Energy-Efficient Pattern Matching Methods on a Fine-Grained Many-Core Platform

By

EMMANUEL OLUFEMI URAI ADEAGBO
B.S. (University of California, Berkeley) June 2009

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dr. Bevan M. Baas

Dr. Rajeevan Amirtharajah

Dr. Hussain Al-Asaad

Committee in charge
2016

© Copyright by Emmanuel Olufemi Urai Adeagbo 2016
All Rights Reserved

Abstract

The matching of one or more occurrences of a keyword within a set of input data is widely used in many datacenter applications such as large string databases, network intrusion detection systems, and search engines. As demand for datacenter performance continues to increase, energy consumption has gone up by nearly 4× within the last decade. It is therefore desirable to have very low energy dissipation per workload with low area overhead and high throughput.

This thesis first presents three energy-efficient methods for searching and filtering streamed data on a fine-grained many-core processor array: parallel, serial, and all-in-one. All three architectures provide programmable flexibility with low energy consumption. Experimental results show that for one keyword search, the parallel and serial architectures consume 2× less energy per workload than the all-in-one architecture. For two or more keyword searches, the all-in-one architecture achieves up to 2.6× higher throughput per area over the parallel architecture, and 25.6× over the serial architecture. Scaled results show that the serial and parallel designs provide 211× increased throughput per area, and yield 155× energy reduction when compared to a traditional processor (Intel Core i7 3667U). The proposed architectures are modular and easily scalable.

In addition to the proposed three energy-efficient methods for searching and filtering strings, this thesis also presents two self-adaptive string search filters for further reducing energy consumption and improving throughput of string search via self-reprogramming. Results show that the self-adaptive implementation with separated statistics block achieves about 2.8× to 4× higher throughput and throughput per area on average than the implementation with combined statistics block in statistics mode. Other performance parameters such as energy per workload, throughput and throughput per area of the main filters are approximately equal.

Next, this thesis investigates regular expression processing and its applications on the AsAP2 fine-grained many-core processor. Results show that ~99% of activity occurs

within the first core of the regular expression filter and less than 27% activity in subsequent cores. The regular expression filters achieve a throughput of 309 MB/s on average when running at the maximum voltage of 1.3 V and 17 MB/s when running at the minimum voltage of 0.675 V.

Finally, this thesis provides brief descriptions of completed projects, and future work. The future work focuses on expanding the capabilities of the regular expression work into a key application such as developing a more sophisticated web search engine.

Acknowledgments

I would like to sincerely thank my adviser Professor Bevan Baas for his support and guidance throughout my years at UC Davis. I was extremely lucky to have the opportunity of working in the VLSI Computation Laboratory (VCL) under his supervision, and it was thoroughly an enriching and delightful experience. I would also like to thank Dr. Rajeevan Amirtharajah for his constant support and valuable advice through my graduate study. I would like to thank Dr. Hussain Al-Asaad for his time and consideration in reviewing my thesis.

There were many people, and without their help this work could not be accomplished. I would like to express my appreciation to Aaron Stillmaker, Jon Pimentel, Brent Bohnenstiehl, and Timothy Andreas from VCL for helping me with reviewing my work and providing valuable feedback, helping with tools, and their endless support and valuable advice through projects. I would also like to thank other VCL members, both current and alumni: Satyabrata Sarangi, Shifu Wu, Mark Hildebrand, Bin Liu, Jeremy Webb, Michael Braly, Dr. Anh Tran, and Dean Truong for providing me with a friendly and exciting environment and inspiring me to keep pursuing my research.

I am grateful for the support from our sponsors at ST Microelectronics, NSF Grants, SRC GRC Grant 2321.001, NSF CCF Grant No. 1321163, 1018972, 0903549, 0430090, C2S2 Grant 2047.002.014, CAREER Award 0546907, DoD and ARL/ARO grant W911NF-13-1-0090, as well as the ECE and BME departmental teaching assistant support.

Finally, a special thanks goes to my beloved wife and our families for all their patience and support throughout the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The matching of one or more occurrences of a keyword within a set of input data is widely used in many datacenter applications such as large string databases [7], network intrusion detection systems [8], [9], and search engines [10].

For example, in large string databases ranging from employee information to protein databases [11], performance is measured in terms of response time to queries. For a protein database containing 122,550 protein strings with an average length of 367 amino acids, with each amino acid encoded as an ASCII character using 1 byte, a Balanced Approximate Substring Search (BASS) -tree indexing scheme [7] achieves an average response times of ~0.122 seconds as shown in Table 1.1. The BASS-tree does this by building a score matrix and making substitutions in different positions based on similarities between protein sequences (using balanced trees) to optimize both area and response time. This search scheme is also referred to as approximate string search, since typographical errors may also lead to a positive match.

The core of network intrusion detection systems rely heavily on high speed string matching capable of processing streamed data on the order of gigabits per second. For example, the Aho-Corasick algorithm [17] based bit-split FSM FPGA architecture [8] targets group sizes of 16 strings with the longest strings between 64 and 128 characters, and achieves throughput of up to 10 Gbit/sec. Table 1.2 shows comparisons to other FPGA based designs where characters per area ($Char/Area$) is how much physical resource the design consumes to store the states needed to process incoming

Table 1.1: SWISS-PROT protein database query response times for different search schemes.

Protein Pattern Length	Response Time (sec)					
	BASS-tree [7]	BLAST [12]	QUASAR [13]	MRS-index [14]	Suffix Tree [15]	Linear Scan [16]
5	0.08	0.81	0.63	2.90	2.10	6.40
10	0.10	1.12	0.95	4.60	3.80	13.80
20	0.13	1.36	1.45	7.50	7.30	22.00
40	0.18	2.05	2.35	12.70	15.20	29.50

Table 1.2: Comparison of several FPGA-based network intrusion detection string matching designs.

Architecture	Throughput (Gbit/sec)	Char/Area ($1/mm^2$)	Throughput/Area ((Gbit/sec)/mm^2)
Bit Split FSM [8]	10.1	55.2	556.3
Pre-coded CAMs [18]	9.7	23.5	228.0
Regular Expressions [19]	0.2	8.1	8.1
Distributed Comparators [20]	2.9	7.9	22.8
NFAs-Shared Decoders [21]	0.8	74.7	59.8

strings. $Throughput/Area$ represents the overall performance of each design in terms of how many incoming characters are processed for the given area.

As demand for datacenter performance (response time and throughput/area) continues to increase, energy consumption has gone up by nearly 4× within the last decade [22] as shown in Figure 1.1. It is therefore desirable to have very low energy dissipation per workload with low area overhead and high throughput. This thesis presents three string search implementations designed on a many-core platform. All three implementations offer very low energy dissipation, while individually offering energy-throughput-area trade-off.

Additionally maintaining low energy dissipation per workload while keeping a low area overhead and high throughput necessitates the system to adapt to changes in its environment, i.e. data changes. Stopping the system, tuning or reprogramming the system then turning it back on is not always an affordable option, and in these cases, the system is left to run with an inefficient

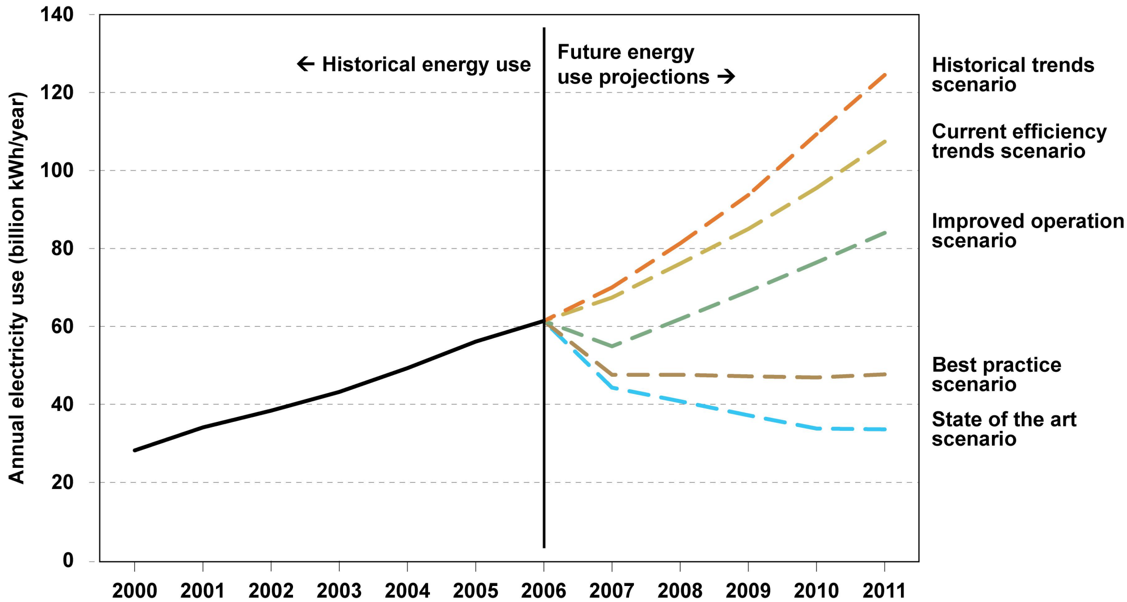

Figure 1.1: US Environmental Protection Agency Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431 (Ernest Orlando Lawrence Berkeley National Laboratory)

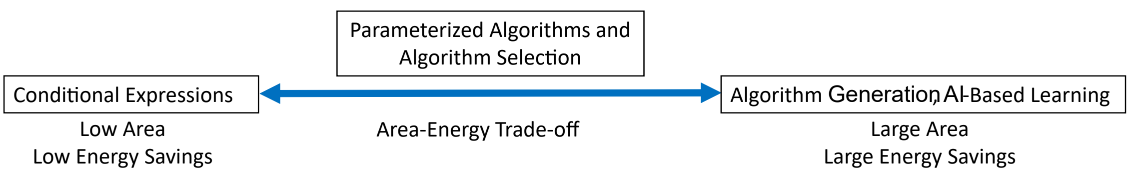

Figure 1.2: Self-adaptability spectrum showing different complexity levels. Parameterized algorithms offer a balance between area and energy.

algorithm (from an energy standpoint). It is therefore also desirable to have a system that is able to self-adapt to changes while running [23]. Figure 1.2 shows the spectrum of self-adaptability [24]. On one extreme of the spectrum are conditional expressions where the system chooses a behavior based on the evaluation of an expression. Although simplistic, these designs tend to have low area overhead. On the other extreme is evolutionary programming which consists of algorithm generation and AI-based learning. The trade-off for these complex systems is the large area required for resource management and other tasks associated with adapting the system. This thesis also presents parameterized string search implementations that adapts to changing data trend by gathering statistics on the history of processed data. The design goal is to further reduce energy consumption while increasing throughput with modest area trade-off. Once this is achieved, creating regular expressions out of the strings allows the system to find a more concise and flexible way of directly automating the matched patterns.

1.2 Thesis Organization

The remainder of this thesis is as follows: The first part of Chapter 2 discusses related work on string search architecture and some traditional methods for string search. The latter part of Chapter 2 briefly discusses related work on self-adaptive systems and their application to string search. Chapter 3 discusses three main implementations of string search, their implementations and comparison to a traditional CPU implementation of string search. Chapter 4 elaborates on self-adaptive string search and techniques for further optimizing the search algorithm. Two implementations are presented and their performances are evaluated. Chapter 5 introduces regular expression processing in hardware and previous work in the field. A regular expression implementation is provided and is applied to database sorting for generating statistics and its performance is also evaluated. Chapter 6 summarizes the thesis, discusses other major projects completed by the author, and provides directions for future work.

Chapter 2

Background

2.1 Related Work on String Search and Self-Adaptive Systems

Previous work in string search has used FPGA [25, 26], traditional CPUs [7], and GPU [27], with increasing throughput often the primary focus. FPGAs and GPUs can provide high performance but typically have high energy demands compared to traditional CPUs and fine-grained many-core arrays [28]. Traditional CPUs and GPUs offer ease of programming, while fine-grained many-core processor arrays can compute complex workloads with high performance and high energy efficiency while being smaller than the aforementioned platforms [29].

String search may also be augmented to other applications such as sorting on the same processor array [30], where the first phase would use string search to filter out undesired data then sort the remaining data. The many-core array would work as a co-processor handling the computationally intense string search and sorting while a general purpose CPU administrates more complex tasks such as deciding the input data to the many-core array, and processing the results, as shown in Figure 2.1.

Self-adaptive systems have been studied in various areas from artificial intelligence and machine learning to biology, all centralizing around the idea of feedback loops [31], where the system monitors system behavior then automatically adjusts itself either for higher performance, increase in accuracy or some other desirable metric such as robustness [24]. By applying the concept of self-adaptability to string search, it may be possible to further reduce energy consumption and achieve higher throughput adjusting to changes in data trend.

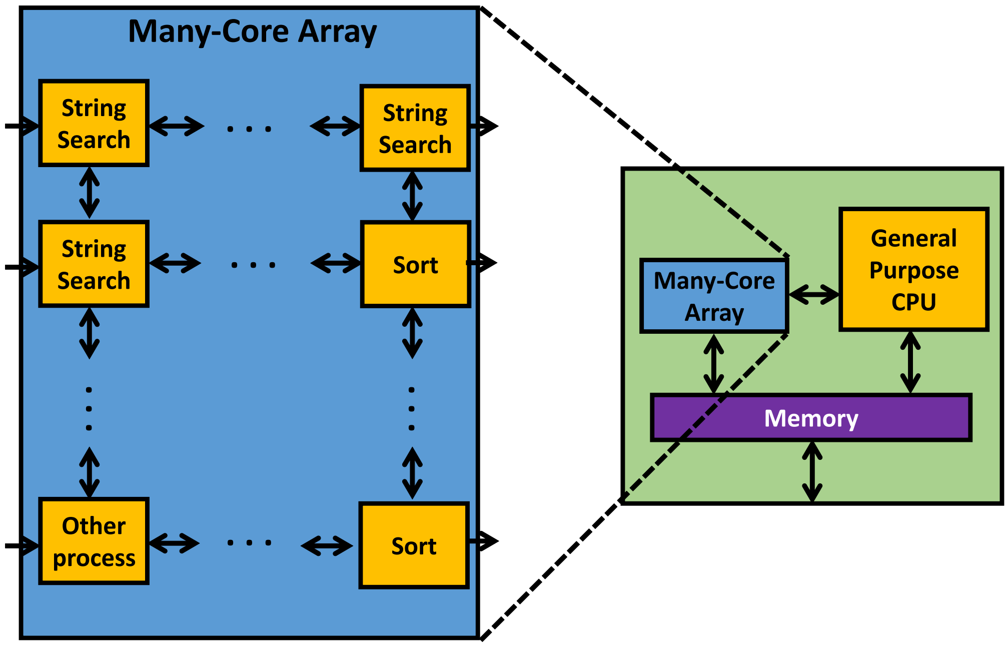

Figure 2.1: String search implemented within a many-core array acting as a co-processor with a general purpose CPU.

2.2 Related Work on Regular Expression Processing in Hardware

Regular expression is a computationally intensive problem requiring high bandwidth and memory [32]. Although there are software implementations of regular expression, increase in data-rate requirements has created a demand for hardware solutions. Divyasree et al. [26] proposed an NFA based regular expression engine for its reduction in logic and parallelism via simultaneous state transitions. Furthermore, the main engine was also designed to be dynamically re-configurable. Figure 2.2 shows a top level diagram of the proposed architecture which consists of several generic blocks cascaded together. Figure 2.3 shows details for one of these generic blocks, where a basic block consists of an AND gate followed by a flip flop for matching functionality. The overall system was built on a Xilinx Virtex2Pro FPGA is targeted at traffic screening for network security and achieved a throughput of 0.8 Gbps.

Bonesana et al. [33] proposed a regular expression architecture that does not follow the NFA or DFA conventions of regular expression typically seen. The design is a processor that reads regular expression in parallel from instruction memory and matches the expression to input from data memory. The authors explored regular expression design space using several Xilinx based

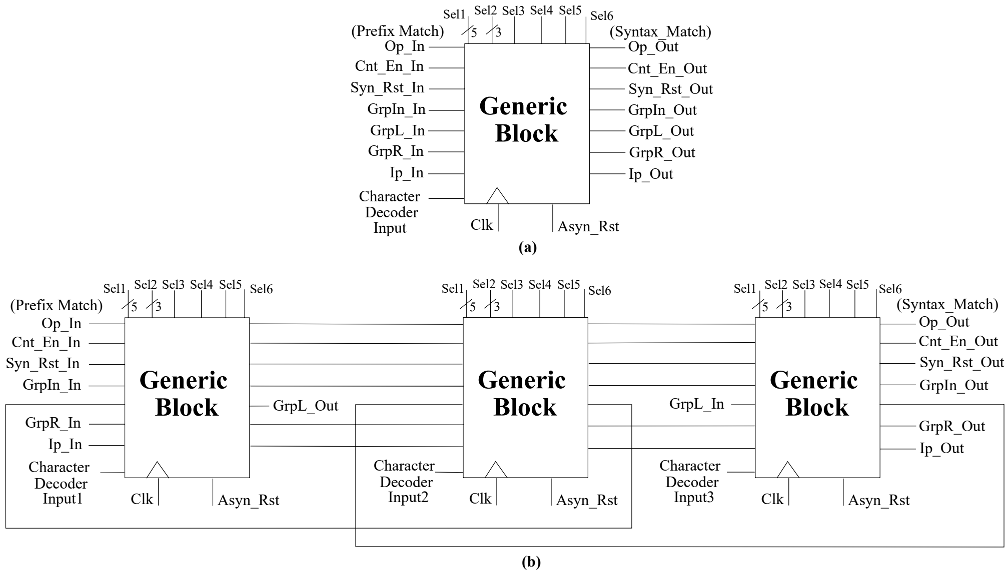

Figure 2.2: Top level diagram of proposed regular expression engine by Divyasree et al.

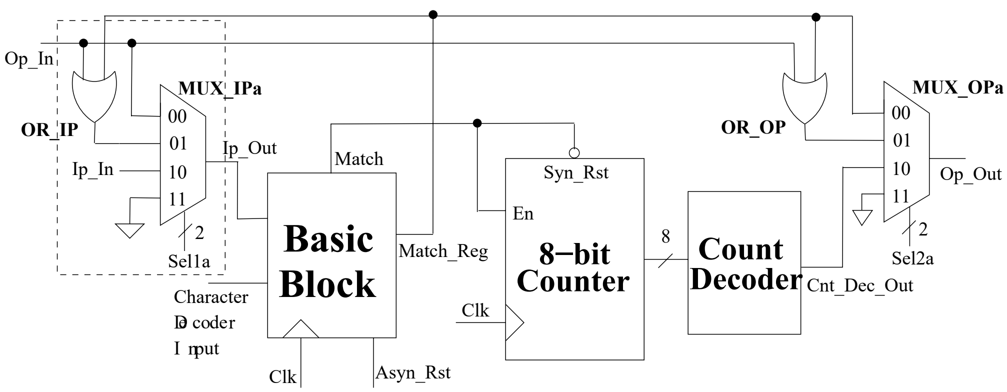

Figure 2.3: Details of a generic block from proposed regular expression engine by Divyasree et al.

FPGAs by varying RAM data width sizes and the number of regular expression units (clusters) operating in parallel within their respective systems.

Chapter 3

String Search Architectures

The primary component of the proposed string search is the filter, whose main operation is to match a keyword to input data. The core code is simple and easily replicated, and only requiring 52 assembly instructions. The pseudo code of the basic filter algorithm is shown in Algorithm 1. The filter starts by reading *inputData* into a buffer. Since a successful search requires a match of all the characters in a keyword, the buffer size is greater than or equal to the keyword length. The strings are scanned using the buffer rather than using more computationally expensive schemes such as String B-Tree, data structures or hash tables [27], [34]. Once filled, the buffer entries are compared to individual characters of the keyword. This process is repeated for as long as the following conditions are true: 1) the number of matches is less than the keyword length, 2) there is more input data to process. Since partial matches are possibilities during mismatches, most of the buffer entries must be preserved while replacing the earliest entry with a new one. The output control block sends out a “1 (True)” when the entire keyword matches, or “0 (False)” when the input data terminates prior to a keyword match. A natural requirement of the proposed string search is that the entire set of strings (e.g. a document) must be preserved when finding multiple keywords. The need to preserve the document presents some challenges for small-memory processors if the data is too large to fit in a processor’s local memory.

