

High-Performance Regular Expression Scanning on the Cell/B.E. Processor

Daniele Paolo Scarpazza
IBM T.J. Watson Research Center
Multicore Computing Department
Yorktown Heights, NY 10598, USA
dpscarpazza@us.ibm.com

Gregory F. Russell
IBM T.J. Watson Research Center
Scalable Systems Department
Yorktown Heights, NY 10598, USA
gfr@us.ibm.com

ABSTRACT

Matching regular expressions (regexps) is a very common workload. For example, tokenization, which consists of recognizing words or keywords in a character stream, appears in every search engine indexer. Tokenization also consumes 30% or more of most XML processors' execution time and represents the first stage of any programming language compiler.

Despite the multi-core revolution, regexp scanner generators like flex haven't changed much in 20 years, and they do not exploit the power of recent multi-core architectures (e.g., multiple threads and wide SIMD units). This is unfortunate, especially given the pervasive importance of search engines and the fast growth of our digital universe. Indexing such data volumes demands precisely the processing power that multi-cores are designed to offer.

We present an algorithm and a set of techniques for using multi-core features such as multiple threads and SIMD instructions to perform parallel regexp-based tokenization.

As a proof of concept, we present a family of optimized kernels that implement our algorithm, providing the features of flex on the Cell/B.E. processor at top performance. Our kernels achieve almost-ideal resource utilization (99.2% of the clock cycles are non-NOP issues). They deliver a peak throughput of 14.30 Gbps per Cell chip, and 9.76 Gbps on Wikipedia input: a remarkable performance, comparable to dedicated hardware solutions. Also, our kernels show speedups of $57\text{--}81\times$ over flex on the Cell.

Our approach is valuable because it is easily portable to other SIMD-enabled processors, and there is a general trend toward more and wider SIMD instructions in architecture design.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel Programming*; F.2.2 [Analysis of Algorithms]: Nonnumerical Algorithms—*Pattern matching*

General Terms

Algorithms, Design, Performance

Copyright ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 22nd Annual International Conference on Supercomputing, ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA. Copyright 2009 ACM 978-1-60558-498-0/09/06 ...\$5.00.

Keywords

Regular Expressions, Multi-core, Cell Processor.

1. INTRODUCTION

Search engines pervade our society. The major search engines perform a cumulative 10 billion searches a month, with an annual increase rate of 15% [1]. However, the search engines' features don't come for free: indexing is a computationally intensive task, and the amount of data to be indexed is growing faster than the available processing power. In fact, between 2006 and 2010 the annual amount of information we produce will grow from 161 to 988 billion Gigabytes [19], i.e., a doubling period of 18.6 months. Tokenization is an application of regular expression matching that consumes a significant portion of a search engine indexer (between 14 and 20%, according to our benchmarks of Lucene [13], a popular open-source search engine library). In XML processing tools, tokenization absorbs 30% or more of the execution time [29, 31].

On the technology side, the uniprocessor scaling that ensured growing performance for many years at little effort for software designers has ceased. Computer architects now achieve higher performance by packing more and more cores on the same die [20]. They have discontinued the effort to extract more instruction-level parallelism, and they are adopting instead more data-parallel architectures, with wider SIMD or SIMT units (Single Instruction Multiple Data, Single Instruction Multiple Thread). For example, Intel's traditional cache-coherent multi-cores currently employ 128-bit-wide SSE (Streaming SIMD Extension [21]) instructions, but they will switch to 256-bit width in AVX (Advanced Vector eXtensions [12]) on *Sandy Bridge*, and to 512-bit width on *Larrabee* [34]. IBM's scratchpad-based Cell/B.E. processor employs 128-bit-wide SIMD units [16]. nVidia's CUDA-enabled GPGPUs [28] use groups of 32 threads called *warps* that coalesce instructions if the threads follow the same control flow.

Low-level details vary, but the impact on programming is similar: irregular, control-dominated code is inefficient because it hinders SIMDization (Intel, Cell), it costs heavy branch misprediction penalties (Cell), and it prevents SIMT operation coalescing (CUDA). Programmers must rethink algorithms bottom-up to avoid these hazards. If they do so, their algorithms will enjoy better performance, no matter which of these platforms dominates in five years. The regexp scanning kernels generated by Vern Paxson's flex [30] (by far the most popular regexp scanner generator) are examples of code with very irregular control flow, that does not compile nor run efficiently on multi-cores. Given the strong need for indexing and the increasing popularity of multi-core architectures, it is surprising that so little attention has been devoted to mapping tokenization efficiently to multi-cores.

	Features			Throughput per chip	
	Backup	Equiv. classes	Max states	Peak	Typical
1.	no	no	1024	14.30	9.76
2.	no	yes	* 6736	13.95	9.52
3.	yes	no	1024	13.14	8.33
4.	yes	yes	* 6736	12.81	8.12

* Data refer to a Lucene-based example with 19 equivalence classes. Your maximum number of states will depend on the number of equivalence classes is subject to alignment issues.

Table 1: Feature/performance trade-offs that our family of automaton kernels offer. The alphabet for kernels without equivalence classes is ASCII (128 symbols).

In this paper we present an algorithm that exploits the large amount of thread-level and data-level parallelism present in modern multi-cores to perform tokenization in parallel on multiple input streams against the same rule set. The algorithm is based on a Deterministic Finite Automaton (DFA) optimized for predication-like branch removal and SIMDization.

To prove the feasibility and the advantages of our approach, we describe in detail and benchmark one optimized implementation, based on the IBM Cell/B.E. processor (the Cell, for short). Although our techniques are general and apply to other general-purpose SIMD-enabled multi-core architectures (as we show in Section 5.7), we choose to present only this implementation, because of the considerable effort involved in hand-tuning low-level code on multiple architectures. Moreover, a second example would not add strength to the proof of concept.

The purpose of the long, technical details of Section 5 is to prove that even regular-expression matching can be attacked and cracked with the same concepts (mainly, predication-like branch elimination and SIMD parallelization) that proved successful [22, 32] in attacking the simpler problem of exact multi-pattern string matching. The higher complexity of regular expression matching pays a throughput penalty of 2–4 \times with respect to the performance of simple exact matching solutions reported by the authors. Portions of Section 5 are needed for completeness but they do not present scientific novelty.

Our implementation consists of four kernels with different features. They exploit all the thread-level and data-level parallelism available on the Cell (see Figure 1). They support scanners with a State Transition Table (STT) small enough to fit in the cores’ local memories (256 kbyte per core). We also present space-saving kernels that allow 5 \times –6 \times more states (depending on the rule set) by reducing redundancy in the STT, at the price of a slight decrease (2.5%) in throughput.

The purpose of our kernels is to demonstrate the viability of our approach, not to provide fully engineered solutions. As a consequence, they exhibit shortcomings (e.g., the lack of explicit support for the Unicode character set), but these shortcomings are not relevant in the scope of this work. Also, the limited size of the pattern sets may prevent our approach from being immediately usable in certain regular-expression-matching applications other than tokenization, e.g., network intrusion detection.

We do not discuss here the challenges involved in supporting large pattern sets: the problem is general and not just related to regex matching, it deserves discussion by itself, and it is not new. We have presented [33] techniques to perform exact multi-pattern string search against dictionaries as large as hundreds of megabytes.

Table 1 summarizes the performance and features of our kernels. We report throughput values in Gbps (billion bits of input processed per second) to ease comparison with network wire speeds, both for peak and typical operating conditions. We define as *peak* the performance of the automaton when the input does not match any rule, which is the norm in a Network Intrusion Detection System (NIDS) not under attack. We define as *typical* the performance of a real tokenizer’s pattern set (taken from Lucene [13]), running on ASCII HTML text files captured from Wikipedia articles.

