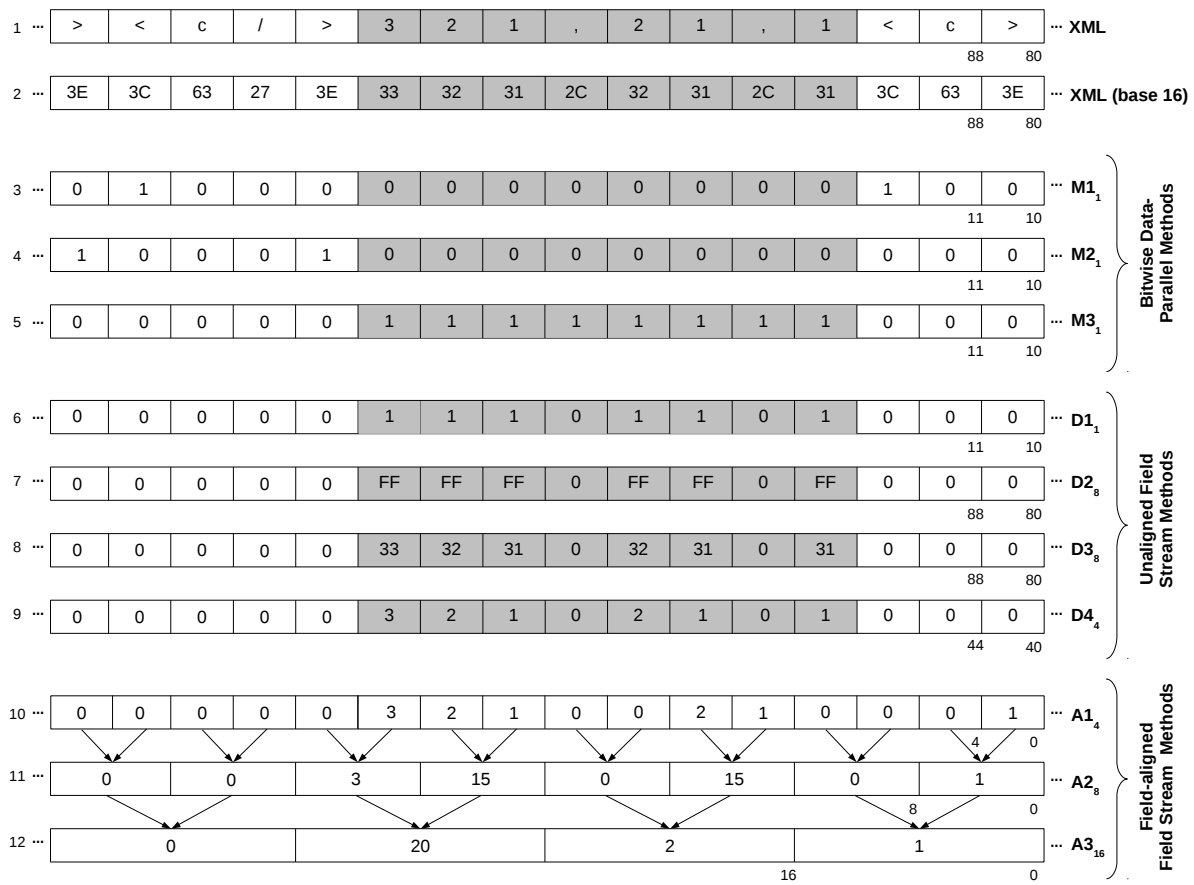

Figure 3.6: s2^k programming styles to convert an 8-bit ASCII stream data to 16-bit binary integer data.

A precondition of this strategy is that the high-order or low-order value of each digit span must be identified. In this example, the most significant value of each span is aligned to a 32-bit stream offset. The algorithm then proceeds through three inductive-doubling stages producing the 16-bit result stream, $A3_{16}$. As shown by this example, in s2^k, flow-parallel, unaligned Fieldstream, and field-aligned programming styles are used in combination to mark, gather, and compute the result.

3.7 8-bit ASCII Stream to 16-bit Binary-integer Stream Conversion

In this section, we illustrate the fundamental concepts of the s2^k programming language by describing the implementation of a simple streaming text processing problem in s2^k, *8-bit ASCII stream*

to 16-bit binary-integer stream conversion. We begin by defining the problem. We then contrast sequential and parallel approaches. We conclude with an example s2^k program that demonstrates a parallel solution to this conversion problem.

3.7.1 Problem Definition.

We define the 8-bit ASCII stream to 16-bit binary-integer stream conversion problem as follows. Given an 8-bit ASCII stream, S, convert each subsequence of consecutive ASCII decimal digits, d in S, into a 16-bit binary integer value. Return the sequence of converted 16-bit binary integer values in the order of the subsequences of d in S. We assume that individual ASCII decimal digit sequences can be variable-length and located at arbitrary stream offsets. We also assume that decimal digits are unsigned, *i.e.*, not prefixed with a sign character, and that the conversion of digit sequences does not result in overflow.

A Sequential Approach. A number of methods for the conversion of ASCII streams to binary integers are known. In general, conversion methods must identify the extents of ASCII decimal digit sequences, subtract the integer value 48 from each decimal digit to obtain the 4-bit binary integer values, multiply the 4-bit value by increasing powers-of-ten to form an intermediate products, and sum intermediate products to form the final result.

This computation can be easily expressed in most sequential programming languages. An intuitive approach is to initialize a variable to zero and then iteratively convert 4-bit binary integer values in sequence. Figure 3.1 shows the inner character-at-a-time loop of this computation expressed in a C-family language. A temporary variable `t` holds intermediate results and an index variable `i` references character buffer elements. The outer character-at-a-time loop is not shown.

```
int sum = 0;
int t = 0;
int d = 0;
char c = buffer[i];
while (c >= '0' && c <= '9')
{
    d = (c - 48);
    t = (t * 10) + d;
    c = buffer[++i];
}
sum = t;
```

Program 3.1: Byte-at-a-time 8-bit ASCII stream to 16-bit binary-integer stream conversion.

Figure 3.7 shows the sequential conversion of a single subsequence of 4-bit ASCII decimal integer values, $d_7d_6d_5d_4d_3d_2d_1d_0$, showing the order in which 4-bit values are combined to form intermediate results. In this figure, d_7 denotes the most significant digit, d_0 denotes the least significant

digit, and t_i denotes the ith intermediate result, where $t_1 = d_7, t_2 = (t_1 * 10) + d_6, ..., t_7 = (t_6 * 10) + d_0$, and t_7 denotes the final result. Figure 3.7 demonstrates a typical computation pattern of sequential text processing algorithms. Since dependencies exist between successive steps, solutions yield linear time complexities proportional to the length of the input sequence.

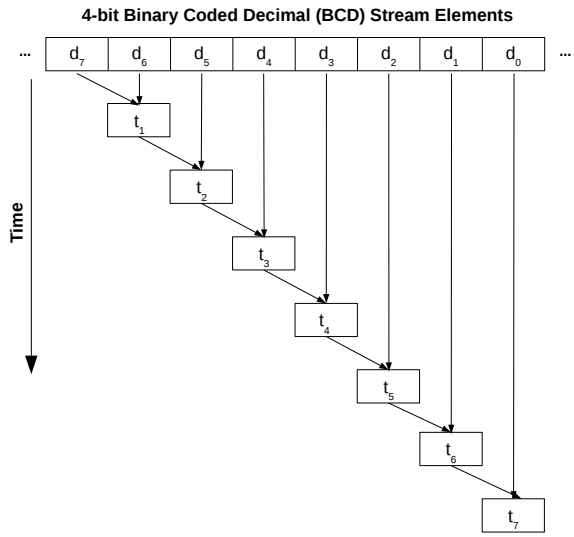

Figure 3.7: Sequential 4-bit BCD to 16-bit binary integer conversion.

A Parallel Approach. A parallel approach to the ASCII to integer conversion problem is less obvious. [11] proposes an inductive-doubling approach. The observation is that given a sequences of 4-bit binary integer values, $d_7d_6d_5d_4d_3d_2d_1d_0$, expression 3.15, can be factored into an equivalent form shown in 3.16 that is suitable for parallel implementation.

$$\begin{aligned}(10^7 * d_7) + (10^6 * d_6) + (10^5 * d_5) + (10^4 * d_4) + \\ (10^3 * d_3) + (10^2 * d_2) + (10^1 * d_1) + (10^0 * d_0)\end{aligned} \tag{3.15}$$

$$\begin{aligned}(10^4 * (10^2 * ((10^1 * d_7) + d_6)) + ((10^1 * d_5) + d_4))) + \\ (10^2 * ((10^1 * d_3) + d_2)) + ((10^1 * d_1) + d_0)\end{aligned} \tag{3.16}$$

As shown in 3.17, evaluating 3.16 outwardly from the innermost parentheses first, the computation can proceed in parallel as follows. Since, t_1, t_2, t_3, t_4, are mutually independent and do not depend on previous results, the computation of t_1, t_2, t_3, t_4 can proceed in parallel. Similarly, given the intermediate results of t_1, t_2, t_3, t_4 and since t_5, t_6 are mutually independent, the computation of t_5, t_6 can also proceed in parallel. Finally, given the intermediate results of t_5, t_6 the computation of

t_7 completes the conversion procedure.

$$
\begin{aligned}
t_1 &= (10^1 * d_7) + d_6) \\
t_2 &= (10^1 * d_5) + d_4) \\
t_3 &= (10^1 * d_3) + d_2) \\
t_4 &= (10^1 * d_1) + d_0) \\
t_5 &= (10^2 * t_1) + t_2) \\
t_6 &= (10^2 * t_3) + t_4) \\
t_7 &= (10^4 * t_5) + t_6)
\end{aligned}
\tag{3.17}
$$

Figure 3.8 illustrates the computation as a binary tree, where 4-bit input values are represented by leaf nodes, and intermediate results are represented by internal nodes. The value of each level i intermediate node is equal to the value of the left subtree of the node multiplied by $10^{2^{i-1}}$ and then added to the value of the right subtree. In Figure 3.8, the rightmost leaf is the value d_0 and the leaf level is index level 0. In this example, the leaf level is shown at the top. The computation is pairwise and proceeds as follows. Pairs of digits are combined by multiplying the high digit of each pair by 10 and adding this result to the low digit. Pairs of two-digit results are then combined by multiplying the high two-digit results by 100 and adding this result to the low two-digit value. Finally, pairs of four-digit results are combined by multiplying the high result by 10000 and adding this result to the low four-digit value.

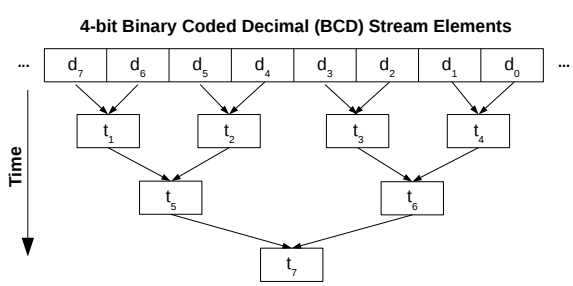

Figure 3.8: Parallel 4-bit BCD to 16-bit binary integer conversion.

Comparing Figure 3.7 and Figure 3.8 we observe that the serial and parallel solutions require the same number of intermediate operations. However, given sufficient parallel resources the parallel approach yields a solution with time complexity that is proportional to log n, demonstrating that a pair-wise inductive-doubling approach to the conversion problem has potential performance

advantages over the sequential approach. Furthermore, we observe that given sufficient parallel resources, the computation can be expressed on an entire stream of appropriately organized decimal digits. One approach is to use a *gather-and-compute* style of stream programming. In the gather phase, the low 4-bits of distributed 8-bit ASCII decimal digit sequence are packed into 32-bit fields with the least significant digit aligned to a 32-bit boundary and the remaining positions padded with zeroes. In the compute phase, the conversion computation proceeds as previously described.

3.7.2 $s2^k$ Implementation.

In this section, we present an annotated example of parallel $s2^k$ implementation of 8-bit ASCII stream to 16-bit binary-integer stream conversion (BCD conversion). We describe the $s2^k$ implementation first at the $s2^k$ stream-graph level to show the overall structure of the $s2^k$ program and then at the filter-level. The complete program listing for this case study is shown in Appendix E.1.

Stream-Graph Level Implementation. Program 3.2 shows the stream-graph level implementation. At the stream-graph level, the $s2^k$ programmer declares stream sets, declares stream filters, and specifies the stream graph.

```
/* Stream Structure Definitions */
struct Input { ...  }

struct Basis { ... }

struct Digits { ... }

struct Output { ... }

/* Stream Filter Definitions */
// Read 8-bit input stream.
filter Read(out: struct Input input) { ... }

// Transpose 8-bit input stream into a set of basis streams.
filter Transpose(in: struct Input input, out: struct Basis basis) { ... }

// Mark the positions of 8-bit ASCII decimal digits, [0-9].
filter Classify(in: struct Basis basis, out: struct Digits digits) { ... }

// Convert 8-bit ASCII decimal digits to 16-bit binary decimal integers.
filter Convert(in: struct Input input, in: struct Digits digits, out: Output output) { ... }

// Write 16-bit results stream.
filter Write(in: Output output) { ... }

/* Stream Graph */
graph
{
  // structure declarations
  struct Input input;
  struct Basis basis;
  struct Digits digits;
  struct Output output;

  // filter declarations
  filter Read read;
  filter Transpose transpose;
  filter Classify classify;
  filter Convert convert
  filter Write write;

  // filter pipeline
  read(input);
  transpose(input, basis);
  classify(basis, digits);
  convert(basis, digits, output);
  write(output);
}
```

Program 3.2: Stream-graph level program listing for BCD conversion.

Filter-level Implementation. In this section, we present the s2^k filter-level implementation. Figure 3.2 begins with the s2^k *Read* filter and goes on to describe the implementation of the remaining filters.

Read. Program 3.3 shows the s2^k *Read* filter. This filter illustrates some of our conventions for the presentation of s2^k programs. Source data and associated meta-information is presented in the figure heading. A left triangle $\triangleleft$ indicates that Fieldstream inputs are shown right to left. We use little-endian byte notation and big-endian bit notation. In essence, Fieldstream values can be interpreted as being *reflected* across a vertical axis. Following this convention, binary addition operations show carry propagation from right to left as is conventionally expressed in mathematics. Below the figure heading are sequences of s2^k program statements. Line numbers are shown at the left hand side of each line. s2^k comments beginning with '`//`' and terminating at the end of each line indicate the results of corresponding s2^k program statements. Vertical bars '`|`' delimit fieldstream elements. Additional whitespace is used to vertically align parallel Fieldstream elements.

Lines 3 and 4 show the s2^k `StreamBind` operation. `StreamBind` associates a host data buffer with an n-bit Fieldstream variable. Line 5 of Program 3.3 shows an 8-bit read operation. The value of this operation is assigned to the 8-bit Fieldstream variable, `s_76543210`. Here, we name variable to indicate bit positions. For example, the variable name `s76543210` indicates an 8-bit Fieldstream in which each stream element contains bit positions 7 to 0 of the 8-bit input data fields where bit-7 denotes the most significant bit and bit-0 denotes the least significant.

```
ASCII data ◁                                  ... |   3    |   b    |   a    |
ASCII data hexadecimal encoding               ... |   33   |   62   |   61   |
byte offset                                   ... |   2    |   1    |   0    |
bit offset                                    ... |76543210|76543210|76543210|
```

```
1  // Read
2
3  StreamBind<8>(input.s_76543210, ibuffer); //
4  StreamBind<16>(output.results, obuffer);  //
5  StreamRead<8>(input.s_76543210);          //    ... |00110011|01100010|01100001|
```

Program 3.3: s2^k Read filter for BCD conversion.

Transpose. Program 3.4 shows an inductive-halving approach to the implementation of serial byte to parallel bitstream transposition filter. Line 3 begins the inductive-halving procedure. In the first stage, an 8-bit Fieldstream is divided to form pairs of 4-bit Fieldstreams, `s7654` and `s3210` respectively. In the second stage, pairs of 4-bit Fieldstream are divided to form pairs of 2-bit Fieldstreams. Finally, in the third stage pairs of 2-bit Fieldstreams are divided to form pairs of 1-bit

Fieldstreams. At each stage, paired `PackHi<n>`, and `PackLo<n>` operations extract the high-order $n/2$ bits, and the low-order $n/2$ bits, respectively. Taken together, the computed 1-bit Fieldstreams, `s_7,s_6,...,s_0`, form a set of basis bitstreams. This set of basis bitstreams serves as the starting point for further parallel bitstream computations.

In Program 3.4, the bitstream `s_76543210` is parallel to `s_7654`, and by transitivity, parallel to `s_7`,`s_6`,`s_5`, and `s_4`, Taken together, the statements of Program 3.4 form a parallel region. Typically, this operation would be performed using the s2^k `Transpose<n>` operations defined in Appendix D.2.2. Here Fieldstream transposition illustrates Fieldstream programming methods.

```
ASCII data ◁                                  ... |   3    |   b    |   a    |
ASCII data hexadecimal encoding               ... |   33   |   62   |   61   |
byte offset                                   ... |   2    |   1    |   0    |
bit offset                                    ... |76543210|76543210|76543210|
```

```
1   // Transpose
2
3   // Stage 1: 8-bit to 4-bit fields.
4   s<4> s_7654
     = PackHi<8>(input.s_76543210);      //    ... |    0110|    0110|    0110|
5   s<4> s_3210
     = PackLo<8>(input.s_76543210);      //    ... |    0011|    0010|    0001|
6
7   // Stage 2: 4-bit to 2-bit fields.
8   s<2> s_76 = PackHi<4>(s_7654);       //    ... |      01|      01|      01|
9   s<2> s_54 = PackLo<4>(s_7654);       //    ... |      10|      10|      10|
10  s<2> s_32 = PackHi<4>(s_3120);       //    ... |      00|      00|      00|
11  s<2> s_10 = PackLo<4>(s_3120);       //    ... |      11|      10|      01|
12
13  // Stage 3: 2-bit to 1-bit fields.
14  basis.s_7 = PackHi<2>(s_76);         //    ... |       0|       0|       0|
15  basis.s_6 = PackLo<2>(s_76);         //    ... |       1|       1|       1|
16  basis.s_5 = PackHi<2>(s_54);         //    ... |       1|       1|       1|
17  basis.s_4 = PackLo<2>(s_54);         //    ... |       0|       0|       0|
18  basis.s_3 = PackHi<2>(s_32);         //    ... |       0|       0|       0|
19  basis.s_2 = PackLo<2>(s_32);         //    ... |       0|       0|       0|
20  basis.s_1 = PackHi<2>(s_10);         //    ... |       1|       1|       0|
21  basis.s_0 = PackLo<2>(s_10);         //    ... |       1|       0|       1|
```

Program 3.4: s2^k Transpose filter for BCD conversion.

Character-class Classification. The heading of Program 3.5 now shows the complete input data stream, ``ab31987c''. The second row of the heading shows the value of each ASCII byte expressed in hexadecimal. The remainder of Program 3.5 shows the derivation of the digits character-class streams, [0-9] using bitwise logic. Given the basis bitstreams, it is possible to derive character-class bitstreams using bitwise logic, *i.e.*, to generate bitstreams that mark the positions at which characters belonging to a particular class occur [15, 18]. Line 3 shows the result of the

character classification computation. Although only a single span of digits is presented, the method presented can be applied without modification to the processing of multiple ASCII digit sequences in parallel. Typically, this computation would be performed using the `CharClass` operation.

```
ASCII data ◁                                  | c| 7| 8| 9| 1| 3| b| a|
ASCII data hexadecimal encoding               |63|37|38|39|31|33|62|61|
byte offset                                   | 7| 6| 5| 4| 3| 2| 1| 0|
```

```
1  // Classify
2
3  s<1> temp1 = (basis.s_7 | basis.s_6);     //  | 1| 0| 0| 0| 0| 0| 1| 1|
4  s<1> temp2 = (basis.s_5 & basis.s_4);     //  | 0| 1| 1| 1| 1| 1| 0| 0|
5  s<1> temp3 = (temp2 &~ temp1);            //  | 0| 1| 1| 1| 1| 1| 0| 0|
6  s<1> temp4 = (basis.s_2 | basis.s_1);     //  | 1| 1| 0| 0| 1| 1| 1| 1|
7  s<1> temp5 = (basis.s_3 & temp4);         //  | 0| 0| 0| 0| 0| 0| 0| 0|
8  digits.digits = (temp3 &~ temp5);         //  | 0| 1| 1| 1| 1| 1| 0| 0|
```

Program 3.5: s2^k Classify filter for BCD conversion.

Convert. A common problem in parallel text processing is the organization of variable-length data items located at arbitrary stream offsets for parallel processing. For example, the heading of Program 3.6 shows the digits span of ``31987'', read right to left, is a span of 5 digits located at stream offset 2. In s2^k, the *parallel gather high-align*, `PGatherSpansHi<`n,m`>` operation gathers and aligns spans of n-bit source data fields to m-bit Fieldstream boundaries. This parallel-gather operation accepts two function parameters, the source Fieldstream field-width and the target Fieldstream field-width, as well as two arguments, a 1-bit index stream, a n-bit source data stream.

Program 3.6 shows the parallel gathering and alignment of 4-bit digit spans to 32-bit fields. Next, an inductive-doubling approach is used to reverse 4-bit digit spans. Alignment of digit spans to 32-bit boundaries is necessary prior to further parallel processing. The remainder of the 8-bit ASCII string to 16-bit binary decimal integer conversion procedure is shown in Program 3.7.

The s2^k filter implementation proceeds as follows. Line 3 forms the 4-bit Fieldstream containing the low 4-bits of each ASCII input byte. Lines 4 to 5 sign extend, `SExt`, the digits character-class stream to generate a 4-bit digits mask, `mask_4`, The 4-bit digits mask is then used to mask off 4-bit decimal digit values. Line 9 gathers and aligns the 4-bit decimal digits to 32-bit Fieldstream boundaries using the `PGatherSpansHi<4,32>(d,d_4)` operation. This operation uses the digits character-class stream, `d`, as a parallel-index stream to the 4-bit digits, `d_4`, and specifies that the target stream is the 32-bit Fieldstream, `g`. The parallel gather high-align operation illustrates the concepts of field alignment and stream indexing using parallel-index streams. In addition, parallel gather marks the starting point of a new parallel region thus demonstrating the concept of a behaviour is undefined. In s2^k, only Fieldstream-movement Operations can modify Fieldstream length dynamically. The

s2^k aligned parallel gather operation provides a number of benefits. First, aligned parallel gather is conceptually simple. Second, aligned parallel gather simplifies stream indexing and eliminates a potential source of programmer error. Third, aligned parallel gather unburdens the programmer from the low-level details of data alignment.