Algorithm 1 filter

```
while true do
    buffer ← inputData[0 : keywordLength − 1]
    i ← 0
    localMatch ← 0
    while inputData[0] ≠ EOF and localMatch == 0 do
        if buffer[i] == keyword[i] then
            if i == keywordLength then
                localMatch ← 1
            else
                i++
            end if
        else
            buffer << 1 char
            buffer[0] ← inputData[0]
            i ← 0
        end if
    end while
    if firstFilter then
        output ← localMatch
    else
        Wait for inMatch
        output ← (inMatch and localMatch)
    end if
end while
```

3.1 Serial Implementation

The serial implementation is a pipelined architecture with preallocated (e.g. 16 KB) block of memory per processor on the many-core array. Each filter in Figure 3.1 is a processor running Algorithm 1. Processing begins when *inputData* streams into both Filter *1* and Memory *1* in parallel. If Filter *1* outputs a “True” *Match*, it is sending a “1” signal to both Memory *1* and Filter *2*. *Data* streams from Memory *1* to Filter *2* and Memory *2* in parallel after which Filter *2* starts processing *Data*. Filter-memory pairs in the latter parts of the chain conditionally run based on match results of previous filter-memory pairs. Filter *N* produces a “True” *Merged Boolean Match* if all subsequent searches were a success. Due to the sequential nature of the serial architecture, if less common or rare keywords are programmed into earlier filters in the chain, subsequent filters are less likely to run as frequently because of the restrictive nature of the filter chain.

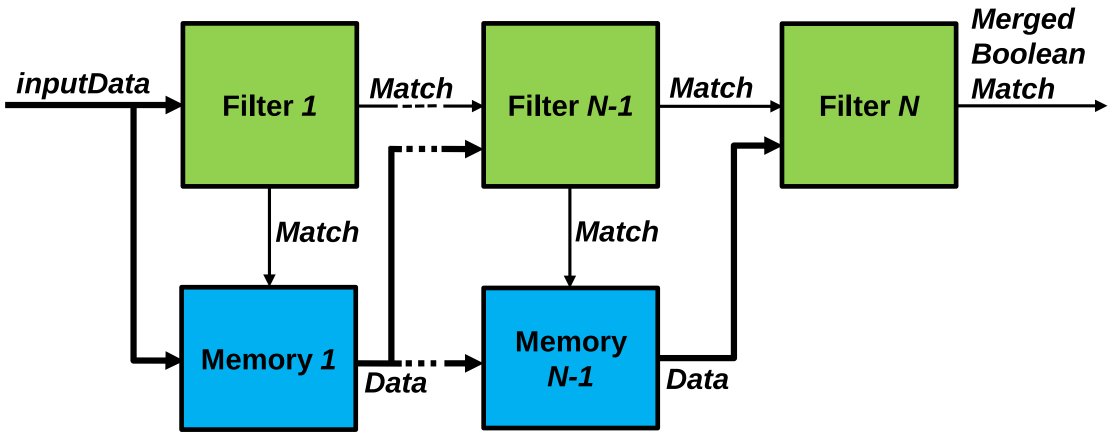

Figure 3.1: Serial architecture data flow highlighting the major control signals of each filter and the on-chip memory. The architecture is pipelined with each filter a processor running Algorithm 1.

3.2 Parallel Implementation

Each filter in Figure 3.2 is a processor running Algorithm 1 where *inputData* is streamed to all of them in parallel for processing. Each *Match* output is boolean merged to *Matches* of subsequent filters and Filter N produces a "True" *Merged Boolean Match* if all subsequent searches are a success. Filters shutdown either if they find a match and wait for other processors, or if *inputData* is empty. The modularity of the parallel architecture enables it to easily scale to larger search queries.

3.3 All-In-One (AIO) Implementation

The AIO architecture combines multiple keyword search operations into the minimum required filter, typically one processor as shown in Figure 3.3. The processor runs Algorithm 1 on multiple keyword searches by using its internal data memory to manage the keywords and *Matches*. When *Match* is "True", a flag corresponding to the matched keyword is asserted thereby disabling future searches of the keyword. When all keyword flags are asserted or when *inputData* is empty, AIO produces *Merged Boolean Match.* Multiple AIO architectures working together allow for dense search queries.

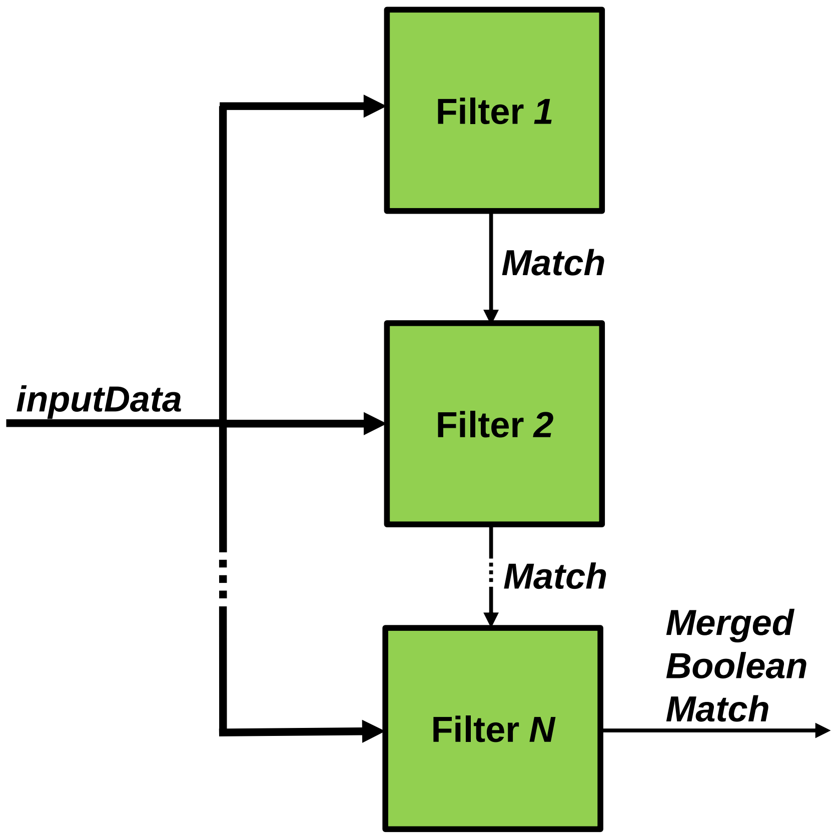

Figure 3.2: Parallel architecture data flow highlighting the major control signals of each filter. Each filter is a processor running Algorithm 1 directly on *inputData*.

3.4 Data Generation and Test Conditions

The string search architecture performances are evaluated using a list of keywords containing ~350,00 words that are randomly generated from the English dictionary [35]. The input data is generated in 8 KB sizes. For a set of keywords, a page excluding these keywords is first generated. A real dictionary is used instead of generating a page of random characters because real words result in more realistic performance data that closely matches real world workloads. Once the pages are generated, the keywords are then inserted in random locations within those pages as shown in Figure 3.4.

The input data for the architectures is generated using three parameters. The first parameter is the number of keywords, and it sets the number of filters per keyword. The second parameter is the keyword length which sets the size of a keyword at one byte per character. The last parameter is the location of a keyword in a data page. When a keyword is chosen, it is assigned

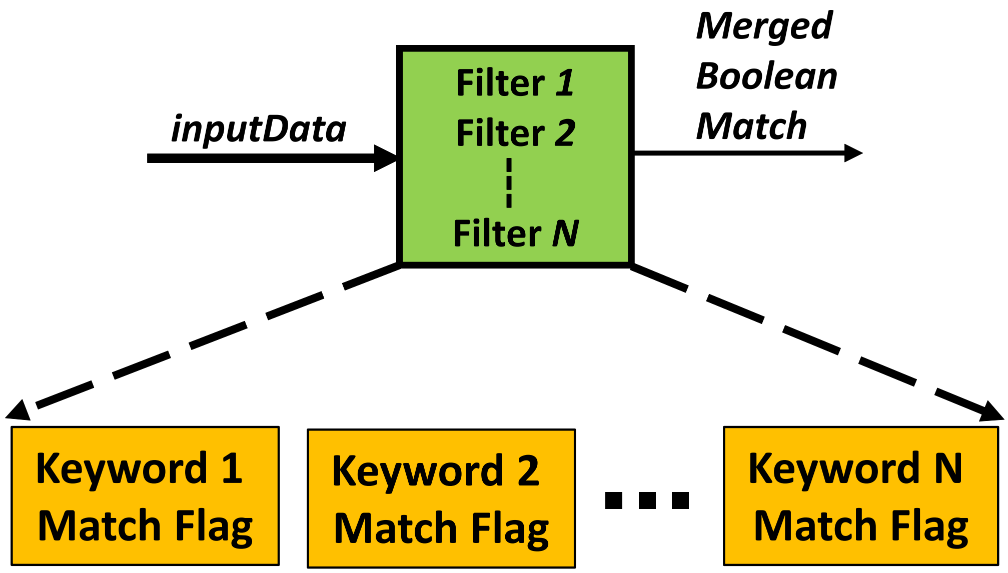

Figure 3.3: AIO architecture data flow with combined multiple keyword search operations. The structure is a processor running Algorithm 1 on multiple keywords.

a random location on a page. 1000 iterations are carried out to produce consistent averaged results.

3.5 Analysis

3.5.1 Experimental Results

The serial, parallel and AIO architectures are simulated on a simulator that uses measured values from the AsAP2 chip [28] operating at a supply voltage of 1.3 V, with 164 independently-clocked homogeneous programmable processors running at 1.2 GHz. Each processor uses 63 simple instruction types within its instruction set. The chip also includes three 16 KB memories with the entire chip connected via a 2D-mesh, allowing for nearest neighbor communication and long distance communication. Each processor contains 128x35-bit instruction memory, 128x16-bit data memory, and two dual clock 64 x 16-bit FIFO buffers for communication between processors [30]. The chip was fabricated in 65 nm technology with each processor occupying 0.17 mm^2.

Energy per workload for each architecture is defined as the total energy consumed when processing *inputData* divided by the total number of bytes in *inputData.* Figure 3.5 shows that for one keyword, the serial and parallel implementations consume 2× less energy per workload

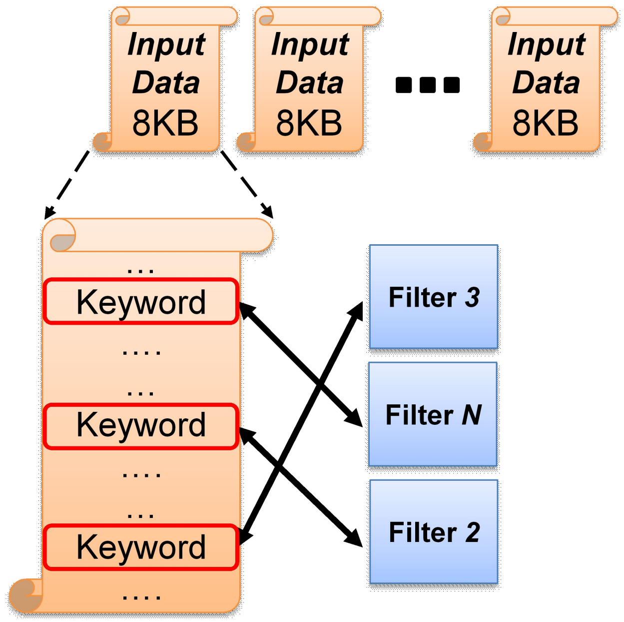

Figure 3.4: Mapping assignments of keywords to filters

than the AIO implementation. For five keywords, the AIO implementation consumes 1.5× less energy per workload over the serial and parallel implementations with majority of its energy consumption from branching overhead. The serial, parallel, and AIO implementations consume 22.55 nJ/byte, 21.16 nJ/byte, and 15.06 nJ/byte, respectively, making the AIO implementation the most energy efficient. The energy overhead in the serial architecture comes from the energy required for communication between its filters and the inclusion of the 16KB memory block(s).

For a given architecture and 16-bit word size, area per throughput is defined as the area occupied by the programmable processors and memory divided by how quickly *inputData* is processed in units of mm^2/(MWords/sec), where a word is 16 bits wide. Figure 3.6 plots the trade offs between energy per workload vs area per throughput for each implementation. For one keyword, the serial and parallel architectures consume approximately 2× less energy per workload and 1.5× less area per throughput than the AIO architecture. For three keywords, AIO occupies approximately 2× and 7× less area per throughput than the parallel and serial architectures, respectively. In contrast, the serial and parallel architectures consumes approximately 2.6× less energy per workload than the AIO architecture, with similar trends at five keywords.

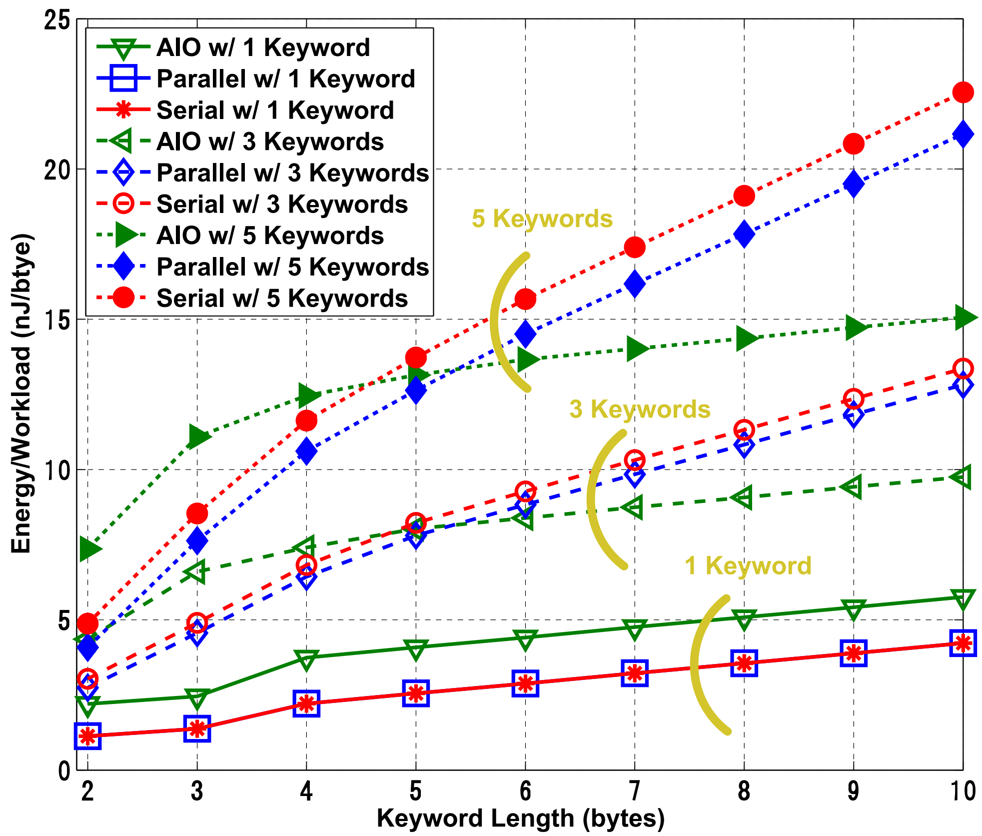

Figure 3.5: Energy per workload versus keyword length for different keywords.

Figure 3.7 shows that for one keyword, the parallel and serial architectures achieve 1.6× higher throughput than the AIO architecture. For five keywords, the parallel architecture is 5.8× higher in throughput than the AIO architecture, and 5× over serial. Longer keyword lengths require more processing, which lead to an average throughput drop from 33 to 6 MWords/sec.

In the case of one keyword, the parallel and serial architectures have the exact same energy, throughput, and area because the serial architecture only requires memory for two or more keywords. The AIO architecture for the one keyword case still has a complexity overhead and therefore consumes slightly more energy with a slightly lower throughput.

3.5.2 Comparisons

As a reference point for how well the architectures perform, similar data inputs are processed in C++ on an Intel Core i7 3667U processor (22 nm fabrication technology) for comparison. The

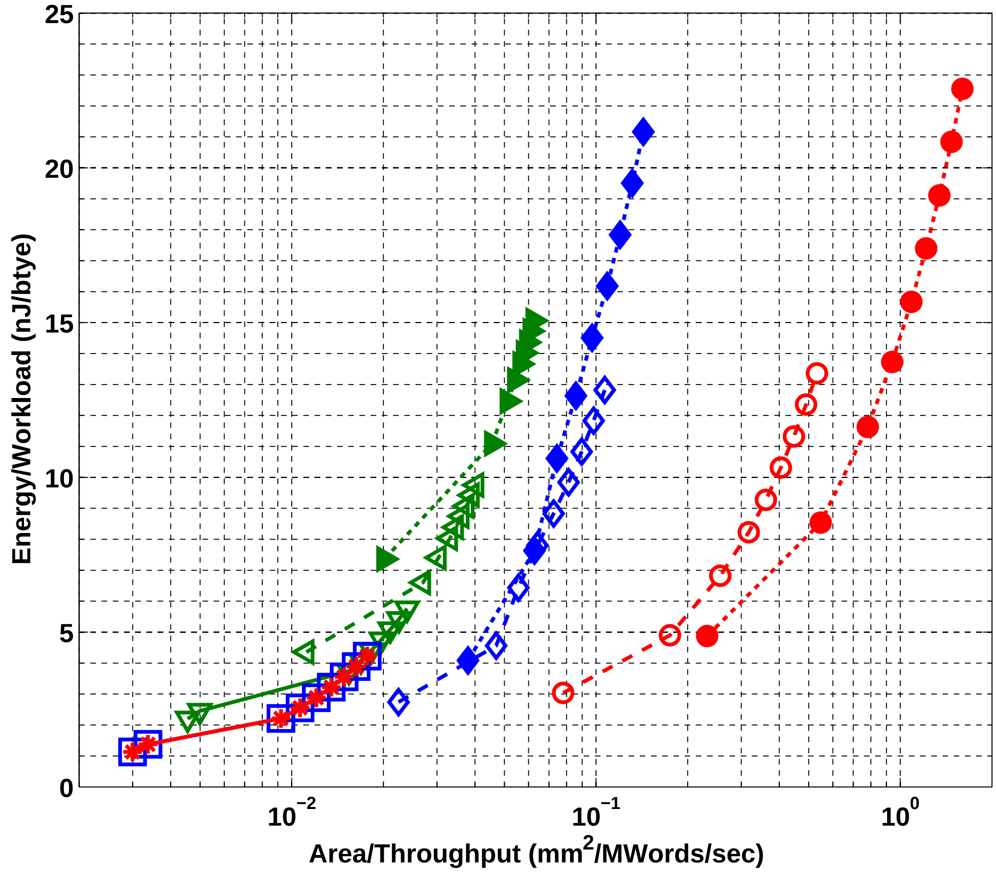

Figure 3.6: Energy per workload versus area per throughput for different keywords. See Figure 3.5 for legend.

fabrication technology for the serial, parallel, and AIO architectures are 65 nm. The results in Table 3.1 and Table 3.2 are for 6 char keyword lengths (6 bytes) showing both unscaled and scaled results. In the scaled columns, the values are scaled from 65 nm to the 22 nm node to match the many-core platform on which the workload was performed to the Intel Core i7 [6]. Table 3.1 shows for one keyword that the serial and parallel architectures provide 155× in energy savings, and with a 211× increased throughput per area over the Intel Core i7 3667U. For five keywords, the AIO architecture provides 17× in energy savings, and with 69× in increased throughput per area over the Intel Core i7 3667U.