Our fastest kernel (Kernel 1 in the table) delivers peak throughput of 14.30 Gbps per chip, and typical throughput of 9.76 Gbps. All the throughput values we report refer to a complete tokenizer workload that matches regular expressions and populates a corresponding output token tables with the matches. We do not consider the impact of load imbalance, but neither do the comparison figures we report later on a commodity processor. Our performance values compare very favorably against many solutions reported in literature (see Section 6), even those based on special-purpose hardware.

Kernel 1 does not support back-up transitions (a feature explained in Section 3), whereas Lucene’s tokenizer requires them. Kernel 3 supports back-up, and runs at a peak and typical throughput of 13.14 Gbps and 8.33 Gbps, respectively. These values are independent of the rule set, provided that the corresponding STT is not larger than the local store (256 kbytes). This allows for 1024 distinct uncompressed states with an ASCII input set of 128 symbols. This limit is sufficient for most tokenizers: Lucene’s only needs 174 states, and a complete lexical scanner for ANSI C [10] requires 245.

Applications requiring more states, can employ *equivalence classes* to increase space capacity at the expenses of throughput. Equivalence classes group input symbols that trigger the same transitions for any given state. This practically reduces the input set, and thus the number of columns in the STT. Lucene’s tokenizer requires only 19 such classes, and the ANSI C scanner only 64. In the table, Kernels 2 and 4 employ equivalence classes. The larger STTs come at the price of slightly lower performance: typical throughput decreases to 9.52 and 8.12 Gbps per socket, respectively.

Our approach focuses on providing optimized run-time kernels and nothing else. We rely on flex for the other tasks: parsing the user’s rule set, generating the automaton state-transition graph, creating the equivalence classes. Then we import the STT, recompile and encode it in an optimized form, and we couple it with the fastest of our kernels that provides the features required by the rule set. We reuse as much as we can of flex to leverage its maturity, reliability, richness in features, and familiarity to developers.

The remainder of this paper is organized as follows. Section 2 introduces scanners and tokenizers. Section 3 presents the specifics of flex’s kernel. Section 4 presents the features of the Cell processor relevant to this domain. Section 5 presents our optimizations and their performance impact. Section 6 reviews the related work. Section 7 presents our conclusions.

2. SCANNERS AND TOKENIZERS

A *lexical scanner* (scanner, for short) is a piece of software that recognizes matches of regular expressions within an input stream of symbols. We call a *rule set* the set of regular expressions that the scanner recognizes. When multiple overlapping substrings of the input match one or more regular expressions, only the *longest of the leftmost* match is accepted. All other matches that overlap it are discarded. Each regular expression in the rule set may have a *semantic action* that the scanner executes when the expression is matched. The same substring may match multiple rules. In flex, the first rule specified wins, and others are discarded. Scanners

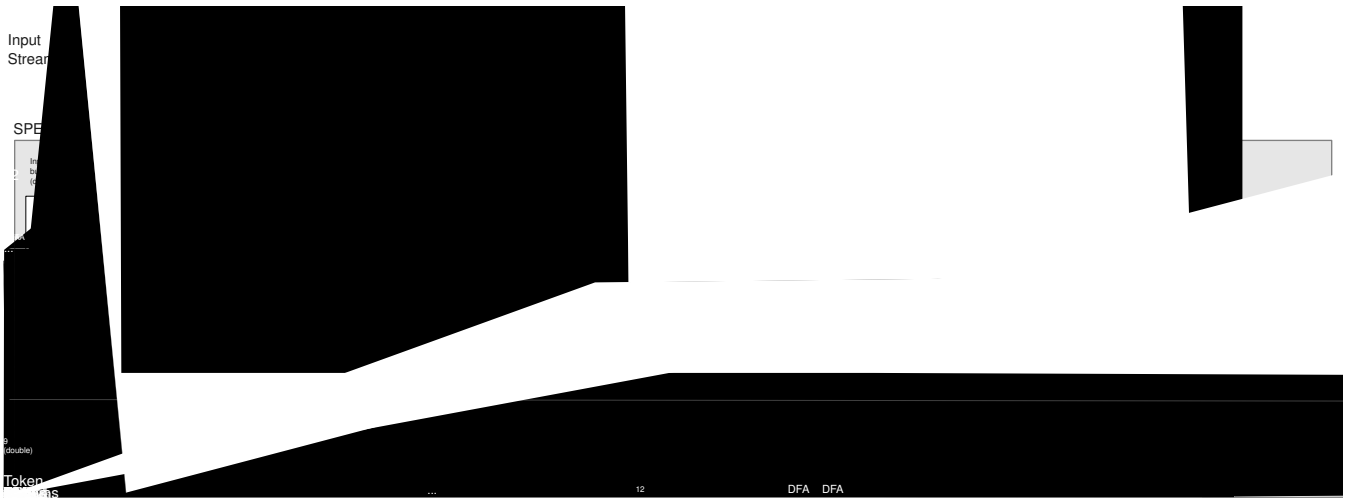


Figure 1: To exploit all the parallelism available on one Cell chip, we run 8 tokenizing automata per each SPE, e.g., automata $DFA_1, DFA_2, \dots, DFA_8$ run on SPE 1. The 8 automata are divided in two groups of 4, and within a group, all the automata share the same SIMD instructions. Each automaton operates on a distinct input stream and generates a distinct output token table, but all automata access the same State-Transition Table (STT) which is kept in the Local Stores (LS). The STT across all the SPEs are identical copies. We employ double buffering for the input streams and for the output tokens.

carry out diverse functions depending on their semantic actions. A *tokenizer* is a scanner whose semantic actions divide the text of an input document into words.

A scanner is more complex than a classic Deterministic Finite Automaton (DFA). A classic DFA is a 5-tuple $(\Sigma, Q, \delta, q_0, F)$: the alphabet, the state set, the transition function, the initial state and the final state set, respectively. Given an input string $I_0 I_1 \dots I_N$, a DFA processes the input as follows. At step 0, the DFA is in state $s_0 = q_0$. At each subsequent step i , the DFA transitions into state $s_i = \delta(s_{i-1}, I_i)$. A DFA can be used as a language acceptor, i.e., to tell whether an input string is a valid string of a language or not: string $I_0 I_1 \dots I_N$ is accepted iff $s_N \in F$, i.e., the DFA is in a final state when all the input is consumed. DFAs are provably equivalent to regular languages, i.e., for each set of regular expressions, a DFA exists that recognizes them, and vice versa. A classic DFA can not be used to: accept multiple matches in larger superstrings, accept only the longest of the leftmost among multiple matches, distinguish among multiple rules, or accept only the highest-priority match when multiple rules match.

3. HOW THE FLEX KERNEL WORKS

To recognize the “longest of the leftmost” match, a scanner does not execute semantic actions as soon as it finds a valid match. Rather, it saves the match in a variable and continues processing the input, in an attempt to find a longer match. It keeps going as long as the symbols it encounters are valid continuations of the match, possibly saving a longer match again and again. As soon as it finds a symbol that cannot continue the match, it stops. Then, depending on the input symbol, either it recognizes the input up to the last but one read symbol, or it recalls and accepts the last saved match.

A classic DFA has just one read head, i.e., the $i \in \{0, 1 \dots N\}$ introduced before, but a scanner has two read heads, the “base pointer” (bp) and the “current pointer” (cp). At any time, the automaton attempts to match the input substring $I_{bp} \dots I_{cp}$, i.e., the *candidate*.