Lines 11 to 27 perform parallel 4-bit field reversal. 4-bit field reversal is not strictly required for 8-bit ASCII stream to 16-bit binary decimal integer conversion but simplifies the example. Line 10 begins the inductive-doubling procedure. In the first stage, 4-bit fields are reversed to form 8-bit fields. In the second stage, 8-bit fields are reversed to form 16-bit fields. Finally, in the third stage 16-bit streams are reversed to form 32-bit fields. At each stage, the n-bit fields of the stream are first re-interpreted as $2n$-bit fields using the `BitCast<`n`>(`s`,`$2n$`)` operation. Next, paired shift-logical left, `SLLI<`$2n$`>(s)`, and shift-logical right, `SLRI<`n`>(`s`)`, operations shift the high-order n bits right, and the low-order n bits left of each $2n$-bit field. Finally, a bitwise logical or operation, `Or<`n`>(`s`)` completes the procedure. The field reversals are not technically necessary but are shown for clarity, since the operands can be selected in reverse order in each multiply-add step.

```
ASCII data ◁                                          | c| 7| 8| 9| 1| 3| b| a|
ASCII data hexadecimal encoding                       |63|37|38|39|31|33|62|61|
byte offset                                           | 7| 6| 5| 4| 3| 2| 1| 0|
```

```
1   // Convert
2
3   s<4> s_3210  = PackLo<8>(input.s_76543210);     //  | 3| 7| 8| 9| 1| 3| 2| 1|
4   s<2> mask_2  = SExt<1>(digits.digits));         //  | 0| 3| 3| 3| 3| 3| 0| 0|
5   s<4> mask_4  = SExt<2>(mask_2);                 //  | 0| F| F| F| F| F| 0| 0|
6   s<4> d_4     = And<4>(s_3210,mask_4);           //  | 0| 7| 8| 9| 1| 3| 0| 0|
7
8   // 32-bit Parallel gather high-align.
9   s<4,G> g     = PGatherSpansHi<4,32>(digits.digits,
                                        d_4);       //  | 7| 8| 9| 1| 3| 0| 0| 0|
10
11  // Stage 1: Reverse 4-bit fields.
12  s<8,M> t_8   = BitCast<4>(g,8);                 //              |78|91|30|00|
13  s<8> hi_4    = SLLI<8>(t_8,4);                  //              |80|10|00|00|
14  s<8> lo_4    = SRLI<8>(t_8,4);                  //              |07|09|03|00|
15  s<8> rot_4   = Or<8>(hi_4,lo_4);                //              |87|19|03|00|
16
17  // Stage 2: Reverse 8-bit fields.
18  s<16,N> t_16 = BitCast<8>(rot_4,16);            //                |8719|0300|
19  s<16> hi_8   = SLLI<16>(t_16,8);                //                |1900|0000|
20  s<16> lo_8   = SRLI<16>(t_16,8);                //                |0087|0003|
21  s<16> rot_8  = Or<16>(hi_8,lo_8);               //                |1987|0003|
22
23  // Stage 3: Reverse 16-bit fields.
24  s<32,P> t_32 = BitCast<16>(rot_8,32);           //                 |19870003|
25  s<32> hi_16  = SLLI<32>(t_32,16);               //                 |00030000|
26  s<32> lo_16  = SRLI<32>(t_32,16);               //                 |00001987|
27  s<32> rot_16 = Or<32>(hi_8,lo_8);               //                 |00031987|
```

Program 3.6: Partial $s2^k$ Convert filter program listing which reverses the bytes of 16-bit fields in parallel for BCD conversion.

Program 3.7 completes the implementation of the *Convert* filter. Lines 28 to 52 perform the parallel conversion. Line 28 begins the parallel conversion using the inductive-doubling procedure described in section 3.7.1. As a final step, line 54 packs the converted value, *i.e.*, the low 16 bits, of each 32-bit field into a 16-bit Fieldstream.

```
28  // Stage 1: Convert 4-bit fields.
29  s<8,M> temp1    = BitCast<32>(rot_16,8);          //    |00|03|17|89|
30  s<8> d_hi_4     = SRLI<8>(4, temp1);              //    |00|00|01|08|
31  s<8> LoMask_8   = LoMask<8>();                    //    |0F|0F|0F|0F|
32  s<8> d_lo_4     = And<8>(rot_16, LoMask8);        //    |00|03|09|07|
33  s<8> c_10       = 10;                             //    |0A|0A|0A|0A|
34  s<8> t1_10      = Mul<8>(d_hi_4, c_10);           //    |00|00|0B|50|
35  s<8> t2_10      = Add<8>(t1_10, d_lo_4);          //    |00|03|13|57|
36
37  // Stage 2: Convert 8-bit fields.
38  s<16,N>  temp2   = BitCast<8>(t2_10,16);           //     |0003|1357|
39  s<16> d_hi_8    = SRLI<16>(8, temp2);             //     |0000|0013|
40  s<16> LoMask_16 = LoMask<16>();                   //     |00FF|00FF|
41  s<16> d_lo_8    = And<16>(d_hi_8, LoMask16);      //     |0003|0057|
42  s<16> c_100     = 100;                            //     |0064|0064|
43  s<16> t1_100    = Mul<16>(d_hi_8, c_100);         //     |0000|076C|
44  s<16> t2_100    = Add<16>(t1_100, d_lo_8);        //     |0003|0763|
45
46  // Stage 3: Convert 16-bit fields.
47  s<32,P>  temp3   = BitCast<16>(t2_100,32);         //      |00030763|
48  s<32> d_hi_16   = SRLI<32>(16, temp3);            //      |00000003|
49  s<32> LoMask_32 = LoMask<32>();                   //      |0000FFFF|
50  s<32> d_lo_16   = And<32>(d_hi_16, LoMask32);     //      |00000763|
51  s<32> c_10000   = 10000;                          //      |00002710|
52  s<32> t1_10000  = Mul<32>(d_hi_16, c_10000);      //      |00007530|
53  s<32> t2_10000  = Add<32>(t1_10000, d_lo_16);     //      |00007CF3|
54  s<16> results   = PackLo<32>(t2_10000);           //          |7CF3|
```

Program 3.7: Partial s2^k Convert filter program listing which transforms packed 4-bit binary integer sequences into 16-bit binary integer values for BCD conversion.

Write. Program 3.8 completes the program example. Line 3 shows a 16-bit write operation. This operation writes the value of the 16-bit fieldstream results to the host language environment.

ASCII data ◁	| c| 7| 8| 9| 1| 3| b| a|
ASCII data hexadecimal	|63|37|38|39|31|33|62|61|
byte offset	| 7| 6| 5| 4| 3| 2| 1| 0|

```
1  // Write
2
3  StreamWrite<16>(output.results); //                    |7CF3|
```

Program 3.8: s2^k Write filter for BCD conversion.

3.8 Discussion

As a starting point from which to assess the design of the s2^k programming language, we consider the following parallel programming language characteristics [20]. First, a parallel language

must insulate the programmer from the target hardware. Second, parallelism must be implicit in the semantics of the language. Third, determinacy must be guaranteed by the language, *i.e.*, a program that implements a deterministic algorithm must always yield the same programmer visible results for the same data.

The $s2^k$ language meets each of these requirements. First, $s2^k$ is a global-view language with portable language constructs that insulate the programmer from the the underlying hardware. Second, $s2^k$ language features operate over unbounded Fieldstreams, and use value semantics. For example, $s2^k$ functions use call-by-value semantics that preclude aliases and side effects. For these reasons, sufficient latent parallelism is implicit in the semantics of the $s2^k$ language. Third, $s2^k$ programs are determinate. The execution order of an $s2^k$ application is ordered by the program text as a series of Fieldstream transformation steps.

3.9 Chapter Summary

This chapter describes the design principles, programming model, and core features of the $s2^k$ language. Key novelties of the $s2^k$ language design include global-view streaming text processing language constructs, and the integration of stream programming abstractions flow-parallel and field-parallel methods.

Chapter 4

The b2^k Intermediate Language

This chapter describes the b2^k intermediate language. Section 4.1 introduces and motivates b2^k. Section 4.2 describes the b2^k block-at-a-time processing model. Section 4.3 presents the features of b2^k and Section 4.4 goes on to conclude this chapter.

4.1 Overview

The b2^k intermediate language is a very high-level IR (intermediate representation) for the expression of the block-at-a-time processing of s2^k Fieldstream operations. b2^k is designed with just the features necessary to support s2^k translation, yet is sufficiently abstract to accommodate a variety of ISAs. b2^k has an internal AST (abstract syntax tree) representation as well as a parsable textual format. Although the textual format of b2^k can be useful for debugging, programmers are not expected to hand-code b2^k. Rather, as described in Chapter 5, an s2^k compiler translates s2^k programs to b2^k IR. b2^k IR is then translated to a back-end target such as C, C++, or LLVM IR. Although the translation of s2^k to b2^k introduces an additional step, translating s2^k programs to b2^k IR introduces useful block-at-a-time abstractions and narrows the overall s2^k program translation distance. In general, b2^k modularizes compiler development, being (i) a straight-forward block-oriented language that can be translated to to a back-end target in a straightforward manner using standard techniques, and (ii) an IR as close to s2^k as possible so that the translation of s2^k to b2^k focuses on the issues of compiling arbitrary-length stream programs to block-at-a-time code.

4.2 Block-at-a-time Processing Model

The b2^k block-at-a-time processing model defines n-bit Fieldstreams operations as block-at-a-time N-bit register operations, where $n = 2^k$, $N = 2^K$ and $0 \leq k \leq K$. The model is defined by four elements.

1. The definition of field-parallel operations defined over the n-bit fields of N-bit SIMD registers.
2. The definition of flow-Parallel (*carry-generating*) N-bit register operations for parallel scanning.
3. The enumeration of sets of carry values in one-to-one correspondence with the carry-generating operations of b2^k kernel functions.
4. The runtime management of carry values in the block-at-a-time processing of Fieldstreams [60, 18].

4.2.1 Bitblock Operations.

b2^k operations are primarily N-bit register operations that produce N-bit values without hidden side effects. As in s2^k, b2^k scalar operations are restricted to constant expressions. Operation categories include the b2^k field-parallel and the b2^k flow-parallel operations.

Field-parallel Operations. The b2^k field-parallel operations are a set of SIMD operations designed to support data parallel computation on the n-bit fields of N-bit registers, where $n = 2^k$, $N = 2^K$ and $0 \leq k \leq K$. The b2^k field-parallel operations are organized into four categories. Definitions of the b2^k field-parallel operations correspond with the s2^k field-parallel in a straightforward manner. Accordingly, we refer the reader to Appendix C for field-parallel operation definitions.

Vertical Operations. Vertical operations accept zero or more SIMD values and return a SIMD value. The nullary through ternary b2^k Vertical operations are denoted $D_n = F_n(), \ldots, D_n = F_n(A_n, B_n, C_n)$ respectively. These operations define the simultaneous calculation of the n-bit fields of an N-bit SIMD register value in accordance with a corresponding function of the form $f_n(), \ldots, f_n(a_n, b_n, c_n)$ defined on n-bit values and that produces an n-bit values for $n = 2^k$ and $N = 2^K$, where $0 \leq k \leq K$.

Horizontal-packing Operations. Horizontal-packing operations accept two consecutive SIMD values and return a SIMD value. Horizontal-packing operations, denoted $D_{n/2} = F_n(A_n, B_n)$, are defined by the simultaneous calculation of the $n/2$-bit fields of $D_{n/2}$ in accordance with a binary function f_n defined on n-bit values and that produces an $n/2$-bit values for $n = 2^k$ and $N = 2^K$, where $1 \leq k \leq K$.

Expansion Operations. Expansion operations accept two parallel SIMD values and return a SIMD value. Expansion operations, denoted $D_{2n} = F_n(A_n, B_n)$, are defined by the simultaneous calculation of the n-bit fields of D_{2n} in accordance with a binary function f_n defined on n-bit values and that produces an $2n$-bit fields for $n = 2^k$ and $N = 2^K$, where $0 \leq k < K$.

Field-movement Operations. Field-movement accept two parallel SIMD values and return a SIMD value. Field-movement operations, denoted $D_n = F_{n,m}(A_n, B_n)$, or $D_n = F_{n,m}(A_n, c)$, are defined by the simultaneous calculation of the n-bit fields of D_n in accordance with a function f_n defined over the m-bit subfields of n-bit values, for constant integer c, $n = 2^k$, $m = 2^j$, and $N = 2^K$, where $0 \leq j \leq k \leq K$.

Flow-Parallel Operations. The b2^k flow-parallel operations are a collection of block-at-a-time operations for parallel scanning that leverage the carry generation properties of bitwise addition and subtraction to propagate carry value information across bit field boundaries. The b2^k flow-parallel operations, denoted $(r_n, co) = F_n(s_n, ci)$ or $(r_n, co) = F_n(s_n, t_n, ci)$ accept one or two n-bit values together with a 1-bit carry-in value, and generate an n-bit value and a 1-bit carry-out value, where $n = 2^k$ for some non-negative integers n and k. The b2^k flow-parallel operations are defined by a pair of functions f_n and g_n defined on n-bit values s_n, t_n, and carry-in value ci. The function f_n produces the n-bit result value, r_n. The function g_n produces the 1-bit carry-out value, co.

To define the flow-parallel operations, we first present some simple definitions on integers. An integer $W > 0$ can be uniquely expressed as the sum

$$W = W_{j-1} * b^{j-1} + \ldots + W_1 * b^1 + W_0$$

for some base b, where $b \geq 2$, $0 \leq W_i < b$, and $W_{j-1} \neq 0$. We denote the base b representation of W as $W = (W_{j-1} \ldots W_1 W_0)_b$, where W_{j-1} denotes the most significant digit of W. To simplify the definition of b2^k operations it is useful to add 0s to the beginning of this representation, *i.e.*, $W = (0 \ldots 0 W_{n-1} \ldots W_1 W_0)_b$. This ensures that the internal representation of W has an equal number of bits for each digit, W_i.

A bitstream S of length L can be expressed as the base 2 representation of an unsigned multiple precision integer. Equivalently, S can be partitioned into n-bit segments and interpreted as a base $b = 2^n$ integer, *i.e.*, a sequence of consecutive base 2^n digits represented internally as n-bit values. Here, we add as many zeroes to the beginning of S as necessary so that N divides L.

Given these preliminaries, we now define the s2^k *Scanning* operation semantics in terms of n-bit b2^k operations. We first consider the s2^k scanning operation, $R = Advance(S)$, that advances all

1-bit fields in S. The semantics of $R = Advance(S)$ is defined as follows.

Algorithm 1: n-bit block-at-a-time s2^k Advance algorithm.

Input: A bitstream $S = (S_{(L/n)-1} \ldots S_1 S_0)_{2^n}$ of bit length L with a sufficient number of zeroes prepended to $S_{(L/n)-1}$ to ensure the bit width of $S_{(L/n)-1} = n$.

Output: A bitstream $R = (R_{(L/n)-1} \ldots R_1 R_0)_{2^n}$ such that $R = Advance(S) = (S + S) \mod 2^L$.

```
1 c = 0                                          // carry bit
2 for i = 0 to (L/n) − 1 do
3     R_i = (S_i + S_i + c) mod 2^n
4     c = ⌊(S_i + S_i + c)/2^n⌋
5 return R
```

The s2^k *Spanning* operations is defined in an analogous manner. A trivial change from the addition of n-bit values to the subtraction of n-bit values suffices to define the s2^k Spanning operation, $R = SpanUpTo(S, F) = (S - F) \mod 2^L$, where S and F denote start and follow positions bitstreams. The semantics of $R = SpanUpTo(S, F)$ is defined as follows.

Algorithm 2: n-bit block-at-a-time s2^k SpanUpTo algorithm.

Input: First and follow positions bitstreams, $S = (S_{(L/n)-1} \ldots S_1 S_0)_{2^n}$ and $F = (F_{(L/N)-1} \ldots F_1 F_0)_{2^n}$ of bit length L with a sufficient number of zeroes prepended to $S_{(L/n)-1}$ and $F_{(L/n)-1}$ respectively to ensure the bit width of $S_{(L/n)-1} = F_{(L/n)-1} = n$.

Output: A bitstream $R = (R_{j-1} \ldots R_1 R_0)_n$ such that $R = S - F \mod 2^L$.

```
1 b = 0                                          // borrow bit
2 for i = 0 to (L/n) − 1 do
3     R_i = (S_i − F_i + b) mod 2^n
4     b = ⌊(S_i + S_i + b)/2^n⌋
5 return R
```

Accordingly, we define each of the block-at-a-time scanning and spanning operations using a pair of functions f_n and g_n defined on n-bit values s_n, t_n and a carry-in value, ci. The function f_n produces the N-bit result value, r_n. The function g_n produces the carry-out value, co.

$$r_n = f_n(s_n, t_n, ci) \tag{4.1}$$

$$co = g_n(s_n, t_n, ci) \tag{4.2}$$

For example, the block-at-a-time function, $(r_n, co) = Advance_n(m_n, ci)$, can be defined by a pair of functions $r_n = (m_n + m_n + ci) \mod 2^n$ and $co = \lfloor (m_n + m_n + ci)/2^n \rfloor$. Alternative definitions are possible. For example, $(r_n, co) = Advance_n(m_n, ci)$ can be equivalently defined by a pair of functions $r_n = (m_n + m_n + ci) \mod 2^N$ and $co = g \vee (p \wedge \neg r_n)$, where $g = m_n \wedge m_n$ and $p = m_n \vee m_n$. Definitions for each of the b2^k flow-parallel operations are provided in Appendix G.

4.2.2 Carry Enumeration

Carry enumeration is the process of assigning *carry value numbers* to each carry value required in the block-at-a-time compilation of b2^k operations. In the translation of s2^k to b2^k, carry enumeration is performed by visiting the carry-generating operations in the AST representation of an s2^k filter in a standard order termed the *carry enumeration order*. This order corresponds to a depth-first and left-to-right traversal of the AST.

4.2.3 Block-at-a-time Management of Carry Values

The management of carry values in the block-at-a-time processing of Fieldstream operations involves the enumeration, initialization, insertion, and assignment of carry values. We illustrate the management of carry values by considering three cases. First, the translation of a sequence of Fieldstream operations without conditional branches. Second, the translation of conditional logic as illustrated by an if statement. Third, the translation of iterative logic as illustrated by a while statement. Here, our intent is to provide a uniform view of the management of carry values in the context of the translation of Fieldstream operations to block-at-a-time operations.

Straight-Line Logic. Figure 4.1 illustrates straight-line logic in the parallel scanning of digits using abstract 1-bit Fieldstream (bitstream) operations. The left-hand column shows abstract operations. The right-hand column shows the effects of each operation. Line numbers are shown at the far left. The figure heading shows a data stream containing two spans of digits as well as non-digit characters indicated with hyphens. As before, the operations of Figure 4.1 are expressed on entire

bitstreams and proceed in sequence beginning at line 1. Line 1 marks with 1s the positions of decimal digits. Lines 2-4 show the calculation of an initial set of marker bits marking initial scanning positions. Line 5 shows the *ScanThru* operation. This operation moves each of the marker positions of m_0 through the corresponding digit spans of d.

Figure 4.2 shows the translation of the bitstream operations of Figure 4.1 to block-at-a-time operations. Subfigure 4.2a shows the translation of the first 16 byte block (Block 1), and Subfigure 4.2b shows the translation of the next block (Block 2). The first row of the headings show the 16-byte partitioned data stream. Block 1 contains a complete digits span `2493711` as well as a partial digits span `56`. Block 2 completes the span beginning with `14`. The second row of the headings show enumerated *carry values* corresponding to each carry-generating operation.

Block-at-a-time processing begins with the initialization of block 1 carry values. Line 3 and line 5 of Subfigure 4.2a show the carry values $c_{0,1}$ and $c_{1,1}$ initialized to zero. Carry variable subscripts indicate carry and block numbers. For example, $c_{0,1}$ and $c_{0,2}$ represent the 0th carry of block 1 and block 2 respectively. Similarly, d_1 and d_2 indicates the block 1 and block 2 16-bit values that compose the bitstream d, where $d = (d_2 * 16^1) + d_1$.

The operations of block 1 are expressed on 16-bit values. Line 1 marks with 1s the positions of digits. Lines 2-4 shows the calculation of an initial set of marker bits marking initial scanning positions. Line 3 shows the block-at-a-time $Advance_n$ operation. This operation accepts the temporary $t0, 1$ and the carry value $c_{0,1}$ and produces the temporary $t_{1,1}$ and the carry value $c_{0,2}$. Line 5 shows the block-at-a-time $ScanThru_n$ operation. This operation accepts $m_{0,1}$, d_1, and $c_{1,1}$ as arguments and moves each of the markers of $m_{0,1}$ through the corresponding spans of digits of d_1 producing the carry value $c_{1,2}$. The generated carry values of block 1 propagate carry information from block 1 to block 2. The sequence of block-at-a-time operations is repeated for block 2. Line 5 shows the final scanning position of the digit `1456` as determined by the scanning of the block 1 and block 2 in sequence.

	8-bit ASCII data $\lhd$	...	`----------1456---2493711----`
1	$d = [0-9]$	...	`.....1111...1111111....`
2	$t_0 = \neg d$	...	`11111111111111....111.....1111`
3	$t_1 = Advance(t_0)$	...	`1111111111111....111.....1111.`
4	$m_0 = d \wedge t_1$	...	`.....1.....1....`
5	$m_1 = ScanThr(m_0, d)$	...	`.....1.....1.....`

Figure 4.1: Straight-line Fieldstream operations.

	8-bit ASCII data $\triangleleft$	...		`56---2493711----`	
	Carry Set	...	$\{c_{0,2}, c_{1,2}\}$		$\{c_{0,1}, c_{1,1}\}$
1	$d_1 = [0-9]$	...		`11...1111111....`	
2	$t_{0,1} = \neg d_1$	...		`..111.......1111`	
3	$(t_{1,1}, c_{0,2}) = Advance_n(t_{0,1}, c_{0,1})$	...	$c_{0,2} = 0$	`.111.......1111.`	$c_{0,1} = 0$
4	$m_{0,1} = d_1 \wedge t_{1,1}$	...		`.1.........1....`	
5	$(m_{1,1}, c_{1,2}) = ScanThru_n(m_{0,1}, d_1, c_{1,1})$	...	$c_{1,2} = 1$	`....1...........`	$c_{1,1} = 0$

(a) Block 1.

	8-bit ASCII data $\triangleleft$	...		`--------------14`	
	Carry Set	...	$\{c_{0,3}, c_{1,3}\}$		$\{c_{0,2}, c_{1,2}\}$
1	$d_2 = [0-9]$	...		`..............11`	
2	$t_{0,2} = \neg d_2$	...		`11111111111111..`	
3	$(t_{1,2}, c_{0,3}) = Advance_n(t_{0,2}, c_{0,2})$	...	$c_{0,3} = 0$	`1111111111111...`	$c_{0,2} = 0$
4	$m_{0,2} = d_2 \wedge t_{1,2}$	...		`................`	
5	$(m_{1,2}, c_{1,3}) = ScanThru_n(m_{0,2}, d_2, c_{1,2})$	...	$c_{1,3} = 0$	`.............1..`	$c_{1,2} = 1$

(b) Block 2.

Figure 4.2: Equivalent straight-line block-at-a-time operations for the Figure 4.1 Fieldstream operations.

Conditional Logic. Figure 4.3 shows conditional logic in the parallel scanning of digits as previously shown in Section 3.5.2. The semantics of the parallel if statement of Figure 4.4 are as follows. If m_0 is non-zero then the statement $m_1 = ScanThru(m_0, d)$ is evaluated, otherwise the statement is skipped. Next, Figure 4.4 shows the translation of the bitstream operations of Figure 4.3 to equivalent block-at-a-time operations. As described, carry values are enumerated and initialized. Line 5 of Subfigure 4.4 shows the block-at-a-time if statement. Each flow-parallel (carry generating) operation in the body of the if statement extends the block-at-a-time if condition by the bitwise ORing of carry-in values corresponding to the carry generating statements contained within the body of the if statement. This ensures that the scanning of the complete span, `1456`, as a pair of partial spans, `56` and `14`, that cross a block boundary are equivalent.

	8-bit ASCII data $\lhd$	...	----------1456---2493711----
1	$d = [0-9]$	...	.....1111...1111111....
2	$t_0 = \neg d$	...	11111111111111....111.....1111
3	$t_1 = Advance(t_0)$	...	1111111111111....111.....1111.
4	$m_0 = d \wedge t_1$	...	.....1.....1....
5	$if(m_0)$	...	.....1.....1....
6	$m_1 = ScanThru(m_0, d)$	...	.....1.....1.....

Figure 4.3: Conditional Fieldstream operations.