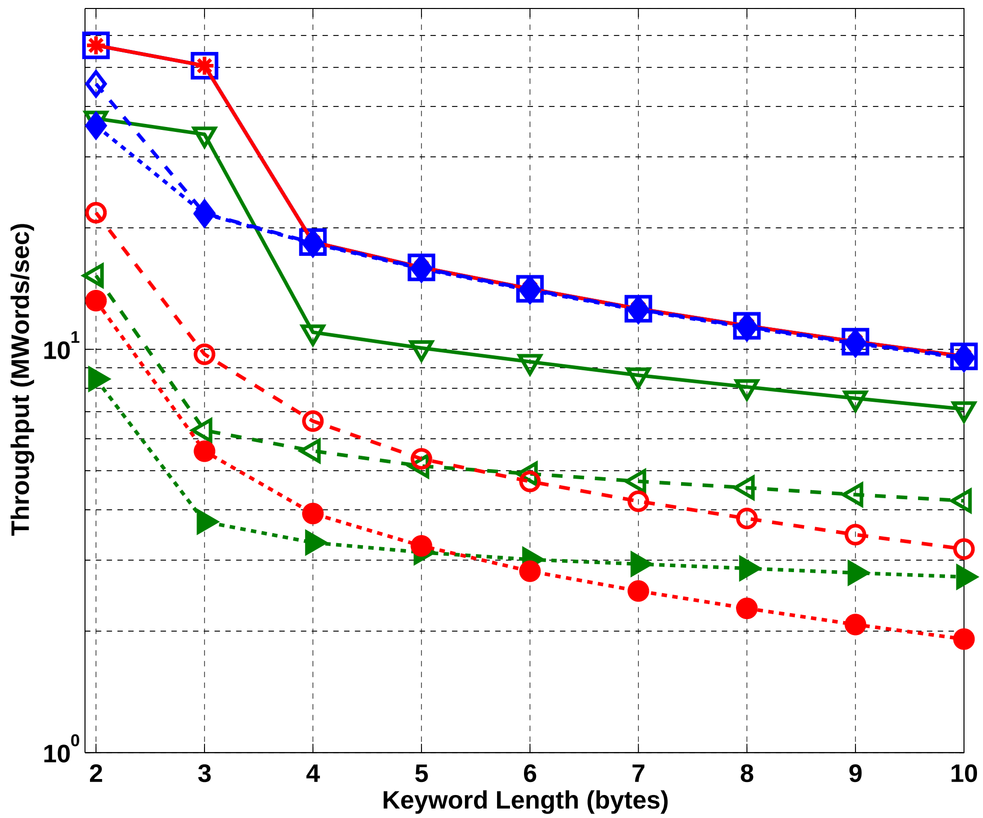

Figure 3.7: Throughput comparison at each keyword length for different keywords. See Figure 3.5 for legend.

Table 3.1: Scaled energy per workload, throughput and throughput per area for keyword length of 6. Values are scaled to 22 nm [6].

	Architecture	Scaled Energy/Workload (nJ/byte)	Scaled Throughput (MWords/sec)	Scaled Throughput/Area ((MWords/sec)/mm^2)
1 Keyword	Intel Core-i7 3667U	77	408	13.8
	Serial	**0.50**	55	**2910**
	Parallel	**0.50**	55	**2910**
	AIO	0.80	36	1920
5 Keywords	Intel Core-i7 3667U	41	256	9.02
	Serial	2.8	11	35.1
	Parallel	2.6	54	362
	AIO	**2.4**	12	**621**

Table 3.2: Unscaled energy per workload, throughput and throughput per area for keyword length of 6.

	Architecture	Unscaled Energy/Workload (nJ/byte)	Unscaled Throughput (MWords/sec)	Unscaled Throughput/Area ((MWords/sec)/mm^2)
1 Keyword	Intel Core-i7 3667U	77	408	13.8
	Serial	**2.8**	14.1	**83.0**
	Parallel	**2.8**	14.1	**83.0**
	AIO	4.4	9.31	54.7
5 Keywords	Intel Core-i7 3667U	41	256	9.02
	Serial	16	2.82	0.920
	Parallel	15	14.0	10.3
	AIO	**14**	3.00	**17.7**

Chapter 4

Self-Adaptive String Search

This Chapter is an extension of Chapter 3, and explores other methods for further reducing energy consumption during string search. Energy consumption by the string search filters are data and structure dependent. For example, in the serial implementation in Figure 3.1, if keywords in the first few filters occur frequently, majority of the chain will process data more often and therefore consume more energy. In an opposite case where keywords in earlier filters occur less frequently in the data stream, majority of the chain will most likely run less frequently since the less common keywords will filter out most of the pages and thereby keep latter filters idle. This will lead to an overall lower power consumption by the architecture. Knowing the data trend ahead of time allows the designer to program a specific sequence of keywords that will lead to the lowest energy consumption per workload. Even then, once the architecture starts running, if the data trend begins to change, the once efficient design starts operating inefficiently, thereby requiring a reprogramming of the system. If the system is able to somehow detect the changes in the data trend, it can be designed to adapt to these changes.

4.1 Statistics Based Processing

Statistics based processing is a form of self-adaptive string search with basic blocks derived from the string search implementations in the Chapter 3. Three main blocks are developed: statistics, reprogram, and main filters shown in Figure 4.1. Details of each are described in their respective sections below.

4.1.1 System Overview

The system starts in a state where keyword placement within each filter is random. This is the training state during which the statistics block counts the number of keyword matches within *inputData*. Filter N of the statistics block outputs *Keyword Match Frequencies* containing the number of occurrences for each keyword. The main filters block run in parallel with the statistics block and also processes the same *inputData*. The reprogram block, using the *Keyword Match Frequencies* sorts the keyword(s) according to configuration conditions then reprograms the main filter blocks using the sorted *Keywords*. Once the reprogram is done the main filters processes new keywords as a reorganized block, sending out final *Merged Boolean Matches*.

The self-adaptive string search filters use an optimized version of the basic filter algorithm introduced in Chapter 3. The pseudo code of the filter is shown in Algorithm 2. When a filter begins, every character has an equal but low probability of been chosen. Once the first character matches, the probability of matching a second character within the keyword increases with subsequent matches which is attributed to character correlation within a word set. The minimum keyword length for filtering is two characters because single character keywords are ubiquitous. Therefore the first step is to check for the first two matching characters in the data stream is a loop. This process is sped up by doing multiple checks via loop unrolling (not shown in pseudo-code for clarity) for the first two matching characters in *inputData*. Core instruction memory size limits the number of times the check for the first two characters may be loop unrolled. Keyword lengths of two characters end processing at this point and the *localMatch* counter increments if a match is found. For longer keyword lengths, the filter reads *inputData* into a buffer then starts checking the buffer for matches referred to as "main match check". When a character in the buffer does not match the corresponding keyword character, the algorithm does a partial match check starting with the second oldest character in the buffer. Once the partial match check goes through the *bufferLength*, the partial match will either end up discarding just the earliest entry in the buffer and going back to main match check if the partial match(es) is/are successful or, flush the entire buffer if there are zero matches and begin a new search for the first two characters in the keyword. Once the filter reaches the end of the data stream (EOF), it sends out keyword matches (*inMatches*) from prior filters if any as well as the current filter's *localMatch*.

Algorithm 2 Optimized Filter

```
while true do
    readControl ← 0
    i ← 0
    localMatch ← 0
    while inputData[0] ≠ EOF do
        buffer ← inputData[0 : 1]
        if (buffer[0] == keyword[0]) and (buffer[1] == keyword[1]) then
            if keywordLength == 2 then
                localMatch++
                readControl ← 1
            else
                keywordCounter ← 2
                i ← 2
                partialMatchCounter = 0
                while (keywordCounter ≠ keywordLength) or
                        (partialMatchCounter! = bufferLength) do
                    buffer[0] ← inputData[0]
                    i++
                    if buffer[i] == keyword[keywordCounter] then
                        keywordCounter++
                    else
                        keywordCounter ← 0
                        while partialMatchCounter! = bufferLength do
                            partialMatchCounter++
                            if buffer[partialMatchCounter] == keyword[keywordCounter] then
                                keywordCounter++
                            else
                                keywordCounter ← 0
                            end if
                        end while
                    end if
                end while
                if keywordCounter == keywordLength then
                    localMatch++
                    readControl ← 1
                end if
            end if
        end if
    end while
end while
```

4.1.2 Statistics Gathering

4.1.2.1 Separated Statistics Block

Statistics based processing requires the training state in order to adjust the main filters to suit the incoming data trend. This adjustment period can be considered overhead since the system can only be reprogrammed after collecting statistics on some data first. The architecture of the

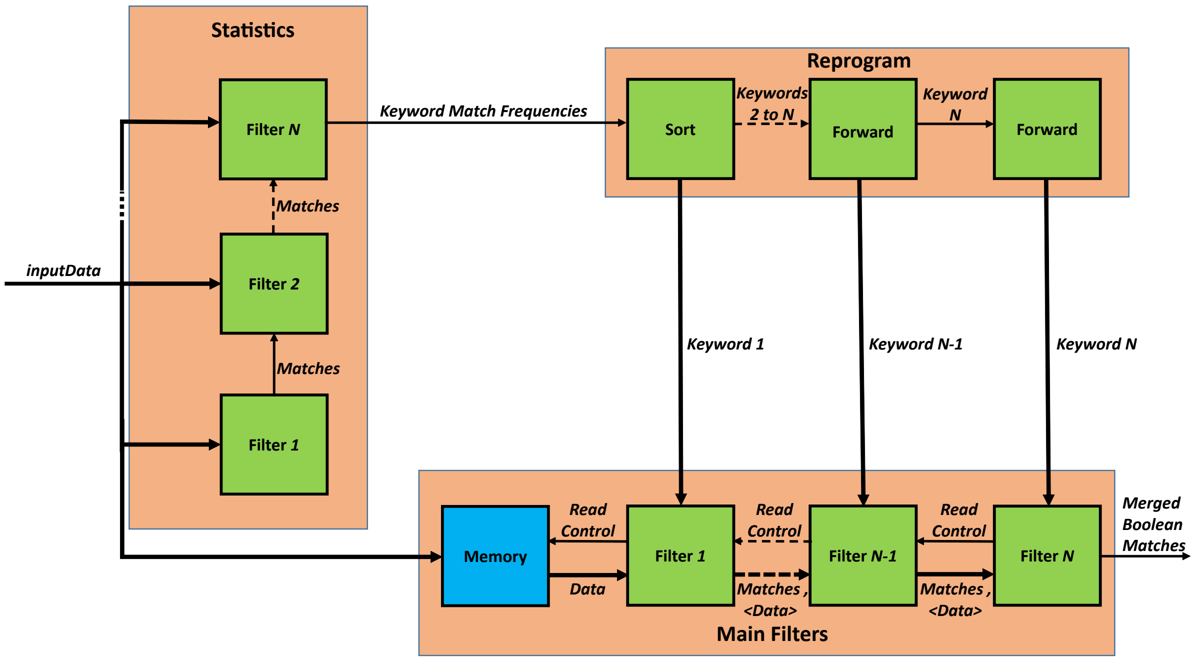

Figure 4.1: Data flow of the three main blocks of statistics based processing: statistics, reprogram, and main filters. This version shows statistics and main filters blocks separated.

Algorithm 3 self-adaptive string search main filter control

if $thisFilter \neq firstFilter$ **then**
    repeat
        $output \leftarrow inMatches$
    until $inMatches$ is empty
end if
$output \leftarrow localMatches$

statistics block sets the latency as well as throughput of the *Keyword Match Frequencies.* Analysis from Chapter 3 shows that the parallel architecture is well suited for the statistics gathering role and is therefore the design base for the statistics block as shown in Figure 4.1. Each of the statistics block filters run Algorithm 2 (excluding *readControl).* Additionally, the statistics block filters also run Algorithm 3 for sending out *Keyword Match Frequencies.*

4.1.2.2 Combined Statistics Block

The additional area overhead from a SSB is eliminated by combining the statistics block's functionality with that of main filters. Figure 4.2 shows the result of combining the two blocks, achieved by restructuring the statistics block as a serial based implementation.

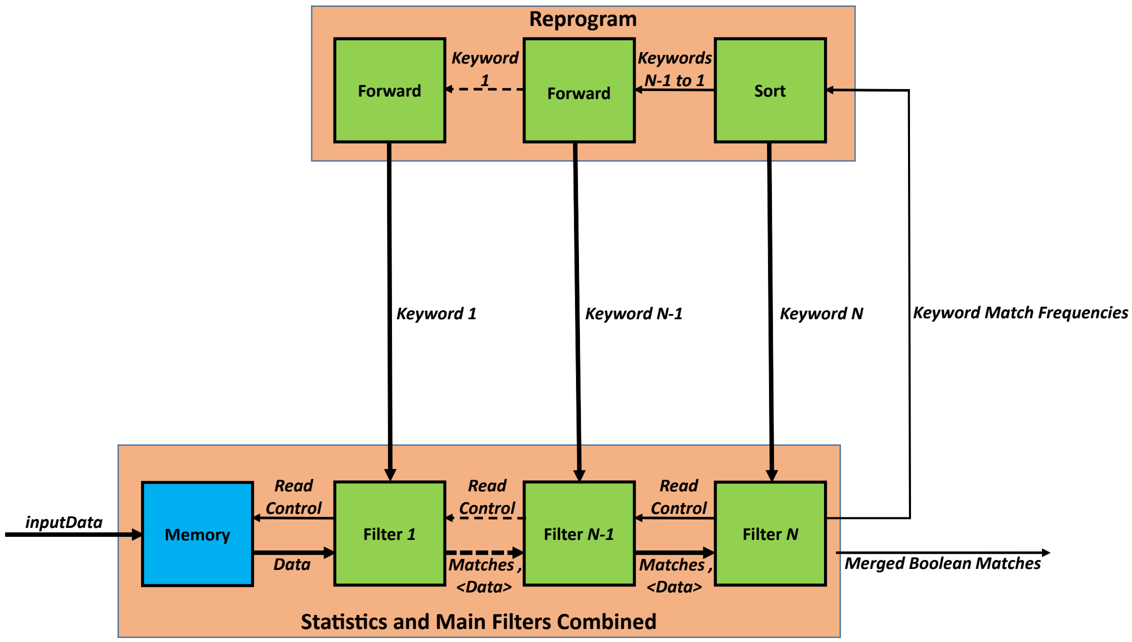

Figure 4.2: Data flow of the three main blocks of statistics based processing: statistics, reprogram, and main filters. This version shows statistics and main filters blocks combined.

During the training state, the system passes all processed keywords from one filter to the next regardless of zero matches. Once the training state produces *Keyword Match Frequencies,* the reprogram block sorts the keyword(s) according to configuration conditions then reprograms the main filter blocks using the sorted *Keywords.* Once the reprogram is done the main filters processes new keywords as a reorganized block, sending out final *Merged Boolean Matches.*

4.1.3 Core Reprogramming

The designed system self organizes main filters based on results from the statistics block. The self-organization may occur as one of two methods: 1) replace assembly instructions of cores as a form of reprogramming or 2) replace the keywords in the data memory. It is more efficient from a programming standpoint to replace the data memory keywords of the filters than to completely replace instructions because the filters are mostly homogeneous with their biggest difference being their respective data memory where keywords are stored. Prior to reprogramming, *Keyword Match Frequencies* are sorted using an adaptive merge-sort algorithm referred to as timsort [36, 37] for speed and memory efficiency. Once sorted, the reprogram block writes the data memory of each main filter block with the sorted keywords. The forward sub-blocks act as helpers for forwarding appropriate keywords to their respective main filters. The reprogram block completes when the last keyword in the forward chain completes.

4.1.4 Main Filter

The main filters block is based on the serial architecture from Chapter 3 with some structural differences. In the serial architecture, each additional filter in the chain also requires an additional pre-allocated block of memory effectively increasing the total area by more than the additional cores. In contrast, the main filters block uses one memory unit for all filters in the chain and uses a *readControl* signal between the filters to communicate with the memory. Each filter runs Algorithm 2, and Algorithm 4 for output control. The control algorithm for the parallel based statistics block and the serial based main filters block differ by the *inputData* forwarding required by the main filters. When a filter has greater than zero matches, when it reaches the end of *inputData* it sends a "True" *readControl* signal (represented by "1") to the previous filter to forward a fresh read of *inputData* for the next filter. The first filter passes the *readSignal* to memory. A "0" *readSignal* indicates a write of new *inputData* to memory. Filters in the latter parts of the main filters block conditionally run based on the match results of previous filters. Filter N produces the final *Boolean Matches* of all previous searches. Due to the sequential nature of the serial architecture, if less common or rare keywords are programmed into earlier filters in the chain, subsequent filters are less likely to run as frequently because of the restrictive nature of the filter chain. Appendix B contains a sample code of main filter designed for 3 keywords.

4.2 Data Generation and Test conditions

Similar to the string search architectures in Chapter 3, the performance of the self-adaptive string search architecture is evaluated using a list of unique keywords containing ~350,000 words that are randomly generated from the English dictionary. The input data is generated in 8 KB sizes. For a set of keywords, a page excluding these keywords is also first generated. The difference between tests in this architecture versus the string search architectures from the previous Chapter is the additional degree of freedom of keyword probabilities. In the previous Chapter all keywords have an

Algorithm 4 Self-adaptive string search main filter control

```
if thisFilter ≠ firstFilter then
    repeat
        output ← inMatches
    until inMatches is empty
end if
output ← localMatches
while readControl == 1 do
    if thisFilter == firstFilter then
        Memory ← readControl
    end if
    repeat
        output ← inputData
    until inputData == EOF
end while
```

equal probability of appearing within *inputData* and do not change over time which will be referred to as non-statistic(constant) searching. In contrast the processing of *inputData* where keywords have unequal probabilities of appearance and in addition may change over time will be referred to as statistic(dynamic) searching. Dynamic searching is used for the self-adaptive string search architecture because it is able to adequately test the self-organization of the design better than a constant search would. The unequal probabilities between the keywords leads to an unbalance between the total amount of each keyword which leads to a trend that the system may adapt to. For realistic performance, keyword probabilities are modeled according to real world data, i.e. the Corpus of Contemporary American English frequency data [38, 39] containing ~450 million words (including their lemmas and variations). Corpora such as these allow testing for occurrences of words. For example, the most common word is "the" with a probability of $^{22,038,615}/_{450,000,000} = 0.049$. The 100th most common is "well" with a probability of $^{411,776}/_{450,000,000} = 0.0009$, while the 5000th most common is "till" with a probability of $^{5079}/_{450,000,000} \approx 0.0$.

inputData for the architectures is generated using four main parameters. The first parameter is the number of keywords, and it sets the number of filters per keyword. The second parameter is keyword probability which leads to the frequency of keywords in *inputData.* The third parameter is the keyword length which sets the size of a keyword at one byte per character. The last parameter is the location of a keyword in a data page. When a keyword is chosen, it is assigned a random location on a page. 1000 iterations are carried out to produce consistent averaged results. Please refer to Appendix A for the generator code.

4.3 Analysis

The self-adaptive string search architecture with separate statistics and combined statistics block are both simulated on a simulator that uses measured values from the AsAP2 chip. A top level script written in Python scripting language is used to sort and forward intermediate results, automating tasks such as generating parameters and inputs to the simulator, and post simulation analysis. Further details on the top level script may be found in Appendix C. MATLAB is used to create the resulting plots, and each design is bench-marked based on *Keyword Match Frequencies* generated by the statistics block and used to reprogram the main filters for 3 cases: 1) reprogram based on HTLF 2) reprogram based on LTHF 3) unsorted frequency of keywords. Figure 4.3, Figure 4.4, and Figure 4.5 plot the self-adaptive filter with SSB and show how this separation affects the design's performance. For one keyword only a single filter is necessary therefore all implementations have equal performance. For two or more keywords *Keyword Match Frequencies* and reprogramming is employed. Figure 4.3 shows that for three keywords, LTHF reprogrammed main filters consume 1.1× less energy than non-reprogrammed main filters and 1.2× less energy than HTLF reprogrammed filters. The LTHF reprogrammed main filters block consumes 1.06× more energy than statistics block, suggesting that a parallel design consumes approximately the same amount of energy as a serial based reprogrammed main filter for small keywords. For five keywords LTHF reprogrammed main filters consume 7.7× less energy than non-reprogrammed main filters and 6.6× less energy than HTLF reprogrammed filters. In addition, the LTHF reprogrammed main filters block consumes 6.6× less energy than the statistics block. At five keywords, the LTHF reprogrammed main filters, the HTLF reprogrammed filters, the non-reprogrammed main filters, and the statistics block consume on average 7.7 nJ/byte, 30.9 nJ/byte, 27.9 nJ/byte, and 20.3 nJ/byte, making the LTHF reprogrammed filters the most energy efficient. As longer and more keywords are processed, the potential to save more energy increases because sorting by low to high frequency priorities rare occurring keywords in front of the main filters chain. Subsequent filters that contain more common words remain idle longer and run only when the rare keywords have been found. The system thereby saves the most energy by only running the needed parts of the main filters.

Figure 4.4 plots the trade-offs between energy per workload vs area per throughput for each implementation. For three keywords, the statistics block consumes approximately 1.2× less energy per workload and 1.6× less area per throughput than the LTHF reprogrammed main filters block, while consuming approximately 1.5× less energy per workload and 3× less area per throughput than either the HTLF reprogrammed or the non-reprogrammed main filters block. For five keywords, the LTHF reprogrammed main filters block consumes approximately 5× less energy per workload and 1.5× less area per throughput than the statistics block, while consuming as low as 6.5 to 7.5× less energy per workload and 6 to 7× less area per throughput than either the HTLF reprogrammed or the non-reprogrammed main filters block, respectively.