Flex scanners have four kinds of transitions: regular, save, final and backup. In a *regular* transition the automaton updates its state according to the transition function, and advances cp by one place. In a *save* transition the automaton also saves a pair of values (cp, a) : cp is the value of the current pointer, and a is the number of the rule which matched the current candidate. a tells which semantic action to execute if the candidate is accepted. Then, cp is advanced one place. In a *final* transition the automaton accepts $I_{bp} \dots I_{cp-1}$ and executes the semantic action associated with the transition. Final transitions have exactly one associated semantic action. The character at cp is not consumed in the transition. The automaton resets (i.e., it transitions into the initial state), and bp takes the value of cp . A new candidate starts with the last examined symbol (which was not consumed) as its first symbol. In a *backup* transition the automaton recalls cp from the the last saved (cp, a) pair, and it executes a , thus accepts $I_{bp} \dots I_{cp}$. Then, it resets. Then, cp is advanced by one place and bp takes the value of cp , so that a new token candidate starts immediately after the recognized one;

Formally, a scanner is a 5-tuple $(\Sigma, Q, A, \Delta, q_0)$. Σ, Q, q_0 are respectively the alphabet, the state set and the initial state as in a classic DFA. A is a set of semantic actions. Δ is an extended transition function that yields the next state, the kind of transition and the associated semantic action: $\Delta : Q \times \Sigma \rightarrow (Q \times \{\text{regular, save, final, backup}\} \times \{A \cup \{\}\})$. The scanner has no final states, only final transitions. Actions are associated with transitions, not states.

Because of the additional complexity, the kernels of scanners and tokenizers exhibit much more irregular control flow than DFAs. Flex-generated kernels include nested loops, multiple selection statements (if, switch) and one unconditional jump (goto).

4. TARGETING THE CELL PROCESSOR

A Cell processor [23, 17, 24] contains a 64-bit PowerPC family Power Processing Element (PPE) with cache memories, and 8 Synergistic Processing Elements (SPEs) [16] with software-based scratchpad memories called Local Stores (LSs). The PPE and the SPEs have a 128-bit wide SIMD instruction set. The PPE is a ser-

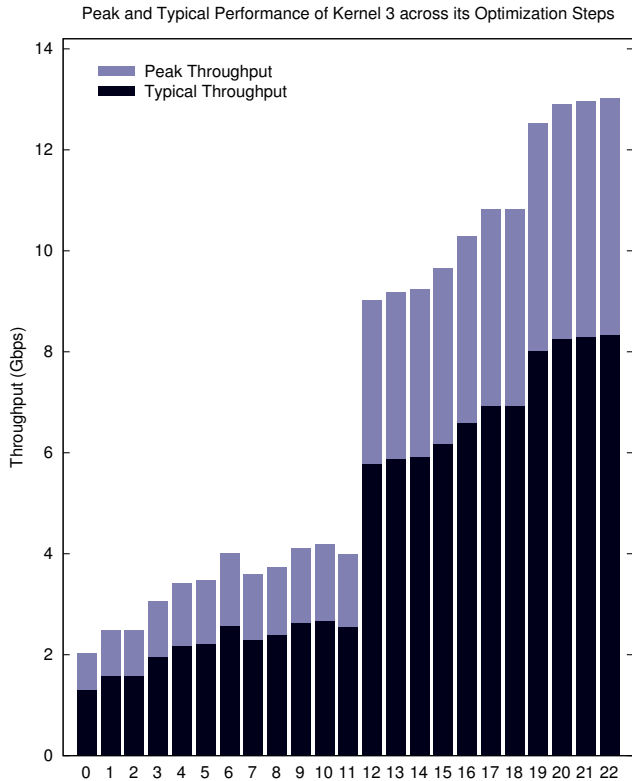


Figure 2: How the throughput grows, as an effect of applying each optimization step, with reference to our Kernel 3. Typical throughput refers to Wikipedia-like input, while peak throughput refers to non-matching input.

vice processor intended to run the OS and coordinate SPE threads; to achieve substantial speedups, programmers must offload their workloads to the SPEs.

An unmodified flex runs Lucene’s tokenizer on the Cell with a throughput of 123.20 Mbit/s, and it uses only one of the two hardware threads available on the PPE (Figure 10)¹. Two clones of the same flex process exploit both the PPE’s hardware threads and process distinct input files at approximately twice the speed (257.52 Mbit/s). Increasing the performance requires porting the flex kernel to the SPEs.

Programming the SPEs efficiently is challenging because: SIMD instructions require appropriate alignment and padding, branches are very expensive and should be avoided (branch predictors are simple, and misprediction penalties are high), load/store instructions only operate on the LS and force the programmer to use an explicit, small working set, and accesses to the main memory only happen via Direct Memory Access (DMA). Programmers should overlap computation with transfers to hide the shorter of the two latencies.

The flex kernel workload is compute-bound in its vanilla version and keeps being compute-bound throughout all our optimization steps. Therefore, we focus primarily on compute optimizations rather than data-transfer ones.

¹Our experiments use input files much larger (200 Mbyte) than the PPE’s L2 cache size (512 kbyte) to exclude the impact of input stream caching, and warm-up runs to eliminate the impact of disk I/O latencies.

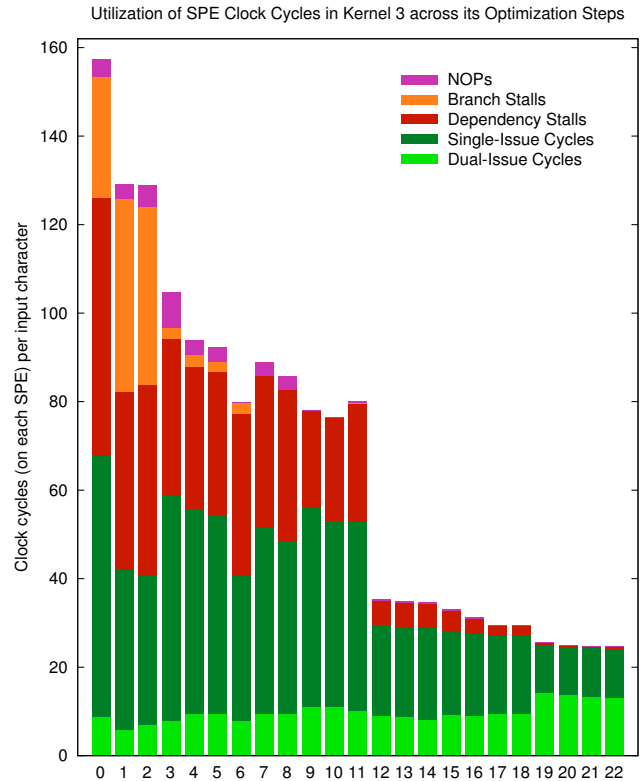


Figure 3: Utilization of the clock cycles across the optimization steps of Kernel 3. Shades of green represent useful cycles (single and dual issues), and shades of red represent stalls and NOPs. Data refer to typical conditions.

5. OPTIMIZATION TECHNIQUES AND RESULTS

We have optimized the code in a sequence of steps, numbered from (1) to (22). Step (0) represents an original flex kernel compiled and run on each of the 8 SPEs. Step (1) represents our first optimized kernel and (22) is our fastest code. Results were obtained with the IBM Cell SDK 3.0 including GNU GCC 4.1.1 on IBM QS21 and QS22 blades². Table 2 and Figure 2 show the impact of each step on our Kernel 3. Our other kernels show similar results; we do not report them for sake of brevity. Figure 3 reports the average number of clock cycles required by one SPE to process an input character, whereas Figure 2 reports the corresponding throughput in Gbps when all the 8 SPEs in a chip are used concurrently to scan independent streams.