	8-bit ASCII data $\lhd$	...		56---2493711----	
	Carry Set	...	$\{c_{0,2}, c_{1,2}\}$		$\{c_{0,1}, c_{1,1}\}$
1	$d_1 = [0-9]$	...		11...1111111....	
2	$t_{0,1} = \neg d_1$	...		..111.....1111	
3	$(t_{1,1}, c_{0,2}) = Advance_n(t_{0,1}, c_{0,1})$	...	$c_{0,2} = 0$	.111.....1111.	$c_{0,1} = 0$
4	$m_{0,1} = d_1 \wedge t_{1,1}$	...		.1.....1....	
5	$if(m_{0,1} \vert c_{1,1})$	...		.....1....	
6	$(m_{1,1}, c_{1,2}) = ScanThru(m_{0,1}, d_1, c_{1,1})$	...	$c_{1,2} = 1$	....1.....	$c_{1,1} = 0$

(a) Block 1.

	8-bit ASCII data $\lhd$	...		----------14	
	Carry Set	...	$\{c_{0,3}, c_{1,3}\}$		$\{c_{0,2}, c_{1,2}\}$
1	$d_2 = [0-9]$	...		11	
2	$t_{0,2} = \neg d_2$	...		11111111111111..	
3	$(t_{1,2}, c_{0,3}) = Advance_n(t_{0,2}, c_{0,2})$	...	$c_{0,3} = 0$	1111111111111...	$c_{0,2} = 0$
4	$m_{0,2} = d_2 \wedge t_{1,2}$	...		.....	
5	$if(m_{0,2} \vert c_{1,2})$	...		.....	
6	$(m_{1,1}, c_{1,2}) = ScanThru_n(m_{0,1}, d_1, c_{1,1})$	...	$c_{1,2} = 1$	.....1..	$c_{1,2} = 1$

(b) Block 2.

Figure 4.4: Equivalent conditional block-at-a-time operations for the Figure 4.3 Fieldstream operations.

Iterative Logic. Figure 4.5 shows iterative logic in the sequential scanning of digits and Figure 4.6 shows the unrolled the statements of Figure 4.5 as previously described in Section 3.5.2. Figure 4.7 shows the translation of the bitstream operations of Figure 4.6 to equivalent block-at-a-time operations. Carry values are enumerated and initialized as previously described. As in the translation of the parallel if statement, additional carry logic ensures that the scanning of the partial spans `56` and `14` that cross block boundaries are processed correctly. Although, not explicitly shown by Figure 4.7

additional logic is required to translate parallel while statement to block-at-a-time equivalents. We defer presentation of these translation steps until Chapter 5.

```
1  d = [0 − 9]
2  m_0 = ScanToFirst(d)
3  while(m_0)
4    m_1 = ScanThru(m_0, d)
5    m_0 = ScanTo(m_1, d)
```

Figure 4.5: Iterative Fieldstream operations.

8-bit ASCII data ◁	...	----------1456---2493711----
1 $d = [0-9]$	...	.....1111...1111111....
2 $m_0 = ScanToFirst(d)$	...	.....1....
3 $m_{1_1} = ScanThru(m_0, d)$	...	.....1.....
4 $m_{0_1} = ScanTo(m_{1_1}, d)$	...	.....1.....
5 $m_{1_2} = ScanThru(m_{0_1}, d)$	...	.....1.....
6 $m_{0_2} = ScanTo(m_{1_2}, d)$	...	.....

Figure 4.6: Unrolled Fieldstream operations of Figure 4.5.

	8-bit ASCII data $\lhd$	...		`56---2493711----`	
	Carry Set	...	$\{c_{0,0}, \ldots\}$		$\{c_{0,1}, \ldots\}$
1	$d_1 = [0-9]$	...		`11...1111111....`	
2	$(m_{0,1}, c_{0,2}) = ScanToFirst_n(d, c_{0,1})$	...	$c_{0,2} = 0$	`..........1....`	$c_{0,1} = 1$
3	$(m_{1_1,1}, c_{1,2}) = ScanThru_n(m_{0,1}, d, c_{1,1})$	...	$c_{1_1,2} = 0$	`....1..........`	$c_{1_1,1} = 0$
4	$(m_{0_1,1}, c_{2,2}) = ScanTo_n(m_{1_1,1}, d, c_{2,1})$	...	$c_{2_1,2} = 0$	`.1.............`	$c_{2_1,1} = 0$
5	$(m_{1_2,1}, c_{3,2}) = ScanThru_n(m_{0_1,1}, d, c_{3,1})$	...	$c_{1_2,2} = 1$	`................`	$c_{1_2,1} = 0$
6	$(m_{0_2,1}, c_{4,2}) = ScanTo_n(m_{1_2,1}, d, c_{4,1})$	...	$c_{2_2,2} = 1$	`................`	$c_{2_2,1} = 0$

(a) Block 1.

	8-bit ASCII data $\lhd$	...		`--------------14`	
	Carry Set	...	$\{c_{0,0}, \ldots\}$		$\{c_{0,2}, \ldots\}$
1	$d = [0-9]$	...		`11...1111111....`	
2	$(m_0, c_{0,2}) = ScanToFirst_n(d, c_{0,1})$	...	$c_{0,3} = 0$	`..............11`	$c_{0,2} = 0$
3	$(m_{1_1,2}, c_{1,2}) = ScanThru_n(m_0, d, c_{1,1})$	...	$c_{1_1,3} = 0$	`................`	$c_{1_1,2} = 0$
4	$(m_{0_1,2}, c_{2,2}) = ScanTo_n(m_{1_1,2}, d, c_{2,1})$	...	$c_{2_1,3} = 0$	`................`	$c_{2_1,2} = 0$
5	$(m_{1_2,2}, c_{3,2}) = ScanThru_n(m_{0_1,2}, d, c_{3,1})$	...	$c_{1_2,3} = 0$	`.............1..`	$c_{1_2,2} = 1$
6	$(m_{0_2,2}, c_{4,2}) = ScanTo_n(m_{1_2,2}, d, c_{4,1})$	...	$c_{2_2,3} = 1$	`................`	$c_{2_2,2} = 1$

(b) Block 2.

Figure 4.7: Iterative block-at-a-time equivalent operations for the Figure 4.6 Fieldstream operations.

4.3 Intermediate Representation Structure

In this section we describe the features of b2^k. We divide the b2^k language description into two parts, graph-level constructs and kernel-level constructs. Many b2^k features are block-at-a-time analogues of previously defined s2^k features, we only briefly describe these features.

4.3.1 Graph-level Constructs

Graph-level features include Bitblock structures, functions, kernels, and graphs.

Bitblock Structures. A Bitblock structure is a non-nestable record type. b2^k restricts Fieldstream structure member types to Bitblocks and Bitblock arrays. The following example defines and declares a b2^k Bitblock structure. The identifier of the structure is `Lex`. `Lex` contains a single Bitblock member, `digits`. The declared instance of `struct Lex` is named `lex`.

```
struct Lex
{
  BitBlock digits;
}
struct Lex lex;
```

Bitblock Functions. A Bitblock function is a subprogram that computes one or more Bitblock values. A function that returns more than one return value must have a return statement as the final statement. Function definitions specify input parameters and output parameters using parameter modes. As in $s2^k$, parameters modes are limited to *in* and *out*.

The following example shows the definition of a Bitblock function that computes the bit counts of the 16-bit fields of a Bitblock.

```
function BitBlock PopCount(in: BitBlock b)
{
  BitBlock t0 = AddHiLo<2>(b);
  BitBlock t1 = AddHiLo<4>(t0);
  BitBlock t2 = AddHiLo<8>(t1);
  BitBlock t3 = AddHiLo<16>(t2);
  return t3;
}
```

Bitblock Kernels. The base unit of computation in $b2^k$ is the Bitblock kernel. Kernels transform sets of input Bitblocks into sets of output Bitblocks. Like $s2^k$ filters, $b2^k$ kernels are independent, self-contained, and non-nestable program units whose function definitions cannot contain calls to other kernel functions. Each $b2^k$ kernel contains an initialization section `init` together with one or more block-at-a-time function definitions. Typically, kernels describe the processing of full blocks that occur in the steady-state processing of a stream graph as well as partial blocks that occur in the stream epilogue with a pair of kernel functions. Other organization strategies are possible depending on the kernel scheduling or problem constraints.

Figure 4.1 shows an example kernel that marks the start and follow positions of decimal digits. The `init` section declares and initializes kernel properties. In this case `init` declares and initializes a set of carry values. The `do_full_block` function defines the processing of full blocks. The `do_partial_block` function defines the processing of partial blocks. This function contains additional logic to process partial blocks and handle end-of-stream stream conditions. The `do_opt_block` function defines optimized processing of full blocks.

```
kernel Match {

  init {
    BitBlock[2] carry_set;
    carry_set[0] = Constant<1>(0);
    carry_set[1] = Constant<1>(0);
  }

  do_full_block(in: struct Lex lex, out: struct Output output) {
    BitBlock t0 = Not<1>(lex.digits);
    BitBlock t1;
    carry_set[0] = Advance(t0, carry_set[0], t1);
    BitBlock m0 = And<1>(d,t1);
    output.starts = Or<1>(output.starts, m0);
    BitBlock m1;
    carry_set[1] = ScanThru(m0, lex.digits, carry_set[1], m1);
    output.follows = Or<1>(output.follows, m1);
  }

  do_partial_block (in: struct Lex lex, in: int32 cnt, out: struct Output output) {
    BitBlock ones = Constant<1>(1);
    BitBlock mask = SRL<1>(ones, BLOCK_SIZE - count + 1); // file extent mask

    BitBlock t0 = Not<1>(And<1>(lex.digits, mask));
    BitBlock t1
    carry_set[0] = Advance(t0, carry_set[0], t1);
    BitBlock m0 = And<1>(d,t1);
    output.starts = Or<1>(output.starts, m0);
    if(Or<1>(m0, carry_set[1])) {
      BitBlock m1
      carry_set[1] = ScanThru(m0, AndC(lex.digits, mask), carry_set[1], m1));
      output.follows = Or<1>(output.follows, m1);
    }
  }

  do_opt_block (in: struct Lex lex, out: struct Output output) {
    BitBlock t0 = Not<1>(lex.digits);
    BitBlock t1
    carry_set[0] = Advance(t0, carry_set[0], t1);
    BitBlock m0 = And<1>(d,t1);
    output.starts = Or<1>(output.starts, m0);
    if(Or<1>(m0, carry_set[1]) {
      BitBlock m1
      carry_set[1] = ScanThru(m0, lex.digits,  carry_set[1], m1);
      output.follows = Or<1>(output.follows, m1);
    }
  }
}
```

Program 4.1: Example $b2^k$ kernel.

Bitblock Graph. The Bitblock Fieldstream graph is the main program unit. A Bitblock graph is a directed acyclic graph. The vertices of the graph are kernels and the edges are structures. Execution of a graph transforms a set of input Bitblocks into a set of output Bitblocks.

4.3.2 Kernel-level Constructs

Type System. b2^k is statically and strongly typed. b2^k scalar types include integer types, floating point types, and string literals. Aggregate types consist of a family of parameterized 2^k-bit Bitblock types as well as Bitblock array type.

Scalar Types. The b2^k scalar types are largely uninteresting and correspond to the s2^k scalars.

Aggregate Types. The b2^k aggregate types are the Bitblock type and the Bitblock array type. The Bitblock type denotes SIMD register values.

The Bitblock array type represents as a sequence of consecutive Bitblocks. The Bitblock array type is useful for the representation of parallel-field values. For example, the arrays `b_10`, `b_1`, and `b_0` denote parallel arrays with indexed values in one-to-one correspondence.

```
BitBlock[2] b_10;
BitBlock[1] b_1;
BitBlock[1] b_0;

b_1[0] = PackHi<2>(b_10[1], b_10[0]);
b_0[0] = PackHi<2>(b_10[1], b_10[0]);
```

Type Conversions and Casts. The Bitblock type is a polymorphic type. Unlike the s2^k Fieldstream type is not a parametric type, rather b2^k operations logically partition Bitblock values according to the function definition. This eliminates value casts and simplifies b2^k intermediate code generation.

Expressions.

Scalar Expressions. Unlike s2^k scalars, b2^k scalars are not limited to constant expressions.

Bitblock Expressions. b2^k Bitblock expressions are collective operations which operate on Bitblock values.

Assignment Statements. The semantics of the $b2^k$ assignment statement is analogous to an $s2^k$ assignment statement. Assignment statements have copy-on-assignment semantics.

Control Flow Statements. As in $s2^k$, $b2^k$ does not allow the mixing of scalar and vector control flow constructs. $b2^k$ control flow is expressed on Bitblocks. Control flow statements include: parallel *if* and parallel *while* statements. As described in Section 4.2.3 the programming of *if* and *while* conditions must include the insertion of additional logic to ensure that each carry-generating operation within the body of an *if* or *while* is evaluated correctly. Note, *Foreach..In* and *For..In* are not necessary for the translation of $s2^k$-to-$b2^k$ and thus not defined by $b2^k$.

Intrinsic Library Functions. The $b2^k$ operations correspond to the abstract operation described in Section 4.2.1. Appendix G defines the $b2^k$ operations.

4.4 Chapter Summary

The $b2^k$ intermediate language is a minimal intermediate language for the expression of $s2^k$ Fieldstream programs as $b2^k$ block-at-a-time programs. The design of $b2^k$ supports the block-at-a-time expression of the graph-level and filter-level constructs of $s2^k$ program. The design of $b2^k$ is simple and flexible and does not restrict implementations to a specific block-at-a-time scheduling or carry value strategy. Kernel dependencies are made explicit by the kernel-graph information flow and kernel property information set items. Compiler writers can leverage this information to schedule parallel executions using multiple forms of parallelism as well as to support streaming text processing features such as fixed lookahead.

Chapter 5

Compilation

In this chapter we describe the $s2^k$ compilation process. The $s2^k$ compilation process requires, first, the translation of $s2^k$ to $b2^k$, and next, the translation of $b2^k$ to C, C++, or LLVM IR for compilation to machine code. The translation of $s2^k$ to $b2^k$ is described in detail. The translation of $b2^k$ to C++ is straightforward and thus only briefly presented. The remainder of the chapter is organized as follows. Section 5.1 presents advantages and disadvantages of the translation of $s2^k$ global-view operations and control-flow constructs. Section 5.2 describes the translation of $s2^k$ to $b2^k$. Here, we focus on the source-to-source translation of flow-parallel and field-parallel operations. Both full- and partial-block operations are presented. Section 5.3 describes the translation of $b2^k$ to C++ and Section 5.4 goes on to conclude this chapter.

5.1 Overview

Programming languages characterized by global-view operations over aggregate types such as arrays, lists, or streams can result in programs that are easier to express, analyze, and optimize. Indeed, much of the appeal of such language designs arises from the decoupling of the specification of algorithms from the implementation. This decoupling allows programmers to focus on the specification of computations rather than on implementation details [91]. Although high-level language abstractions can benefit programmer productivity, abstract language features widen the translation distance between a source language and machine code. This commonly leads to additional compiler phases that gradually lower the program representation, thus making program compilation more complex.

A number of reasons exist to perform source-to-source translation. Typically, source translations are performed to improve application performance. Additional reasons include programmer productivity, program reliability, and software integration [71]. The primary motivation for $s2^k$ source translation is to simplify streaming text programming using parallel Fieldstream operations. The $s2^k$

translation strategy follows a systematic approach that uses compiler tools to analyze and translate $s2^k$ to $b2^k$. The $b2^k$ language is then translated to C, C++, or LLVM IR [59, 85]. In general, this approach relies on an existing back-end compiler to optimize and generate machine code. Compilers developed in this manner can lead to an expedited implementations with the back-end compiler performing the heavy-lifting in the generation of optimized machine code [16].

5.2 $s2^k$ to $b2^k$ Transformations

The translation of $s2^k$ to $b2^k$ requires a number of source-to-source transformations. Source translations range from simple translation of types to more sophisticated transformations involving looping constructs. Since many of these techniques are well-known [5, 54, 62, 91] we do not present each translation case. Rather, we provide a uniform view of the translation process emphasizing techniques unique to $s2^k$.

We organize the description of the translation of $s2^k$ to $b2^k$ into phases and proceed in a bottom-up manner. We begin with a description of parallel Fieldstream translations. We then describe two important categories of transformations associated with $s2^k$ operations, field-parallel translations and flow-parallel translations. Next, we consider transformations associated with the block-at-a-time processing of full- and partial-blocks. The processing of full blocks occurs in the steady-state iterations of a kernel-graph. Conceptually, the processing of partial-blocks occurs in the graph epilogue when the number of remaining data items is less than the block size.

In general, $s2^k$ filter translation strategies must define both full- and partial-block source translations to $b2^k$. Full- and partial-block translations generate independent block-at-a-time kernel functions that can be invoked based on the runtime processing mode (full- or partial-block mode). Typically, partial-block translations simply mask (or equivalently zero-extend) partial block operations. We describe both full- and partial-block translations strategies in the description of $s2^k$ filter to $b2^k$ kernel translations.

We assume that the following items are available in our description of $s2^k$ source-to-source translations.

1. AST (Abstract Syntax Tree) representation. An AST representation of a source program is available.

2. Virtual Query Methods. A set of virtual methods is available to query properties associated with AST nodes. Typically, compilers take multiple information gathering passes that populate a symbol table information. Here, virtual methods simplify our description.

3. Virtual AST Manipulation and Generation Methods. A set of virtual methods is available to translate source constructs to target language constructs.

4. Virtual Code Generation Methods. A set of virtual methods is available to produce the textual representation of AST constructs.

In addition, the application of one-or-more prior AST translations applied in a non-random order may be assumed. Indeed, many source translations depend on prior translation operations. In practice many difficult software engineering problems are concerned with program analysis and transformation orderings. Here, however, we are interested in the identification of source language patterns and target language replacements. Rather than provide detailed syntax-directed translation algorithms we present simple before and after examples and assume that the reader can deduce the form of each source translation. Program translations are shown by the replacement of parameterized source text fragments with target text fragments. Parameters are indicated with italics. Surrounding text fragments are indicated by mono spaced font.

We next describe s2^k to b2^k translations that support the block-at-a-time processing of n-bit Fieldstreams. As described in Chapter 4, the b2^k block-at-a-time processing model is an incremental block-by-block processing model that supports the translation of Fieldstreams operations to Bitblock operations. The s2^k to b2^k translations presented follow directly from our previous description of the b2^k block-at-a-time processing model in which we translate n-bit Fieldstream operations to n consecutive Bitblock operations. Thus, n-bit Fieldstreams are decomposed into n segments for block-at-a-time processing. For example, an operation defined on 1-bit Fieldstreams is translated to a single Bitblock operation, whereas, an operation defined on 4-bit Fieldstreams is translated to four consecutive Bitblock operations. In each of the following transformation definitions, the left-hand side shows s2^k code fragments and the right-hand side shows resultant b2^k code fragments. We assume prior translation to three-address form.

5.2.1 Fieldstream Variable Declaration Statements.

An s2^k n-bit Fieldstream declaration statement is translated to a b2^k Bitblock array declaration of size n.

stream<n> T; ==> **BitBlock**[n] T;

5.2.2 Field-parallel Function Call Assignments.

Assignment statements in which the right-hand side is a n-bit field-parallel function call are transformed in one of three ways. The transformation strategy depends on whether an s2^k n-bit field-parallel operation produces an n-bit Fieldstream, an $n/2$-bit Fieldstream, or an $2n$-bit Fieldstream. We term these transformations n-bit, $n/2$-bit, and $2n$-bit field-parallel function call assignments.

n-bit Field-parallel Function Call Assignments. n-bit Vertical and Field-movement function call assignments are replaced by n consecutive block-at-a-time function calls. Block-at-a-time resultant values are assigned to consecutive indexes of a Bitblock array of size n.

```
stream<n> S;      ==>   BitBlock[n] S;
stream<n> T;            BitBlock[n] T;
stream<n> R;            BitBlock[n] R;
R = F<n>(S,T);          for i in 0 .. n by 1
                        {
                           R[i] = F<n>(S[i],T[i]);
                        }
```

$n/2$-bit Field-parallel Function Call Assignments. $n/2$-bit Horizontal-packing function call assignments are replaced by $n/2$ corresponding block-at-a-time function calls. Block-at-a-time resultant values are assigned to consecutive indexes of a Bitblock array of size $n/2$.

```
stream<n> S;      ==>   BitBlock[n] S;
stream<n/2> R;          BitBlock[n/2] R;
R = F<n>(S,T);          for i in 0 .. n/2 by 2
                        {
                           R[i] = F<n>(S[i+1],S[i]);
                        }
```

$2n$-bit Field-parallel Function Call Assignments. $2n$-bit Expansion function call assignments are replaced by n pairs of corresponding block-at-a-time function calls which determine the *high* and the *low* Bitblock values. Block-at-a-time resultant values are assigned to consecutive indexes of a Bitblock array of size $2n$.

```
stream<n> S;      ==>   BitBlock[n] S;
stream<n> T;            BitBlock[n] T;
stream<2n> R;           BitBlock[2n] R;
R = F<n>(S,T);          for i in 0 .. n by 1
                        {
                           R[i + 1] = FHi<n>(S[i + 1],T[i]);
                           R[i] = FLo<n>(S[i + 1],T[i]);
                        }
```

A concrete example of this form of translation is the translation of an s2^k `Merge`$<n>(S,T)$ operation to a pair of b2^k `MergeHi`$<n>(S,T)$ and `MergeLo`$<n>(S,T)$ operations.

5.2.3 Flow-parallel Transformations.

This section builds on the notion of the carry value management in the translation of s2^k to b2^k. As described in Section 4.2.3 we consider three cases. First, the translation of straight-line

sequences of s2^k statements, in this case flow-parallel assignments. Second, the translation of s2^k conditional statements. Third, the translation of s2^k iteration statements. Whereas the translation of field-parallel operations is context-free the translation of flow-parallel operations is context-dependent. In the context-dependent translation of operation we distinguish between two contexts, the *carry-in-out* context and the *carry-out* context. In the carry-in-out context incoming carry values are read and out-going carry values are written. In the carry-out context incoming carry values set to zero and out-going carry values are written. The carry-in-out context is the default translation context. The carry-out context is entered in translation of block-at-a-time *inner while* loops as described below.

In our description of the flow-parallel transformations we make the following assumptions. First, we assume three-address form. Second, we assume carry values are enumerated and initialized on a filter-by-filter basis and that the ith carry value of a filter is available through the indexing of a virtual filter carry set, *e.g.*, $carry_set[\ i\ ]$. Third, we assume the *carry position* and the *carry count* of each *carry item* is available. s2^k carry items include: flow-parallel function calls, conditional statements, iterative statements, and filter definitions. The carry position of an item is the ordinal position of that item within of a s2^k filter as defined by the carry enumeration order. The carry count of a carry item is the count of flow-parallel functions calls that are descendants of an AST carry item node. For example, the carry position of an s2^k filter is zero and the carry count of the filter is the number of flow-parallel operation calls within the filter.

Flow-Parallel Operations. Flow-Parallel function call assignments are translated to block-at-a-time function calls that read carry-in values and write carry-out values.

```
stream<1> S;          ==>  BitBlock[1] S;
stream<1> T;               BitBlock[1] T;
R = Advance(S);            carry_set[ position ] = Advance(S, carry_set[ position ], R);
R = ScanTo(S, T);          carry_set[ position ] = ScanTo(S, T, carry_set[ position + 1 ], R);
```

If Statements. s2^k if statements are translated to block-at-a-time if statements that insert additional conditional logic to test that each carry value within the if statement body is non-zero.

```
if(Condition)   ==>  if(Or<1>(Condition, CarryTest(position, count))
{                    {
  Body()               gen_if_body(Body())
}                    }
```

While Statements. s2^k while body statements are translated to block-at-a-time statements in accord with statically defined outer-while and inner-while translation contexts. The first iteration of a

block-at-a-time loop is translated in the outer-while translation context. In the outer-while context, both incoming and outgoing carry values are set to the input and output carry values of carry generating functions respectively. Subsequent loop iterations are translated in the inner-while context. In the inner-while translation context, incoming carry values are set to zero and outgoing carry values are set to the output carry values of carry generating functions respectively.

```
while(Condition)   ==>   if(Or<1>(Condition, CarryTest(position, count))
{                        {
  Body()                   gen_outer_while_body(Body())
}                          while(Condition)
                           {
                             gen_inner_while_body(Body())
                           }
                         }
```

Partial-block Translations. The translation to partial-blocks requires additional considerations. As a general strategy we generate n-bit Fieldstream masks to mark valid input Fieldstream positions and control final block processing. We illustrate two cases in the following examples.