Figure 4.5 shows that for three keywords, the statistics block achieves 1.4× and 2.6× higher throughput on average than the LTHF reprogrammed main filters block or the non-reprogrammed main filters block, respectively. This implies that the throughput bottleneck at three keywords is the serial based main filters block before and after it is reprogrammed. Therefore a parallel non-reprogrammed structure operates faster than a reprogrammed serial based main filter on smaller sets of keywords. For five keywords, the LTHF reprogrammed main filters block achieves 1.7× higher throughput on average than the statistics block, and 5.8× over the non-reprogrammed main filters block. More keywords and longer keyword lengths require more processing time, thereby shifting the throughput bottleneck to the statistics block. The reprogrammed main filters block also achieves a throughput increase of 5.8× over the non-reprogrammed main filters block, with an average throughput of 270 MWords/sec after reprogramming.

Figure 4.6, Figure 4.7, and Figure 4.8 plot the self-adaptive string search architecture with CSB's performance. From one keyword to three keywords there is no change in performance between the different modes of operation. Figure 4.6 shows that for five keywords the LTHF reprogrammed main filters consume approximately 8× less energy than non reprogram and 12× less energy than when in statistics mode.

Figure 4.7 plots the trade offs between energy per workload vs area per throughput for each mode of the self-adaptive filter with CSB. For five keywords, the LTHF reprogrammed main filters block consumes approximately 11.5× less energy per workload and 8× less area per throughput than when in statistics mode, while consuming as low as 8× less energy per workload and 7× less area per throughput than either the HTLF reprogrammed or the non-reprogrammed modes, respectively.

Figure 4.7 shows that for five keywords, the LTHF reprogrammed main filters block achieves 8× higher throughput on average than in statistics mode, and 6× higher throughput than non-reprogrammed mode, with an average throughput of 313 MWords/sec after reprogramming.

Table 4.1 compares the performance of the two self-adaptive string search architectures with SSB versus CSB. The SSB and CSB main filters consume roughly the same amount of energy, while the SSB statistics consumes as low as 2× less energy than CSB in statistics mode at 5 keywords. For three keywords, the SSB main filters achieve about 1.8× higher throughput on average over the CSB main filters, while for five keywords the CSB main filters achieve about 1.2× higher throughput on average than the SSB main filters. For three keywords, the SSB statistics achieves about 2.8× higher throughput than CSB in statistics mode and increasing to 4× higher throughput at five keywords. This is due to the fact that the SSB statistics operates on multiple keywords in parallel while CSB in statistics mode operates on keywords sequentially. For three keywords, the SSB statistics achieves about 2.7× higher throughput per area on average than CSB in statistics mode, and increasing to 3.9× higher throughput per area at five keywords. Although the trade-off for the SSB statistics' higher throughput is a larger area (since extra cores are used for the SSB statistics), this trade-off has a negligible impact on the SSB statistics' throughput per area versus CSB in statistics mode.

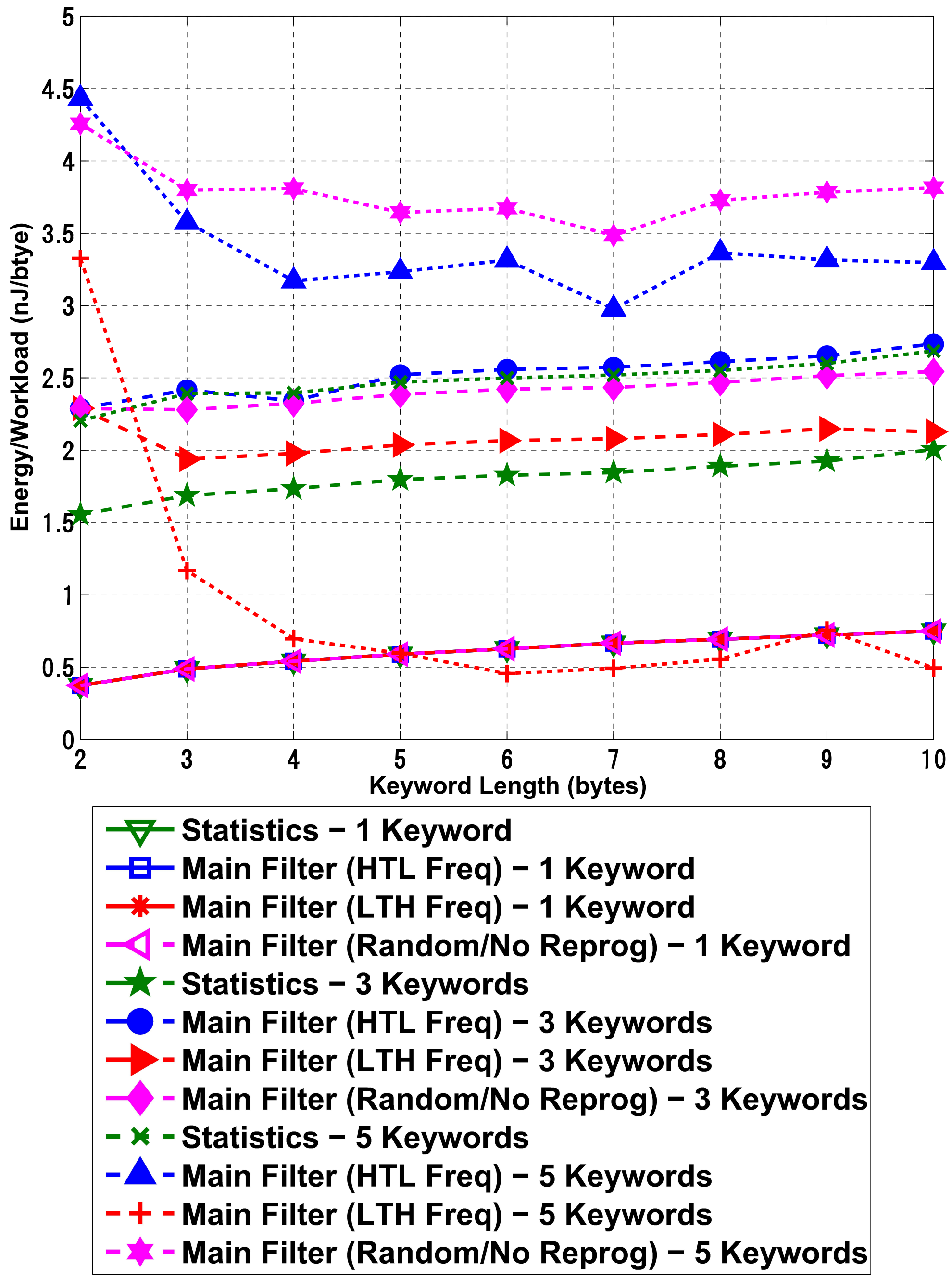

Figure 4.3: Energy per workload versus keyword length at different keywords for statistics based processing with separated statistics and main filters blocks.

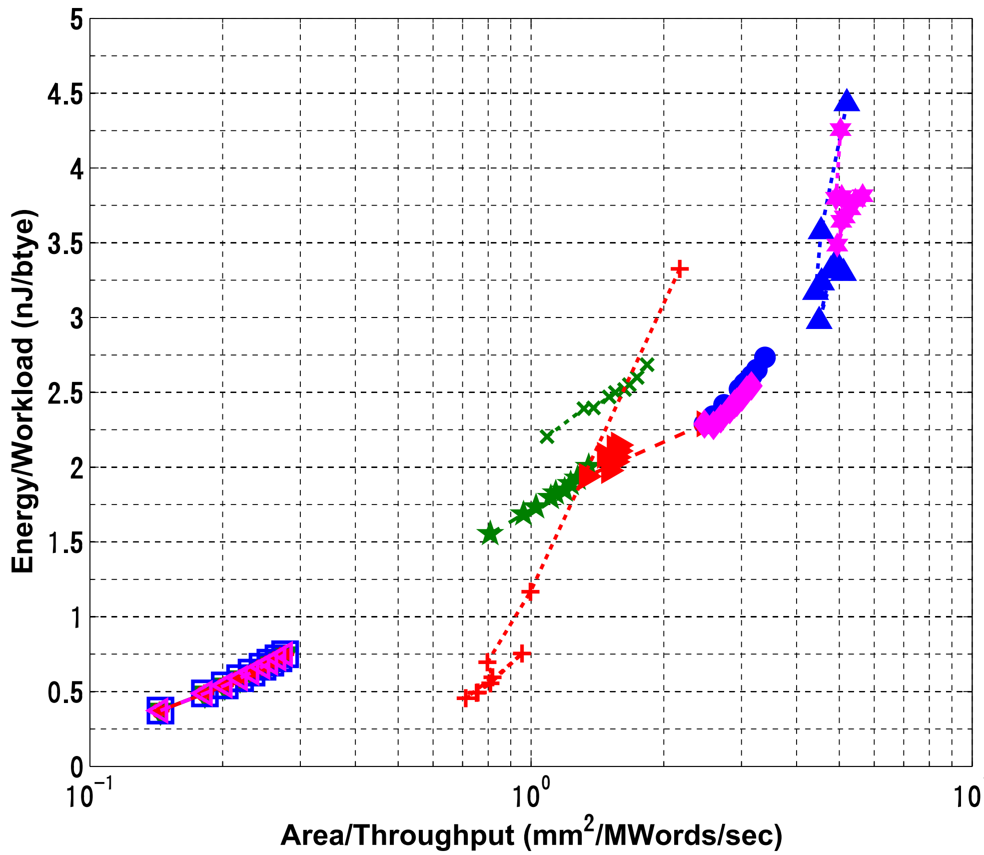

Figure 4.4: Energy per workload versus area per throughput at different keywords for statistics based processing with separated statistics and main filters blocks. See Figure 4.3 for legend.

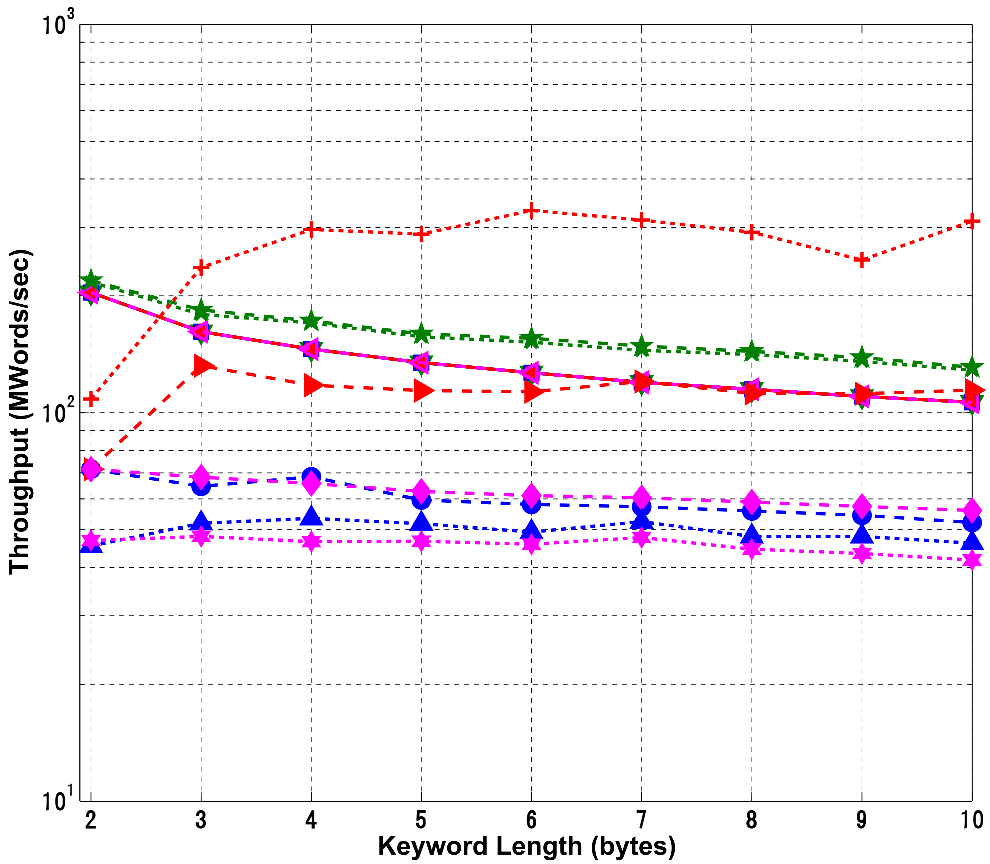

Figure 4.5: Throughput comparison of different keywords for statistics based processing with separated statistics and main filters blocks. See Figure 4.3 for legend.

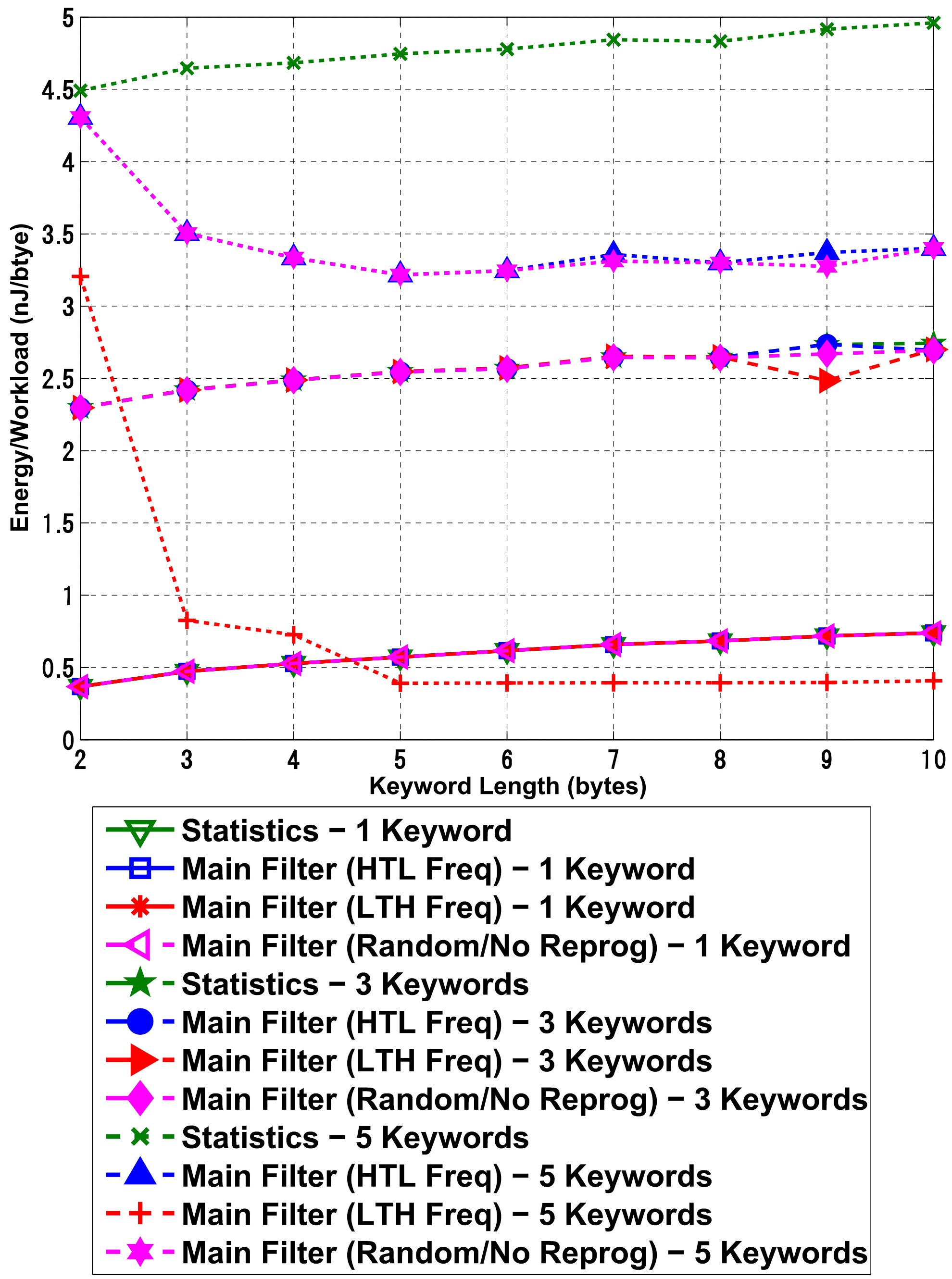

Figure 4.6: Energy per workload versus keyword length at different keywords for statistics based processing with combined statistics and main filters blocks.

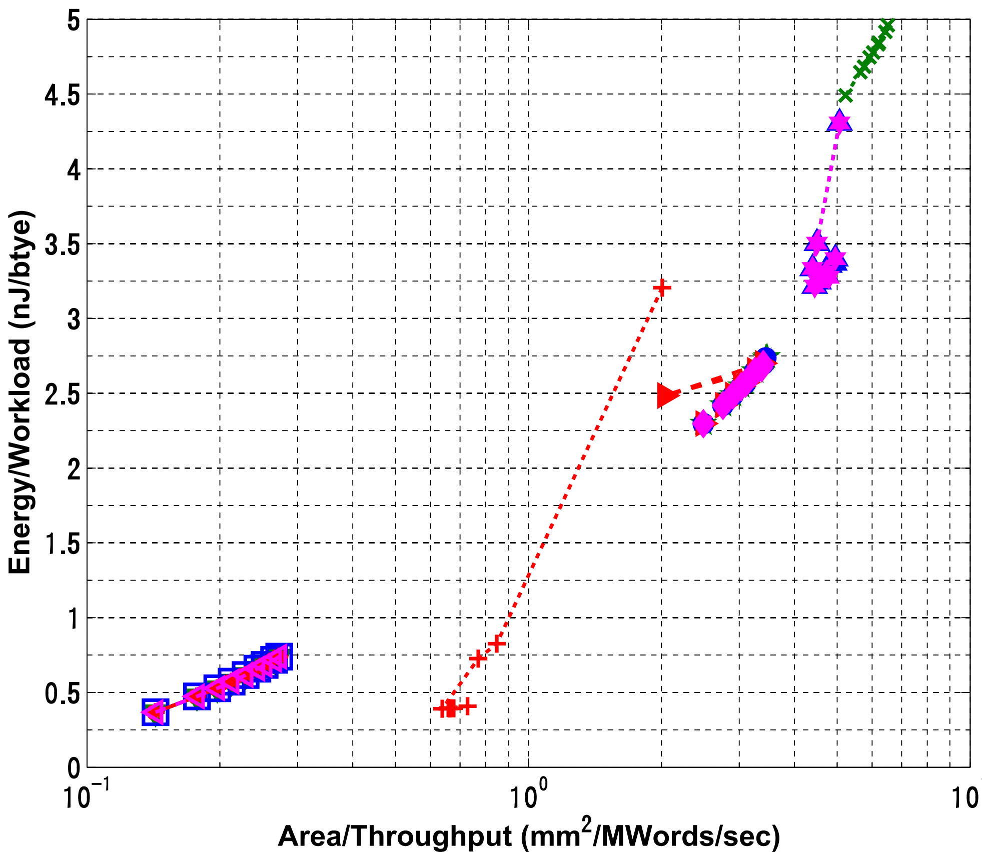

Figure 4.7: Energy per workload versus area per throughput at different keywords for statistics based processing with combined statistics and main filters blocks. See Figure 4.6 for legend.

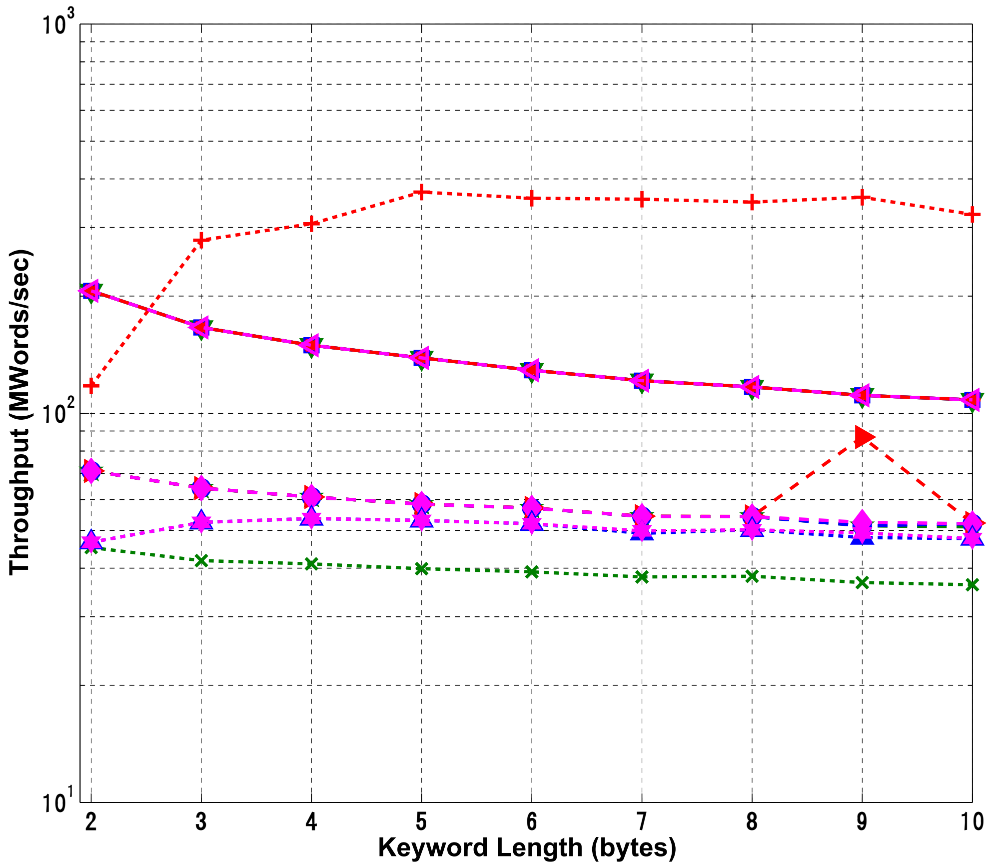

Figure 4.8: Throughput comparison of different keywords for statistics based processing with separated statistics and main filters blocks. See Figure 4.6 for legend.