5.1 Summary and speedup break-down

Globally, our optimizations improve performance by using fewer instructions per transition (the instructions per character decrease from 89.58 to 40.52) and making the code schedule denser (the CPI decreases from 1.76 to 0.61). Altogether, we transform sequential code into branchless, unrolled, SIMDized code, with manually re-ordered loads/stores, and output table updates which are speculative, aligned and padded.

²The difference between the processors used in these two blades, i.e., is fully pipelined double-precision floating-point units on QS22, is irrelevant for our purposes. Our string matching algorithm makes no use of floating point arithmetics.

Some steps have precedences among them (e.g., one must make the code branchless before SIMDizing it) but there, is in general, freedom in their order. We choose the order which appears the most sensible to us, but in no way do we hide the fact that low-level optimization is a trial-and-error process. Some steps like (7) locally decrease performance, but they are necessary to enable more beneficial steps at a later stage (e.g., SIMDization). We apply loop unrolling multiple times to evaluate its effect on diverse code.

Globally, on Kernel 3, we obtain a speedup of $67.7\times$ using 8 SPEs, when compared against original flex on the PPE. This speedup can be roughly broken down as follows:

- $1.32\times$ due to the SPE being slightly faster than the PPE on this code,
- $8\times$ due to using the 8 SPEs,
- $2.02\times$ due to our more efficient single-table STT encoding, reduced pointer arithmetics, and branch elimination,
- $2.20\times$ due to 4-way SIMDization; this is far from a theoretical $4\times$ because many instructions (e.g., loads and stores) are not SIMD,
- $1.44\times$ achieved by duplicating the number of engines; the added code fills dependency stalls with useful instructions and increase the double issue rate.

5.2 Efficient STT access

This section describes steps (1), (2), (4), (5) and (8).

Most kernel instructions deal with performing a state transition. Therefore, even a small decrease in the cycles per transition improves the overall performance significantly. Unfortunately, compilers cannot optimize this code aggressively, because it would require making data-representation choices that violate the C standard. We have co-designed the STT representation and the code that access it at the instruction level. We employ aligned pointers in place of state numbers to simplify STT addressing. The overall cost of an STT access decreases from five instructions (two integer additions, two integer multiplications, and a load) to a single instruction in the kernel.

5.3 Efficient encoding of the STT

Flex’s original kernel (0) uses two separate tables to represent the STT and each state’s semantic action (if any). Each transition

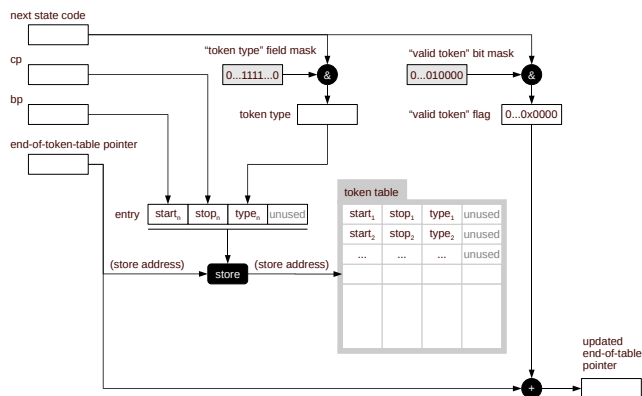


Figure 5: This data flow represents the branchless, speculative, aligned update of the output token table that each automaton employs.

accesses these two tables. We halve the loads by storing their information into one table, the STT. The details described here affect all steps (1)–(22).

We encode semantic action information in the least significant bits of the state pointers. These bits are unused because of row alignment in the STT. STT entries are 2-byte wide (16 bits) so, if we adopt a 128-symbol alphabet, row addresses are 256-byte aligned, and the 8 least significant bits are available. In general, with a 2^n -symbol alphabet, $(n + 1)$ bits are available.

We use these bits to encode the following information: whether the transition is final, whether the token is accepted (i.e., it is not ignored) and the type of token. Note that a transition can be final and the token not accepted, e.g., with stopwords. If the automaton supports back-up transitions, two additional flags tell whether the transition is of type save or restore. Additional $\lceil \log_2 t \rceil$ bits are needed to represent the type of token recognized, if t is the number of non-ignored token types.

Kernel 1 uses 128 input symbols; it uses 2 of the 8 available bits to encode the *final* and *token* flag, and it can encode up to $2^6 = 64$ distinct token types with the remaining ones. Kernel 3, which supports back-up, needs two additional bits (*save* and *restore*), so only $2^4 = 16$ token types are supported. Depending on the chosen alphabet, the available bits may not be enough to represent all the above information. If so, the designer can either pad the STT rows to a larger alignment or use a separate table, as flex does.

5.4 Generate Output Efficiently

This section describes steps (7) and (8). As an output, a tokenizer populates the *output token table*, i.e., the list of the non-ignored tokens that match the rules. We organize this table as a sequence of tuples (start, stop, type), i.e., telling where each token begins and ends in the input stream, and what is the token’s type. We tuned this code to maximize performance: we pad table entries to the register size (128 bits) to avoid expensive unaligned writes and we update the table speculatively to avoid branches.

This mechanism is illustrated in Figure 5 (for clarity we draw only one automaton, although our final implementation runs multiple automata in parallel). The next state code, as read from the STT, contains bits to indicate whether the candidate is a valid token and which type it is. Simple bitwise AND operations are sufficient to isolate the two values.

5.5 Reduce the Cost of Back-up Transitions

Depending on the rule set, flex may generate automata that require back-up transitions. We present an original optimization that mitigates the cost of back-up by transforming the state-transition graph. This optimization affects steps (2)–(19).

Let us consider a minimal, meaningful rule set which translates into an automaton with back-up transitions:

```

LETTER      [A-Za-z]
DIGIT       [0-9]
ALPHA       {LETTER}+
TOKEN       ({LETTER}|{DIGIT})+
ACRONYM     {ALPHA} "." ({ALPHA} ".")+
%%
.\|n       ; /* Do nothing */ /* Action 1 */
{TOKEN}    emit_token( yytext ); /* Action 2 */
{ACRONYM}  emit_acronym( yytext ); /* Action 3 */

```

The rule set matches (and prints) alphanumeric tokens and acronyms, ignoring everything else. The corresponding automaton is in Figure 6. We label each transition with the range of input symbols that trigger it, the kind of transition if non-regular (save, back-up or final), and its action number. Let us consider how the scanner processes the example input “a.b.c” (note the final space):