Input-stream Extents. Since Fieldstream operations derive output streams from input streams it is sufficient to mask input stream values. Here, an input selection mask is applied to the bitstream members of a set of basis bitstreams. We assume that a single basis bitstream set is used to derive the streams throughout the program.

```
basis.bit_7;   ==>   BitBlock ones = Constant<1>(1);
basis.bit_6;         BitBlock mask = SRL<1>(ones, BLOCK_SIZE - count + 1);
...                  And<1>(basis.bit_7, mask);
basis.bit_0;         And<1>(basis.bit_6, mask);
                     ...
                     And<1>(basis.bit_0, mask);
```

Flow-parallel Operations. A sentinel position is required to ensure $ScanTo$ variants terminate at the extents of an input stream. It is sufficient to mask the scanned stream with the complement of the input selection mask.

```
R = ScanTo(S, T);   ==>   carry_set[ position ] = ScanTo(S, AndC<1>(T, mask),
                                                          carry_set[ position + 1 ], R);
```

End-of-stream Considerations. Block-at-a-time processing may end with the processing of a full block. In this case, carry-out values may indicate an incomplete syntactic constructs or a final match position. In practice we simply process an additional partial-block that processes the carry-in values of the final full block to ensure such cases are correctly handled.

5.2.4 Fieldstream-filter Definitions.

s2^k filters are translated to block-at-a-time processing kernel functions in a straightforward manner. Here, the body of the partial-block kernel function, $Body'()$, is produced by translating the body of the Fieldstream filter in partial-block translation mode.

```
filter Identifier (Parameters()) {    ==>    kernel Identifier {
  Body()                                        init{...}
}                                               function do_full_block (Parameters()) {
                                                  gen_full_block_kernel_body(Body())
                                                }
                                                function do_partial_block (Parameters()) {
                                                  gen_partial_block_kernel_body(Body())
                                                }
                                              }
```

5.3 b2^k to C++ Transformations

The translation of b2^k to C++ is straightforward and by and large mechanical. As such, we show the translation of a simple b2^k kernel to a C++ struct with appropriate runtime support. This b2^k kernel is the block-at-a-time equivalent of the s2^k *Match* filter of the s2^k Grep exact string search program listed in Appendix E.2.

```
kernel Match {

   init {
     BitBlock [5] carry_set;
   }

   function do_block(in: struct Lex lex, out: struct Output output) {
     BitBlock cursor;
     BitBlock t0;
     BitBlock t1;
     BitBlock t2;
     BitBlock t3;
     carry_set[0] = bitblock.srli<128>(127, flow.Advance(lex.a, carry_set[0], cursor));
     t0 = simd.and<1>(cursor,lex.p);
     carry_set[1] = bitblock.srli<128>(127, flow.Advance(t0, carry_set[1], cursor));
     t1 = simd.and<1>(cursor,lex.p);
     carry_set[2] = bitblock.srli<128>(127, flow.Advance(t1, carry_set[2], cursor));
     t2 = simd.and<1>(cursor,lex.l);
     carry_set[3] = bitblock.srli<128>(127, flow.Advance(t2, carry_set[3], cursor));
     t3 = simd.and<1>(cursor,lex.e);
     carry_set[4] = bitblock.srli<128>(127, flow.Advance(t3, carry_set[4], cursor));
     output.match_follows = cursor;
   }

}

   ==>

struct Match

  Match{}

  inline void do_block(struct Lex & lex, struct Output & output) {

    BitBlock cursor;
    BitBlock t0;
    BitBlock t1;
    BitBlock t2;
    BitBlock t3;
    carry_set[0] = bitblock::srli<127>(flow.Advance(t3,carry_set[0],cursor));
    t0 = simd_and(cursor,lex.p);
    carry_set[1] = bitblock::srli<127>(flow.Advance(t0,carry_set[1],cursor));
    t1 = simd_and(cursor,lex.p);
    carry_set[2] = bitblock::srli<127>(flow.Advance(t1,carry_set[2],cursor));
    t2 = simd_and(cursor,lex.l);
    carry_set[3] = bitblock::srli<127>(flow.Advance(t2,carry_set[3],cursor));
    t3 = simd_and(cursor,lex.e);
    carry_set[4] = bitblock::srli<127>(flow.Advance(t3,carry_set[4],cursor));
    output.match_follows = cursor;
  }

  BitBlock carry_set[4];

};
```

5.4 Chapter Summary

The design of source-to-source translation strategies from $s2^k$ to $b2^k$, and from $b2^k$ to a portable back-end language is a valuable technique. A number of benefits were identified. First, an inherent modularization developed from the definition of the $s2^k$ and $b2^k$ languages simplifying the compilation of $s2^k$. Second, since $b2^k$ has a textual form we were able to parse and evaluate $b2^k$ language features prior to implementing $s2^k$ translation logic. Third, information gathering and optimization phases can be independently applied to the streaming or block-at-a-time program representations, simplifying analysis. Fourth, porting the $s2^k$ compiler to support additional back-end languages is straightforward.

Chapter 6

Inverted-index Construction: An s2^k Case Study

This chapter presents a case study in inverted-index construction with s2^k. Section 6.1 introduces the problem of inverted-index construction. Section 6.2 presents an implementation of inverted-index construction in s2^k and discusses advantages and disadvantages that stem from the language design. Section 6.3 concludes the chapter.

6.1 Overview

Of recent interest are text processing problems that search large document collections. Commercial search engines commonly solve search problems with offline index construction, using programming models such as MapReduce together with sequential text processing methods to construct indexes [32]. Text search has also increased in importance at the other end of the hardware spectrum [92]. For instance, applications to efficiently search local file systems on personal computers and handheld devices are of increasing interest. While it is relatively straightforward to scale text applications on clusters, it is non-trivial to scale equivalent applications on commodity computers, for example, relatively inexpensive and widely available standard personal computers or mobile devices. The s2^k programming language addresses this limitation allowing programmers to express scalable text processing operations on such hardware architectures.

6.1.1 Inverted-index Construction

An inverted-index is a data structure used to accelerate search that associates a dictionary of terms with a list of documents within a collection. An inverted-index consists of two parts: a dictionary of terms and an inverted-index file. Terms are commonly expressed using regular expressions

to describe sets of words. Inverted-index files associate terms with document identifiers and additional search parameters. If an index file indicates only per term document identifiers it is called a document-level inverted-index. If in addition, an index file indicates the ordinal positions of terms within a document it is called a word-level inverted-index. Additional statistics may be collected to improve search results such as the term frequency per document, counts of documents containing a term, and counts of occurrences of a term within a collection [92]. Figure 6.1 shows an example document collection excerpted from [92].

Document ID	Document Content
1	the old night keeper keeps the keep in the town
2	in the big old gown in the big old house
3	the house in the town had the big old keep
4	where the old night keeper never did sleep
5	the night keeper keeps the keep in the night
6	and keeps in the dark and sleeps in the light

Figure 6.1: The *Keeper* database of single-line documents.

Figure 6.2 shows the record-level inverted-index for the *Keeper* database. Each term is associated with a list of pairs (d,$f_{d,t}$) where d is the document identifier and $f_{d,t}$ is the frequency of a term t in d. For instance, there are two occurrences of the term *big* in document 2 and a single occurrence in document 3.

Term t	Inverted List for t
and	(6, 2)
big	(2, 2), (3, 1)
dark	(6, 1)
did	(4, 1)
gown	(2, 1)
had	(3, 1)
house	(2, 1), (3, 1)
in	(1, 1), (2, 2), (3, 1), (5, 1), (6, 2)
keep	(1, 1), (3, 1), (5, 1)
keeper	(1, 1), (4, 1), (5, 1)
keeps	(1, 1), (5, 1), (6, 1)
light	(6, 1)
never	(4, 1)
night	(1, 1), (4, 1), (5, 2)
old	(1, 1), (2, 2), (3, 1), (4, 1)
sleep	(4, 1)
sleeps	(6, 1)
the	(1, 3), (2, 2), (3, 3), (4, 1), (5, 3), (6, 2)
town	(1, 1), (3, 1)
where	(4, 1)

Figure 6.2: Inverted-index for the *Keeper* database of single-line documents.

6.2 Inverted-index Construction with s2^k

We now present our length-sorted approach to document-level inverted-index construction with s2^k. At a conceptual level inverted-index construction in s2^k can be divided into four stages: tokenization, length sorting, per-document index construction, and per-document index merging. Tokenization transforms an input text stream into a token stream with tokens defined with a set of regular expressions. Length sorting partitions the token stream into an set of length-partitioned token streams. Our prior work indicates that length-partitioning tokens using bitstream techniques can reduce the cost of hashing and variable-length string comparisons [17]. Thus, we length-partition tokens prior to the construction of per-document indexes. Next, document-index construction transforms token streams into per-document inverted-indexes. Finally, document-index merging fuses per-document inverted-indexes into a single inverted-index for the collection.

Prior to describing the s2^k implementation we first describe our length-sorting techniques. We then present the s2^k implementation at two levels. First at the stream-graph level to show the overall structure of the s2^k program and then at the filter-level. A complete program listing for this case study is shown in Appendix E.3.

6.2.1 Length-partitioning

Length partitioning groups tokens into length classes using parallel bitstream techniques. We present three partitioning strategies termed *identity length-partitioning*, *logarithmic length-partitioning*, and *division length-partitioning*. We characterize each strategy by a function that partitions tokens into length groups. For example, the identity partitioning strategy groups tokens of length k into length groups defined by the identify function, $f(k) = k$.

The key idea behind our approach is that given bitstreams S and F that mark the start and follow positions of tokens, we can partition S and F into sets of bitstreams $\{S_1, ..., S_k\} \cup S_{>k}$ and $\{F_1, ..., F_k\} \cup F_{>k}$, where S_i and F_i, mark the start and follows positions of tokens in the ith length group for $1 \leq i \leq k$, and where $S_{>k}$ and $F_{>k}$ mark start and follow positions of a final group that completes the partitioning of tokens on length.

We use the following notations in our description of the length-partitioning strategies. S and F denotes bitstreams that mark token start and follow positions of tokens irrespective of length group. S_i denotes a bitstream that marks the start positions of tokens in the ith length group, and $S_{>i}$ denotes a bitstream that marks the start positions of tokens of length greater than the ith length group. R_i denotes a bitstream mask used for partitioning R_i from $R_{>i}$ at each step of the partitioning procedure. R_i is defined in terms of the bitstream $LookAheadN$ operation and logical operations on F. Similarly, F_i denote a bitstream that marks the follow positions of tokens within the ith length group, and $F_{>i}$ denotes a bitstream that marks the follow positions of tokens of length greater than the ith length group. N_i denotes a bitstream mask used for partitioning F_i from $F_{>i}$ at each step of the calculation. N_i is defined in terms of the bitstream $AdvanceN$ operation and logical operations on S. We use the notation $n(S, c)$ to denote the $AdvanceN(S, c)$ operation, and $r(F, c)$ to denote the $LookAheadN(F, c)$ operation as well as bitstreams S and F and integer constant c. In essence, the operations $n(S, c)$ and $r(F, c)$ forward-shift and reverse-shift an input bitstream by c positions respectively.

Identity Length-partitioning Identity length-partitioning sorts tokens into $n + 1$ disjoint length groups $P_1, P_2, \ldots, P_n, P_{>n}$ with the first n groups defined by the function $f(n) = n$. The identity strategy independently generates start and follow positions bitstreams. To produce partitioned start position bitstreams we apply the following pattern of operations.

$$
\begin{aligned}
R_1 &= r(F, 1), \\
S_1 &= R_1 \wedge S, \\
S_{>1} &= S \wedge \neg S_1, \\
R_i &= r(R_{i-1}, 1), \\
S_i &= R_i \wedge S_{>(i-1)}, \\
S_{>i} &= S_{>(i-1)} \wedge \neg S_i \quad \text{for } i \geq 2.
\end{aligned}
$$

Here, the $LookAhead$ operation together with bitwise logic marks start positions. For example, we first partition S into the bitstreams S_1 marking token start positions of length 1, and the bitstreams $S_{>1}$ marking token start position of length greater than 1.

$$
\begin{aligned}
R_1 &= r(F, 1) \\
S_1 &= R_1 \wedge S \\
S_{>1} &= S \wedge \neg S_1
\end{aligned}
$$

Next, we produce the bitstreams S_2 marking token start positions of length 2, and the bitstreams $S_{>2}$ marking token start position of length greater than 2.

$$
\begin{aligned}
R_2 &= r(R_1, 1) \\
S_2 &= R_2 \wedge S_{>1} \\
S_{>2} &= S_{>1} \wedge \neg S_2
\end{aligned}
$$

Repeating this pattern k times we produce the bitstreams S_k marking token start positions of length k, and the bitstreams $S_{>k}$ marking token start position of length greater than k. Similarly, to partition follow positions we apply the following pattern of operations. In this case, we use the $Advance$ operation together with bitwise logic to mark follow positions [1].

$$
\begin{aligned}
N_1 &= n(S, 1), \\
F_1 &= N_1 \wedge F, \\
F_{>1} &= F \wedge \neg F_1, \\
N_i &= n(N_{i-1}, 1), \\
F_i &= N_i \wedge F_{>(i-1)}, \\
F_{>i} &= F_{>(i-1)} \wedge \neg F_i \quad \text{for } i \geq 2.
\end{aligned}
$$

Division Length-partitioning Division length-partitioning groups tokens into $n+1$ disjoint length groups $P_0, P_1, \ldots, P_n, P_{>n}$ with the first n groups defined by the function $f(n) = \lceil n/2 \rceil$.

To produce partitioned start positions we apply the following pattern of operations.

[1] In Identity Length-Partitioning F_i is simply $n(S_i, i)$ and thus the N_i is strictly not necessary, however, to remain consistent in our description of each strategy we include the calculation of N_i.

$$
\begin{aligned}
R_1 &= r(F,1) \vee F, \\
S_1 &= R_1 \wedge S, \\
S_{>1} &= S \wedge \neg S_1, \\
R_i &= r(R_{i-1},2), \\
S_i &= R_i \wedge S_{>(i-1)}, \\
S_{>i} &= S_{>(i-1)} \wedge \neg S_i \quad \text{for } i \geq 2.
\end{aligned}
$$

To produce partitioned follow positions we apply the following pattern of operations.

$$
\begin{aligned}
N_1 &= n(S,1) \vee S, \\
F_1 &= N_1 \wedge F, \\
F_{>1} &= F \wedge \neg F_1, \\
N_i &= n(N_{i-1},2), \\
F_i &= N_i \wedge F_{>(i-1)}, \\
F_{>i} &= F_{>(i-1)} \wedge \neg F_i \quad \text{for } i \geq 2.
\end{aligned}
$$

Logarithmic Length-partitioning Logarithmic length-partitioning groups tokens into $n+1$ length groups $P_0, P_1, \ldots, P_n, P_{>n}$ with the first n groups defined by the function $f(n) = \lceil \log_2(n) \rceil + 1$.

In the logarithmic method, we again independently generate start and follow positions bitstreams. To produce partitioned start position bitstreams we apply the following pattern of operations.

$$
\begin{aligned}
R_1 &= r(F), \\
S_1 &= R_1 \wedge S, \\
S_{>1} &= S \wedge \neg S_1, \\
R_i &= r(R_{i-1}, 2^{i-2}) \vee R_{i-1}, \\
S_i &= R_i \wedge S_{>(i-1)}, \\
S_{>i} &= S_{>(i-1)} \wedge \neg S_i \qquad \text{for } i \geq 2.
\end{aligned}
$$

To produce partitioned follow positions we apply the following pattern of operations.

$$
\begin{aligned}
N_1 &= n(S), \\
F_1 &= N_1 \wedge F, \\
F_{>1} &= F \wedge \neg F_1, \\
N_i &= n(N_{i-1}, 2^{i-2}) \vee N_{i-1}, \\
F_i &= N_i \wedge F_{>(i-1)}, \\
F_{>i} &= F_{>(i-1)} \wedge \neg F_i \qquad \text{for } i \geq 2.
\end{aligned}
$$

Additional partitioning schemes are possible. For example, a hybrid approach that applies the division strategy on higher frequency short tokens together with the logarithmic strategy for lower frequency longer tokens can improve the distribution of tokens.

Figure 6.3 illustrates the partitioning of an input text of white-space-delimited words into length-partitioned start and follow position streams using our logarithmic strategy. The figure heading shows an input data stream that contains several white-space-delimited words. A right arrow indicates the input stream is read left-to-right. We use C for alphabetic characters and W for whitespace characters, 0's are represented with a period. The first two lines mark character class positions. The next two lines mark the start and follow positions of each word (spans of non-whitespace characters). S denotes a starts stream and F denotes a follows stream. [2] The remaining statements are grouped into triples. Each group of statements produces mask streams and partitioned starts and follows streams. The first and second groups produce length 1 starts and follows streams, S_1 and F_1. The third and fourth groups produce length 2 starts and follows streams, S_2 and F_2. The fifth and sixth groups produces lengths 3 to 4 starts and follows streams, $S_{3,4}$ and $F_{3,4}$. The final pair of groups produces lengths 5 to 8 starts and follows streams, $S_{5,8}$ and $F_{5,8}$. The streams $S_{>5,8}$ and $F_{>5,8}$ mark any remaining tokens of length greater than 8 although none are present in this example.

[2] The term *starts stream* refers to a 1-bit property streams that marks the start positions of each word. Likewise, the term *follows stream* refers to 1-bit property stream that marks the position immediately each word.

8-bit ASCII Data ▷	`in the big old gown in the big old house`
$C = characters$	`11.111.111.111.1111.11.111.111.111.11111.`
$W = whitespace$	`..1...1...1...1....1..1...1...1...1.....1`
$S = n(\neg W) \wedge W$	`1..1...1...1...1....1..1...1...1...1.....`
$F = n(W) \wedge (\neg W)$	`..1...1...1...1....1..1...1...1...1.....1`
$R_1 = r(S, 1)$	`.1...1...1...1....1..1...1...1...1.....1.`
$S_1 = R_1 \wedge S$	`.....`
$S_{>1} = S_{>1} \wedge \neg S_1$	`1..1...1...1...1....1..1...1...1...1.....`
$N_1 = n(S, 1)$	`.1..1...1...1...1....1..1...1...1...1....`
$F_1 = N_1 \wedge F$	`.....`
$F_{>1} = F_{>1} \wedge \neg F_1$	`..1...1...1...1....1..1...1...1...1.....1`
$R_2 = r(N_1, 1) \vee N_1$	`11..11..11..11...11.11..11..11..11....11.`
$S_2 = R_2 \wedge S_{>1}$	`1.....1.....`
$S_{>2} = S_{>1} \wedge \neg S_2$	`...1...1...1...1.....1...1...1...1.....`
$N_2 = (N_1, 1) \vee N_1$	`.11.11..11..11..11...11.11..11..11..11...`
$F_2 = N_2 \wedge F_{>1}$	`..1.....1.....`
$F_{>2} = F_{>1} \wedge \neg F_2$	`.....1...1...1....1.....1...1...1.....1`
$R_{3,4} = r(R_2, 2) \vee R_2$	`11111111111111.111111111111111111..1111.`
$S_{3,4} = R_{3,4} \wedge S_{>2}$	`...1...1...1...1.....1...1...1.....`
$S_{>3,4} = S_{>2} \wedge \neg S_{3,4}$	`.....1.....`
$N_{3,4} = n(N_2, 2) \vee N_2$	`.111111111111111111.111111111111111111.`
$F_{3,4} = N_{3,4} \wedge F_{>2}$	`.....1...1...1....1.....1...1...1.....`
$F_{>3,4} = F_{>2} \wedge \neg F_{3,4}$	`.....1`
$R_{5,8} = r(R_3, 4, 4) \vee R_{3,4}$	`1111111111111111111111111111111111111111.`
$S_{5,8} = R_{5,8} \wedge S_{>3,4}$	`.....1.....`
$S_{>5,8} = S_{>5,8} \wedge \neg S_{3,4}$	`.....`
$N_{5,8} = n(N_{3,4}, 4) \vee N_{3,4}$	`.11111111111111111111111111111111111111111`
$F_{5,8} = N_{5,8} \wedge F_{>3,4}$	`.....1`
$F_{>5,8} = F_{>3,4} \wedge \neg F_{5,8}$	`.....`

Figure 6.3: Logarithmic length-partitioning of an input text stream.

6.2.2 Stream-graph Level Implementation.

In our approach to inverted-index construction, tokenization and length sorting are expressed on a per document basis with the $s2^k$ language; document-index construction and document-index merging are expressed with a host language. The interface between the $s2^k$ language output and the host language is defined by the binding of $s2^k$ variables with a set of host data buffers termed an array set. Each array holds length-sorted tokens padded and aligned with null bytes. A final array of packed arbitrary length tokens is indexed with parallel start and follow positions arrays. In document inversion and merging, the host language iterates through these arrays, constructing per-document indexes, and fusing per-document indexes into a single global inverted-index. The implementation per-document inverted lists and merging with a host language are straightforward. We thus omit any further description of these stages.

Figure 6.4 shows stream-graph of the $s2^k$ inverted-index construction program. As before, the vertices of the stream-graph denote Fieldstream filters, and the edges denote Fieldstream structures. Document inversion begins at the $s2^k$ *Read* filter. The *Read* filter transfers a document text stream from the host language space to the $s2^k$ program space. Next, the *Tokenize* filter produces token starts and follows streams defined here by a simple regular expression. In addition, the Tokenize filter initializes mask streams for logarithmic length-partitioning. Next, the filter chain *Mark_S_1*, *Mark_S_2*, *Mark_S_3_4*, and *Mark_S_5_8* partitions token start positions into sets of length-sorted starts streams. Likewise, the filter chain *Mark_F_1*, *Mark_F_2*, *Mark_F_3_4*, and *Mark_F_5_8* partitions token follows streams. The *Gather* filter then performs parallel gather operations based on length-partitioned starts and follows streams. Finally, the *Write* filters writes the set of length-sorted gather streams to the host language space. Listing 6.1 shows the overall structure of the $s2^k$ program listing for Figure 6.4.