Table 4.1: Averaged comparison of the self-adaptive string search architecture with separated statistics block(SSB) versus combined statistics block(CSB).

	Architecture: Block (Reprog Mode)	Energy/Workload (nJ/byte)	Throughput (MWords/sec)	Throughput/Area ((MWords/sec)/mm^2)
1 Keyword	SSB: Main Filters (LTHF)	0.6	136	4.6
	CSB: Main Filters (LTHF)	0.6	139	4.7
	SSB: Main Filters (No Reprog)	0.6	136	4.6
	CSB: Main Filters (No Reprog)	0.6	139	4.7
	SSB: Statistics	0.6	136	4.6
	CSB: Statistics	0.6	139	4.7
3 Keywords	SSB: Main Filters (LTHF)	2	**112**	0.63
	CSB: Main Filters (LTHF)	2.5	62.2	0.35
	SSB: Main Filters (No Reprog)	2.5	60.2	0.34
	CSB: Main Filters (No Reprog)	2.6	58.2	0.33
	SSB: Statistics	1.8	**161**	**0.91**
	CSB: Statistics	2.6	58.1	0.33
5 Keywords	SSB: Main Filters (LTHF)	0.9	270	1.14
	CSB: Main Filters (LTHF)	0.8	**313**	1.33
	SSB: Main Filters (No Reprog)	3.4	49.6	0.21
	CSB: Main Filters (No Reprog)	3.4	50.3	0.21
	SSB: Statistics	**2.5**	**158**	**0.67**
	CSB: Statistics	4.8	40.0	0.17

Chapter 5

Regular Expression Processing

5.1 Introduction

While string search represents the matching of one or more occurrences of a keyword within a set of input data, regular expression offers a broader construct for the types of occurrences and keywords. In particular, regular expressions represent a set of strings in terms of adjacency, repetition, and alternation, and are a general-purpose method of describing and matching patterns [26, 40, 32]. The main purpose of regular expression sequences is to find the most concise and flexible way of directly automating matched patterns e.g. text processing.

5.1.1 Quantifiers

The simplest example of a regular expression is string of characters such as "xyz" where xyz is the fixed pattern of interest. From here different symbols known as quantifiers may be used to express more complex terms:

- **"+"** the plus quantifier indicates one or more occurrences of the preceding character
  - e.g. **ab+a** matches "aba", "abba", "abbba" and so on.
- **"*"** the asterisk quantifier indicates zero or more occurrences of the preceding character
  - e.g. **ab*a** matches "aa", "aba", "abba" and so on.
- **"?"** the question mark quantifier indicates zero or one occurrences of the preceding character

  - e.g. **ab?a** matches “aa”, “aba”, but not “abba” since “b” occurs twice.

- **“[a-z]” or “[0-9]”** the range quantifiers match any element within the square parenthesis, inclusive.
  - e.g. **ab[0-8]a** matches “ab0a”, “ab8a”, “ab5a” and so on.

5.1.2 Grouping and Boolean OR

In addition to quantifiers multiple regular expressions may be grouped with **“( )”** and Boolean OR’ed with **“|”** to indicate alternatives [41].

- For example, **“x(y|z)*(a|b|c)”** matches “xa”, “xya”, “xzc”, “xyzzzyzyyyzb” and so on.

The previously described basic regular expressions are used in forming more complex regular expressions depending on system and platform [40]. A system changes a regular expression into a state machine then uses the machine to match incoming strings.

5.1.3 Regular Expression Types

There are two major types of regular expression, NFA and DFA both of which determine how a regular expression engine is constructed.

5.1.3.1 Non-deterministic Finite Automaton (NFA)

An NFA is a state machine that allows simultaneous state transitions as well as state transitions with no input. The NFA algorithm described in this Chapter is based on Thompson’s NFA graph algorithm [3]which is converted to regular expression [4]. The NFA regular expression building blocks are as follows:

- No input state transition
  - ϵ = no input
- Single input state transition
- OR/alternation state transition

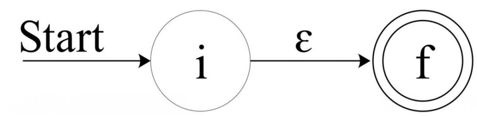

Figure 5.1: An NFA state diagram showing a transition given no input to reach the final state f [1].

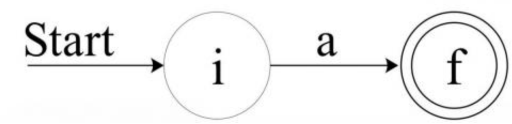

Figure 5.2: An NFA state diagram showing a transition given a single input to reach the final state f [1].

- AND/concatenation state transition

- Star state transition

The NFA construction for a regular expression such as **"(a|b)*abb"** would look like Figure 5.6 where the generated state diagram is based on the rules for the NFA building blocks described above.

5.1.3.2 Deterministic Finite Automaton (DFA)

A DFA is a state machine that takes a finite number of input sequences before arriving at its final state. In contrast to an NFA, a DFA state machine does not allow simultaneous state transitions and every state transition also requires an input [32].

Figure 5.7 shows the DFA construction for the regular expression **"(a|b)*abb"**. This DFA is also referred to as a subset construction of its NFA since it only shows a subset of state transitions that lead to the final state rather than all possible states that lead to the final state.

5.1.3.3 NFA vs DFA

Table 5.1 shows the advantages and disadvantages of NFA and DFA that aid with deciding what regular expression type to employ.

DFAs only allow single state transitions, which implies that they require only a single memory operation per character processed. This makes DFAs more attractive in applications such

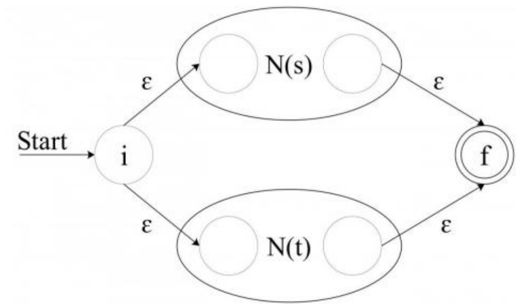

Figure 5.3: An NFA state diagram showing a transition based on alternation. The final state f is reached either through N(s) or N(t) but not through both [1].

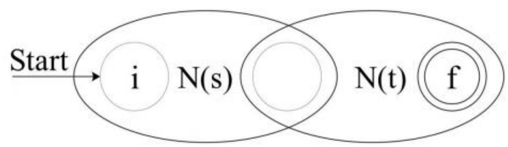

Figure 5.4: An NFA state diagram showing a transition based on concatenation. The final state f is reached when conditions for N(s) is satisfied, followed by satisfying the conditions for N(t) [1].

as high speed networking [32]. On the other hand, complex regular expressions lead to exponentially large number of states in DFAs where this would not be the case in an NFA. This limits the complexity of regular expressions DFA based regular expression processors can handle.

NFAs are able to accept more complex regular expressions than DFAs since NFAs allow multiple state transitions as well as empty inputs [41]. An NFA that has n states may have an equivalent DFA with exponentially larger states. The low state requirement leads to a low memory requirement for NFA based regular expression processing. NFA, as the name suggests is non-deterministic, meaning state transitions are not necessarily finite. This increases the NFA's time complexity since every path that leads to its final state must be checked. This also creates a problem in construction since computing systems have finite space, often requiring converting the NFAs to equivalent DFAs in practice.

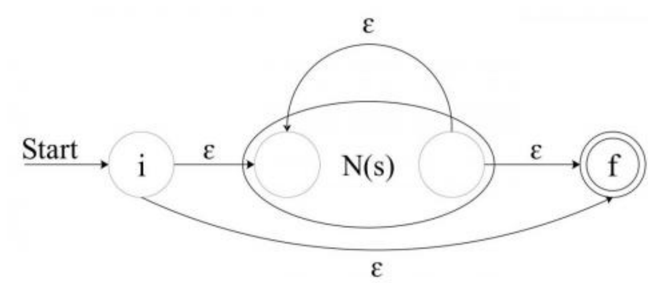

Figure 5.5: An NFA state diagram showing a transition given zero or more of the input to reach the final state f [1].

Table 5.1: NFA vs DFA

	NFA	DFA
Advantage	Low Memory Requirement $O(n)$	Constant processing time complexity $O(1)$
Disadvantage	Linear processing time complexity $O(n)$	Large memory requirement $O(2^n)$
Comment	Often requires conversion to DFA	Requires simplification of complex regular expressions

5.2 Implementation

Every regular expression can be converted into an equivalent finite automaton and vice versa. In addition, each state of the automaton can be executed on parallel hardware to efficiently implement the regular expression. A DFA based regular expression processor was implemented on the AsAP2 [28] to benchmark AsAP2's performance and show the feasibility of developing regular expression on the platform. In order to program the AsAP2 chip an external tool flow was designed to streamline the process as shown in Figure 5.8.

The tool accepts one or several regular expressions and parses them for the AsAP2. The parsed regular expression is separated into smaller cells that could then be programmed on the AsAP2's the 2D mesh of processors. Though the programmable cells have been generated, the route placement of these cells on the chip affects activity and thus power consumption. To counter this issue, a BAMSE developed by Mohammad H Faroozannejad [42] was integrated into the process. BAMSE is a constructive approach that incrementally maps the concurrent tasks (e.g. parsed regular expression cells) of a task graph into the cores of the given hardware platform. The key idea is to arrange the concurrent tasks in a sequence called Task Sequence and read through this sequence to gradually construct the final mapping solution. The algorithm can have as high as 65% improvement over manual placing and routing in terms of longest connection, thereby guaranteeing a higher level of optimization of the cell placement and routing on the AsAP2 chip. Several regular expression elements were created and tested. A cell to be programmed on the AsAP2 contains a parameterized program that interprets each pattern as having a literal or special meaning. An element may be interpreted as an alphanumeric character, space or a "." which stands for any character. Compatible expressions and their implications are as follows:

- "fixed_d" tagged cells signify possible trailing empty space up until the element within that cell is matched.
- "fixed_s" tagged cells matches just the element within that cell.
- "+" tagged cells match when the element within the cell occur one or more times
- "*" tagged cells match when the element within the cell occur zero or more times
- "?" tagged cells match when the element within the cell occur zero or one time only
- "range [a-z]" or "range [0-9]" "?" tagged cells match when the element within the specified range occurs once.

5.3 Results Summary

Most activity (99%) occurred within the first core, allowing scaling down of the supply voltage for subsequent cores which had less than 27% activity. Table 5.2 shows the average throughput performance throughput as ~587 M/sec for each parameterized program.

Table 5.2: Performance for each parameterized program [5]

Parameterized program	**Example regexp**	**Program size (# intruc.)**	**Throughput**	
			miss - hit	**@ 1.2 GHz (chars/sec)**
fixed_s	^ab.d	29	1.29 - 3.26	527 M/sec
fixed_d	.*a..d	28	1.29 - 3.09	548 M/sec
range	[d-v]	28	1.20 - 2.30	686 M/sec
quest	a? (0 or 1)	25	2.25 - 2.25	533 M/sec
plus	b+ (1 or more)	27	1.20 - 2.30	686 M/sec
star	c* (0 or more)	27	2.20 - 2.20	545 M/sec

With an example regular expression such as thal*ia_*... [a-z]n+e_*.._Davis? the implementation achieved a throughput of 309 MB/s @ 1.3 V Dynamic Voltage Frequency Scaling (DVFS) using 59 mW and 181 pJ/Byte as shown in Table 5.3 [5]. Minimizing power consumption to 1.4 mW and using only 76 pJ/Byte allows the design to achieve 17 MB/s throughput @ 0.675 V. Additionally Table 5.3 compares performance results to related works on regular expression processing in hardware. Unfortunately the respective authors often reported max frequency and throughput but not the power consumption or energy per byte.

Table 5.3: Example regular expression performance comparison

Platform	**Supply Voltage (V)**	**Max Frequency (MHz)**	**Throughput (MB/s)**	**Power (mW)**	**Energy (pJ/Byte)**
Virtex2Pro [26]	2.5 (typical)	Not reported	800	Not reported	Not reported
Virtex2Pro [33]	2.5 (typical)	103	664 (Averaged)	Not reported	Not reported
Virtex-IV [33]	3.3 (Commercial)	132	860 (Averaged)	Not reported	Not reported
Spartan3 [33]	3.3 (Commercial)	58	377 (Averaged)	Not reported	Not reported
AsAP2 [5]	1.3	1210	309	133	411
AsAP2 [5]	1.3, DVFS	1210	309	59	181
AsAP2 [5]	1.2, DVFS	1070	273	44	154
AsAP2 [5]	0.675	66	17	1.4	76

Regular expression processing can be expanded to other applications. Figure 5.9 shows a diagram of how regular expression was combined with database sorting to generate statistics and histograms [5]. Performance results are shown in Table 5.4. The results were generated using 200-Byte records achieving a throughput of 1520 MB/s at 1.2 V while using 47 mW at 30 pJ/Byte.

Table 5.4: Performance of database regular expression with sort and statistics [5]

Supply Voltage (V)	**Max Frequency (MHz)**	**Throughput (MB/s)**	**Power (mW)**	**Energy (pJ/Byte)**
1.2	1070	1520	47	30
0.75	260	369	3.4	8.8
0.675	66	94	0.61	6.2

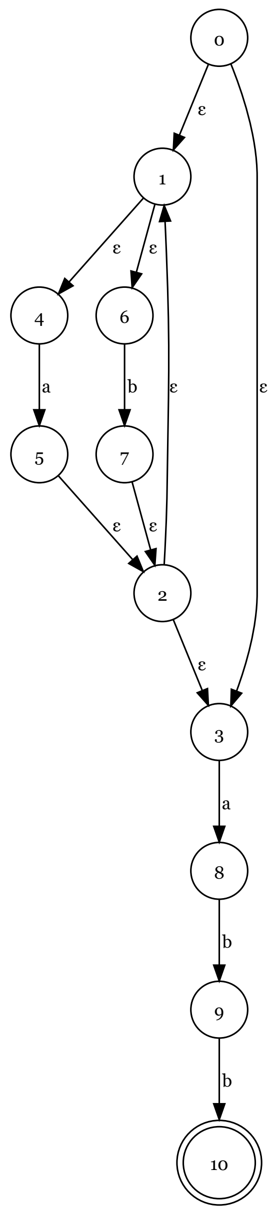

Figure 5.6: A generated NFA state diagram [2] of **"(a|b)*abb"** regular expression based on [3, 4]

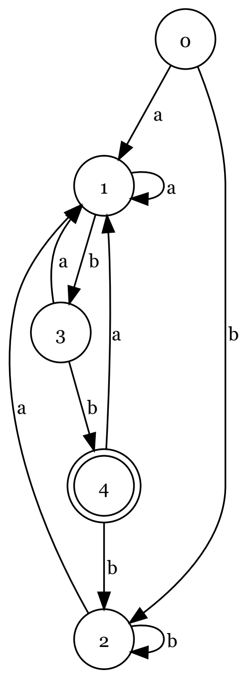

Figure 5.7: A generated DFA state diagram of **“(a|b)*abb”** regular expression [2]. This DFA also represents a subset construction of the NFA in Figure 5.6

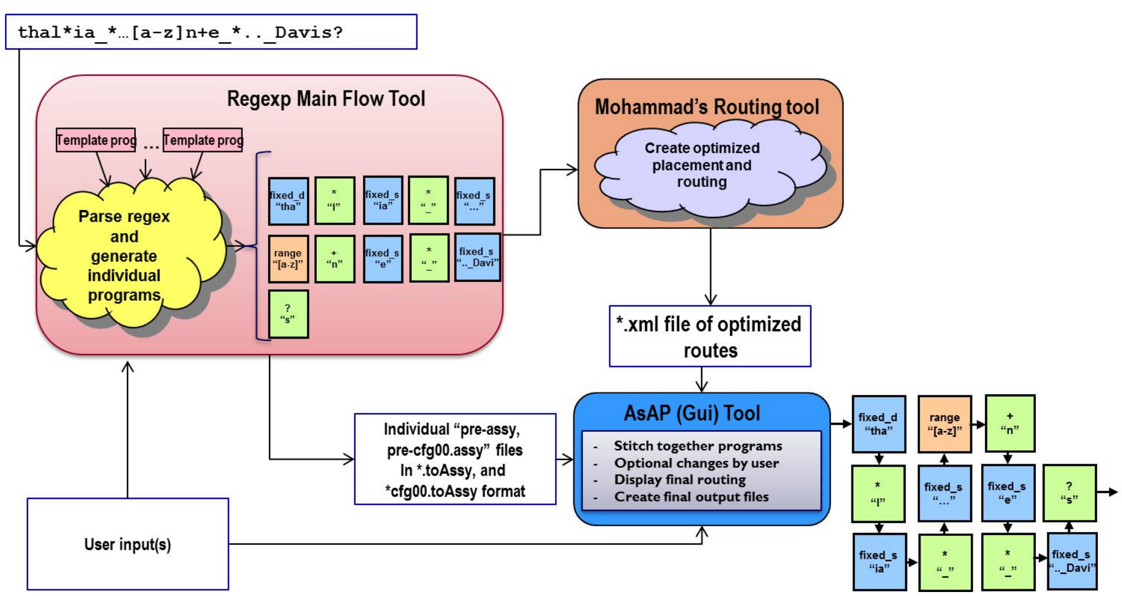

Figure 5.8: Regular expression flow process, starting from the regular expression main flow tool (top left) to the parameterized program cells (bottom right)

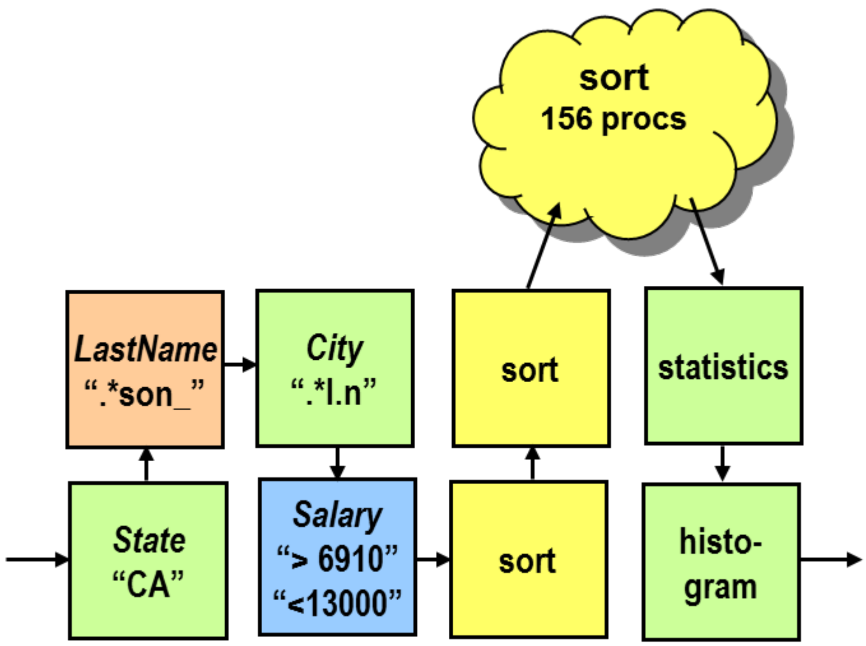

Figure 5.9: An example regular expression combined with database sort and statistics [5]

Chapter 6

Summary and Future Work

Three energy-efficient architectures are presented utilizing a fine-grained many-core processor array for searching and filtering streamed data. The serial architecture is optimal for small keyword searches while the parallel architecture is well suited for larger keyword searches. The all-in-one architecture combines filtering operations and ensures the smallest area footprint. The designs achieve 211× increased throughput per area, and yield 155× energy reduction when compared to string search on a traditional processor (Intel Core i7 3667U).