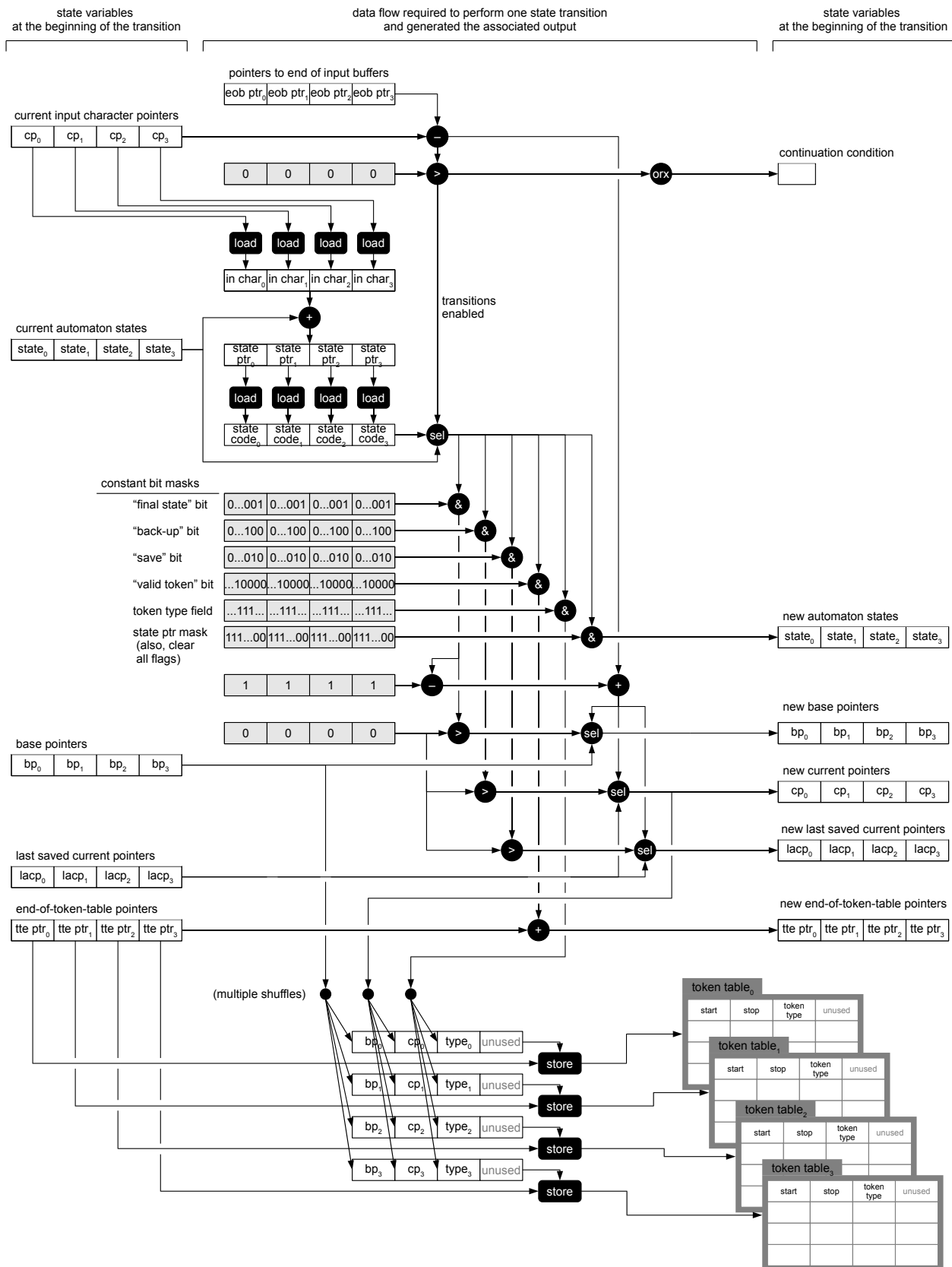


Figure 4: The data-flow of a branchless, SIMDized state transition for four concurrent automata, with support for back-up transitions (as in Kernels 3 and 4). Our kernels that do not support backup (1 and 2) have a subset of this data-flow (i.e., without 'laccp', 'backup bit' mask, 'save bit' mask, and the associated operators).

Optimization Step	Typical Throughput (Gbps)	Cycles/char (on each SPE)	CPI	Insts per char	NOPs	Branch Stall Rate	Dep. Stall Rate	Single Issue Rate	Dual Issue Rate	Used Regs
(0) Vanilla flex ×8 SPEs	1.30	157.31	1.76	89.58	2.48%	17.38%	36.97%	37.60%	5.57%	76
(1) Single table	1.59	129.08	2.29	56.33	2.52%	33.87%	31.04%	28.02%	4.56%	34
(2) Final-to-initial State Shortcut	1.59	128.89	2.43	52.94	3.73%	31.28%	33.31%	26.29%	5.39%	34
(3) Partial Branch Speculation	1.96	104.62	1.34	77.97	7.57%	2.35%	33.66%	48.85%	7.57%	37
(4) STT Relocation	2.18	93.73	1.25	74.87	3.47%	2.62%	34.25%	49.55%	10.11%	36
(5) Input SHL 1 bit	2.22	92.27	1.23	75.17	3.54%	2.52%	34.80%	48.80%	10.35%	56
(6) Local Variable Copies	2.57	79.81	1.59	50.27	0.19%	2.92%	46.08%	40.79%	10.02%	53
(7) Speculative Output Update	2.30	88.95	1.29	68.80	3.52%	0.01%	38.50%	47.38%	10.59%	59
(8) Token bit flag=16	2.39	85.84	1.31	65.68	3.64%	0.01%	39.89%	45.47%	10.97%	59
(9) Branchless	2.62	78.06	1.04	75.03	0.02%	0.01%	27.92%	57.99%	14.06%	68
(10) Branchless Unroll ×2	2.68	76.50	1.10	68.80	0.02%	0.01%	30.52%	55.10%	14.35%	82
(11) Branchless Unroll ×4	2.56	80.00	1.20	66.46	0.51%	0.02%	33.56%	53.17%	12.75%	88
(12) 4 Engines, SIMD	5.77	35.47	0.86	41.33	1.11%	0.01%	15.36%	58.24%	25.28%	89
(13) 4 Engines, SIMD, Unroll ×2	5.87	34.89	0.86	40.55	1.13%	0.01%	15.62%	58.10%	25.15%	86
(14) 4 Engines, SIMD, Unroll ×4	5.92	34.60	0.87	39.97	0.86%	0.01%	14.62%	60.84%	23.67%	86
(15) 4 Engines, SIMD, U×2, SW/pipe	6.18	33.14	0.81	40.74	1.19%	0.01%	14.09%	57.06%	27.65%	89
(16) 4 Engines, SIMD, U×4, SW/pipe	6.59	31.09	0.79	39.48	0.64%	0.01%	10.64%	60.19%	28.53%	89
(17) 4 Engines, SIMD, U×8, SW/pipe	6.93	29.54	0.76	38.75	0.34%	0.01%	8.23%	59.57%	31.85%	89
(18) 4 Engines, SIMD, U×16, SW/pipe	6.97	29.38	0.75	39.37	0.54%	0.01%	7.42%	59.62%	32.40%	89
(19) 8 Engines, SIMD, U×2, SW/pipe	8.02	25.54	0.60	42.56	0.02%	0.02%	2.07%	41.93%	55.97%	72
(20) 8 Engines, SIMD, U×4, SW/pipe	8.26	24.81	0.60	41.59	0.02%	0.02%	0.94%	43.57%	55.45%	72
(21) 8 Engines, SIMD, U×8, SW/pipe	8.30	24.68	0.61	40.43	0.02%	0.02%	1.14%	44.88%	53.94%	72
(22) 8 Engines, SIMD, U×16, SW/pipe	8.33	24.57	0.61	40.52	0.02%	0.02%	1.05%	45.09%	53.83%	72
Unoptimized Reference	Throughput	(1 PPE)								
Original flex on 1 PPE	0.12	207.79								

Table 2: The impact of our optimization steps on the performance and quality of code of Kernel 3. Column ‘Gbps’ reports typical throughput per chip (8 SPEs), obtained running Lucene’s tokenizing rule set on Wikipedia HTML pages.

step	1	2	3	4	5	6
input symbol	a	.	b	.	c	(space)
transition kind	save	regular	regular	save	regular	back-up
next state	6	8	10	11	10	1
last saved (cp, a)	(1,1)	(1,1)	(1,1)	(4,3)	(4,3)	(4,3)

At step 4, the scanner recognizes substring “a.b.” as a valid acronym by saving Action 3. Then it reads “c”, which is a valid continuation of an acronym. Then it reads the space, which is not a valid acronym continuation. So, at step 6, the scanner backs up to the saved $(cp, a) = (4, 3)$: it accepts candidate $I_1..I_4 = \text{“a.b.”}$ as an acronym (Action 3). At the next iteration the scanner restarts, in the initial state, at input symbol “c”.