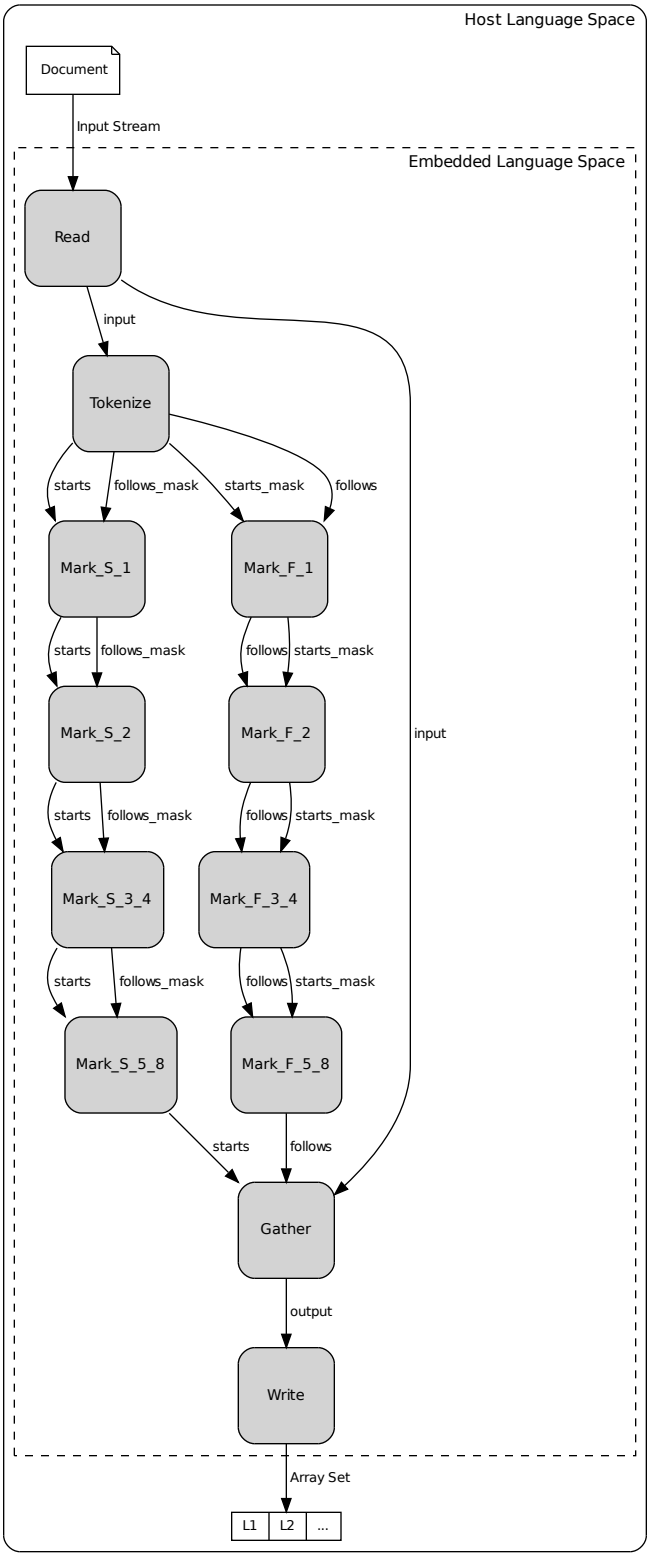

Figure 6.4: Stream-graph for length-sorted inverted-index construction.

```
/* Stream Structure Definitions */
struct Input { ... }
struct Output { ... }
struct TokenStarts { ... }
struct TokenFollows { ... }
struct FollowsMask { ... }
struct StartsMask { ... }

/* Filter Definitions */
filter Read(out: struct Input input) { ... }
filter Tokenize(in: struct Input input,
                in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                in: TokenFollows follows,      out: struct FollowsMask starts_mask) { ... }
filter Mark_S_1(in: struct TokenStarts starts,  out: struct TokenStarts starts,
            in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) { ... }
filter Mark_S_2(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                in: TokenFollows follows,      out: struct FollowsMask starts_mask) { ... }
filter Mark_S_3_4(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                  in: TokenFollows follows,       out: struct FollowsMask starts_mask) { ... }
filter Mark_S_5_8(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                  in: TokenFollows follows,       out: struct FollowsMask starts_mask) { ... }
filter Mark_F_1(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                in: TokenFollows follows,      out: struct FollowsMask starts_mask) { ... }
filter Mark_F_2(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                in: TokenFollows follows,      out: struct FollowsMask starts_mask) { ... }
filter Mark_F_3_4(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                  in: TokenFollows follows,       out: struct FollowsMask starts_mask) { ... }
filter Mark_F_5_8(in: struct TokenStarts starts, out: struct FollowsMask follows_mask,
                  in: TokenFollows follows,       out: struct FollowsMask starts_mask) { ... }
filter Gather(in: struct Input input,
              in: struct TokenStarts starts, in: struct TokenFollows follows,
              out: struct Output output) { ... }
filter Write(in: struct Output output) { ... }

graph pipeline() {

  // structure declarations ...

  // filter pipeline
  read(input);
  tokenize(input, starts, follows_mask, follows, starts_mask);
  mark_S_1(starts, follows_mask);
  mark_S_2(starts, follows_mask);
  mark_S_3_4(starts, follows_mask);
  mark_S_5_8(starts, follows_mask);
  mark_F_1(follows, starts_mask);
  mark_F_2(follows, starts_mask);
  mark_F_3_4(follows, starts_mask);
  mark_F_5_8(follows, starts_mask);
  gather(input, starts, follows, output);
  write(output);
}
```

Program 6.1: Stream-graph level program listing for $s2^k$ length-sorted inverted-index construction.

6.2.3 Filter-level Implementation

In this section we describe the s2^k filter-level implementation. We present s2^k language features throughout support programmer productivity and performance.

Read. Listing 6.2 shows the implementation of the *Read* filter. Lines 1 to 8 bind host data arrays to s2^k variables. Line 10 transfers an input text stream from the host language space to the s2^k language space. The expression of bulk stream transfers provides flexibility for the system to pre-dimension input and output buffers and use local memory efficiently.

```
1   StreamBind<8> (input.data, buffer);
2   StreamBind<8> (output.L_1, L_1);
3   StreamBind<16>(output.L_2, L_2);
4   StreamBind<32>(output.L_3_4, L_3);
5   StreamBind<32>(output.L_5_8, L_5);
6   StreamBind<8> (output.L_gt_8, L_gt_8);
7   StreamBind<32>(output.L_gt_8_S, L_gt_8_S);
8   StreamBind<32>(output.L_gt_8_F, L_gt_8_F);
9
10  StreamRead<8>(input.data);
```

Program 6.2: s2^k Read filter for inverted-index construction.

Tokenize. Listing 6.3 shows the implementation of the *Tokenize* filter. Line 1 shows the *CharClass* (character class) operation applied to an input data stream. Given a source stream $input.data$ and a regular expression the *CharClass* operation returns a bitstream L marking each position at which a letter occurs. Line 2 and Line 3 mark the start and follow positions of each token. Lines 4 and 5 initialize masks for length-partitioning.

Listing 6.3 demonstrates that s2^k program statements can be expressed concisely and in a straightforward manner. For example, the identification of all character class positions within a data stream can be expressed with a single operation. This conciseness arises from the use of abstract global-view operations. Furthermore, simple programming abstractions such as marker streams allow programmers to reason about stream operations without consideration of lower-level implementation details.

```
1  stream<1> L      = CharClass(input.data, "[\p{L}]");
2  starts.S         = Advance(~L) & L;
3  follows.F        = Advance(L) & ~L;
4  follows_mask.R_0 = follows.F;
5  starts_mask.N_0  = starts.S;
```

Program 6.3: s2^k Tokenize filter for inverted-index construction.

Mark Start and Follow Positions. Listings 6.4 and 6.5 show the implementation of the *Mark_S_1* and the *Mark_F_1* filters. These filters demonstrate the length sorting techniques described in Section 6.2.1 expressed in s2^k. The correspondence between the abstract operations with the concrete operations is straightforward and direct.

```
follows_mask.R_1 = LookAheadN(follows_mask.R_0, 1);
starts.S_1       = mask.R_1 & starts.S;
starts.S_gt_1    = starts.S &~ starts.S_1;
```

Program 6.4: s2^k Mark_S_1 filter for inverted-index construction.

```
starts_mask.N_1  = AdvanceN(starts_mask.N_0, 1);
follows.F_1      = starts_mask.N_1 & follows.F;
follows.F_gt_1   = follows.F &~ follows.F_1;
```

Program 6.5: s2^k Mark_F_1 filter for inverted-index construction.

Gather. Listing 6.6 shows the implementation of the *Gather* filter. Lines 1 to 5 produce a span stream for each token length group. Lines 7 to 10 gather, align, and pad tokens with null bytes for postprocessing. In this example, the set of span streams serve as index streams to the s2^k gather operations. Lines 12 and 13 produce integer index values to access the start and follow positions of the final length group. The *Gather* filter shows that s2^k supports the gather-and-compute style of stream programming such as Brook [10], ZPL [81], and HPF [63], which benefits programmer productivity and application performance.

```
1  stream<8>  spans_1         = SpanUpTo(starts.S_1, follows.F_1);
2  stream<16> spans_2         = SpanUpTo(starts.S_2, follows.F_2);
3  stream<32> spans_3_4       = SpanUpTo(starts.S_3_4, follows.F_3_4);
4  stream<64> spans_5_8       = SpanUpTo(starts.S_5_8, follows.F_5_8);
5  stream<8>  spans_gt_5_8    = SpanUpTo(starts.S_gt_5_8, follows.F_gt_5_8);
6
7  output.L_1                 = PGatherSpansLo<8,8>(input.data, spans_1);
8  output.L_2                 = PGatherSpansLo<8,16>(input.data, spans_2);
9  output.L_3_4               = PGatherSpansLo<8,32>(input.data, spans_3_4);
10 output.L_5_8               = PGatherSpansLo<8,64>(input.data, spans_5_8);
11 output.L_gt_5_8            = PGather<8>(input.data, spans_gt);
12 output.output.L_gt_8_S     = Positions(starts.S_gt, 32);
13 output.output.L_gt_8_F     = Positions(follows.F_gt, 32);
```

Program 6.6: s2^k Gather filter for inverted-index construction.

Write. Listing 6.7 shows the implementation of the *Write* filter. Lines 1 to 7 write the results of the $s2^k$ operations to a host language space. Each target array is associated with a particular property of the text stream. In this case token start and follow positions. We term this data organization an Array Set Model (ASM) representation [12].

The association of set of parallel arrays with a set of $s2^k$ variables provides a number of benefits. To conclude this subsection, we list these benefits.

1. Prefetching. Processors perform software and hardware prefetching to ensure that data is available in the processor cache when required. In general, prefetching is most effective in conjunction with the regular memory accesses of processing long data arrays [44].

2. Bulk Memory Operations. In processing environments such as CPU-GPU platforms independent and clearly defined data streams allow system software to schedule bulk data transfers, thereby increasing cache and memory bus efficiency.

3. Structure of Arrays (SOA). While Array of Structures (AOS) data organization may be appropriate for encapsulation it is generally not suitable for vector processing [50]. The $s2^k$ array set model supports SOA data organization for vector processing without additional data organization steps.

4. Multicore Processing. The processing of sets of arrays can enable the distribution of work to the individual cores of a multicore system. Provided that sequential dependencies can be minimized, data arrays can be segmented and processed in parallel. Alternatively, software pipelining strategies can be used when dependencies cannot be overcome.

```
1  StreamWrite<8>(output.L_1);
2  StreamWrite<16>(output.L_2);
3  StreamWrite<32>(output.L_3_4);
4  StreamWrite<64>(output.L_5_8);
5  StreamWrite<8>(output.L_gt);
6  StreamWrite<8>(output.L_gt_S);
7  StreamWrite<8>(output.L_gt_F);
```

Program 6.7: $s2^k$ Write filter for inverted-index construction.

6.3 Chapter Summary

In application to the problem of inverted-index construction, the $s2^k$ programming language is effective and productive. Using only $s2^k$ collective operations it is possible to express the stages of tokenization, length-sorting, and results gathering in inverted-index construction. Token hashing is

deferred to postprocessing where the processing of packed arrays of fixed width values is efficient. Likewise, per-document indices are merged in postprocessing to form a single global index of all terms which appear in all documents. Overall, our approach suggests a clean and well-defined architecture with distinct parallel and sequential phases. A clean separation of phases supports data locality and provides flexibility in the processing granularity of input streams.

Chapter 7

Conclusion

7.1 Conclusions

This dissertation defines the s2^k programming language for streaming text extraction and transformations. The s2^k language design demonstrates that the integration of stream programming concepts with parallel bitstream programming methods in the design of a parallel programming language for streaming text extraction and transformations can simplify programming and maintain application performance. The s2^k language unburdens programmers from the many low-level details of parallel streaming text programming by abstracting over processor features such as SIMD register width. In s2^k, the programmer need not be concerned with issues of data alignment, data movement, boundary crossing, or block-at-a-time implementation strategies. Furthermore, through the provision of operations defined over sequences of 2^k-bit fields, the s2^k language supports the expression of a broader class of applications than previously possible with the Pablo language and Parabix application framework. We support these claims with the following contributions.

1. We define the s2^k programming language and programming model for streaming text extraction and transformations. s2^k provides novel language features for computation and program control flow. Language features include, but are not limited to, marker streams, index streams, 1-bit flow-parallel operations, and 2^k-bit field-parallel operations. In addition, the s2^k design supports parallel if, parallel while, and sequential foreach constructs for program control flow, as well as, for..in expressions for the expression of inductive-doubling algorithms. Overall, the s2^k language design simplifies parallel text programming and allows implementations to scale s2^k operations with SIMD register widths. Furthermore, the s2^k language makes explicit the producer-consumer relationships of the stream graph representation of a program, eliminating the reliance on hand-coded templates for code generation, and providing additional opportunities for task and data parallelism on multicore processors.

2. We define the $b2^k$ intermediate language and processing model for the block-at-a-time processing of Fieldstream operations. The design of $b2^k$ language is simple and flexible. $b2^k$ implementations are not restricted to specific block-at-a-time scheduling strategies nor particular carry-value propagation strategies. The semantics of the $b2^k$ processing model are well-defined. Taken together, the $s2^k$ programming model and the $b2^k$ processing model provide a sound basis for the translation of the $s2^k$ language to the $b2^k$ intermediate language.

3. We define a source-to-source translation scheme for the translation of $s2^k$ to $b2^k$, as well as a translation scheme from $b2^k$ to C, C++, or LLVM IR for compilation to machine code.

4. We present example $s2^k$ programs that allows programmers to express parallel streaming text processing algorithms in a portable manner and without concern for the details of block-at-a-time processing.

7.2 Limitations

$s2^k$ is a domain-specific language and thus by definition restricts programmers to the domain of streaming text processing. As such, problems that can be easily expressed with $s2^k$ are ideal candidates. Applications without natural stream processing abstractions or that require frequent interaction with a host language are considerably more difficult express with $s2^k$ and may perform poorly. A further limitation of the $s2^k$ language design corresponds to the limited breadth of benchmark applications. This benchmark set focuses on validation, transformation and extraction problems that involves document formats definable by regular grammars. Further exploration and development of benchmark application areas would likely lead to improvements in the design of $s2^k$.

7.3 Future Directions and Work

There are a number of areas where $s2^k$ can be extended. Various extensions were identified throughout the design and implementation of $s2^k$. For instance, support for arbitrary-length lookahead, recursive data formats, and multicore scheduling strategies are future goals of $s2^k$. In other cases, deficiencies in the language were identified as the design and implementation of $s2^k$ progressed such as improvements in library interfaces and the translation of $s2^k$ to LLVM IR.

Arbitrary-length Lookahead. A long term goal of $s2^k$ is the support of arbitrary-length stream lookahead. However, a number of challenges arise such as (i) the efficient management of memory and (ii) complexity in the design and implementation of dynamic scheduling strategies for multiple block lookahead.

Recursive Data Formats. Parallel bitstream methods are well-suited to the processing of free or semi-structured text formats that are definable by regular grammars. A goal of $s2^k$ is the definition and implementation of language-level support for processing text formats not definable by regular grammars such as XML and JSON. For example, whereas the regular aspects of XML parsing such as character validation and XML tag-scanning are supported by $s2^k$ [17, 15], non-regular features must be processed with a general purpose language such as C++.

Multicore Scheduling. The stream programming paradigm is suitable for the of programming of multicore processors [55]. Programming languages such as Brook and StreamIt that define explicit producer-consumer filter relationship leverage this information to schedule parallel executions.

LLVM Backend. The $s2^k$ compiler translates $s2^k$ to $b2^k$ and then $b2^k$ to C++ for compilation into machine code. This approach supports the reuse of C++ optimizers and code generators and supports efficient implementation. However, treating the backend C++ compiler as a black box prevents the specification of the lower-level optimization passes available with compiler frameworks such as GCC[3] or LLVM [59]. At this time, the LLVM framework [85, 55, 74] is considered superior since the library-based design of LLVM allows implementors to organize backend optimizations for specific audiences. Hence, translation of $s2^k$ to LLVM IR is a reasonable objective.

Bibliography

[1] IEEE standard for software productivity metrics. *IEEE Std 1045-1992*, 1993.

[2] *SSE4.2 SIMD Programming For Text-Processing/Lexing/Parsing*, volume 1 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, chapter 10, page 289. 2006. 9, 10

[3] GCC, the GNU compiler collection., January 15, 2015 2015. 99

[4] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. Structure and interpretation of computer programs, 1996. 18

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 2007. 74

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006. 1, 2, 10

[7] Guy E. Blelloch. Prefix sums and their applications. 1990. 2

[8] Guy E. Blelloch. NESL: A nested data-parallel language.(version 3.1). Technical report, DTIC Document, 1995. 19

[9] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of parallel and distributed computing*, 21(1):4--14, 1994. 19

[10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777--786. ACM, 2004. 3, 4, 7, 11, 12, 13, 14, 15, 18, 25, 94

[11] R. D. Cameron and D. Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 337--348. ACM, 2009. 1, 2, 15, 27, 44, 48

[12] Rob Cameron, Ken Herdy, and Ehsan Amiri. Parallel bit stream technology as a foundation for XML parsing performance. In *International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, volume 8, 2009. 5, 15, 95

[13] Robert D. Cameron. Pablo: A language and compiler for parallel bit stream programming.

[14] Robert D. Cameron. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 91--98. ACM, 2008. 1, 4, 5, 10, 15, 16, 41, 44

[15] Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011 Parallel Processing*, pages 2--13. Springer, 2011. 5, 6, 11, 15, 16, 20, 30, 31, 53, 99, 108

[16] Robert D. Cameron and Anthony H. Dixon. *Symbolic computing with Lisp*. Prentice-Hall, Inc, 1992. 74

[17] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance XML parsing using parallel bit stream technology. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 17. ACM, 2008. 1, 4, 5, 9, 10, 15, 16, 20, 84, 99

[18] Robert D. Cameron, Kenneth S. Herdy, Dan Lin, Meng Lin, Ben Hull, Thomas S. Shermer, and Arrvindh Shriraman. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd international conference on Parallel architectures and compilation techniques*, 2014. 30, 31, 53, 60

[19] Robert D. Cameron and Hua Huang. IDISA toolkit project.

[20] David Cann. Retire Fortran? a debate rekindled. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 264--272. ACM, 1991. 57

[21] David C. Cann. Vectorization of an applicative language: Current results and future directions. In *Compcon Spring'91. Digest of Papers*, pages 396--402. IEEE, 1991.

[22] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP*, volume 11, pages 47--56, 2010. 18, 19, 22

[23] Bradford L. Chamberlain. IEEE technical committee on scalable computing (TCSC) myths about scalable parallel programming languages part 1: Productivity and performance. 3

[24] Bradford L. Chamberlain. IEEE technical committee on scalable computing (tcsc) myths about scalable parallel programming languages part 6: Performance of higher-level languages. 3, 22

[25] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291--312, 2007. 3, 17, 22

[26] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, pages 66--75. Citeseer, 2004. 3, 17, 22

[27] Bradford L. Chamberlain, Sung-Eun Choi, C. Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering, IEEE Transactions on*, 26(3):197--211, 2000.

[28] Norman H Cohen. *Ada as a second language*. McGraw-Hill Higher Education, 1995. 32

[29] Mohamad Dabbagh. Introducing a new language for stream applications. 4, 11

[30] William J Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003. 18

[31] Mark Davis and A. Heninger. Unicode technical standard 18, Unicode regular expressions. *The Unicode Consortium*, 2012. 41

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107--113, 2008. 10, 82

[33] Zhao Hui Du, Zhenying Liu, Mohan Rajagopalan, Byoungro So, and Zhi Gang Wang. A flexible parallel programming model for tera-scale architectures. 2007. 4, 20

[34] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus:(via language support for architectures and compilers). In *ACM SIGPLAN Notices*, volume 47, pages 1--12. ACM, 2012. 12

[35] R. Kent Dybvig. *The SCHEME programming language*. 2003. 18

[36] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 93--104. ACM, 2007. 4, 10, 13

[37] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Lapresté Jean-Thierry. Boost. SIMD: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing*, pages 1--8. ACM, 2014.

[38] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-proof data parallel algorithms and software on Intel multi-core architecture. *Intel Technology Journal*, 11(4):333--348, 2007. 20

[39] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151--162. ACM, 2006. 3, 11, 12

[40] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGPLAN Notices*, volume 37, pages 291--303. ACM, 2002. 14

[41] Michael I. Gordon. Compiler techniques for scalable performance of stream programs on multicore architectures, May 2010. Ph.D. Thesis. 12

[42] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12--pp. IEEE, 2005. 18

[43] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851--876, 2007. 2

[44] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. 95

[45] K. S. Herdy, D. S. Burggraf, and R. D. Cameron. High performance GML to SVG transformation for the visual presentation of geographic data in web-based mapping systems. In *Proceedings of SVG Open*, 2008. 16

[46] Armando Faz Hernández. Yet another survey on SIMD instructions. 2013. 4

[47] Mark D. Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18--21, 1990.

[48] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170--1183, 1986. 2, 44

[49] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35--35. IEEE, 2005.

[50] INC INTEL. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3b: System Programming Guide (Part 2)*, 2013. 4, 95

[51] Knezovic Josip, Kovac Mario, and Mlinaric Hrvoje. Integrating streaming computations for efficient execution on novel multicore architectures. *Automatika--Journal for Control, Measurement, Electronics, Computing and Communications*, 51(4), 2011. 11, 18, 22

[52] Ken Kennedy, Charles Koelbel, and Robert Schreiber. Defining and measuring the productivity of programming languages. *International Journal of High Performance Computing Applications*, 18(4):441--448, 2004.

[53] Josip Knezovic. Streaming model of computation for image and video processing.

[54] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261--301, 1974. 74

[55] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *ACM SIGPLAN Notices*, volume 43, pages 114--124. ACM, 2008. 99

[56] Rajendra Kumar. *Theory Of Automata*. Tata McGraw-Hill Education, 2010. 10

[57] Susana Ladra, Oscar Pedreira, Jose Duato, and Nieves R. Brisaboa. Exploiting SIMD instructions in current processors to improve classical string algorithms. 10

[58] Chris Lattner. *LLVM*, chapter 11, pages 155--170. The Architecture of Open Source Applications. 2011.

[59] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75--86. IEEE, 2004. 74, 99

[60] Dan Lin, Nigel Medforth, Ken Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the 18rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '18, Louisianna, USA, 2012. IEEE Computer Society. 2, 4, 5, 6, 7, 15, 16, 17, 20, 60

[61] Chris Lomont. Introduction to Intel advanced vector extensions. In *Proceedings of the 2nd Annual ASCI Conference*, pages 132--137, 2011. 4

[62] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*, 24(1):121--145, 1977. 74

[63] David B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25--42, 1993. 94

[64] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(01):89--102, 2007.

[65] Raymond Manley and David Gregg. *Mapping streaming languages to general purpose processors through vectorization*, pages 95--110. Languages and Compilers for Parallel Computing. Springer, 2010. 18

[66] Michael D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816--831, 2008. 6, 25, 26

[67] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 5--5. USENIX Association, 2010. 9, 18, 22

[68] Nigel Medforth, Dan Lin, Kenneth Herdy, Rob Cameron, and Arrvindh Shriraman. icXML: Accelerating a commercial XML parser using SIMD and multicore technologies. In *Balisage: The Markup Conference 2013*, volume 10, 2013. 1, 4, 5, 11, 16, 17, 43

[69] Chris J Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, et al. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on*, pages 224--235. IEEE, 2011. 1, 7, 9, 18, 20, 22, 25

[70] David Padua. *Encyclopedia of parallel computing*, volume 4. Springer, 2011. 32

[71] David A. Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, 6:30, 2013. 73

[72] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc, 2010. 10

[73] Guido Van Rossum. Python programming language - official website, September 15th, 2006. 2

[74] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. ErLLVM: an LLVM backend for Erlang. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 21--32. ACM, 2012. 99

[75] N. R. Satish and Y. Yi. Finite state machine pattern. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/finitestatemachine.pdf. 10

[76] D. P. Scarpazza and G. W. Braudaway. Workload characterization and optimization of high-performance text indexing on the Cell Broadband Engine (Cell/BE). In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 13--23. IEEE, 2009. 4

[77] Daniele Paolo Scarpazza. High-performance, data-parallel document inversion for the Cell/BE processor. *world*, 1:11.