Two self-adaptive string search filters are also presented for further reducing energy consumption and improving throughput of string search via self-reprogramming. The optimized self-adaptive string search filters consume 5× less energy and achieve 4.8× higher throughput over the three previously designed string-search architectures. The self-adaptive implementation with separated statistics block achieves about 2.8× to 4× higher throughput and throughput per area on average than the implementation with combined statistics block in statistics mode. Other performance parameters such as energy per workload, throughput and throughput per area of the main filters are approximately equal. The area trade-off for having a separated statistics block has negligible impact on the performance of the overall system.

6.1 Completed Projects

In addition to energy-efficient string search methods on a fine-grained many-core platform, the author has also worked extensively on several other major projects listed below.

6.1.1 Many-Core Digital Oscillator Design in 32 nm SOI

The VLSI Computation Laboratory designed four generations of many-core processing arrays. The author was a member of the design team responsible for creating the third generation 1000 core generation chip (Kilocore). The author helped with the physical design and wrote several tests programs for verifying logic and functionally. In the fourth generation chip, the author designed and implemented the digital oscillators for the on-chip cores and routers. The oscillator features course and fine tuning delay stages, and two clock dividers with short-cycle prevention circuitry. VCL has published papers on Kilocore [43], [44], with another submitted for publication [45]. The fourth generation chip has been fabricated and is currently under testing with results soon to be published as well.

6.1.2 Implantable Radio Transmitters for Long Range Health Monitoring

In this project, surveys and simulations are carried out on several implantable radio transmitters for health monitoring. The first half of this project focuses on gathering data on recent research in the area of Medical Implant Communication Systems (MICS), a standard aimed at improving communication distances to ~2 meters. Next we broaden our scope of coverage to include other bands outside the MICS band which achieve link distance over 2 meters by investigating Ultra Wide Band (UWB) systems and other potential long range radios. In addition we discuss various link performance parameters between several papers to gain a better understanding of the system. Finally we run several simulations, using ADS Momentum to model the power gain between an implanted transmitter antenna (loop) in muscle tissue to a receiver antenna (dipole) in free space. Using data collected and the results from the simulations, a performance metric is formed for quantifying the power gain as a function of free space, tissue depth, and frequency. Please see technical report [46] for further details.

6.1.3 A Band-Gap Reference with Internal Digital Signal Processing

The project objective was to design a low-power band-gap voltage reference that uses internal digital processing to compute its analog output. A conventional band-gap reference creates an output voltage by summing scaled incoming voltages in the analog domain. The idea behind this project was that by digitizing the incoming voltages, the sum and scaling is carried out in the digital domain, which may be less expensive in die area and power dissipation than using standard analog techniques in modern CMOS processes. Please see technical report [47] for further details.

6.2 Future Work

The author has published the three energy-efficient architectures [48], with self-adaptive string search filters the continuation of that work. The next step is to make the regular expression filters to include self-adaptive, similar to the implementation in Chapter 4. This will involve changing the regular expressions into state machines, mapping them to the many-core array then configuring them to work with statistics and reprogram blocks. Once the work is extended to regular expressions it may be used in several key applications. For example a more sophisticated web search engine may be developed where the system would support complex searches that are not supported by pre-built index tables. The self-adaptive search algorithm would run simultaneously across many cores and assign scores to each page of result, then merge results based on those scores. The searches would support both basic string search, regular expressions, and page-level expressions.

Acronyms

AIO all-in-one. vii, 10, 12–17

AsAP2 Asynchronous Array of simple Processors, 2nd version. 12, 25, 39, 40, 42

BAMSE Balanced Mapping Space Exploration Algorithm. 39

BASS Balanced Approximate Substring Search. 1, 2

BLAST Basic Local Alignment Search Tool. 2

CSB Combined Statistics Block. ix, 26, 27, 34

DFA Deterministic Finite Automaton. viii, ix, 6, 36–39, 44

FPGA Field Programmable Gate Array. 5–7

GPU Graphics Processing Unit. 5

HTLF High To Low Frequency. 25, 26

LTHF Low To High Frequency. 25, 26, 34

MRS Multi Resolution String Index. 2

NFA Non-deterministic Finite Automaton. viii, ix, 6, 36–39, 43, 44

QUASAR Q-gram Alignment based on Suffix ARrays. 2

SSB Separated Statistics Block. ix, 21, 25, 27, 34

Glossary

AsAP2 A 167-Processor Computational Platform in 65 nm CMOS with 164 independently-clocked homogeneous programmable processors running at 1.2 GHz. Each processor uses 63 simple instruction types within its instruction set. The chip also includes three 16 KB memories with the entire chip connected via a 2D-mesh, allowing for nearest neighbor communication and long distance communication. Each processor contains 128x35-bit instruction memory, 128x16-bit data memory, and two dual clock 64 x 16-bit FIFO buffers for communication between processors. 12

BASS Balanced Approximate Substring Search (BASS) is a fully balanced tree that organizes all position points by recursively grouping together position points that lead similar segments in the string database. 1

BLAST A tool that finds regions of similarity between biological sequences. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance. 2

DFA A state machine that takes a finite number of input sequences before arriving at its final state. 36

filter A string search component whose main operation is to match a keyword to input data. vii, viii, 8–11, 13, 18–33, 46, 48

NFA A state machine that allows simultaneous state transitions as well as state transitions with no input. 36

QUASAR A database searching algorithm that was designed to quickly detect sequences with strong similarity to the query in a context where many searches are conducted on one database. 2

String B-Tree A combination of B-trees and Patricia tries for internal-node indices that is made more effective by adding extra pointers to speed up search and update operations. 8

Suffix Tree A substring of text defined by its starting position and continuing to the right as far as possible to make the string unique. 2

Appendix A

Input Data Generator (Python)

```python
"""
Author: Eman Adeagbo (Created: Sept/11/2014)
Description: Generates keywords and a page of 8192 characters
                                based on defined metrics
"""
import gen_excel_dat, clear_directory
import os,shutil, itertools, re, random
from numpy.random import choice, random_integers
from tables import table

#Any, all, or none of the input parameters to this function may be set before calling this function.
#page_locations='random' when it is not replaced by the caller
#Mainly for compatibility with older code that calls generate_pages() with no input args but use a parameter file
def generate_pages(dynamic_in=None,keyword_popsize_in=None,number_of_keywords_in=None,keyword_probs_in=None,
                                    keyword_lengths_in=None, page_locations_in=['random'],page_size_limit_in=8192):
        """
        Main page generation module
        """

        #Clear working directory

        clear_directory.rm_all(os.path.join(os.path.dirname(os.curdir),'..','..','GeneratePage','GeneratePage','out_pages'))


        if dynamic_in is None: #For now, keep old generate_pages code separate from new one

         pass
         ####  --  Old Code --- ####

         #
         #
         #

        ####################################################################################################

        ############################# ---- NEW CODE BELOW  -----  ****************#########################

        ####################################################################################################
        else:
```

```python
    #generate_pages(dynamic,keyword_popsize,number_of_keywords,keyword_probs,keyword_lengths)
    #generate_pages(dynamic_in=None,keyword_popsize_in=None,number_of_keywords_in=None,keyword_probs_in=None,
    #keyword_lengths_in=None, page_locations_in='random'):

    number_of_keywords = number_of_keywords_in
    keyword_lengths = keyword_lengths_in

    keywords_sizes_list_len = len(keyword_lengths)
    keyword_list = []

    #Generate a random distribution of keyword appearances within document streams
    multiple_pages = 2 #choice(5) #2 = 2*8KB (8KB=8192) Start low for now, then randomize to a larger value later

    if len(keyword_probs_in) != number_of_keywords:
        raise AssertionError("length of keyword_probs_in, "+str(len(keyword_probs_in))+
                             " does not match number of keywords, "+str(number_of_keywords))

    #Generate the list of keywords needed for search within pages
    for k in range(number_of_keywords):
        for i in keyword_lengths:
            keyword_list.append(gen_keyword.keyword(i))
        #keyword_list.append('')

    with open(os.path.join(os.path.dirname(os.curdir),'..','..',
                           'GeneratePage','GeneratePage','out_pages','keyword_list.txt'),'w') as f:
        for k in range(number_of_keywords):
            for i in range(keywords_sizes_list_len):
                f.write(keyword_list[keywords_sizes_list_len*k+i]+'\n')

    #Generate page of 8192 characters -> adjust payload to 8175
    page_size_limit = page_size_limit_in-17 #8192 #128 is padded 16 times and #129 is padded once into the file in StringtoAscii

    word = ''

    no_kw_page = []

        #first create the page without worrying about overflow of words
        #make sure to exclude keywords
    for page_count in range(multiple_pages):
        page = []
        page_size = 0
        while page_size < page_size_limit:
            word = choice(gen_keyword.word_list)
            if word not in keyword_list:
                page.append(word+' ')
                page_size = page_size+len(word)+1

        #trim the page down since the last word might have gone over the page limit
        no_kw_page = trim_page(page, page_size_limit)

        locations = page_locations_in #['random'] #['top','mid','bot']
        #locations_num = [0.1,0.5,0.9]
        histogram = [keyword_list,[]]
        for _ in range(len(keyword_list)):
            histogram[1].append(0)
        #Generate the pages
```

```

        for loc_index in range(len(locations)):
            for num_kw in range(number_of_keywords): #*****#*****#*****

                #enumerate the keywords
                keywords_enum = []
                keyword_probs_limited = []
                for kw in range(num_kw+1):
                    keywords_enum.append(kw+1)
                    keyword_probs_limited.append(keyword_probs_in[kw])

                #generate the keyword distribution for set number_of_keywords
                pop_size = choice(keyword_popsize_in)
                keyword_distr = randsample(keywords_enum, pop_size, keyword_probs_limited)
                #print(keyword_distr)
                for kw_length in keyword_lengths:  #*****#*****#*****

                    #Make a new copy of the generated page that had no keywords
                    new_page = []
                    for chunk in no_kw_page:
                        new_page.append(chunk)


                    pruned_keyword_list = []
                    for keyword in keyword_list:
                        if len(keyword) == kw_length:
                            if len(pruned_keyword_list) < num_kw+1:
                                pruned_keyword_list.append(keyword) #contains all keywords of interest

############################ Only want to insert keywords that are in the keyword_distribution list
############################ At the corresponding keyword length
############################ For example if keyword_distr is 1 1 1 3, then only randomly insert keywords 1, 3 times then
############################ randomly insert keyword 3 once

                    #Go through randomly sampled keyword list
                    for rand_picked_kw in keyword_distr:
                        #Randomize every placement of each inserted keyword
                        #Calculate the location of where to insert the keywords
                        stop_loc = random_integers(page_size_limit-55) #limit stop location to
                        #no more than end of page minus 10 characters to prevent keyword overflow.
                        char_count = 0
                        loc_in_page = 0
                        for char in no_kw_page:
                            char_count += len(char)
                            if (char_count < stop_loc):
                                loc_in_page += 1

                        #insert randomly picked keyword into random location in page
                        new_page.insert(loc_in_page,pruned_keyword_list[rand_picked_kw-1]+' ')

                    new_page = trim_page(new_page, page_size_limit) #Reduce the page size down to the page limit

                    #*****Debug code start*******#
                    temp_pg_size = 0
                    for temp_word in new_page:
                        temp_pg_size += len(temp_word)
                    #print('size of new_page after trim: '+str(temp_pg_size))
```

```
                        if temp_pg_size != page_size_limit:
                            raise AssertionError("Trimmed size, "+str(temp_pg_size)+
                            " does not match page size limit, "+str(page_size_limit))

                        #*****Debug code end*******#


                        #Write out for each set of keyword at a given length
                        out_page_name = os.path.join(os.path.dirname(os.curdir),'..','..',
                                    'GeneratePage','GeneratePage','out_pages','loc_'+
                                    locations[loc_index]+'_num_keywrd_'+
                                    str(num_kw+1)+'_keywrd_len_'+
                                    str(kw_length)+'.txt')

                        with open(out_page_name,'a') as f:
                            #Write finished generated page to output file
                            f.writelines( "%s" % line for line in new_page )

                        #Create histogram for number of keywords, not for each length
                        histogram = hist(new_page,page_size_limit,keyword_list,histogram)

                    with open(os.path.join(os.path.dirname(os.curdir),'..','..',
                            'GeneratePage','GeneratePage','out_pages','loc_'+
                            locations[loc_index]+'_num_keywrd_'+
                            str(num_kw+1)+
                            '_histogram'+'.txt'),'a') as f:
                            #Write histogram to output file
                            for index in range(len(keyword_list)):
                                f.write("{}\t\t{}\n".format(histogram[1][index],histogram[0][index]))

                    histogram = [keyword_list,[]]
                    for _ in range(len(keyword_list)):
                        histogram[1].append(0)

"""
Essential function(s)
"""

#customized randsample that behaves similar to MATLAB function randsample.
#Given elements, a population size, and a list of corresponding probabilities, returns
#a list of the elements of the the population size according to their weighted probabilities.
#numpy.choice seems unreliable
def randsample(elements,population_size,weights):
    if len(elements) != len(weights):
        raise IndexError("Elements size must match probability list size. \nElement size: "+
                    str(len(elements))+" weight size: "+str(len(weights)))
    def weighted_choice(weights):
        rnd = random.random() * sum(weights)
        for i, w in enumerate(weights):
            rnd -= w
            if rnd < 0:
                return i
    sampled_pop = []
    pop_size = population_size
    for _ in range(pop_size):
        sampled_pop.append(elements[weighted_choice(weights)])
    return sampled_pop
```

```python


#trim_page trims a page (characters) down to the set limit, and returns the resulting new page
def trim_page(in_page, page_size_limit):
	""" Function for trimming down a page to the page_size_limit """
	word = ''
	end_of_page_size = 0
	page_sz = 0
	out_page =[]
	#confirm the size of the incoming page
	for chunk in in_page:
		page_sz += len(chunk)

	#print('last few words before trim in trimFunc are: \n'+in_page[-3]+'\n'+in_page[-2]+'\n'+in_page[-1]+'\n')
	while page_sz > page_size_limit :
		word = in_page.pop()
		page_sz = page_sz - len(word)

	end_of_page_size = page_size_limit - page_sz	#calculate space between last word and page_limit
	last_word=''
	for num in range(end_of_page_size):
		last_word += ' '
	page_sz_check = 0
	for chunk in in_page:
		out_page.append(chunk)
		page_sz_check += len(chunk)
	out_page.append(last_word)
	page_sz_check += len(last_word)
	return out_page


def hist(in_page, page_size_limit, keyword_list, kw_match_tally_list):
	"""Function for creating a histogram of keywords within a page """
	#for each keyword in the keyword list count how many times it occurs in the page

	for word in in_page:
		word = word.rstrip(' ')
		if word in keyword_list:
			kw_match_tally_list[1][keyword_list.index(word)] +=1  #increment the corresponding kw position
	return kw_match_tally_list

def keyword(size):
	with open(os.path.join(os.path.dirname(os.curdir),'..','..','GeneratePage','GeneratePage',
						'dictionary_words','dictionary.txt')) as f:
		word_list = list(word.strip().lower() for word in f)
	out_word = ''
	while len(out_word) != size:
		out_word = random.choice(word_list)


	return out_word
```