To reduce the cost involved in save and back-up transitions, we only save cp rather than (cp, a) and encode a in each transition. This optimization is not possible when multiple distinct values of a may correspond to one transition. We now propose a transformation of the state-transition graph to counter that.

We call a state *ambiguous* if it can be reached with multiple saved values of a . We call a transition *ambiguous* if it originates from an ambiguous state. We call a scanner *ambiguous* if it has at least one ambiguous back-up transition. The scanner in our example is ambiguous: Figure 7 shows the possible saved values of a for each state in curly braces. States 8 and 10 are ambiguous and have ambiguous back-up out-transitions.

We now describe how to transform an ambiguous scanner into an equivalent, non-ambiguous one. We say that two scanners are equivalent iff they accept the same substrings with the same semantic actions. Given an ambiguous scanner $T = (\Sigma, Q, A, \Delta, q_0)$, the transformation yields a non-ambiguous new scanner $T' =$

$(\Sigma, Q', A, \Delta', (q_0, \{\}))$ that is equivalent to T . T' uses the same alphabet and semantic actions as T . The state set of T' is $Q' = Q \times (A \cup \{\})$, i.e., for each state q_i of T , there are $|A| + 1$ states in Q' as follows: $(q_i, \{\}), (q_i, a_1), (q_i, a_2), \dots$. The new transition function is:

$$\Delta' : Q' \times \Sigma \rightarrow (Q' \times \{\text{regular, save, final, backup}\}),$$

constrained as follows:

$$(s, i) \xrightarrow{\Delta'} (q, \text{kind}) \wedge (\text{kind} = \text{final} \vee \text{kind} = \text{backup}) \Rightarrow q = q_0.$$

The new Δ' is computed from Δ according to the following rules:

- if $(s, i) \xrightarrow{\Delta} (q, \text{regular}, \{\})$,
then $\forall a \in \{A \cup \{\}\}$ add $((s, a), i) \xrightarrow{\Delta'} ((q, a), \text{regular})$;
- if $(s, i) \xrightarrow{\Delta} (q, \text{save}, a)$,
then $\forall m \in \{A \cup \{\}\}$ add $((s, m), i) \xrightarrow{\Delta'} ((q, a), \text{save})$.
- if $(s, i) \xrightarrow{\Delta} (q_0, \text{final}, a)$,
then add $((s, a), i) \xrightarrow{\Delta'} ((q_0, \{\}), \text{final})$;
- if $(s, i) \xrightarrow{\Delta} (q_0, \text{backup}, \{\})$,
then $\forall a \in \{A \cup \{\}\}$ add $((s, a), i) \xrightarrow{\Delta'} ((q_0, \{\}), \text{backup})$.

The disambiguated scanner operates similarly to that described in Section 3, with the following differences: in a *save* transition, only cp is saved rather than (cp, a) ; in a *final* transition from state

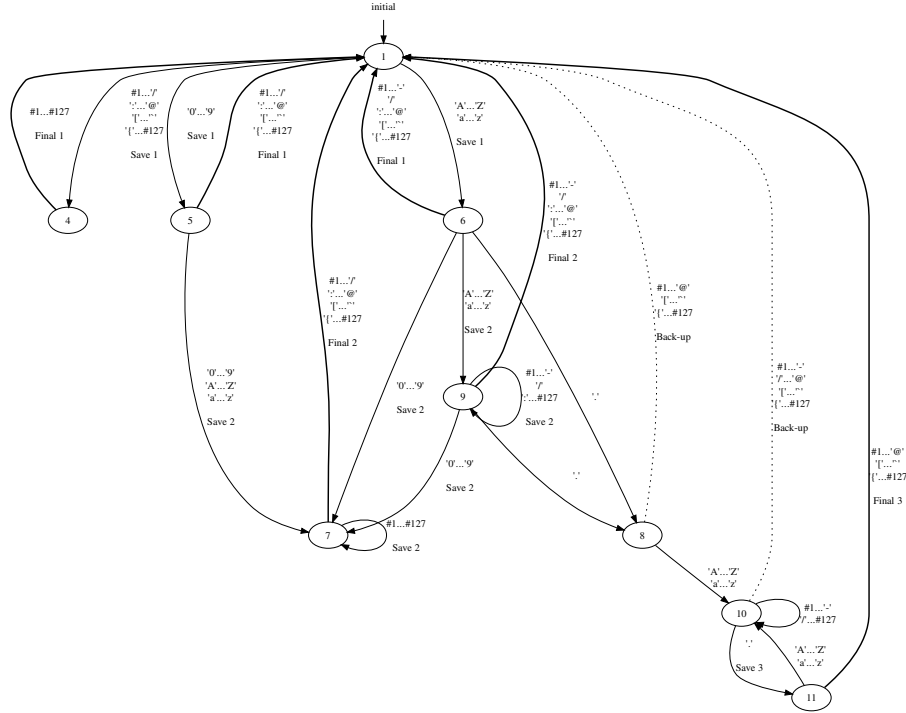


Figure 6: The original automaton generated by flex from the example rule set. In this automaton, a transition may ‘Save’ a semantic actions, or ‘Back-up’ to semantic action saved earlier.

(q, a) to state $(q_0, \{\})$, action a is executed; in a *backup* transition from state (q, a) to state $(q_0, \{\})$, the automaton recalls the last saved cp and executes a . The transformed automaton for our example rule set is in Figure 8. It processes the same input string “a.b.c” as follows:

step	1	2	3	4	5	6
input	a	.	b	.	c	(space)
transition kind	save	regular	regular	save	regular	back-up
next state	(6,1)	(8,1)	(10,1)	(11,3)	(10,3)	(1,{})
last saved cp	1	1	1	4	4	4

Disambiguation increases the number of states by a factor of $1 + |A|$ (one plus the number of actions), since $Q' = Q \times (A \cup \{\})$. Fortunately, most states in Q' may be unreachable. They can be safely eliminated, together with their out-transitions. In the example, our transformation increases the number of reachable states from 9 to 12. In a realistic scenario, the increase in reachable states caused by disambiguation is acceptable: Lucene’s tokenizer grows from 159 to 173 states (+9%).

5.6 Branch removal and SIMDization

Branches disrupt the benefits of SIMDization because they allow different automata to take different paths. In addition, branches are expensive on the SPEs, where branch predictors are simple, and the misprediction penalty is large. A branch miss can cost as much as 26 clock cycles, and there are multiple branches in a single automaton transition, in flex’s kernel. A delay of 26 cycles is large, especially compared to the 36.5 cycles required by our optimized Kernel 1 to complete one transition. This section describes the branch-avoiding optimization techniques we use in steps (3),(9)–(19).

In steps (3), (7) and (9) we remove branches via software speculation: i.e., computing the results of both branches and select-

ing the right result a posteriori. Software speculation cuts branch miss stalls, facilitates loop unrolling, and creates long basic blocks, which lead to better scheduling alternatives for the compiler.