[78] Eric Schweitz, Peter Szilagi, and Richard Lethin. R-Stream: Enabling efficient development of portable, high-performance, parallel applications peter mattson, allen leung, ken mackenzie. 12, 14

[79] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Performance impact of misaligned accesses in simd extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, pages 334--342, 2006. 9

[80] Guangyu Shi, Min Li, and Mikko Lipasti. Accelerating search and recognition workloads with SSE 4.2 string and text processing instructions. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 145--153. IEEE, 2011. 11

[81] Lawrence Snyder. The design and development of ZPL. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 8--1. ACM, 2007. 19, 94

[82] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491--541, 1997. 18

[83] Jon Stokes. Peakstream unveils multicore and CPU/GPU programming solution. *Retrieved April*, 17(2008):20060918--7763, 2006. 11

[84] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25--35, 2002. 18

[85] David A. Terei and Manuel MT Chakravarty. An LLVM backend for GHC. In *ACM Sigplan Notices*, volume 45, pages 109--120. ACM, 2010. 74, 99

[86] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 49--84. Springer, 2002. 3, 4, 7, 11, 12, 13, 14, 18, 19, 22

[87] William Thies. Language and compiler support for stream programs, Feb 2009. Ph.D. Thesis. 4, 11, 18

[88] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 365--376. ACM, 2010. 3, 4

[89] William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe. Streamit: A compiler for streaming applications. 2002. 11, 13

[90] Adam Trendowicz and Jurgen Munch. Factors influencing software development productivity - state-of-the-art and industrial experiences. *Advances in computers*, 77:185--241, 2009.

[91] Ben Wegbreit. Goal-directed program transformation. *Software Engineering, IEEE Transactions on*, (2):69--80, 1976. 73, 74

[92] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006. 82, 83

Appendix A

Glossary of Terms and Notations

A.1 General Terms and Notations

SIMD. In Flynn's taxonomy, the term SIMD (Single Instruction Multiple Data) refers to a vector computer architecture in which one instruction controls the execution of multiple ALUs that execute in lockstep.

SWAR. SWAR (SIMD Within A Register) refers to a range of techniques used to perform data parallel short vector operations.

SPMD. SPMD (Single Program Multiple Data) is a parallel programming model that models the execution of a single program on different processing elements with independent flow of control. The term SPMD is occasionally confused with the term SIMD, however, SIMD is a hardware classification whereas SPMD is a software classification.

A.2 s2^k Language Terms and Notations

Fieldstream. A Fieldstream, denoted T_{2^k}, is a homogeneous sequence of 2^k-bit fields for some non-negative integer k. A Fieldstream may optionally be prefixed by 2^k*-bit* to indicate the bit width of individual stream elements. The prefix 2^k is optional if brevity improves clarity.

Fieldstream Field. A Fieldstream field, denoted t_{2^k}, is an element of a Fieldstream defined as an ordered sequence of 2^k 0s and 1s for some non-negative integer k. A field may optionally be prefixed by 2^k*-bit* to indicate the bit width of a field. The prefix 2^k is optional if brevity improves clarity. In addition, the term stream may be used in place of the term bitstreamif brevity improves clarity.

Bitstream. A bitstream, denoted T_1, is a Fieldstream of 1-bit fields. In s2^k, bitstreams can be interpreted as finite length unsigned integers that can be processed as a single entities using bitwise

arithmetic, logical, and shifting operations as provided by the s2^k language definition. The subscript is optional if brevity improves clarity. In addition, the term stream may be used in place of the term bitstreamif brevity improves clarity.

Fieldstream Length. The length of a Fieldstream T_{2^k}, denoted $Length(T_{2^k})$, is the number of fields in T_{2^k}.

i^{th} Field of a Fieldstream. The ith field of a Fieldstream T_{2^k}, denoted $T_{2^k}[i]$, is the stream element at position i in T_{2^k} for $0 \leq i < Length(T)$.

j^{th} Subfield Element of the ith Field of a Fieldstream. The jth 2^c-bit subfield of the ith 2^k-bit field of a Fieldstream T_{2^k}, denoted $T_{2^k}[i][j]$, is the c-bit subfield element at position j in $T_{2^k}[i]$, for some non-negative integers c, k for $c \leq k$, $0 \leq j < (Width(T_{2^k})/c)$, and $0 \leq i < Length(T_{2^k})$.

Field Width. The width of each field of a Fieldstream T_{2^k}, denoted $Width(T_{2^k})$, is the number of bits in each field of T_{2^k}. Since Fieldstreams are homogeneous sequences of fields, the width of all fields of a Fieldstream are equal.

Fieldstream Size. The size of a 2^k-bit Fieldstream T_{2^k}, denoted $Size(T_{2^k})$, is $Length(T_{2^k}) * 2^k$.

Parallel Fieldstreams. Two or more Fieldstreams in one-to-one correspondence. Fieldstreams S_{2^k} and T_{2^j} are parallel if and only if $S[i]$ is associated with $T[i]$ by a particular property of the data values represented, $T[i]$ is associated with $S[i]$ by a particular property of the data values represented, and $Length(S_{2^k})$ equals $Length(T_{2^j})$ for some non-negative integers k and j.

Bit-space and Byte-space Processing Domains. The terms bit-space and byte-space refer to two important processing domains that frequently occur in streaming text processing using parallel stream methods. Bit-space processing refers to operations performed on the transpose representation of byte-oriented stream data, whereas byte-space processing to operations performed on non-transposed byte-oriented stream data.

Fieldstream Transposition. Given a 2^k-bit Fieldstream T_{2^k}, the transpose of T_{2^k} is the set of 1-bit Fieldstreams, denoted $t_0 \ldots t_{2^k-1}$, such that the ith bit of the jth field of T_{2^k} is the jth bit of the ith field of t_i for some non-negative integers i, j, k.

Fieldstream Inverse Transposition. Given a 2^k-bit Fieldstream T_{2^k} and the set of 1-bit Fieldstreams of the transpose of T_{2^k}, denoted $t_0 \ldots t_{2^k-1}$, the inverse stream transpose of $t_0 \ldots t_{2^k-1}$ is T_{2^k} for some non-negative integer k.

Basis Bitstream Set. Given the transpose of the 2^k-bit stream T_{2^k}, denoted $t_0 \ldots t_{2^k-1}$, where t_0 is the stream of bits consisting of bit 0 of each byte in the input byte stream, t_1 the stream of bits consisting of bit 1 of each byte in the input stream and so on. The set of streams $t_0 \ldots t_{2^k-1}$ are called the basis bitstream of T_{2^k} or simply the basis streams if brevity improves clarity.

Property Bitstream. A property bitstream or *property stream* is a 2^k-bit Fieldstream consisting of fields in one-to-one correspondence with the fields of a parallel 2^j-bit Fieldstream for some non-negative integers k and j.

Character Class Bitstream. A character class bitstream or *character class stream* is a 1-bit property stream that marks the positions of the occurrences of characters that belong to a particular character class in an associated parallel 2^k-bit Fieldstream for some non-negative integer k [15].

Marker Bitstream. A marker bitstream or *marker stream* is a 1-bit property stream that marks the positions of matches in the parallel bit stream pattern matching process [15].

Spans Bitstream. A spans bitstream or *spans stream* is a 1-bit property stream that marks contiguous runs of a property of interest. Each contiguous run of a property of interest is termed a span.

Start Positions Bitstream. A start positions bitstream or *starts stream* is a 1-bit property stream that marks the start position of each span in a spans stream.

Follow Positions Bitstream. A follow positions bitstream or *follows stream* is a 1-bit property stream that marks the position immediately following each span in a spans stream.

Appendix B

$s2^k$ Grammar

This appendix specifies the syntax of the $s2^k$ language. The grammar presented follows the EBNF conventions described in B.1.

B.1 EBNF Notations

This appendix adopts a simple EBNF (Extended Backus Naur Form) notation to describe the syntax of the $s2^k$ language. Each rule consists of three parts (1) a left-hand side, (2) a right-hand side, and (3) the symbol '::=' which separates the left-hand from the right-hand side and denotes ``is defined as''. The left-hand side of each rule contains a single nonterminal that names the rule, the right-hand side supplies the rule definition. Each rule definition consists of combinations of of the four control forms defined in Table B.1. Parenthesis, `(' `)', are used to enforce precedence in the grouping of control forms.

Control Form	Notation	Description
Sequence		Items appear left-to-right. Order is important.
Choice	'|'	Alternative items are separated by a vertical bar. One item is chosen from the list of alternatives. Order is unimportant.
Optional	'[' `]'	An optional item is enclosed between square brackets.
Repetition	'{' and '}'	A repeatable item is enclosed between curly braces. The item can be repeated zero or more times.

Table B.1: EBNF control forms.

Syntactic Category	Notation	Description
Terminals	`!'	Terminal symbols are represented as single-quoted sequences of symbols.
Keywords	`**keyword**'	Keywords are single-quoted and shown in bold-face type.
Comments	`(*' `*)'	(* an EBNF comment *)

Table B.2: Additional EBNF notations.

To improve readability, rule names, rule definitions, and '::=' symbols are vertically aligned. Additional spacing has no effect on the meaning of rules. Rules are unordered, however, whenever possible we adopt the convention of writing rules in a top-down manner with the first rule denoting the program start symbol.

B.2 s2^k Grammar Definition

This section defines the s2^k programming language grammar. The grammar is presented using the EBNF conventions described in Section B.1.

Fieldstream Program

program ::= { structureDef | functionDef | filterDef } graphDef

Fieldstream Filter Graph

graphDef ::= `**graph**' graphBody
graphBody ::= `{' structDeclStmtList `}'
structDeclStmtList ::= { structType id `;' }

Fieldstream Filter Definition

filterDef ::= `**filter**' id `(' filterParameterList `)' blockStmt
filterParameterList ::= filterParameter { ‘,’ filterParameter }
filterParameter ::= parameterMode structType id
parameterMode ::= `**in**' | `**out**'

Fieldstream Function Definition

```
functionDef            ::=  `function' returnType id parameterList functionBody
returnType             ::=  type
functionParameterList  ::=  `(' functionParameter { ‘,’ functionParameter } `)'
functionParameter      ::=  type id
functionBody           ::=  blockStmt
```

Fieldstream Structure Definition

```
structDef              ::=  `struct' id structDefBody `;'
structDefBody          ::=  `{' structDefMemberList `}'
structDefMemberList    ::=  structDefMember `;' { structDefMember `;' }
structDefMember        ::=  streamType id
```

Statements

```
stmt                   ::=  localVarDeclStmt | assignmentStmt | funcCallStmt |
                            ifStmt | whileStmt | foreachStmt
                            returnStmt | annotationStmt |
                            assertStmt
localVarDeclStmt       ::=  type id [ simpleAssignOp expr ] `;'
assignmentStmt         ::=  id assignOp expr `;'
funcCallStmtCall       ::=  funcCall `;'
ifStmt                 ::=  `if' `(' expr `)' blockStmt
whileStmt              ::=  `while' `(' expr `)' blockStmt
foreachStmt            ::=  `foreach' id `in' expr blockStmt
returnStmt             ::=  `return' [ expr ] `;'
assertStmt             ::=  `assert' `(' exprList `)' `;'
annotationStmt         ::=  `@' id `(' exprList `)' blockStmt
blockStmt              ::=  `{' { stmt } `}'
assignOp               ::=  simpleAssignOp | cmpdAssignOp
simpleAssignOp         ::=  `='
cmpdAssignOp           ::=  `+=' | `-=' | `\=' | `*=' | `|=' | `&=' | `^='
```

Expressions

```
expr                              ::=  prefixExpr | infixExpr | postfixExpr |
                                       id | cmpdId | constant |
                                       `(' expr `)'
prefixExpr                        ::=  prefixOp expr
bitwisePrefixOp                   ::=  `~'
logicalPrefixOp                   ::=  `!'
infixExpr                         ::=  expr infixOp expr
infixOp                           ::=  arithmeticInfixOp | bitwiseShiftInfixOp |
                                       bitwiseInfixOp | logicalInfixOp |
                                       relationalInfixOp
arithmeticInfixOp                 ::=  `*' | `\' | `**' | `%' | `+' | `-'
bitwiseShiftInfixOp               ::=  ` | `<<' | `>>'
bitwiseInfixOp                    ::=  `|' | `&' | `^'
logicalInfixOp                    ::=  `||' | `&&'
postfixExpr                       ::=  funcCall | forinExpr
funcCall                          ::=  simpleFunctionCall | parameterizedFunctionCall
simpleFunctionCall                ::=  id `(' exprList `)'
parameterizedFunctiontiontionCall ::=  id `<' [ integerConstant ] `>' `(' exprList `)'
forinExpr                         ::=  `for' id `in' rangeExpr [ `by' stride ] blockStmt
rangeExpr                         ::=  expr `..' expr
stride                            ::=  expr
```

Types

```
type                ::=  compositeType | primitiveType
compositeType       ::=  structType | parameterizedType
structType          ::=  `struct' id
parameterizedType   ::=  streamType
streamType          ::=  `stream' [ `<' [ integerConstant ] `>' ]
primitiveType       ::=  `int8' | `int16' | `int32' |
                         `int64' | `int128' | `int256' |
                         `uint8' | `uint16' | `uint32' |
                         `uint64' | `uint128' | `uint256' |
                         `float16' | `float32' | `float64' | `float128' |
                         `string8' | `string16'
```

Low-level Definitions

```
exprList        ::=  expr { ‘,’ expr }
cmpdId          ::=  id `.' id
id              ::=  letter { letter | digit }
constant        ::=  stringConstant | integerConstant
stringConstant  ::=  ```' stringElement `'''
integerConstant ::=  `0' | { nonZeroDigit }
stringElement   ::=  escapeChar |
                     ``any printable Unicode character except the double quote''
charEscape      ::=  `\' ``any printable Unicode character''
hexEscape       ::=  `\x' hexDigit hexDigit
unicodeEscape   ::=  `\u' hexDigit hexDigit hexDigit hexDigit
escapeChar      ::=  charEscape | hexEscape | unicodeEscape
letter          ::=  lower | upper | `_'
lower           ::=  `a' | ··· | `z'
upper           ::=  `A' | ··· | `Z'
digit           ::=  `0' | ··· | `9'
nonZeroDigit    ::=  `1' | ··· | `9'
hexDigit        ::=  `0' | ··· | `7' | `a' | ··· | `f' | `A' | ··· | `F'
comment         ::=  `/*' ``any sequence of characters'' `*/' |
                     `//' ``any sequence of characters up to end of line'' |
                     `#' ``any sequence of characters up to end of line''
```

Appendix C

s2^k Fieldstream Operations

This appendix defines the s2^k flow-parallel and field-parallel operations.

C.1 Flow-parallel Operations

The s2^k flow-parallel bitstream operations, denoted $R = F(S)$, $R = F(S, T)$, or $R = F(S, c)$ are defined using bitwise logic and arithmetic operations on bitstreams S, T of length L, or an integer constant c. The flow-parallel operations consist of the parallel scanning and parallel spanning operations. Given marker bitstreams and property bitstreams, parallel-scanning operation advance multiple 1-bit marker positions independently using addition and bitwise logical operations, thereby producing a follows bitstream. Given starts bitstreams and follows bitstreams, parallel-spanning operations form multiple contiguous runs of 1-bits (spans) using subtraction and bitwise logical operations, thereby producing a spans bitstream.

C.1.1 Parallel-scanning Operations

The flow-parallel scanning operations accept a marker stream M, a character-class property stream C, and an integer constant c.

Operation	Description
$Advance(M)$	The operation $R = Advance(M)$ advances a set of markers positions in M by a single bit position. $R = Advance(M) = (M + M) \mod 2^L$.
$AdvanceN(M, c)$	The operation $R = AdvanceN(M, c)$ advances a set of markers positions in M by c bit positions. $R = Advance(M, c) = (M \ll c) \mod 2^L$.

$ScanToFirst(M)$	The operation $R = ScanToFirst(M)$ scans to the first occurrence of a non-zero marker bit in M. $R = ScanToFirst(M) = ((F + \neg C) \wedge C)$ mod 2^L, where F is a bitstream such that $F[0] = 1$ and $F[k] = 0$ for $0 < k < L$.
$ScanTo(M, C)$	The operation $ScanTo(M, C)$ scans from an initial set of marker positions in M through the negated spans of C. $R = ScanTo(M, C) = ((M + \neg C) \wedge C)$ mod 2^L.
$AdvThenScanTo(M, C)$	The operation $R = AdvThenScanTo(M, C)$ (advance then scan to) scans from an initial set of advanced marker positions through the negated spans of C. $R = AdvThenScanTo(M, C) = ((M + M) + (\neg C)) \wedge C)$ mod 2^L.
$ScanThru(M, C)$	The operation $R = ScanThru(M, C)$ (scan through) scans from an initial set of marker positions in M through spans of C. $R = ScanThru(M, C) = ((M + C) \wedge (\neg C))$ mod 2^L.
$AdvThenScanThru(M, C)$	The operation $R = AdvThenScanThru(M, C)$ (advance then scan through) scans from an initial set of advanced marker positions in M through the spans of C. $R = AdvThenScanThru(M, C) = (((M + M) + C) \wedge (\neg C))$ mod 2^L.

$MatchStar(M, C)$	The operation $R = MatchStar(M, C)$ marks all positions that can be reached by 0 or more character-class markers in C from each marker position in M . $R = MatchStar(M, C) = ((((M \wedge C) + C) \oplus C) \vee M)$ mod 2^L.

Table C.1: Scanning operations.

C.1.2 Spanning Operations

The flow-parallel spanning operations accept a starts positions stream S, a follows positions stream F, and return a spans stream R. A precondition of each spanning operation is that the 1-bits in S and F must be well paired. That is, $PopCount(S) = PopCount(F)$ and the ith non-zero field in S precedes the ith non-zero field in F such that the bitwise logical OR of S and F, $S|F$, interleaves the 1-bit fields of S and F. Failure to satisfy this precondition results in an instance of an erroneous program.

Operation	Description
$SpanUpTo(S, F)$	The operation $R = SpanUpTo(S, F)$ marks all positions in S up to but not including positions in F. $R = SpanUpTo(S, F) = (F - S)$ mod 2^L.
$InclusiveSpan(S, F)$	The operation $R = InclusiveSpan(S, F)$ marks all positions in S and in F, inclusively. $R = InclusiveSpan(S, F) = ((F - S)\|F)$ mod 2^L.
$ExclusiveSpan(S, F)$	The operation $R = ExclusiveSpan(S, F)$ marks all positions in S and in F inclusively. $R = ExclusiveSpan(S, F) = ((F - S) \wedge (\neg S))$ mod 2^L.

Table C.2: Spanning operations.

C.1.3 LookAhead Operations

The lookahead operations accept a marker stream M, or an integer constant c.

Operation	Description
$LookAhead(M)$	The operation $R = LookAhead(M)$ reverse shifts a set of markers positions in M by a single bit position. $R = Advance(M) = (M \gg 1) \mod 2^L$.
$LookAheadN(M, c)$	The operation $R = LookAhead(M)$ reverse shifts a set of markers positions in M by constant integer c bit positions. $R = LookAheadN(M) = (M \gg c) \mod 2^L$.

Table C.3: LookAhead operations.

C.1.4 Masking Operations

The masking operations accept a bitstream S.

Operation	Description
$InFile(S)$	The operation $R = InFile(S)$ (in file) marks the position immediately past the end of a stream.
$AtEOF(S)$	The operation $R = AtEOF(S)$ (at end of file) marks all positions within a stream.

Table C.4: Masking operations.

C.1.5 Positional Operations

The positional operations accept a marker stream M and an integer constant c that denotes the bit width of integer return values.

Operation	Description
Positions(M,c)	The $Positions(M)$ operation returns a c-bit Fieldstream of stream indices for each 1-bit marker in M.
Lengths(M,c)	The $Lengths(M)$ operation returns a c-bit Fieldstream of stream indices for each span in M.

Table C.5: Indexing operations.

C.2 Field-parallel Operations.

Field-parallel operations are a set of collective operations defined to support the simultaneous calculation of n-bit Fieldstream fields, for $n = 2^k$ where $k \leq K$.

C.2.1 Auxiliary Operations.

Table C.6 presents n-bit auxillary operations used to describe the semantics of the field-parallel operations. The auxillary operations define a set of binary functions on n-bit fields for $n = 2^k$ for $0 \leq k \leq K$.

Auxiliary Operation	Description
$hi_n(s_n)$	The operation $r_{n/2} = hi_n(s_n)$ returns the high $n/2$-bits of an n-bit value s_n. $r_{n/2} = hi_n(s_n) = s_n/(n/2)$.
$lo_n(s_n)$	The operation $r_{n/2} = lo_n(s_n)$ returns the low $n/2$-bits of an n-bit value s_n. $r_{n/2} = lo_n(s_n) = s_n$ mod $(n/2)$.

Table C.6: Virtual operations.

C.3 Vertical Operations

Vertical operations, defined over n-bit Fieldstreams of length L and denoted $R_n = F_n(S_n), \ldots,$ $R_n = F_n(S_n, T_n, U_n)$, are defined by the element-wise calculation of the n-bit fields of Fieldstream arguments $S_n, T_n,$ or U_n. Vertical operations produce an n-bit Fieldstream result R_n of length L in accordance with a function $f_n(s_n) \ldots f_n(s_n, t_n, u_n)$ defined on n-bit values $s_n, t_n,$ or u_n for $n = 2^k$ where $0 \leq k \leq K$.

C.3.1 Unary Operations

Operation	Description
$Not_n(S_n)$	The operation $R_n = Not_n(S_n)$ computes the bitwise logical negation of each n-bit field of S_n. $R_n[i] = \neg S_n[i]$.
$Any_n(S_n)$	The operation $R_n = Any_n(S_n)$ sets each n-bit field to all 1's if any bit within a field is 1 otherwise 0. $Any_n(S_n)[i] = \begin{cases} (1 \ll n) - 1 & \text{if } \exists j,\ R_n[i][j] = 1,\ 0 \leq j < n \\ 0 & \text{otherwise.} \end{cases}$
$All_n(S_n)$	The operation $R_n = All_n(S_n)$ sets each n-bit field to all 1's if all bit within a field are 1 otherwise 0. $All_n(S_n)[i] = \begin{cases} (1 \ll n) - 1 & \text{if } \forall j,\ R_n[i][j] = 1,\ 0 \leq j < n \\ 0 & \text{otherwise.} \end{cases}$
$Abs_n(S_n)$	The operation $R_n = Abs_n(S_n)$ computes the absolute value of each n-bit field of S_n. $R_n[i] = \lvert S_n[i] \rvert$.
$Neg_n(S_n)$	The operation $R_n = Neg_n(S_n)$ negates each n-bit field of S_n. $R_n[i] = -1 * S_n[i]$.

$AddHiLo_n(S_n)$	The operation $R_n = AddHiLo_n(S_n)$ adds the high $n/2$ bits of each field of S_n to the low $n/2$ bits of S_n, respectively. $R_n[i] = Hi_n(S_n)[i] + Lo_n(S_n)[i]$.
$XorHiLo_n(S_n)$	The operation $R_n = XorHiLo_n(S_n)$ computes the bitwise exclusive logical OR of the high $n/2$ bits each field of S_n with the to the low $n/2$ bits of S_n. $R_n[i] = Hi_n(S_n)[i] \oplus Lo_n(S_n)[i]$.
$PopCount_n(S_n)$	The operation $R_n = PopCount_n(S_n)$ computes the n-bit Field-stream of 1-bit field counts of S_n. $R_n[i] =$ The count of 1 bits in $S_n[i]$.

Table C.7: Unary operations.