Appendix B

Main Filter AsAP2 Simulator Code for 3 Keywords (C++/Assembly)

```
#include "stdafx.h"
#include "asapsim.h"

Function( Filter )


//Convert variables to something more convenient
#define keyword_char_pi   ag0pi
#define keyword_char          ag0
#define work_buf_char_pi  ag1pi
#define work_buf_char         ag1
#define input_char                Ibuf0
#define work_buf_counter  DMEM[1]
#define keyword_counter   DMEM[2]
#define keyword_length        DMEM[3]
#define eop                           DMEM[4]
#define temp                          DMEM[0]

Start_Initialization


switch(m->storage[0]){
case 1: //core 0 1
        //begin 1,0
        // DCMEM 0 set's the keywords APTR0
        // DCMEM 1 set's the work buffer end APTR2
        MOVI(DCMEM[2], 32)     // ag0 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[3], 1536)  //ag0 start = DMEM 6 and end addresses = 0 (change end address at runtime) keyword ptr
        MOVI(DCMEM[4], 383)    // ag0 stride=1, sml=1111111
        MOVI(DCMEM[5], 32512)  // ag0 and_mask=1111111 or_mask = 0000000
        MOVI(DCMEM[6], 32)     // ag1 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[7], 25856)  //ag1 start = DMEM 101 and end addresses = 0 (change end address at runtime) work buf ptr
        MOVI(DCMEM[8], 383)    // ag1 stride=1, sml=1111111
        MOVI(DCMEM[9], 32512)  // ag1 and_mask=1111111 or_mask = 0000000
```

```
        // DMEM 0 temp
        // DMEM 1 **Not currently used *****
        // DMEM 2 contains keyword counter
        //MOVI( DMEM[3], 3)     // keyword length
        MOVI( DMEM[4], 128)         // Code for end of page
        MOVI( DMEM[5], 6)   // Value of 6 represents the address to
                                                    // DMEM 6 = the first char in keyword chars  list


        //MOVI( DMEM[6], 112)    // keyword char 1 "p"
        //MOVI( DMEM[7], 108)    // keyword char 2 "l"
        //MOVI( DMEM[8], 97)    // keyword char 3 "a"
        //MOVI( DMEM[9], 99)    // keyword char 4 "c"
        //MOVI( DMEM[10], 97)    // keyword char 5 "a"
        //MOVI( DMEM[11], 116)    // keyword char 6 "t"
        //MOVI( DMEM[12], 105)    // keyword char 7 "i"
        //MOVI( DMEM[13], 111)    // keyword char 8 "o"
        //MOVI( DMEM[14], 110)    // keyword char 9 "n"
        //MOVI( DMEM[15], 115)    // keyword char 10 "s"

        MOVI(DMEM[99], 512) //for RPT block
        MOVI( DMEM[100], 101)         // Value of 101 represents the address to DMEM 101
        //DMEM 101 and below reserved for work buffer, with size = keyword length

        //Additional required constants for keyword ptr reset (ag0) and work buf resets ag1
        MOVI(DMEM[125], 1536)        //Reset for  keyword ptr reset (ag0)
        MOVI(DMEM[126], 25856)        //Reset for  work buf reset (ag1)
        MOVI(DMEM[127], 26112)        //Reset for  work buf reset, then advance by 1 (ag1) skip oldest char
break;
case 2: //core 0 2
        //begin 2,0
        output( east, west)
        //#longdist coreinLdis LDinLen LDoutL2R
        // DCMEM 0 set's the keywords APTR0
        // DCMEM 1 set's the work buffer end APTR2
        MOVI(DCMEM[2], 32)     // ag0 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[3], 1536)  //ag0 start = DMEM 6 and end addresses = 0 (change end address at runtime) keyword ptr
        MOVI(DCMEM[4], 383)    // ag0 stride=1, sml=1111111
        MOVI(DCMEM[5], 32512)  // ag0 and_mask=1111111 or_mask = 0000000
        MOVI(DCMEM[6], 32)     // ag1 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[7], 25856)  //ag0 start = DMEM 101 and end addresses = 0 (change end address at runtime) work buf ptr
        MOVI(DCMEM[8], 383)    // ag1 stride=1, sml=1111111
        MOVI(DCMEM[9], 32512)  // ag1 and_mask=1111111 or_mask = 0000000
        // DMEM 0 temp
        // DMEM 1 **Not currently used *****
        // DMEM 2 contains keyword counter
        //MOVI( DMEM[3], 3)     // keyword length
        MOVI( DMEM[4], 128)         // Code for end of page
        MOVI( DMEM[5], 6)   // Value of 6 represents the address to
                                            // DMEM 6 = the first char in keyword chars  list
        //MOVI( DMEM[6], 101)    // keyword char 1 "e"
        //MOVI( DMEM[7], 110)    // keyword char 2 "n"
        //MOVI( DMEM[8], 100)    // keyword char 3 "d"
        //MOVI( DMEM[9], 101)    // keyword char 4 "e"
        //MOVI( DMEM[10], 114)    // keyword char 5 "r"
        //MOVI( DMEM[11], 108)    // keyword char 6 "l"
```

```
        //MOVI( DMEM[12], 105)    // keyword char 7 "i"
        //MOVI( DMEM[13], 110)    // keyword char 8 "n"
        //MOVI( DMEM[14], 39)    // keyword char 9 "'"
        //MOVI( DMEM[15], 115)    // keyword char 10 "s"

        MOVI(DMEM[99], 512) //for RPT block
        MOVI( DMEM[100], 101)         // Value of 101 represents the address to DMEM 101
        //DMEM 101 and below reserved for work buffer, with size = keyword length

        //Additional required constants for keyword ptr reset (ag0) and work buf resets ag1
        MOVI(DMEM[125], 1536)        //Reset for  keyword ptr reset (ag0)
        MOVI(DMEM[126], 25856)        //Reset for  work buf reset (ag1)
        MOVI(DMEM[127], 26112)        //Reset for  work buf reset, then advance by 1 (ag1) skip oldest char
break;
case 3: //core 0 3
        //begin 2,0
        output( east )
        //#longdist coreinLdis LDinLen LDoutL2R
        // DCMEM 0 set's the keywords APTR0
        // DCMEM 1 set's the work buffer end APTR2
        MOVI(DCMEM[2], 32)     // ag0 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[3], 1536)  //ag0 start = DMEM 6 and end addresses = 0 (change end address at runtime) keyword ptr
        MOVI(DCMEM[4], 383)    // ag0 stride=1, sml=1111111
        MOVI(DCMEM[5], 32512)  // ag0 and_mask=1111111 or_mask = 0000000
        MOVI(DCMEM[6], 32)     // ag1 br=0, dir=1, shr_amt=0
        MOVI(DCMEM[7], 25856)  //ag0 start = DMEM 101 and end addresses = 0 (change end address at runtime) work buf ptr
        MOVI(DCMEM[8], 383)    // ag1 stride=1, sml=1111111
        MOVI(DCMEM[9], 32512)  // ag1 and_mask=1111111 or_mask = 0000000

        // DMEM 0 temp
        // DMEM 1 **Not currently used *****
        // DMEM 2 contains keyword counter
        //MOVI( DMEM[3], 3)     // keyword length
        MOVI( DMEM[4], 128)         // Code for end of page
        MOVI( DMEM[5], 6)   // Value of 6 represents the address to
                                                    // DMEM 6 = the first char in keyword chars  list

        //MOVI( DMEM[6], 101)    // keyword char 1 "e"
        //MOVI( DMEM[7], 110)    // keyword char 2 "n"
        //MOVI( DMEM[8], 100)    // keyword char 3 "d"
        //MOVI( DMEM[9], 101)    // keyword char 4 "e"
        //MOVI( DMEM[10], 114)    // keyword char 5 "r"
        //MOVI( DMEM[11], 108)    // keyword char 6 "l"
        //MOVI( DMEM[12], 105)    // keyword char 7 "i"
        //MOVI( DMEM[13], 110)    // keyword char 8 "n"
        //MOVI( DMEM[14], 39)    // keyword char 9 "'"
        //MOVI( DMEM[15], 115)    // keyword char 10 "s"

        //MOVI(DMEM[99], 512) //for RPT block
        MOVI( DMEM[100], 101)         // Value of 101 represents the address to DMEM 101
        //DMEM 101 and below reserved for work buffer, with size = keyword length

        //Additional required constants for keyword ptr reset (ag0) and work buf resets ag1
        MOVI(DMEM[125], 1536)        //Reset for  keyword ptr reset (ag0)
        MOVI(DMEM[126], 25856)        //Reset for  work buf reset (ag1)
        MOVI(DMEM[127], 26112)        //Reset for  work buf reset, then advance by 1 (ag1) skip oldest char
        break;
```

```
}
NOP(nop3)

End_Initialization

//begin 1,0 (measure activity, measure energy)
prestart:
if(m->storage[0] == 1){

        output(west)        // Set output direction to MC1 control input
        NOP(nop3)
        MOVE(Obuf, _0)                                  // Send request to MC1 to write first page to mem16k
        NOP(nop3)
        MOVE(Obuf, _1)                                  // Send request to MC1 to read first page from mem16k
        output( east, west)                         // Reset output direction to normal operation directions
        NOP(nop3)
}


start:

if(m->storage[0] == 2 || m->storage[0] == 3){
        XOR( NULL, Ibuf0, _1, NOP2)        // start this core if
                                                                        // incoming core was  match
        BRZ(start_1st_char_loop)                // truly start by jumping to 1st char loop

//before_true_start:



        MOVE( Obuf, _0)
        BR(start)
}

//true_start:


//Before setting up work buffer scan for first match with loop unrolling
start_1st_char_loop:
XOR(NULL, Ibuf0, DMEM[6], nop2)                //compare input char to first keyword char
BRZ(check_2nd_char_block)                        //branch to second char block if match found
XOR(NULL, Ibuf0, DMEM[6], nop2)                //compare input char to first keyword char
BRZ(check_2nd_char_block)                        //branch to second char block if match found
XOR(NULL, Ibuf0, DMEM[6], nop2)                //compare input char to first keyword char
BRZ(check_2nd_char_block)                        //branch to second char block if match found
XOR(NULL, Ibuf0, DMEM[6], nop2)                //compare input char to first keyword char
BRZ(check_2nd_char_block)                        //branch to second char block if match found
XOR(NULL, Ibuf0, DMEM[6], nop2)                //compare input char to first keyword char
BRZ(check_2nd_char_block)                        //branch to second char block if match found


XOR(NULL, Ibuf0nap, eop, nop2)        //Check to see if at the end of page
BRNZ(start_1st_char_loop)                        //if nothing found, restart loop

//handle_end_of_page:

MOVE( Obuf, _0)                                        //Send code to next block indicating no match
```

```
ADD(eop,eop,_1,nop3)                          // Create a modified end of page
goto_end_of_page:                                 // Advance Ibuf0 till end of page
XOR( NULL, eop, Ibuf0,        NOP2)        // check if at the end of the page
BRNZ( goto_end_of_page)                        // If not end of page, repeat this block
SUB(eop,eop,_1,nop3)                        //Reset end of page to original

if(m->storage[0] == 1){
        output(west)                                // Set output direction to MC1 control input
        NOP(nop3)
        MOVE(Obuf, _1)                                // Send request to MC1 to read first page from mem16k
        output( east, west)                        // Reset output direction to normal operation directions
        NOP(nop3)
}
//else if (m->storage[0] == 2 || m->storage[0] == 3){
//        output(west)                                // Set output direction to MC1 control input
//        NOP(nop3)
//        MOVE( Obuf, _0)                        // Send "processing done" message to previous filter
//        NOP(nop3)
//        output( east)                        // Reset output direction to normal operation directions
//        NOP(nop3)
//}

BR(reset_all)                                //Reset everything once at the end of page

check_2nd_char_block:
XOR(NULL, Ibuf0, DMEM[7], nop2)        //compare next input char to 2nd keyword char
BRNZ(start_1st_char_loop)                        //branch to 1st char loop if no match

//At this point we have both chars matching.
SUB(NULL, DMEM[3], _2,nop2)                        //check if keyword length reached.
BRZ(send_match_result)                        //if reached send out approriate code

//More words to search through so setup work buffer and use
MOVE(work_buf_char_pi, keyword_char_pi, nop3)    //save 1st char match to work buffer
MOVE(work_buf_char_pi, keyword_char_pi, nop3)    //save 2nd char match to work buffer
MOVI(keyword_counter, _2)                                        //update the keyword counter to reflect the 2 matched chars
MOVI(work_buf_counter, _2)                                //update the work buffer counter to reflect the 2 matched chars

variable_char_check_block:

MOVE(work_buf_char, input_char, nop3)    //grow work buffer by 1 char from input buffer
ADD(work_buf_counter, work_buf_counter, _1, nop3)        //grow work buffer by 1
XOR(NULL, keyword_char_pi, work_buf_char_pi, nop2) //compare the 3rd/var keyword char to 3rd/var char in work buffer
BRNZ(var_char_nomatch)                                                        //handle matched char

//Character matched
ADD(keyword_counter, keyword_counter, _1, nop3)         //increment keyword counter
XOR(NULL, keyword_length, keyword_counter, nop2)        //check if all keyword chars checked
BRNZ(variable_char_check_block)                                        //If chars left, branch back up

//All chars matched
BR(send_match_result)

//Character did not match
var_char_nomatch:
//SUB(work_buf_counter, work_buf_counter, _1)        //Need to decrease work buf size by one
```

```
MOVE(DCMEM[3], DMEM[125])                                          //Reset keyword pointer (ag0)
MOVE(DCMEM[7], DMEM[127])                                          //Reset work buf ptr (ag1)to second to oldest work buf location
MOVI(keyword_counter, _0)                                          //Reset keyword counter
MOVI(temp, _1, nop3)                                                  //Reset temp which is used here as sub work buf counter (ignore oldest char)

//Submatch check
submatch_check:

ADDC(temp, temp, _1)                                                        //increment sub work buf counter
XOR(NULL, work_buf_char_pi, keyword_char_pi, nop2) //compare char from work buf to keyword char
BRZ(submatch_check_success)

//submatch check fail
XOR(NULL, temp, work_buf_counter, nop2)        //check if search is at the end of the work buffer
BRZ(partial_reset)                                                 //if at the end of work buffer continue main character checking block


//Some chars left in work buf to check
MOVE(DCMEM[3], DMEM[125])                                          //Reset keyword pointer (ag0)
NOP()
MOVI(keyword_counter, _0, nop2)                         //Reset keyword counter
BR(submatch_check)                                                 //Need to check the remaining chars in work buf


partial_reset:
MOVE(DCMEM[3], DMEM[125], nop2)                  //Reset keyword pointer (ag0)
MOVE(DCMEM[7], DMEM[126], nop3)                 //Reset work buf ptr (ag1)
BR(start_1st_char_loop)                                 //After resetting everything go back to first char loop


reset_all:
MOVE(DCMEM[3], DMEM[125], nop2)                 //Reset keyword pointer (ag0)
MOVE(DCMEM[7], DMEM[126], nop3)                 //Reset work buf ptr (ag1)

if(m->storage[0] == 1){//Filter 1
        BR(prestart)                                                 //After resetting everything go back to prestart
}else{//all other filters
        BR(start)                                                         //full reset for all other filters
}

submatch_check_success:
ADD(keyword_counter, keyword_counter, _1)        //increment keyword counter
NOP()
XOR(NULL, temp, work_buf_counter, nop2)        //check if search is at the end of the work buffer
BRZ(variable_char_check_block)                        //if at the end of work buffer continue main character checking block

BR(submatch_check)                                                 //Need to check the remaining chars in work buf



send_match_result:

ADD(eop,eop,_1,nop3)                                 // Create a modified end of page
zip_to_end_of_page:                                         // Advance Ibuf0 till end of page
XOR( NULL, eop, Ibuf0,        NOP2)                // check if at the end of the page
BRNZ( zip_to_end_of_page)                         // If not end of page, repeat this block
//SUB(eop,eop,_1,nop3)                                 //Reset end of page to original
```

```
if(m->storage[0] == 1){
MOVE( Obuf, _1)                                    //Send code to next block indicating successful match
                                                            //--For filter 2 and 3 --
                                                            //These cores by definition only runs when previous core output==1
                                                            //therefore if current core matches send out a 1
}else if(m->storage[0] == 2){
MOVE( Obuf, _1)                                    //Send code to next block indicating successful match
                                                            //--For filter 2 and 3 --
                                                            //These cores by definition only runs when previous core output==1
                                                            //therefore if current core matches send out a 1
}else{//Filter 3
MOVE( Obuf, _1)                                    //Send code to next block indicating successful match
                                                            //--For filter 2 and 3 --
                                                            //These cores by definition only runs when previous core output==1
                                                            //therefore if current core matches send out a 1
}
NOP(nop3)


if(m->storage[0] == 1){//Filter 1
        //Configure current core as a pass-through until last filter finishes
        //MOVI(DMEM[98],_0)
        //ADD(eop,eop,_1)
        //fresh_read_out:
        output( east)                        // Reset output direction to normal operation directions
        //NOP(nop3)

        //set pass-through
        pass_through:
        RPT(DMEM[99],NOP3)
        MOVE( Obuf, Ibuf0)                        // broadcast input to outputs. Loop unroll rather than branch
        }

        XOR( NULL, eop, Ibuf0nap, NOP2) //Next core completes when the last sent char is eop
                                        //Technically eop+1 since the SUB earlier was commented out
        BRNZ(pass_through)                                //continue to set current core as pass_through if not (eop+1)


        SUB(eop,eop,_1)                                        //Reset end of page to original
        XOR ( NULL, Ibuf1, _0, NOP2)        // check if next core wants a fresh read of document or not
        BRZ( reset_all)                                        // Restart program since subsequent cores are done

        ////////---------loop unroll(last few lines modified)---------

        //set fresh read out
        output(west)                                        // Set output direction to MC1 control input
        ADD(eop,eop,_1)                                        // Create a modified end of page
        //XOR( NULL, DMEM[98], _1, NOP2)
        //BRZ( prestart)
        //ADD(DMEM[98],DMEM[98],_1)
        NOP(nop2)
        MOVE(Obuf, _1)                                        // Send request to MC1 to read page from mem16k
        //NOP(nop3)

        output( east)                        // Reset output direction to normal operation directions
        //NOP(nop3)
```

```
        //set pass-through
        pass_through_1rpt:
        RPT(DMEM[99],NOP3)
        MOVE( Obuf, Ibuf0)                                // broadcast input to outputs. Loop unroll rather than branch
        }

        XOR( NULL, eop, Ibuf0nap, NOP2)        //Next core completes when the last sent char is eop
                                                                    //Technically eop+1 since the SUB earlier was commented out
        BRNZ(pass_through_1rpt)                                //continue to set current core as pass_through if not (eop+1)


        SUB(eop,eop,_1)                                        //Reset end of page to original
        //XOR ( NULL, Ibuf1, _0, NOP2)        // check if next core wants a fresh read of document or not
        BR( reset_all)                                        // Restart program since subsequent cores are done

        //////////-----------------------

        //BR(fresh_read_out)                                //jump back to fresh_read_out label
}else if(m->storage[0] == 2){//Filter 2
        //Configure current core as a pass-through until last filter finishes
        //ADD(eop,eop,_1)
        output( east)                                        // Reset output direction to normal operation directions
        //NOP(nop3)

        //set pass-through
        pass_through_2:
        RPT(DMEM[99],NOP3)
        MOVE( Obuf, Ibuf0)                                // broadcast input to outputs. Loop unroll rather than branch
        }

        XOR( NULL, eop, Ibuf0nap, NOP2)        //Next core completes when the last sent char is eop
                                                                    //Technically eop+1 since the SUB earlier was commented out
        BRNZ(pass_through_2)                        //continue to set current core as pass_through if not (eop+1)

        output(east, west)                        // Set output direction to MC1 control input
        SUB(eop,eop,_1)                                //Reset end of page to original
        //NOP(nop3)
        BR( reset_all)                                // Restart program since subsequent cores are done
}else{ //Filter 3
        SUB(eop,eop,_1)                                //Reset end of page to original
        BR( reset_all)                                // Restart program
}
//NOP()
}//end
```

Appendix C

String Search Top Level Script (Python)

```
"""
Author: Eman Adeagbo (Created: Jul/07/2015)
Description: Manages the statistics based filters.
                          Generates the config and input files for the simulator,
                          calls the simulator, parses the statistics,
                          programs the proper cores, calls simulator again,
                          parses the resulting output and
                          creates the appropriate excel files for data plots etc.
"""

#python system modules
import subprocess, sys, os, shutil, time, operator, itertools

#custom modules
import sim_keywords_cfg, clear_directory, gen_excel_dat, gen_page , StringtoAscii

architectures = ['statistic','mainfilter','mainfilter_compare','mainfilter_compare_reverse'] #must make sure that statistic runs first
#In post processing statistics -> statistics , mainfilter -> mainfilter_sorted_descend ,
#mainfilter_compare -> mainfilter_unsorted, mainfilter_compare_reverse -> mainfilter_sorted_ascend

keyword_lengths = [2,3,4,5,6,7,8,9,10]
number_of_keywords = 5 #5
dynamic = 1 #1 = probability based string search, 0 = constant string search, no probability
keyword_probs = [0.049, 0.0009, 0.025, 0.0000011, 0.0] # These don't necessarily have to sum up to 1.0 but each value must be less than 1.0
keyword_popsize = 10000  #Keyword population size sets the limit on the number of keywords per set.
                                             #For example: for 3 unique keywords, and keyword_popsize of 100, any of the 3 unique keywords
                                             #can occur multiple times(based on weighted probability), but their sum must be 100 or less.

#Note, the value below must also be changed in gen_page.py to take effect
#maybe not...Checking....
page_locations = ['random'] #,'top','bot'] #'mid'
page_size_limit = 8192 #Page limit set as a consequence of Asap2 memory size