Furthermore, branchless automata can easily be run in parallel via SIMD instruction. We perform this SIMDization in step (12) and following. Figure 4 shows the SIMDized data flow corresponding to one state transition. Four automata run on separate input buffers and enqueue their output into the respective token tables (at the bottom right corner of the figure). White boxes enclosing 4 values represent 128-bit quadwords, where each 32-bit word belongs to one automaton. Each automaton ($i \in \{0, 1, 2, 3\}$) has its individual current character pointer (cp_i), base pointer (bp_i), current state, last saved current pointer (lcp_i), and pointer to the end of its output token table ($tteptr_i$). Grey boxes represent constant operands like bit masks. Black circles represent SIMD operators. Black rounded rectangles represent LS load/store operations. A “sel” circle represents a *select bits* instruction, and an “orx” circle represents an *or across* instruction; both are explained below. The boxes in the left-hand column of the chart represent values at the beginning of each iteration; those in the right-hand column represent the newly computed values.

Each automaton may finish processing its input at a different time. The data flow of the entire picture is repeated until all the automata complete, as follows. When one automaton hits the end of its buffer, its word in the “transitions enabled” quadword becomes zero, which inhibits further state transitions. New states keep being computed, but they are ignored: a “sel” instruction selects the old value of current state to be stored in the automaton’s current state variable. When all four automata have finished processing their inputs, all words in “transitions enabled” drop to zero, and so does the “continuation condition”, which is their bitwise *or*.

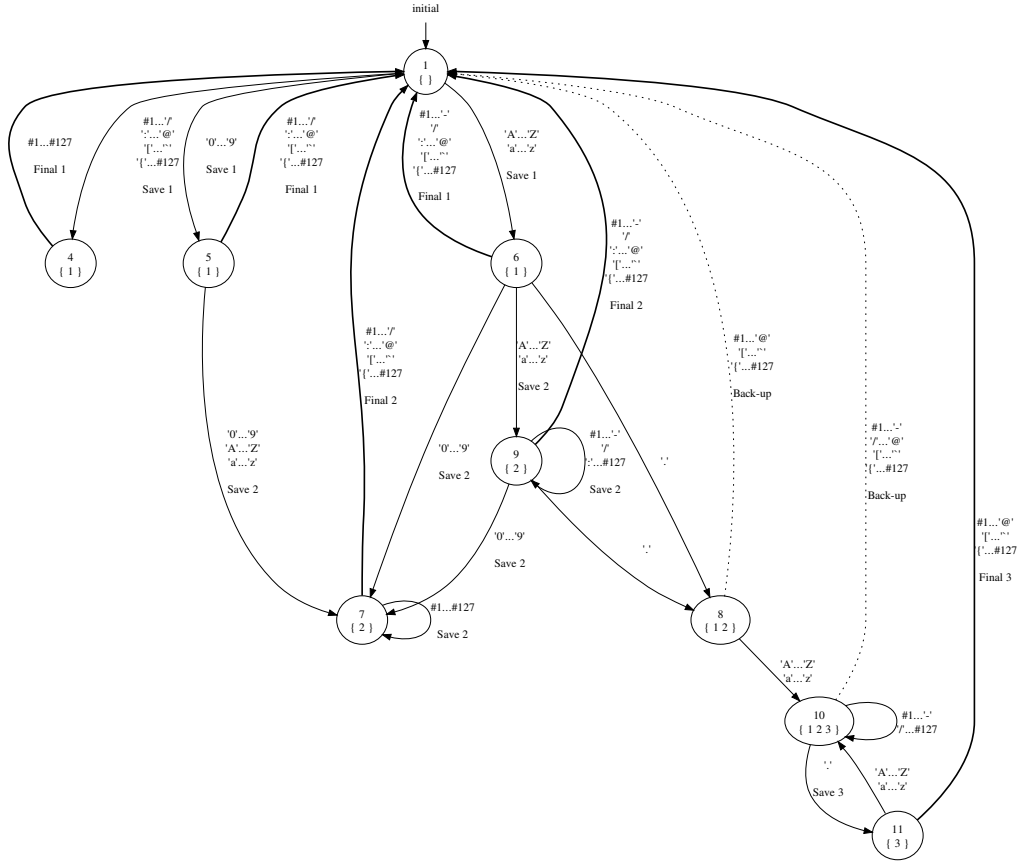


Figure 7: In this state chart, we explicitly list the semantic actions that might be saved upon entry into each state (the list is in curly braces). A few states (8 and 10) are ambiguous, because they have more than one such saved value. Our approach solves this ambiguity.

The bit masks (the gray boxes in the middle of the figure) extract control bits from the state code, and govern the update of variables bp_i , cp_i , lcp_i . The group of instructions *and*, > 0 , *sel* occurs frequently: its purpose is to use a 1-bit flag to select an entire 32-bit word from two possible sources. This effectively avoids branches. The token table update mechanism is a four-fold replica of that in Figure 5.

The use of quadword “transitions enabled” inhibits both unwanted state transitions and unwanted side effects. Upon end-of-buffer conditions, each automaton keeps using a value of current state which was cleared from any control bits by means of the “state pointer mask”. Thanks to the “transitions enabled” latch, we can unroll the loop an arbitrary number of times without need for specialized loop heads or tails. We explore the impact of loop unrolling in steps (13) and following.

The GCC compiler does not reorder loads and stores efficiently. In steps (15) and following, we manually overlap unrolled iterations, so that input character loads take place before output table writes. This causes the compiler to generate more compact code while preserving correctness.

The SIMDization we introduced in step (12) does not provide sufficient parallelism to nullify the dependency stalls. Even after $16\times$ unroll, in Step (18), 7.64% of the clock cycles are wasted in dependency stalls. We fix this in Steps (19)–(22) by doubling the amount of automata running at the same time. These steps use 8 concurrent automata, SIMDized 4 at a time. This optimization

reduces dependency stalls to less than 1% of the cycles and leads to a cycle utilization equal to 99.24%. Steps (19)–(22) lead to our final software architecture depicted in Figure 1.

For reference, Figure 9 presents the performance of a recent Intel quad-core processor when running vanilla flex scanners in peak and typical conditions. Since these results do not employ SIMD-optimized code, they are not intended for direct comparison against our optimized kernels.

5.7 Portability to other architectures

Nothing in the approach we have presented is strictly specific to the Cell architecture. The SIMDized instructions that appear in the flow chart of our branchless automaton transition (represented as black circles in Figure 4) either have a one-to-one mapping to corresponding instructions in the SIMD instruction set of other architectures, or have a relatively inexpensive multi-instruction translation.

Porting to the other cores of the IBM Power family is trivial because the SPU intrinsics have a direct mapping to VMX. Porting to Intel x86 machines is also possible. For example, all the SIMD instructions in Figure 4 have a direct SSE match, with the notable exception of the “sel” operator (intrinsic `spu_sel`), that we translate into three instructions with the following intrinsics: `_mm_or_si128(_mm_and_si128(...), _mm_andnot_si128(...))`. Further investigation is needed to leverage SSE primitives that are not available on the Cell.