C.3.2 Binary Operations

Operation	Description
$And_n(S_n, T_n)$	The operation $R_n = And_n(S_n, T_n)$ computes the bitwise logical AND of corresponding 2^k-bit fields of S_n and T_n. $R_n[i] = S_n[i] \wedge T_n[i]$.
$AndC_n(S_n, T_n)$	The operation $R_n = AndC_n(S_n, T_n)$ computes the bitwise logical AND of corresponding 2^k-bit fields of S_n and the bitwise logical negation of T_n. $R_n[i] = S_n[i] \wedge \neg T_n[i]$.

$Or_n(S_n, T_n)$

The operation $R_n = Or_n(S_n, T_n)$ computes the bitwise logical OR of corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = S_n[i] \vee T_n[i].$$

$Nor_n(S_n, T_n)$

The operation $R_n = Nor_n(S_n, T_n)$ computes the bitwise logical negation of corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = \neg(S_n[i] \vee T_n[i]).$$

$Xor_n(S_n, T_n)$

The operation $R_n = Xor_n(S_n, T_n)$ computes the bitwise exclusive logical OR of corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = S_n[i] \oplus T_n[i].$$

$Add_n(S_n, T_n)$

The operation $R_n = Add_n(S_n, T_n)$ adds corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = S_n[i] + T_n[i].$$

$Sub_n(S_n, T_n)$

The operation $R_n = Sub_n(S_n, T_n)$ subtracts corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = S_n[i] - T_n[i].$$

$Eq_n(S_n, T_n)$

The operation $R_n = Eq_n(S_n, T_n)$ compares corresponding 2^k-bit fields of S_n and T_n for the Boolean equality comparison operator.

$$Eq_n(S_n, T_n)[i] = \begin{cases} (1 \ll 2^k) - 1 & \text{if } S_n[i] = T_n[i] \\ 0 & \text{otherwise.} \end{cases}$$

$Lt_n(S_n, T_n)$

The operation $R_n = Lt_n(S_n, T_n)$ compares the corresponding 2^k-bit fields of S_n and T_n for the Boolean less-than comparison operator.

$$R_n[i] = \begin{cases} (1 \ll 2^k) - 1 & \text{if } S_n[i] < T_n[i] \\ 0 & \text{otherwise.} \end{cases}$$

$Gt_n(S_n, T_n)$

The operation $R_n = Gt_n(S_n, T_n)$ compares corresponding 2^k-bit fields of S_n and T_n for the Boolean greater-than comparison operator.

$$R_n[i] = \begin{cases} (1 \ll 2^k) - 1 & \text{if } S_n[i] > T_n[i] \\ 0 & \text{otherwise.} \end{cases}$$

$Max_n(S_n, T_n)$

The operation $R_n = Max_n(S_n, T_n)$ returns the maximal value of corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = \begin{cases} S_n[i] & \text{if } S_n[i] >= T_n[i] \\ T_n[i] & \text{otherwise.} \end{cases}$$

$Min_n(S_n, T_n)$

The operation $R_n = Min_n(S_n, T_n)$ returns the minimal value of corresponding 2^k-bit fields of S_n and T_n.

$$R_n[i] = \begin{cases} S_n[i] & \text{if } S_n[i] <= T_n[i] \\ T_n[i] & \text{otherwise.} \end{cases}$$

$SLL_n(S_n, T_n)$

The operation $R_n = SLL_n(S_n, T_n)$ (shift left logical) logically left shifts 2^k-bit fields of S_n by the corresponding values of T_n bit positions.

$$R_n[i] = S_n[i] \ll T_n[i].$$

$SRL_n(S_n, T_n)$	The operation $R_n = SRL_n(S_n, T_n)$ (shift right logical) logically right shifts the 2^k-bit fields of S_n by the corresponding values of T_n bit positions. $R_n[i] = S_n[i] \gg T_n[i]$.
$SLLI_n(s, c)$	The operation $R_n = SLLI_n(s, c)$ (shift left logical immediate) logically left shifts the 2^k-bit fields of S_n by c bit positions. $R_n[i] = S_n[i] \ll c$.
$SRLI_n(s, c)$	The operation $R_n = SRLI_n(s, c)$ (shift right logical immediate) logically right shifts the 2^k-bit fields of S_n by constant integer c bit positions. $R_n[i] = S_n[i] \gg c$.
$SLAI_n(s, c)$	The operation $R_n = SLAI_n(s, c)$ (shift left arithmetic immediate) arithmetically left shifts the 2^k-bit fields of S_n by constant integer c bit positions. $R_n[i] = S_n[i] \ll c$.
$SRAI_n(s, c)$	The operation $R_n = SRAI_n(s, c)$ (shift right arithmetic immediate) arithmetically right shifts the 2^k-bit fields of S_n by constant integer c bit positions. $R_n[i] = S_n[i] \gg c$.

Table C.8: Binary operations.

C.3.3 Ternary Operations

Operation	Description
IfHi$_n(S_n, T_n, U_n)$	$R_n[i] = \begin{cases} T_n[i] & \text{if } S_n[i][n-1] = 1 \\ U_n[i] & \text{otherwise.} \end{cases}$

Table C.9: Ternary operations.

C.4 Horizontal-packing Operations

The Horizontal-packing operations, defined over n-bit Fieldstreams of length L and denoted $R_{n/2} = F_n(S_n)$, are defined by the element-wise calculation of $n/2$-bit fields given an n-bit Fieldstream argument S_n. The Horizontal-packing operations produce an $n/2$-bit Fieldstream result, $R_{n/2}$ of length L, in accordance with a function f_n defined on n-bit values that produce an $n/2$-bit Fieldstream for $n = 2^k$ where $1 \leq k \leq K$.

Operation	Description
$PackHi_n(S_n)$	The operation $R_{n/2} = PackHi_n(S_n)$ returns the $n/2$-bit Fieldstream, $R_{n/2}$, containing the high $n/2$-bits of each corresponding n-bit field of S_n. $R_{n/2}[i] = hi_n(S_n[i])$.
$PackLo_n(S_n)$	The operation $R_{n/2} = PackLo_n(s)$ operation returns the $n/2$-bit Fieldstream, $R_{n/2}$, containing the high $n/2$-bits of each n-bit field of S_n. $R_{n/2}[i] = lo_n(S_n[i])$.

Table C.10: Horizontal-packing operations.

C.5 Expansion Operations

The Expansion operations, defined over n-bit Fieldstreams of length L and denoted $R_{2n} = F_n(S_n, T_n)$, are defined by the element-wise calculation of the $2n$-bit fields of n-bit Fieldstream arguments S_n and T_n. The Expansion operations produce an $2n$-bit Fieldstream result, R_{2n} of length L, in accordance with a function f_n defined on n-bit values that produces $2n$-bit values for $n = 2^k$ where $0 \leq k \leq K$. Here, field concatentation is denoted by the symbol, $||$.

Operation	Description
$Merge_n(s,t)$	The operation $R_{2n} = Merge_n(s,t)$ concatenates corresponding n-bit element fields of s and t. $R_{2n}[i] = s[i] \mid\mid t[i]$.
$Mult_n(s,t)$	The operation $R_{2n} = Mult_n(s,t)$ multiplies corresponding n-bit element fields of s and t. $R_{2n}(s,t)[i] = s[i] * t[i]$.
$ZExt_n(s)$	The operation $R_{2n} = ZExt_n(s)$ (zero extend) prepends n 0-bit copies to each field of s. $R_{2n}[i] = n$ 0 bits $\mid\mid s[i]$.
$SExt_n(s)$	The operation $R_{2n} = SExt_n(s)$ (sign extend) prepends n sign-bit copies to each field of s. $R_{2n}[i] = n$ sign bits $\mid\mid s[i]$.

Table C.11: Expansion operations.

C.6 Field-movement Operations

Field-movement operations, defined over n-bit Fieldstreams of length L and denoted $R_n = F_{n,m}(S_n, T_n)$ or $R_n = F_{n,m}(S_n, c)$, are defined by the simultaneous calculation of the individual n-bit fields of n-bit Fieldstream arguments S_n,T_n, or constant integer c. Field-movement operations produce an n-bit Fieldstream result, R_n of length L, in accordance with a function f_n defined over the m-bit subfields of n-bit values of the following forms, for constant integer c, $n = 2^k$, and $m = 2^j$, where $0 \leq j \leq k \leq K$.

Operation	Description
$SLL_{n,m}(S_n, T_n)$	The operation $R_n = SLL_{n,m}(S_n, T_n)$ logically left shifts m-bit subfields of each n-bit fields of S_n by corresponding values of T_n. $$R_n[i] = \begin{cases} S_n[i] \ll (T_n[i] * m) & \text{if } (T_n[i] * m) < n, \\ 0 & \text{otherwise.} \end{cases}$$
$SRL_{n,m}(S_n, T_n)$	The operation $R_n = SRL_{n,m}(S_n, T_n)$ logically right shifts m-bit subfields of each n-bit fields of S_n by corresponding values of T_n. $$R_n[i] = \begin{cases} S_n[i] \gg (T_n[i] * m) & \text{if } (T_n[i] * m) < n, \\ 0 & \text{otherwise.} \end{cases}$$
$SLLI_{n,m}(S_n, c)$	The operation $R_n = SLLI_{n,m}(S_n, c)$ logically left shifts m-bit subfields of each n-bit fields of S_n by some constant integer c. $$R_n[i] = \begin{cases} S_n[i] \ll (c * m) & \text{if } (c * m) < n, \\ 0 & \text{otherwise.} \end{cases}$$
$SRLI_{n,m}(S_n, c)$	The operation $R_n = SRLI_{n,m}(S_n, c)$ logically right shifts m-bit subfields of each n-bit fields of S_n by some constant integer c. $$R_n[i] = \begin{cases} S_n[i] \gg (c * m) & \text{if } (c * m) < n, \\ 0 & \text{otherwise.} \end{cases}$$

$Splat_{n,m}(S_n, c)$	The operation $R_n = Splat_{n,m}(S_n, c)$ broadcasts the cth m-bit subfield of each n-bit field of S_n into every position of each m-bit subfield of S_n. $Splat_{n,m}(S_n, c)[i][j] = S_n[i][c]$.

Table C.12: Field-movement operations.

C.7 Casting Operations

The casting operation $R_c = BitCast_n(S_n, c)$ re-interprets a 2^k-bit Fieldstream, S_n, as a c-bit Fieldstream, R_c, for $n = 2^k$ and constant integer $c = 2^j k$, where $0 \leq k \leq K$ and $0 \leq j \leq K$.

Cast Operation	Description
$BitCast_n(s, c)$	The operation $R_c = BitCast_n(S_n, c)$ re-interprets a 2^k-bit Fieldstream, S_n, as a c-bit Fieldstream R_c, for $n = 2^k$ and constant integer $c = 2^j k$, where $0 \leq k \leq K$, $0 \leq j \leq K$.

Table C.13: Cast operations.

C.8 Fieldstream-movement Operations

Fieldstream-movement operations, denoted $R_n = F_{n,m}(S_n, I)$, define the parallel movement of the fields of an n-bit Fieldstream R_n of length L in accordance with an index stream I for $n = 2^k$, $m = 2^j$, where $0 \leq k \leq j \leq K$.

Operation	Description
$PGather_n(S_n, I)$	The $PGathern(S_n, I)$ (parallel gather) operation gathers and packs indexed n-bit source data fields to a target n-bit Fieldstream.

$PScatter_n(S_n, I)$	The $PScattern(S_n, I)$ (parallel scatter) operation scatters packed n-bit source data fields to the indexed field of a target n-bit Fieldstream.
$PGatherSpansHi_{n,m}(S_n, I)$	The $PGatherSpansHi_{n,m}(S_n, I)$ (parallel gather align high) operation gathers and aligns spans of n-bit source data fields to high-order m-bit boundaries.
$PGatherSpansLo_{n,m}(S_n, I)$	The $PGatherSpansLo_{n,m}(S_n, I)$ (parallel gather align low) operation gathers and aligns spans of n-bit source data fields to low-order m-bit boundaries.

Table C.14: Fieldstream-movement operations.

Appendix D

$s2^k$ Variables and Libraries

This Appendix defines a concrete syntax for the $s2^k$ variables and libraries. Variables and operations are denoted by $\langle library_name \rangle . \langle variable_name \rangle$ or $\langle library_name \rangle . \langle function_name \rangle$. Library names are shown in parentheses following each heading.

D.1 $s2^k$ Variables (s2k)

$s2^k$ provides a small number of built-in variables.

Variable	Description
BufferSize	The number of fields buffered by a host-program for $s2^k$ processing.
SegmentSize	The number of fields processed per segment in $s2^k$ segment-by-segment processing.
BlockSize	The number of bits processed per block in $b2^k$ block-by-block processing. Alternatively, the bit width of the SIMD registers of a target SIMD ISA.

Table D.1: $s2^k$ variables.

D.2 s2^k Libraries

D.2.1 Buffer Management Library (io)

A collection of s2^k operations for the expression of data access operations between a host program an s2^k-program. The Buffer Management Library operations define procedures that do not return values. Buffer Management operations are of the form op<n>(S_n,$Addr$) or op<n>(S_n), where op<n> denotes a polymorphic operation name for $n = 2^k$ where 1 $\leq k \leq$ s2k.BlockSize, S_n denotes a n-bit Fieldstream, and $Addr$ denotes a host-program memory address.

Operation	Description
StreamBind<n>(S_n, $Addr$)	The StreamBind<n>(S_n,$Addr$) operation associates an s2^k Fieldstream S with a host-program memory region.
StreamRead<n>(S_n)	The StreamRead<n>(S_n) operation copies n-bit fields from a bound host-program memory region to an associated s2^k Fieldstream variable, S_n.
StreamWrite<n>(S_n)	The StreamWrite<n>(S_n) operation copies n-bit fields from a bound s2^k variable S_n to a host-program memory region.

Table D.2: Buffer management library.

D.2.2 Standard Library (std)

A collection of operations for the expression of standard algorithms for streaming text processing using parallel bit streams. The transposition and inverse transposition functions are of the are of the form $B_n = op$<n>(S_n) or $S_n = op$<n>(B_n) or where $n = 8, n = 16, n = 32$, S_n denotes an n-bit Fieldstream, B_n denotes a n-element *Basis* stream structure.

Operation	Description
Transpose<n>(S_n)	The Transpose<n>(S_n) operation returns the transpose of the n-bit stream S_n as a n-element Basis stream set structure, B_n.

InverseTranspose<n>(B_n)	The InverseTranspose<n>(B_n) operation returns the inverse-transpose of an n-element Basis stream set structure as a n-bit Fieldstream.

Table D.3: Standard library.

D.2.3 Regular Expression Library (regex)

A collection of regular expression operations for streaming text search. The Regular Expression library supports a limited subset of the Unicode Level 1 requirements set out in Unicode Technical Standard #18 expression syntax and functionality.

s2^k regular expression search computes longest-leftmost matches. Match positions are marked with a 1-bit immediately following each matched string with a 1-bit.

The Regular Expression library functions are of the form, $R = op$<n>(S_n,p), where $n = 8, n = 16, n = 32$, R is a bitstream, S_n denotes an n-bit Fieldstream, p is a string constant that describes a Unicode Level 1 regular expression.

Operation	Description
CharClass<n>(S_n,p)	The CharClass<n>(S_n,p) (character class) operation marks all positions of character class occurrences in S_n. For example, CharClass<n>(s_n,``$[\backslash p\{Lu\}]''$) returns the Fieldstream that marks all lower-case character positions in S_n, an n-element Basis stream set structure.
Search<n>(S_n,p)	The Search<n>(S_n,p) operation marks the follows position of each longest left-most match of p in S_n.

Table D.4: Regular expression library.

D.3 Flow-parallel Library (flow)

A collection of flow-parallel operations for the expression of abstract flow-parallel defined in Section C. Flow-parallel functions are of the form $R = op(S)$ or $R = op(S, T)$, where op denotes the operation name, and R, S and T denote bitstreams.

D.4 Field-parallel Libraries (field)

A collection of field-parallel operations for the expression of the abstract field-parallel operations defined in Section C.2. The flow-parallel functions are of the form op<n>(...) or op<n, m>(...), where op denotes the operation name, and op<n> denotes an operation is applied to n-bit Fieldstreams, and op<n,m> denotes an operation is applied to the m-bit subfields of n-bit Fieldstreams. The arguments and return values of field-parallel functions correspond to the previously described abstract operations.

Appendix E

$s2^k$ Case Study Program Listings

This appendix contains program listings for example $s2^k$ programs.

E.1 8-bit ASCII Stream to 16-bit Binary-integer Conversion

```
/*
 * s2k 8-bit ASCII Stream to 16-bit Binary Integer Conversion.
 *
 * The s2k grep exact-string search program is organized into the following
 * pipeline stages.
 *
 * Stage 1 - Read
 * Stage 2 - Transpose
 * Stage 3 - Classify
 * Stage 4 - Convert
 * Stage 5 - Write
 *
 */

/* Stream Structure definitions */
struct Input
{
  s<8> s_76543210;
}

struct Basis
{
  s<1> s_7; s<1> s_6; s<1> s_5; s<1> s_4;
  s<1> s_3; s<1> s_2; s<1> s_1; s<1> s_0;
}

struct Digits
```

```
{
  s<1> digits;
}

struct Output
{
  s<16> results;
}

/* Filter Definitions */
// Read 8-bit input stream.
filter Read(in: struct Input input, out: struct Input s76543210) {
  StreamBind<8>(input.s_76543210, ibuffer);
  StreamBind<16>(output.results, obuffer);
  StreamRead<8>(input.s_76543210);
}

// Transpose 8-bit input stream into a set of basis streams.
filter Transpose(in: struct Input input, out: struct Basis basis) {
  s<4> s_7654 = PackHi<8>(input.s_76543210);
  s<4> s_3210 = PackLo<8>(input.data);

  Stage 2: 4-bit to 2-bit fields.
  s<2> s_76 = PackHi<4>(s_7654);
  s<2> s_54 = PackLo<4>(s_7654);
  s<2> s_32 = PackHi<4>(s_3120);
  s<2> s_10 = PackLo<4>(s_3120);

  Stage 3: 2-bit to 1-bit fields.
  basis.s_7 = PackHi<2>(s_76);
  basis.s_6 = PackLo<2>(s_76);
  basis.s_5 = PackHi<2>(s_54);
  basis.s_4 = PackLo<2>(s_54);
  basis.s_3 = PackHi<2>(s_32);
  basis.s_2 = PackLo<2>(s_32);
  basis.s_1 = PackHi<2>(s_10);
  basis.s_0 = PackLo<2>(s_10);
}

// Mark the positions of 8-bit ASCII decimal digits, i.e., [0-9].
// Character class bit stream equations are generated with
// the character-class compiler tool.
filter Classify(in: struct Basis basis, out: struct Digits digits) {
  s<1> temp1 = (basis.s_7 | basis.s_6);
  s<1> temp2 = (basis.s_5 & basis.s_4);
```

```
  s<1> temp3 = (temp2 &~ temp1);
  s<1> temp4 = (basis.s_2 | basis.s_1);
  s<1> temp5 = (basis.s_3 & temp4);
  digits.digits = (temp3 &~ temp5);
}

// Convert 8-bit ASCII decimal digits to 16-bit binary decimal integers.
filter Convert(in: struct Input input, in: struct Digits digits, out: Output output) {
  s<4> s_3210  = PackLo<8>(input.s_76543210);
  s<4> mask_4  = ZExt<2>(ZExt<1>(digits.digits));
  s<4> d_4     = And<4>(s_3210,mask_4);

  // 32-bit Parallel gather align-high.
  s<4,G> g     = PGatherSpansHi<4,32>(digits.digits,d_4);

  // Stage 1: Reverse 4-bit fields.
  s<8,M> t_8   = BitCast<4>(g,8);
  s<8> hi_4    = SLLI<8>(t_8,4);
  s<8> lo_4    = SRLI<8>(t_8,4);
  s<8> rot_4   = Or<8>(hi_4,lo_4);

  // Stage 2: Reverse 8-bit fields.
  s<16,N> t_16 = BitCast<8>(rot_4,16);
  s<16> hi_8   = SLLI<16>(t_16,8);
  s<16> lo_8   = SRLI<16>(t_16,8);
  s<16> rot_8  = Or<16>(hi_8,lo_8);

  // Stage 3: Reverse 16-bit fields.
  s<32,P> t_32 = BitCast<16>(rot_8,32);
  s<32> hi_16  = SLLI<32>(t_32,16);
  s<32> lo_16  = SRLI<32>(t_32,16);
  s<32> rot_16 = Or<32>(hi_8,lo_8);

  // Stage 1: Convert 4-bit fields.
  s<8,M> temp1    = BitCast<32>(rot_16,8);
  s<8> d_hi_4     = SRLI<8>(4, temp1);
  s<8> lomask_8   = LoMask<8>();
  s<8> d_lo_4     = And<8>(rot_16, lomask8);
  s<8> c_10       = 10;
  s<8> t1_10      = Mul<8>(d_hi_4, c_10);
  s<8> t2_10      = Add<8>(t1_10, d_lo_4);

  // Stage 2: Convert 8-bit fields.
  s<8,N> temp2    = BitCast<8>(t2_10,16);
  s<16> d_hi_8    = SRLI<16>(8, temp2);
```

```
  s<16> lomask_16 = LoMask<16>();
  s<16> d_lo_8    = And<16>(d_hi_8, lomask16);
  s<16> c_100     = 100;
  s<16> t1_100    = Mul<16>(d_hi_8, c_100);
  s<16> t2_100    = Add<16>(t1_100, d_lo_8);

  // Stage 3: Convert 16-bit fields.
  s<8,P>  temp3   = BitCast<16>(t2_100,32);
  s<32> d_hi_16   = SRLI<32>(16, temp3);
  s<32> lomask_32 = LoMask<32>();
  s<32> d_lo_16   = And<32>(d_hi_16, lomask32);
  s<32> c_10000   = 10000;
  s<32> t1_10000  = Mul<32>(d_hi_16, c_10000);
  s<32> t2_10000  = Add<32>(t1_10000, d_lo_16);

  // Pack 16-bit result fields.
  s<16> results   = PackLo<32>(t2_10000);

}

// Write 16-bit results stream.
filter Write(in: Output output) {
  StreamWrite<16>(output.results);
}

/* Stream Graph */
graph
{
  // structure declarations
  struct Input input;
  struct Basis basis;
  struct Digits digits;
  struct Output output;

  // filter declarations
  filter Read read;
  filter Transpose transpose;
  filter Classify classify;
  filter Convert convert
  filter Write write;

  // filter pipeline
  read(input);
  transpose(input, basis);
  classify(basis, digits);
```

```
  convert(basis, digits, output);
  write(output);
}
```