```

```
num_iterations = 1000


#absolute paths
gen_page_path = os.path.join(os.path.dirname(os.curdir),'..','..','GeneratePage','GeneratePage')

main_sim_path = os.path.join(os.path.dirname(os.curdir),'..','..','SBFProjManager','SBFProjManager','simulation')
string_to_ascii_path = os.path.join(os.path.dirname(os.curdir),'..','..','StringtoAscii','StringtoAscii')



#Create the input files needed for the simulator


#Before generating the input files and running the simulator, need to remove any old files
#excluding the .exe sim program from the current simulation directory

for architecture in architectures:
        for cur_num_kw in range(number_of_keywords):
                cur_sim_path = os.path.join(main_sim_path,architecture,str(cur_num_kw+1)+'keywords')
                clear_directory.rm_files_ext(cur_sim_path,'txt')


for iteration in range(num_iterations):
        #Reset the keyword list
        keyword_list = []
        #Clear the generation path output directory
        clear_directory.rm_files(os.path.join(gen_page_path,'out_pages'))
        #Next, execute page generation script
        #gen_page_script = os.path.join(gen_page_path,'gen_page.py')
        #arguments =  ""


        #os.system(gen_page_script)
        gen_page.generate_pages(dynamic,keyword_popsize,number_of_keywords,keyword_probs,keyword_lengths,page_locations,page_size_limit)


        #Import the created keywords from the page generation out_pages directory
        with open(os.path.join(gen_page_path,'out_pages','keyword_list.txt')) as f:
                for keyword in f:
                        if keyword is not '\n':                                         #skip new lines
                                keyword_list.append(keyword)

        for architecture in architectures:
                if architecture is  'mainfilter':
                        #At this point statistic gathering is done

                        for pg_loc in page_locations:
                                for cfg_f_num_of_keywrds in range(number_of_keywords):
                                        for kw_length in keyword_lengths:
                                                pcr_list = [] #per core raw list
                                                pcr_group_list = [] #make groupings of 3s for the per core raw list for each keyword
                                                                #[(keyword1,output1,energy1),...,(...)]
                                                #relative path
                                                cur_sim_path = os.path.join(main_sim_path,
                                                                  'statistic',
                                                                  str(cfg_f_num_of_keywrds+1)+
```

```
                                  'keywords')
                        with open(os.path.join(cur_sim_path,'pcr_iter_'+str(iteration+1)+'_'+'loc_'+
                             pg_loc+'_num_keywrd_'+
                             str(cfg_f_num_of_keywrds+1)+
                             '_keywrd_len_'+str(kw_length)+
                             '.txt')) as f_pc_raw:
                            pcr_list=list(f_pc_raw.readlines())
                        #print(int(len(pcr_list)/3))
                        for groupings in range(int(len(pcr_list)/3)):
                            #pcr_kw_list.append(str(pcr_list[(item+1)%1]))
                            #pcr_out_list.append(int(pcr_list[(item+1)%2]))
                            #pcr_energy_list.append(float(pcr_list[(item+1)%3]))
                            pcr_group_list.append([str(pcr_list[groupings*3+0]),
                              int(pcr_list[groupings*3+1]),
                              float(pcr_list[groupings*3+2])])


                        pcr_group_list.sort(key=operator.itemgetter(2),reverse=True) #0=keyword, 1=output, 2=energy
                        #At this point pcr_group_list is sorted in ascending order (highest to lowest)

                        #Reconfigure the work cores with the code below#########

                        #Extract only the sorted keywords from the sorted group list
                        sorted_keyword_list =[]
                        for core_stat in pcr_group_list:
                            #sorted_keyword_list.append(core_stat[0].rstrip('\n'))
                            sorted_keyword_list.append(core_stat[0])
                        config_list = []
                        config_list = sim_keywords_cfg.config(architecture,sorted_keyword_list,
                                                                cfg_f_num_of_keywrds+1,kw_length)
                        #write config file to 'mainfilter' sim directory
                        #relative path
                        cur_sim_path = os.path.join(main_sim_path,architecture,str(cfg_f_num_of_keywrds+1)+'keywords')
                        cfg_name = 'cfg_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                        with open (os.path.join(cur_sim_path,cfg_name),'w') as f: #TODO: make it iterate up to 5 later
                            #Special case for singular filter. core location should be configured to [0][0]
                            if cfg_f_num_of_keywrds+1 is 1:
                                config_list[1] = '0\n'
                                config_list[2] = '0\n'
                            f.writelines( "%s" % line for line in config_list )

        elif architecture is  'mainfilter_compare':
            #Need to measure how much better the reprogrammed mainfilter is
            #Run a version of the mainfilter as if it were not repgrogrammed
            for pg_loc in page_locations:
                for cfg_f_num_of_keywrds in range(number_of_keywords):
                    for kw_length in keyword_lengths:
                        pcr_list = [] #per core raw list
                        pcr_group_list = [] #make groupings of 3s for the per core raw list for each keyword
                                        #[(keyword1,output1,energy1),...,(...)]
                        #relative path
                        cur_sim_path = os.path.join(main_sim_path,'statistic',str(cfg_f_num_of_keywrds+1)+'keywords')
                        with open(os.path.join(cur_sim_path,'pcr_iter_'+
                             str(iteration+1)+'_'+'loc_'+pg_loc+'_num_keywrd_'+
                             str(cfg_f_num_of_keywrds+1)+
                             '_keywrd_len_'+str(kw_length)+
                             '.txt')) as f_pc_raw:
```

```
                        pcr_list=list(f_pc_raw.readlines())

                    for groupings in range(int(len(pcr_list)/3)):
                        pcr_group_list.append([str(pcr_list[groupings*3+0]),
                          int(pcr_list[groupings*3+1]),
                          float(pcr_list[groupings*3+2])])

                    unsorted_keyword_list =[]
                    for core_stat in pcr_group_list:
                        #sorted_keyword_list.append(core_stat[0].rstrip('\n'))
                        unsorted_keyword_list.append(core_stat[0])
                    config_list = []
                    config_list = sim_keywords_cfg.config(architecture,unsorted_keyword_list,
                                                cfg_f_num_of_keywrds+1,kw_length)
                    #write config file to 'mainfilter' sim directory
                    #relative path
                    cur_sim_path = os.path.join(main_sim_path,architecture,str(cfg_f_num_of_keywrds+1)+'keywords')
                    cfg_name = 'cfg_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                    with open (os.path.join(cur_sim_path,cfg_name),'w') as f: #TODO: make it iterate up to 5 later
                        #Special case for singular filter. core location should be configured to [0][0]
                        if cfg_f_num_of_keywrds+1 is 1:
                            config_list[1] = '0\n'
                            config_list[2] = '0\n'
                        f.writelines( "%s" % line for line in config_list )

    elif architecture is  'mainfilter_compare_reverse':
        #At this point statistic gathering is done

        for pg_loc in page_locations:
            for cfg_f_num_of_keywrds in range(number_of_keywords):
                for kw_length in keyword_lengths:
                    pcr_list = [] #per core raw list
                    pcr_group_list = [] #make groupings of 3s for the per core raw list for each keyword
                            #[(keyword1,output1,energy1),...,(...)]
                    #relative path
                    cur_sim_path = os.path.join(main_sim_path,'statistic',str(cfg_f_num_of_keywrds+1)+'keywords')
                    with open(os.path.join(cur_sim_path,'pcr_iter_'+str(iteration+1)+'_'+'loc_'+pg_loc+
                         '_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+
                         str(kw_length)+'.txt')) as f_pc_raw:
                        pcr_list=list(f_pc_raw.readlines())
                    for groupings in range(int(len(pcr_list)/3)):
                        pcr_group_list.append([str(pcr_list[groupings*3+0]),
                          int(pcr_list[groupings*3+1]),float(pcr_list[groupings*3+2])])


                    pcr_group_list.sort(key=operator.itemgetter(2),reverse=False) #0=keyword, 1=output, 2=energy
                    #At this point pcr_group_list is sorted in ascending order (lowest to highest)

                    #Reconfigure the work cores with the code below##########

                    #Extract only the sorted keywords from the sorted group list
                    sorted_keyword_list =[]
                    for core_stat in pcr_group_list:
                        #sorted_keyword_list.append(core_stat[0].rstrip('\n'))
                        sorted_keyword_list.append(core_stat[0])
                    config_list = []
                    config_list = sim_keywords_cfg.config(architecture,sorted_keyword_list,
```

```python
                                        cfg_f_num_of_keywrds+1,kw_length)
                    #write config file to 'mainfilter' sim directory
                    #relative path
                    cur_sim_path = os.path.join(main_sim_path,architecture,str(cfg_f_num_of_keywrds+1)+'keywords')
                    cfg_name = 'cfg_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                    with open (os.path.join(cur_sim_path,cfg_name),'w') as f: #TODO: make it iterate up to 5 later
                        #Special case for singular filter. core location should be configured to [0][0]
                        if cfg_f_num_of_keywrds+1 is 1:
                            config_list[1] = '0\n'
                            config_list[2] = '0\n'
                        f.writelines( "%s" % line for line in config_list )

    else: #architecture is statistic
        for cfg_f_num_of_keywrds in range(number_of_keywords):

            #relative path
            cur_sim_path = os.path.join(main_sim_path,architecture,str(cfg_f_num_of_keywrds+1)+'keywords')

            #generate the config file
            for kw_length in keyword_lengths:
                config_list = []
                config_list = sim_keywords_cfg.config(architecture,keyword_list, cfg_f_num_of_keywrds+1,kw_length)
                #write config file to appropriate sim directory
                cfg_name = 'cfg_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                with open (os.path.join(cur_sim_path,cfg_name),'w') as f:
                    #Special case for singular filter. core location should be configured to [0][0]
                    if cfg_f_num_of_keywrds+1 is 1:
                        config_list[1] = '0\n'
                        config_list[2] = '0\n'
                    f.writelines( "%s" % line for line in config_list )


    #Regardless of which architecture run all the code below

    for cfg_f_num_of_keywrds in range(number_of_keywords):

        #relative path
        cur_sim_path = os.path.join(main_sim_path,architecture,str(cfg_f_num_of_keywrds+1)+'keywords')


        for kw_length in keyword_lengths:
            #config file name
            cfg_name = 'cfg_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
        #Next, convert corresponding page from page generation directory to ASCII and place in simulation directory

            for pg_loc in page_locations:
                #clear string, ascii, and temp workspace directories prior to using them
                clear_directory.rm_files(os.path.join(string_to_ascii_path,'string_files'))
                clear_directory.rm_files(os.path.join(string_to_ascii_path,'ascii_files'))
                clear_directory.rm_files(os.path.join(main_sim_path,'temp_workspace'))

                #copy over page from page generation out_pages directory to string to ascii string directory
                page_name = 'loc_'+pg_loc+'_num_keywrd_'+str(cfg_f_num_of_keywrds+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                shutil.copy2(os.path.join(gen_page_path,'out_pages',page_name),
             os.path.join(string_to_ascii_path,'string_files'))

                #set page line count and multiple eop insertion flag. 1 if mainFilter, 0 for statistics based architectures
```

```
with open(os.path.join(string_to_ascii_path,'input_params','params.txt'),'w') as f:
        f.write('SBFProjManager'+'\n')
        if architecture is  'statistic': #Disable line count plus eop padding insertion if
                #statistic (Not really used in StringToAscii.py but left as placeholder)
                f.write(str(0)) #This assumes parallel based structure
        else: #Enable mainFilter architecture
                f.write(str(1)) #This assumes parallel based structure

#Next, execute string to ascii script
#string_to_ascii_script = os.path.join(string_to_ascii_path,'StringtoAscii.py')
#os.system(string_to_ascii_script)
StringtoAscii.strings_to_ascii(page_size_limit)

#Confirm script ran successfully and output ascii file is in the temporary workspace
if not os.path.isfile(os.path.join(main_sim_path,'temp_workspace','input.txt')):
        raise FileNotFoundError(
                "input.txt, the converted string to ascii file was not created. "+
                "\nMake sure StringtoAscii script was properly"+
                "called and it's string_files directory is not empty")

#move ascii page from temporary workspace to simulation directory
shutil.move(os.path.join(main_sim_path,'temp_workspace','input.txt'),os.path.join(cur_sim_path,page_name))

#
#
####SIM SHOULD ALWAYS RUN THE STATISTICS FIRST
#THEN REPROGRAM THE MAIN FILTERS OTHERWISE ERROR####
#
#

#Finally run the simulator with the config file and input file in the simulation directory
#Input format: <cfg filename> <input_data filename> <sim_out filename> <stats filename> <raw_data filename>
#print("Now running: "+cur_sim_path+'_out_'+page_name+'\n')
program        = os.path.join(cur_sim_path,'Generic_asap.exe')
arg_cfg        = os.path.join(cur_sim_path,cfg_name)
arg_input = os.path.join(cur_sim_path,page_name)
arg_output = os.path.join(cur_sim_path,'out_'+page_name)
arg_stats = os.path.join(cur_sim_path,'stats_'+page_name)
arg_all_cores_raw = os.path.join(cur_sim_path,'raw_iter_'+
                                   str(iteration+1)+'_'+page_name)
arg_per_core_raw                = os.path.join(cur_sim_path,'pcr_iter_'+
                                  str(iteration+1)+'_'+page_name) #needed for statistic arch run
try:
        subprocess.check_output([program, arg_cfg,arg_input,
                     arg_output,arg_stats,arg_all_cores_raw,
                     arg_per_core_raw],shell=True,
                    stderr=subprocess.STDOUT)
        #retcode = subprocess.call("mycmd" + " myarg")
except OSError as e:
        print("Execution failed:", e, file=sys.stderr)

#check if output from sim was 1
#time.sleep(1) #Give the program time to finish writing out the output to file
out = ''
retry = 0 #Number of times to retry a failed run to success
t_delay = 0
while (os.stat(os.path.join(cur_sim_path,'out_'+page_name)).st_size <= 0) and (t_delay <100):
```

```python
                                                subprocess.check_output([program, arg_cfg,arg_input,arg_output,
                                                           arg_stats,arg_all_cores_raw,arg_per_core_raw],
                                                          shell=True,stderr=subprocess.STDOUT)
                                                t_delay +=1
                                                time.sleep(t_delay) #Give the program time to finish writing out the output to file

                                        if os.stat(os.path.join(cur_sim_path,'out_'+page_name)).st_size <= 0:
                                                raise AssertionError("Failed. Blank output file: ")
                                #sort energy and other metrics here
print("\nPost processing...\n")

#Average out the raw data iterations
for architecture in architectures:
        for cur_num_kw in range(number_of_keywords):
                cur_sim_path = os.path.join(main_sim_path,architecture,str(cur_num_kw+1)+'keywords')
                for pg_loc in page_locations:
                        for kw_length in keyword_lengths:
                                page_name = 'loc_'+pg_loc+'_num_keywrd_'+str(cur_num_kw+1)+'_keywrd_len_'+str(kw_length)+'.txt'
                                with open(os.path.join(cur_sim_path,'raw_'+page_name),'w') as f:
                                        performance_params = [0,0,0,0,0]
                                        for iteration in range(num_iterations):
                                                with open(os.path.join(cur_sim_path,'raw_iter_'+str(iteration+1)+
                                                         '_'+'loc_'+pg_loc+'_num_keywrd_'+str(cur_num_kw+1)+
                                                         '_keywrd_len_'+str(kw_length)+'.txt')) as f_raw:
                                                        temp = list(f_raw.readlines()) #Needed values in list must be converted to floats!!!
                                                        performance_params[0] += float(temp[0])
                                                        performance_params[1] += float(temp[1])
                                                        performance_params[2] += float(temp[2])
                                                        performance_params[3] += float(temp[3])
                                                        performance_params[4] += float(temp[4])
                                                        # Energy (uJ) = performance_params[0]
                                                        # Throughput (MWords/sec) = performance_params[1]
                                                        # Runtime (ms) = performance_params[2]
                                                        # Core Area = performance_params[3]*Unit_Area
                                                        # Mem Area = performance_params[4]*2*Unit_Area
                                        f.write(str(performance_params[0]/num_iterations)+'\n')
                                        f.write(str(performance_params[1]/num_iterations)+'\n')
                                        f.write(str(performance_params[2]/num_iterations)+'\n')
                                        f.write(str(performance_params[3]/num_iterations)+'\n')
                                        f.write(str(performance_params[4]/num_iterations))



#Put together the generated raw files and stats to be processed by excel
gen_excel_dat.combine_data_and_stats(main_sim_path,
                                main_sim_path,number_of_keywords,
                                keyword_lengths,
                                keyword_list,
                                page_locations,
                                architectures)
```

Bibliography

[1] E. Bendersky, "Finite state machines and regular expressions," June 2013. [Online]. Available: https://www.gamedev.net/resources/_/technical/general-programming/finite-state-machines-and-regular-expressions-r3176

[2] HackingOff, "Regular expression to nfa (non-deterministic finite automata) generator," 2012. [Online]. Available: http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

[3] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, Jun. 1968. [Online]. Available: http://doi.acm.org/10.1145/363347.363387

[4] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 1, pp. 39–47, March 1960.

[5] E. Adeagbo *et al.*, *C2S2 Annual Review*, Carnegie Melon University, Pittsburgh, PA. Team members: Regular Expression Processing: E. Adeagbo, J. Pimentel, M. Braly, Statistics and Histogram: J. Pimentel, Database Sorting and Table of Performances: A. Stillmaker, Regular Expression Processing, Database Sorting and Statistics Figure: B. Baas, Std., October 2011, 5. [Online]. Available: http://www.ece.cmu.edu/research/sheets/c2s2.pdf

[6] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of CMOS circuits from 180 nm to 22 nm," VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4, Dec. 2011.

[7] J. Yang, W. Wang, and P. Yu, "BASS: approximate search on large string databases," in *Scientific and Statistical Database Manage., 2004. Proc. 16th Int. Conf. on*, June 2004, pp. 181–190.

[8] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Comput. Architecture, 2005. ISCA '05. Proc. 32nd Int. Symp. on*, June 2005, pp. 112–122.

[9] N.-F. Huang *et al.*, "A deterministic cost-effective string matching algorithm for network intrusion detection system," in *Commun., 2007. ICC '07. IEEE Int. Conf. on*, June 2007, pp. 1292–1297.

[10] S. Rus, R. Ashok, and D. Li, "Automated locality optimization based on the reuse distance of string operations," in *Code Generation and Optimization (CGO), 2011 9th Annu. IEEE/ACM Int. Symp. on*, April 2011, pp. 181–190.

[11] A. R. Bairoch A, "The SWISS-PROT protein sequence data bank," in *Nucleic Acids Research*. Nucleic Acids Research, 2000. [Online]. Available: http://www.ebi.ac.uk/swissprot/

[12] S. F. Altschul *et al.*, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022283605803602

[13] S. Burkhardt *et al.*, "Q-gram based database searching using a suffix array (quasar)," in *Proceedings of the Third Annual International Conference on Computational Molecular Biology*, ser. RECOMB '99. New York, NY, USA: ACM, 1999, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/299432.299460

[14] T. Kahveci and A. K. Singh, "An efficient index structure for string databases," in *VLDB*, vol. 1, 2001, pp. 351–360.

[15] A. Andersson and S. Nilsson, "Efficient implementation of suffix trees," *Software: Practice and Experience*, vol. 25, no. 2, pp. 129–141, 1995.

[16] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[17] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975. [Online]. Available: http://doi.acm.org/10.1145/360825.360855

[18] I. Sourdis and D. Pnevmatikatos, "Pre-decoded cams for efficient and high-speed nids pattern matching," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004, pp. 258–267.

[19] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 111–120.

[20] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, *Specialized Hardware for Deep Network Packet Filtering*, M. Glesner, P. Zipf, and M. Renovell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-46117-5_48

[21] C. R. Clark and D. E. Schimmel, *Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 956–959. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45234-8_94

[22] M. Pedram, "Energy-efficient datacenters," *Comput-Aided Des. Integr. Circuits Syst., IEEE Trans.*, vol. 31, no. 10, pp. 1465–1484, Oct 2012.

[23] F. J. Rammig *et al.*, "Designing self-adaptive embedded real-time software - towards system engineering of self-adaptation," *Brazilian Symposium on Computing System Engineering, SBESC*, vol. 2015-April, pp. 37–42, 2015.

[24] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May/June 1999.

[25] I. Sourdis and D. Pnevmatikatos, "Pre-decoded cams for efficient and high-speed NIDS pattern matching," in *Field-Programmable Custom Comput. Mach., 2004. FCCM 2004. 12th Annu. IEEE Symp. on*, April 2004, pp. 258–267.

[26] J. Divyasree, H. Rajashekar, and K. Varghese, “Dynamically reconfigurable regular expression matching architecture,” *International Conference on Application-Specific Systems Architectures and Processors*, vol. 2-4, pp. 120–125, July 2008.

[27] W. Ong *et al.*, “A parallel bloom filter string searching algorithm on a many-core processor,” in *Open Syst. (ICOS), 2013 IEEE Conf. on*, Dec 2013, pp. 1–6.

[28] D. Truong *et al.*, “A 167-processor computational platform in 65 nm CMOS,” *Solid-State Circuits, IEEE J. of*, vol. 44, no. 4, pp. 1130–1144, April 2009.

[29] ——, “A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling,” in *VLSI Circuits, 2008 IEEE Symp. on*, June 2008, pp. 22–23.

[30] A. Stillmaker, L. Stillmaker, and B. Baas, “Fine-grained energy-efficient sorting on a many-core processor array,” *IEEE International Conference on Parallel and Distributed Systems*, December 2012.

[31] Y. Brun *et al.*, “Engineering self-adaptive systems through feedback loops,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5525 LNCS, pp. 48–70, 2009.

[32] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS ’07. New York, NY, USA: ACM, 2007, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323573

[33] I. Bonesana, M. Paolieri, and M. D. Santambrogio, “An adaptable fpga-based system for regular expression matching,” *Design, Automation and Test in Europe*, pp. 10–14, March 2008.

[34] P. Ferragina and R. Grossi, “The string b-tree: A new data structure for string search in external memory and its applications,” *J. ACM*, vol. 46, no. 2, pp. 236–280, Mar. 1999.

[35] Kevina, “Spell checker oriented word lists (scowl),” September 2014. [Online]. Available: http://app.aspell.net/

[36] T. Peters, “Timsort description,” 2002, http://bit.ly/2kSv2g2.

[37] C. P. Nicolas Auger, Cyril Nicaud, “Merge strategies: from merge sort to timsort <hal-01212839v2>,” Universit?aris-Est Marne-la-Vallée, HAL, Tech. Rep., 2015. [Online]. Available: https://hal-upec-upem.archives-ouvertes.fr/hal-01212839

[38] M. Davies, “The 400 million word corpus of historical american english (1810-2009),” *Irén Hegedus (ed.) English Historical Linguistics 2010,*, 2010.

[39] ——, “Word frequency data (corpus of contemporary american english),” 2008. [Online]. Available: http://www.wordfrequency.info/free.asp?s=y

[40] G. Berry and R. Sethi, “From regular expressions to deterministic automata,” *Theoretical Computer Science*, vol. 48, pp. 117 – 126, 1986. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397586900885

[41] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, pp. 227–238.

[42] M. H. Foroozannejad, B. Bohnenstiehl, and S. Ghiasi, "Bamse: A balanced mapping space exploration algorithm for gals-based manycore platforms," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 22–25, January 2013.

[43] B. Bohnenstiehl *et al.*, "A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000−processor array," *Symposium on VLSI Technology and Circuits*, 2016.

[44] ——, "Kilocore: A 32 nm 1000-processor array," in *IEEE HotChips Symposium on High-Performance Chips*, 2016.

[45] ——, "Kilocore: A 32 nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits (JSSC), In press*, 2017.

[46] E. Adeagbo and S. O'Driscoll, "Implantable radio transmitters for long range health monitoring," VLSI Computation Lab and SSCRL, ECE Department, University of California, Davis, Tech. Rep. ECE-2010, Dec. 2010, http://vcl.ece.ucdavis.edu/pubs/2010.12.techreport.implanttransmitter/.

[47] E. Adeagbo and S. Lewis, "A band-gap reference with internal digital signal processing," VLSI Computation Lab and SSCRL, ECE Department, University of California, Davis, Tech. Rep. ECE-2011, Jul. 2011, http://vcl.ece.ucdavis.edu/pubs/2011.07.techreport.dualslopeadc/.

[48] E. Adeagbo and B. Baas, "Energy-efficient string search architectures on a fine-grained many-core platform," in *Technology and Talent for the 21st Century (TECHCON 2015)*, 2015.