E.2 Grep Exact String Search

```
/*
 * s2k Grep Exact-string Search.
 *
 * The s2k grep exact-string search program is organized into the following
 * pipeline stages.
 *
 * Stage 1 - Read
 * Stage 2 - Transpose
 * Stage 3 - Classify
 * Stage 4 - Match
 * Stage 5 - MatchLines
 * Stage 6 - Write
 *
 */

/* Stream Structure Definitions */
struct Input{
  stream<8> data;
}

struct Basis_bits{
  stream<1> s_7;
  stream<1> s_6;
  stream<1> s_5;
  stream<1> s_4;
  stream<1> s_3;
  stream<1> s_2;
  stream<1> s_1;
  stream<1> s_0;
};

struct Lex{
  stream<1> a;
  stream<1> p;
  stream<1> l;
  stream<1> e;
  stream<1> LF;
};

struct Output{
  stream<1> match_follows;
  stream<1> lines;
  stream<1> line_starts;
```

```
  stream<1> line_ends;
};

/* Filter Definitions */
// Read 8-bit input stream.
filter Read(out: struct Input input) {
  StreamBind<8>(input.data, buffer);
  StreamBind<1>(output.starts, starts);
  StreamBind<1>(output.ends, ends);
  StreamBind<1>(output.markers, markers);
  StreamRead<8>(input.data);
}

// Transpose 8-bit input stream into a set of basis streams.
filter Transpose(in: struct Input input, out: struct Basis_bits basis_bits) {
  Stage 1: 8-bit to 4-bit fields.
  s<4> s_7654 = PackHi<8>(input.data);
  s<4> s_3210 = PackLo<8>(input.data);

  Stage 2: 4-bit to 2-bit fields.
  s<2> s_76 = PackHi<4>(s_7654);
  s<2> s_54 = PackLo<4>(s_7654);
  s<2> s_32 = PackHi<4>(s_3120);
  s<2> s_10 = PackLo<4>(s_3120);

  Stage 3: 2-bit to 1-bit fields.
  basis.s_7 = PackHi<2>(s_76);
  basis.s_6 = PackLo<2>(s_76);
  basis.s_5 = PackHi<2>(s_54);
  basis.s_4 = PackLo<2>(s_54);
  basis.s_3 = PackHi<2>(s_32);
  basis.s_2 = PackLo<2>(s_32);
  basis.s_1 = PackHi<2>(s_10);
  basis.s_0 = PackLo<2>(s_10);
}

// Mark the positions of 8-bit ASCII characters in the fixed string `apple', i.e., [aple].
// Character class bit stream equations are generated with
// the character-class compiler tool.
filter Classify(in: struct Basis_bits basis_bits, out: struct Lex lex) {
  stream<1> temp1 = (basis_bits.s_1 & (~ basis_bits.s_0));
  stream<1> temp2 = (basis_bits.s_2 & (~ basis_bits.s_3));
  stream<1> temp3 = (temp1 & temp2);
  stream<1> temp4 = (basis_bits.s_4 | basis_bits.s_5);
  stream<1> temp5 = (basis_bits.s_7 & (~ basis_bits.s_6));
```

```
  stream<1> temp6 = (temp5 & (~ temp4));
  lex.a = (temp3 & temp6);
  stream<1> temp7 = (basis_bits.s_2 & basis_bits.s_3);
  stream<1> temp8 = (temp1 & temp7);
  stream<1> temp9 = (basis_bits.s_6 | basis_bits.s_7);
  stream<1> temp10 = (temp4 | temp9);
  lex.p = (temp8 & (~ temp10));
  stream<1> temp11 = (basis_bits.s_4 & basis_bits.s_5);
  stream<1> temp12 = (temp11 & (~ temp9));
  lex.l = (temp3 & temp12);
  stream<1> temp13 = (basis_bits.s_5 & (~ basis_bits.s_4));
  stream<1> temp14 = (temp13 & temp5);
  lex.e = (temp3 & temp14);
  stream<1> temp15 = (basis_bits.s_0 | basis_bits.s_1);
  stream<1> temp16 = (basis_bits.s_2 | basis_bits.s_3);
  stream<1> temp17 = (temp15 | temp16);
  stream<1> temp18 = (basis_bits.s_4 & (~ basis_bits.s_5));
  stream<1> temp19 = (basis_bits.s_6 & (~ basis_bits.s_7));
  stream<1> temp20 = (temp18 & temp19);
  lex.LF = (temp20 & (~ temp17));
}

// Mark exact matches.
filter Match(in: struct Lex lex, out: struct Output output) {
  stream<1> cursor = Advance(lex.a);
  cursor = Advance((cursor & lex.p));
  cursor = Advance((cursor & lex.p));
  cursor = Advance((cursor & lex.l));
  cursor = Advance((cursor & lex.e));
  output.match_follows = cursor;
}

// Mark lines containing matches.
filter MatchLines(in: struct Lex lex, out: struct Output output) {
  stream<1> all_line_starts = (ScanToFirst((~ lex.LF))
                              | (Advance(lex.LF) & (~ lex.LF)));
  stream<1> all_line_ends = lex.LF;
  stream<1> last_line_start = ScanToFirst(all_line_starts);
  stream<1> cursor = last_line_start;
  while (InFile(cursor)) {
      if ((cursor & all_line_starts)) {
          last_line_start = cursor;
      }
      if ((cursor & output.match_follows)) {
          cursor = ScanTo(cursor, lex.LF);
```

```
          output.lines |= InclusiveSpan(last_line_start, cursor);
      }
      cursor = AdvanceThenScanTo(cursor, (all_line_starts | output.match_follows));
  }
  output.line_starts = (output.lines & all_line_starts);
  output.line_ends = ScanTo(output.line_starts,
                                 (output.lines & all_line_ends));
}

// Write 16-bit results stream.
filter Write(in: struct Output output) {
  StreamWrite<1>(output.line_starts);
  StreamWrite<1>(output.line_ends);
  StreamWrite<1>(output.match_follows);
}

/* Stream Graph */
graph pipeline() {

  // structure declarations
  struct Input input;
  struct Basis_bits basis_bits;
  struct Lex lex;
  struct Output output;

  // filter declarations
  filter Read read;
  filter Transpose transpose;
  filter Classify classifyBytes;
  filter Match match;
  filter MatchLines matchLines;
  filter Write write;

  // filter pipeline
  read(input);
  transpose(input, basis_bits);
  classifyBytes(basis_bits, lex);
  match(lex, output);
  matchLines(lex, output);
  write(output);

}
```

E.3 Inverted-index Construction

```
/*
 * s2k Length-Sorted Inverted Index Construction.
 *
 * The s2k length-sorted inverted index program is organized into the following stages.
 *
 * Stage 1 - Read
 * Stage 2 - Tokenize
 * Stage 3 - Mark Start Group Positions (composed of substages)
 * Stage 4 - Mark Follow Group Positions (composed of substages)
 * Stage 5 - Gather
 * Stage 6 - Write
 *
 */

/* Stream Structure Definitions */
struct Input {
  stream<8> data;
}

struct Output {
  stream<8>  L_1;
  stream<16> L_2;
  stream<32> L_3_4;
  stream<64> L_5_8;
  stream<8>  L_gt_8;
  stream<32> L_gt_8_S;
  stream<32> L_gt_8_F;
}

struct TokenStarts {
  stream<1> S;
  stream<1> S_1;
  stream<1> S_2;
  stream<1> S_3_4;
  stream<1> S_5_8;
  stream<1> S_gt_1;
  stream<1> S_gt_2;
  stream<1> S_gt_3_4;
  stream<1> S_gt_5_8;
}

struct TokenFollows {
  stream<1> F;
```

```
  stream<1> F_1;
  stream<1> F_2;
  stream<1> F_3_4;
  stream<1> F_5_8;
  stream<1> F_gt_1;
  stream<1> F_gt_2;
  stream<1> F_gt_3_4;
  stream<1> F_gt_5_8;
}

struct FollowsMask {
  stream<1> R_0;
  stream<1> R_1;
  stream<1> R_2;
  stream<1> R_3_4;
  stream<1> R_5_8;
}

struct StartsMask {
  stream<1> N_0;
  stream<1> N_1;
  stream<1> N_2;
  stream<1> N_3_4;
  stream<1> N_5_8;
}

/* Filter Definitions */
// Bind and read.
filter Read(out: struct Input input) {

  StreamBind<8> (input.data         , buffer);
  StreamBind<8> (output.L_1         , L_1);
  StreamBind<16>(output.L_2         , L_2);
  StreamBind<32>(output.L_3_4       , L_3);
  StreamBind<32>(output.L_5_8       , L_5);
  StreamBind<8> (output.L_gt_8      , L_gt_8);
  StreamBind<32>(output.L_gt_8_S    , L_gt_8_S);
  StreamBind<32>(output.L_gt_8_F, L_gt_8_F);

  StreamRead<8>(input.data);
}

// Mark the starts positions of tokens. Initialize starts and follows masks.
filter Tokenize(in: struct Input input,
                out: struct TokenStarts starts, out: struct FollowsMask follows_mask,
```

```
                out: TokenFollows follows,      out: struct FollowsMask starts_mask) {
  stream<1> L      = CharClass("[\p{L}]");
  starts.S         = Advance(~L) & L;
  follows.F        = Advance(L) & ~L;
  follows_mask.R_0 = follows.F;
  starts_mask.N_0  = starts.S;
}

// Mark start group positions.
filter Mark_S_1(in: struct TokenStarts starts,       out: struct TokenStarts starts,
in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  follows_mask.R_1   = LookAheadN(follows_mask.R_0, 1);
  starts.S_1         = mask.R_1 & starts.S;
  starts.S_gt_1      = starts.S &~ starts.S_1;
}

filter Mark_S_2(in: struct TokenStarts starts,       out: struct TokenStarts starts,
in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  follows_mask.R_2   = LookAheadN(follows_mask.R_1, 1) | follows_mask.R_1;
  starts.S_2         = follows_mask.R_2 & starts.S_gt_1;
  starts.S_gt_2      = starts.S_gt_1 &~ starts.S_2;
}

filter Mark_S_3_4(in: struct TokenStarts starts,       out: struct TokenStarts starts,
  in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  follows_mask.R_3_4 = LookAheadN(follows_mask.R_2, 2) | follows_mask.R_2;
  starts.S_3_4       = follows_mask.R_3_4 & starts.S_gt_2;
  starts.S_gt_3_4    = starts.S_gt_2 &~ starts.S_3_4;
}

filter Mark_S_5_8(in: struct TokenStarts starts,       out: struct TokenStarts starts,
      in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  follows_mask.R_5_8 = LookAheadN(follows_mask.R_3_4, 4) | follows_mask.R_3_4;
  starts.S_5_8       = follows_mask.R_5_8 & starts.S_gt_3_4;
  starts.S_gt_5_8    = starts.S_gt_3_4 &~ starts.S_5_8;
}

// Mark follow group positions.
filter Mark_F_1(in: struct TokenStarts starts,       out: struct TokenStarts starts,
in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  starts_mask.N_1    = AdvanceN(starts_mask.N_0, 1);
  follows.F_1        = starts_mask.N_1 & follows.F;
  follows.F_gt_1     = follows.F &~ follows.F_1;
}
```

```
filter Mark_F_2(in: struct TokenStarts starts,       out: struct TokenStarts starts,
in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  starts_mask.N_2    = AdvanceN(starts_mask.N_1, 1) | starts_mask.N_1;
  follows.F_2        = starts_mask.N_2 & follows.F_gt_1;
  follows.F_gt_2     = follows.F_gt_1 &~ follows.F_2;
}

filter Mark_F_3_4(in: struct TokenStarts starts,       out: struct TokenStarts starts,
  in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  starts_mask.N_3_4  = AdvanceN(starts_mask.N_2, 2) | starts_mask.N_2;
  follows.F_3_4      = starts_mask.N_3_4 & follows.F_gt_2;
  follows.F_gt_3_4   = follows.F_gt_2 &~ follows.F_3_4;
}

filter Mark_F_5_8(in: struct TokenStarts starts,       out: struct TokenStarts starts,
  in: struct FollowsMask follows_mask, out: struct FollowsMask follows_mask) {
  starts_mask.N_5_8  = AdvanceN(starts_mask.N_3_4, 4) | starts_mask.N_3_4;
  follows.F_5_8      = starts_mask.N_5_8 & follows.F_gt_3_4;
  follows.F_gt_5_8   = follows.F_gt_3_4 &~ follows.F_5_8;
}

// Gather results.
filter Gather(in: struct Input input,
              in: struct TokenStarts starts, in: struct TokenFollows follows,
              out: struct Output output) {

  stream<8>  spans_1         = SpanUpTo(starts.S_1, follows.F_1);
  stream<16> spans_2         = SpanUpTo(starts.S_2, follows.F_2);
  stream<32> spans_3_4       = SpanUpTo(starts.S_3_4, follows.F_3_4);
  stream<64> spans_5_8       = SpanUpTo(starts.S_5_8, follows.F_5_8);
  stream<8>  spans_gt_5_8    = SpanUpTo(starts.S_gt_5_8, follows.F_gt_5_8);

  output.L_1                 = PGatherSpansLo<8,8>(input.data, spans_1);
  output.L_2                 = PGatherSpansLo<8,16>(input.data, spans_2);
  output.L_3_4               = PGatherSpansLo<8,32>(input.data, spans_3_4);
  output.L_5_8               = PGatherSpansLo<8,64>(input.data, spans_5_8);
  output.L_gt_5_8            = PGather<8>(input.data, spans_gt);
  output.output.L_gt_8_S     = Positions(starts.S_gt, 32);
  output.output.L_gt_8_F     = Positions(follows.F_gt, 32);
}

// Write result streams.
filter Write(in: struct Output output) {
  StreamWrite<8>(output.L_1);
  StreamWrite<16>(output.L_2);
```

```
  StreamWrite<32>(output.L_3_4);
  StreamWrite<64>(output.L_5_8);
  StreamWrite<8>(output.L_gt);
  StreamWrite<8>(output.L_gt_S);
  StreamWrite<8>(output.L_gt_F);
}

/* Stream Graph */
graph pipeline() {

  // structure declarations
  struct Input input;
  struct TokenStarts starts;
  struct FollowsMask follows_mask;
  struct TokenFollows follows;
  struct StartsMask starts_mask;
  struct Output output;

  // filter pipeline
  read(input);
  tokenize(input, starts, follows_mask, follows, starts_mask);
  mark_S_1(starts, follows_mask);
  mark_S_2(starts, follows_mask);
  mark_S_3_4(starts, follows_mask);
  mark_S_5_8(starts, follows_mask);
  mark_F_1(follows, starts_mask);
  mark_F_2(follows, starts_mask);
  mark_F_3_4(follows, starts_mask);
  mark_F_5_8(follows, starts_mask);
  gather(starts, follows, output);
  write(output);
}
```

Appendix F

b2^k Grammar

This appendix specifies the syntax of the b2^k intermediate representation. The grammar presented follows the EBNF conventions described in Section B.1.

F.1 b2^k Grammar Definition

Bitblock Program

program ::= { structureDef | functionDef | kernelDef } graphDef

Bitblock Kernel Graph

graphDef ::= `**graph**' graphBody
graphBody ::= `{' structDeclStmtList `}'
structDeclStmtList ::= { structType id `;' }

Bitblock Kernel Definition

kernelDef ::= `**kernel**' id kernelBody
kernelBody ::= `{' kernelInitDef KernelFunctionDef { KernelFunctionDef } `}'
kernelInitDef ::= `**init**' `{' kernelPropertyDeclStmt kernelPropertyInitStmt `}'
kernelPropertyDeclStmt ::= { varDeclStmt }
kernelPropertyInitStmt ::= { assignmentStmt }
kernelFunctionDef ::= id kernelFunctionParameterList kernelFunctionBody
kernelFunctionParameterList ::= functionParameterList
kernelFunctionParameter ::= functionParameter
kernelFunctionBody ::= functionBody

Bitblock Function Definition

```
functionDef            ::=  `function' returnType id parameterList functionBody
returnType             ::=  bitBlockType
functionParameterList  ::=  (' functionParameter { ‘,’ functionParameter } `)'
functionParameter      ::=  parameterMode type id
parameterMode          ::=  `in'  |  `out'
functionBody           ::=  blockStmt
```

Bitblock Structure Definition

```
structDef              ::=  `struct' id structDefBody `;'
structDefBody          ::=  `{' structDefMemberList `}'
structDefMemberList    ::=  structDefMember `;' { structDefMember `;' }
structDefMember        ::=  streamType id
```

Statements

```
stmt                ::=  localVarDeclStmt  |  assignmentStmt  |  funcCallStmt  |
                         ifStmt  |  whileStmt  |  returnStmt  |  annotationStmt  |
                         assertStmt
varDeclStmt         ::=  type id `;'
assignmentStmt      ::=  id assignOp expr `;'
funcCallStmtCall    ::=  funcCall `;'
ifStmt              ::=  `if' `(' expr `)' blockStmt
whileStmt           ::=  `while' `(' expr `)' blockStmt
returnStmt          ::=  `return' [ expr ] `;'
assertStmt          ::=  `assert' `(' exprList `)' `;'
blockStmt           ::=  `{' { stmt } `}'
assignOp            ::=  simpleAssignOp  |  cmpdAssignOp
simpleAssignOp      ::=  `='
cmpdAssignOp        ::=  `+='  |  `-='  |  `\='  |  `*='  |  `|='  |  `&='  |  `^='
```

Expressions

```
expr                          ::= prefixExpr | infixExpr | postfixExpr |
                                  id | cmpdId | constant |
                                  `(' expr `)'
prefixExpr                    ::= prefixOp expr
bitwisePrefixOp               ::= `~'
logicalPrefixOp               ::= `!'
infixExpr                     ::= expr infixOp expr
infixOp                       ::= arithmeticInfixOp | bitwiseShiftInfixOp |
                                  bitwiseInfixOp | logicalInfixOp |
                                  relationalInfixOp
arithmeticInfixOp             ::= `*' | `\' | `**' | `%' | `+' | `-'
bitwiseShiftInfixOp           ::= ` | `<<' | `>>'
bitwiseInfixOp                ::= `|' | `&' | `^'
logicalInfixOp                ::= `||' | `&&'
postfixExpr                   ::= funcCall | arrayIndexExpr
funcCall                      ::= simpleFuncCall | parameterizedFuncCall
simpleFuncCall                ::= id `(' exprList `)'
parameterizedFunctiontionCall ::= id `<' [ integerConstant ] `>' `(' exprList `)'
arrayIndexExpr                ::= id `[' expr `]'
```

Types

```
type                ::= compositeType | primitiveType
compositeType       ::= bitblockStructType | bitblockArrayType
bitblockStructType  ::= `struct' id
bitblockArrayType   ::= `BitBlock' `[' expr `]'
bitblockType        ::= `BitBlock'
primitiveType       ::= `int8' | `int16' | ... `intBitBlock' |
                        `uint8' | `uint16' | ... `intBitBlock' |
                        `float16' | `float32' | `float64' | `float128' |
                        `string8' | `string16' | `string32'
```

Low-level Definitions

```
exprList        ::=  expr { ‘,’ expr }
cmpdId          ::=  id `.' id
id              ::=  letter { letter | digit }
constant        ::=  stringConstant | integerConstant
stringConstant  ::=  ```' stringElement `'''
integerConstant ::=  `0' | { nonZeroDigit }
stringElement   ::=  escapeChar |
                     ``any printable Unicode character except the double quote''
charEscape      ::=  `\' ``any printable Unicode character''
hexEscape       ::=  `\x' hexDigit hexDigit
unicodeEscape   ::=  `\u' hexDigit hexDigit hexDigit hexDigit
escapeChar      ::=  charEscape | hexEscape | unicodeEscape
letter          ::=  lower | upper | `_'
lower           ::=  `a' | ··· | `z'
upper           ::=  `A' | ··· | `Z'
digit           ::=  `0' | ··· | `9'
nonZeroDigit    ::=  `1' | ··· | `9'
hexDigit        ::=  `0' | ··· | `7' | `a' | ··· | `f' | `A' | ··· | `F'
comment         ::=  `/*' ``any sequence of characters'' `*/' |
                     `//' ``any sequence of characters up to end of line'' |
                     `#' ``any sequence of characters up to end of line''
```

Appendix G

b2^k Bitblock Operations

This appendix defines the b2^k flow-parallel operations.

G.1 Flow-parallel Operations.

The b2^k flow-parallel operations, denoted $(r_n, co) = F_n(s_n, ci)$ or $(r_n, co) = F_n(s_n, t_n, ci)$ accept one or two n-bit values together with a 1-bit carry-in value, and generate an n-bit value and a 1-bit carry-out value, where $n = 2^k$ for some non-negative integers n and k. The b2^k flow-parallel operations are defined by a pair of functions f_n and g_n defined on n-bit values s_n, t_n, and carry-in value ci. The function f_n produces the n-bit result value, r_n. The function g_n produces the 1-bit carry-out value, co.

G.2 Scanning Operations

Operation	Description
$Advance_n(m_n, ci)$	The operation $(r_n, co) = Advance_n(m_n, ci)$ advances a set of positions in m_n. $r_n = (m_n + m_n + ci) \mod 2^n$. $co = \lfloor (m_n + m_n + ci)/2^n \rfloor$.

$ScanToFirst_n(m, ci)$	The operation $(r, co) = ScanToFirst_n(m, ci)$ (scan to first) scans to the first occurrence of a marker in m_n. $r_n = ((1 + (\neg c_n) + ci) \mod 2^n) \wedge c_n$. $co = \lfloor((1 + (\neg c_n) + ci))/2^n\rfloor$.
$ScanTo_n(m_n, ci)$	The operation $(r_n, co) = ScanTo_n(m_n, c_n, ci)$ scans from an initial set of marker positions in m_n through the negated spans of characters belonging to a character class in c_n. $r_n = ((m_n + (\neg c_n) + ci) \mod 2^n) \wedge c_n$. $co = \lfloor((m_n + (\neg c_n) + ci))/2^n\rfloor$.

$AdvThenScanTo_n(m_n, c_n, ci)$	The operation $(r_n, co) = AdvThenScanTo_n(m_n, c_n, ci)$ (advance then scan to) scans from an initial set of advanced marker positions through the negated spans of characters belonging to a character class in c_n. $r_n = (((m_n + m_n + ci) + (\neg c_n)) \mod 2^n)) \wedge c_n$. $co = \lfloor((m_n + m_n + ci) + (\neg c_n))/2^n\rfloor$.
$ScanThru_n(m_n, c_n, ci)$	The operation $(r_n, co) = ScanThru_n(m_n, c_n, ci)$ (scan through) scans from an initial set of marker positions in m_n through spans of characters belonging to a character class in c_n. $r_n = ((m_n + c_n + ci) \mod 2^n) \wedge (\neg c_n)$. $co = \lfloor(m_n + c_n + ci)/2^n\rfloor$.
$AdvThenScanThru_n(m_n, c_n, ci)$	The operation $(r_n, co) = AdvThenScanThru_n(m_n, c_n, ci)$ (advance then scan through) scans from an initial set of advanced marker positions through the spans of characters belonging to a character class in c_n. $r_n = (((m_n + m_n) + c_n + c_i) \mod 2^n) \wedge (\neg c_n)$. $co = \lfloor((m_n + m_n) + c_n + c_i)/2^n\rfloor$.
$MatchStar_n(m_n, c_n, ci)$	The operation $(r_n, co) = MatchStar_n(m_n, c_n, ci)$ marks all positions that can be reached by 0 or more occurrences of characters in class c_n from each position in m_n . $r_n = ((((m_n \wedge c_n) + c_n + ci) \mod 2^n) \oplus c_n) \vee m_n$.

Table G.1: Scanning operations.

G.3 Spanning Operations

Intrinsic Function	Description
$SpanUpTo_n(s_n, f_n, bi)$	The operation $(r_n, bo) = SpanUpTo_n(s_n, f_n, bi)$ marks all positions in s_n up to but not including positions in f_n. $r_n = (f_n - s_n + bi) \mod 2^n$. $b_i = \lfloor (f_n - s_n + bi)/2^n \rfloor$.
$InclusiveSpan_n(s_n, f_n, bi)$	The operation $(r_n, bo) = InclusiveSpan_n(s_n, f_n, bi)$ marks all positions in s_n and in f_n, inclusively. $r_n = ((f_n - s_n + bi) \mod 2^n) \vert f_n$. $b_i = \lfloor (f_n - s_n + bi)/2^n \rfloor$.
$ExclusiveSpan_n(s_n, f_n, bi)$	The operation $(r_n, bo)R = ExclusiveSpan_n(s_n, f_n, bi)$ operation marks all positions in s_n and in f_n inclusively. $r_n = ((f_n - s_n + bi) \mod 2^n) \wedge (\neg s_n)$. $b_i = \lfloor (f_n - s_n + bi)/2^n \rfloor$.

Table G.2: Spanning operations.

G.4 Field-parallel Operations.

The b2^k field-parallel operations are a set of collective operations defined to support SIMD operations on the n-bit fields of SIMD registers of size N, for $n = 2^k$ and $N = 2^K$, where $0 \leq k \leq K$. The b2^k field-parallel operations are organized into four categories. The b2^k operation definitions correspond to the s2^k field-parallel operations defined in Appendix C.