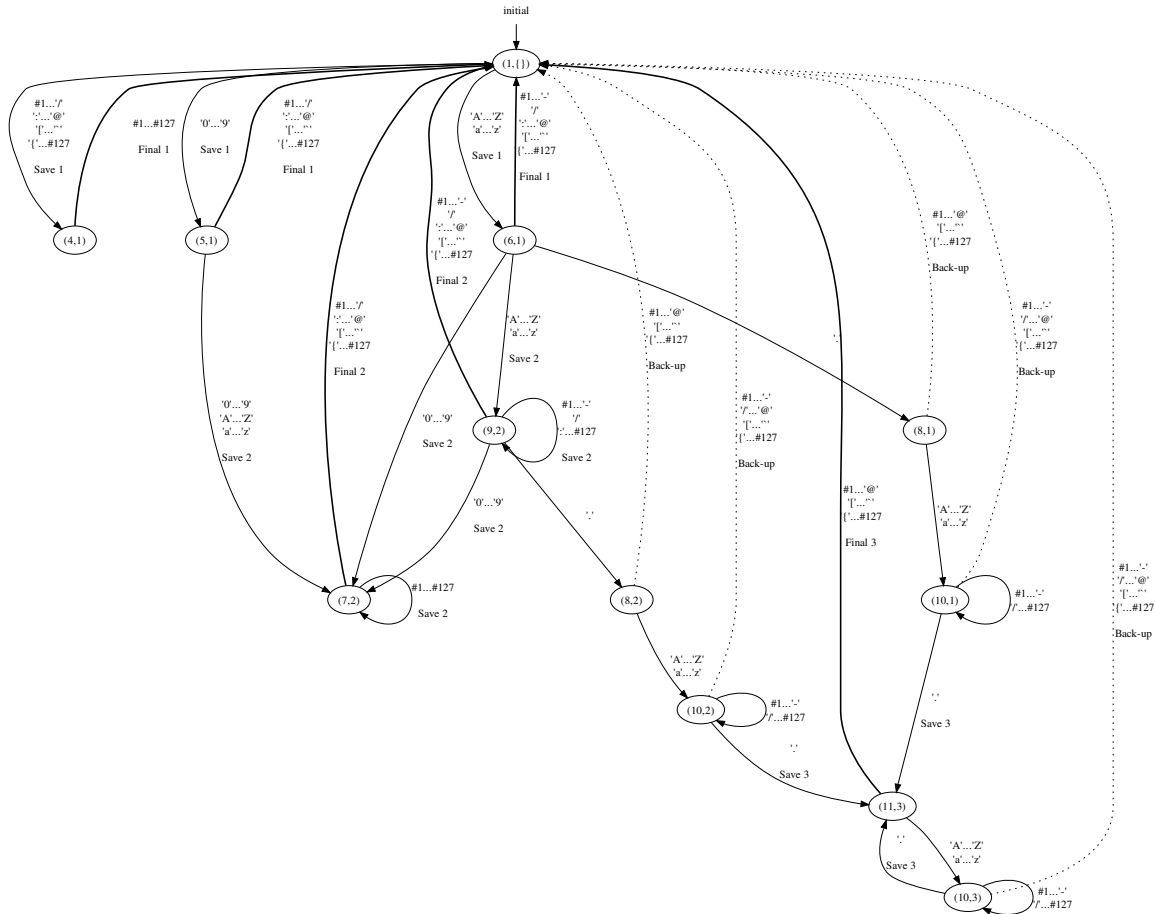


Figure 8: The disambiguated state-transition graph equivalent to the one in Figure 7.

6. RELATED WORK

Multi-pattern string matching, either against exact dictionaries or against regular expressions, is a prolific research area, where many algorithms have been employed, such as Bloom filters [4], Aho-Corasick [2], specialized state machines [39], and Content-Addressable Memories (CAMs).

Specialized hardware, especially FPGAs (Field Programmable Gate Arrays), is very popular. FPGAs have been proposed to match exact patterns with Bloom filters [11, 26, 37], CAMs [5, 36], or Aho-Corasick automata [9, 18, 35], and to match regular expressions with direct hardware synthesis [18, 25] or DFAs [27]. Most of these solutions are fast enough for use in a Network Intrusion Detection System, which filters incoming traffic at wire speed (several Gbps of throughput), although they usually exhibit a very limited dictionary capacity. For example, the parallel regular expression matching solution proposed by Lee et al. [25] delivers a throughput of 4.4 Gbps but supports only 25 rules, while Suresh et al. [37] match exact patterns with a Bloom filter-based solution capable of delivering 18 Gbps with 80-144 patterns.

Solutions based on commodity hardware support larger dictionaries and rule sets, but their performance can be rather low [3]. Even recent works [15, 6, 38] on lexical scanner generation for general-purpose processors do not attempt to exploit any thread- or data-level parallelism, and often generate code (e.g., go-to tables [38]) that resists any attempt of efficient parallel translation.

Recently, Cameron et al. [8, 7] proposed a bit-parallel approach to exploit the wide SIMD units available in contemporary hardware. Fast DFA-based solutions are appearing for exact matching on multi-core processors. On the Cell, Iorio and van Lunteren [22] propose a BFSM implementation that compresses the STT in the register file and achieves throughput in excess of 50 Gbps per chip, and Scarpazza et al. propose two Aho-Corasick implementations: one based on the local store [32], which delivers 40 Gbps, and one capable of supporting dictionaries as large as the available main memory [33], which delivers 3–4 Gbps. On a 128-CPU Cray XMT, Villa et al. [40] deliver 28 Gbps with large dictionaries. On nVidia GPGPUs, Goyal et al. [14] propose an automaton-based regular expression matching implementation that delivers 400 Mbps.

Nevertheless, despite the great attention to string matching, no work so far has explored the potential of the Cell processor for general regular expression matching.

7. CONCLUSIONS

We propose an algorithm to perform regular expression matching against small rule sets which suits the needs of search engine tokenizers and maps well to SIMD multi-core architectures. Implementations of this algorithm can replace the traditional kernels generated by the flex (the popular regexp scanner generator), while we still use flex’s front-end to parse the rule sets and generate the

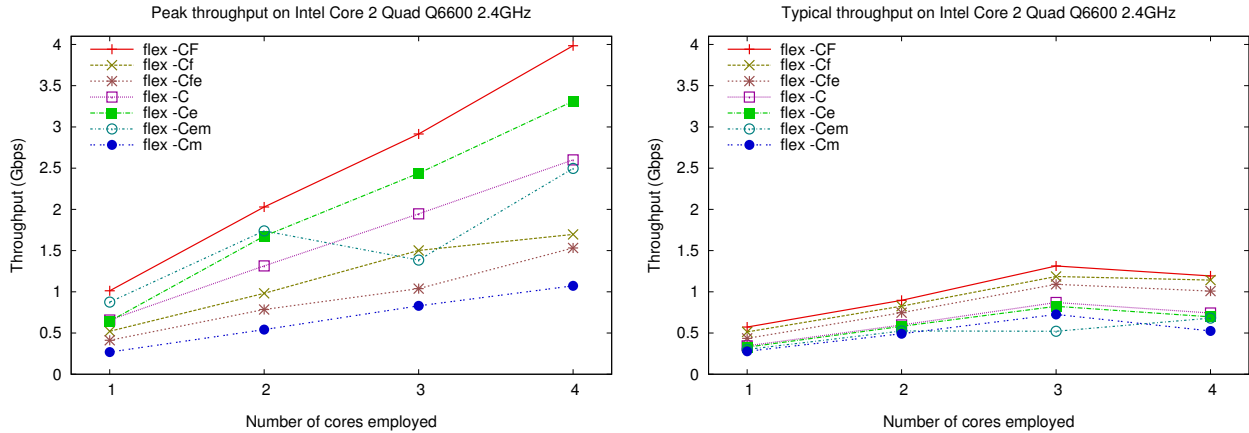


Figure 9: Peak (left) and typical (right) performance of the flex kernel (including token table update) on an Intel Core 2 Quad Q6600 processor, compiled with different STT representations (options -CF, -Cf, ...). The processor contains 4 cores running at 2.4 GHz.

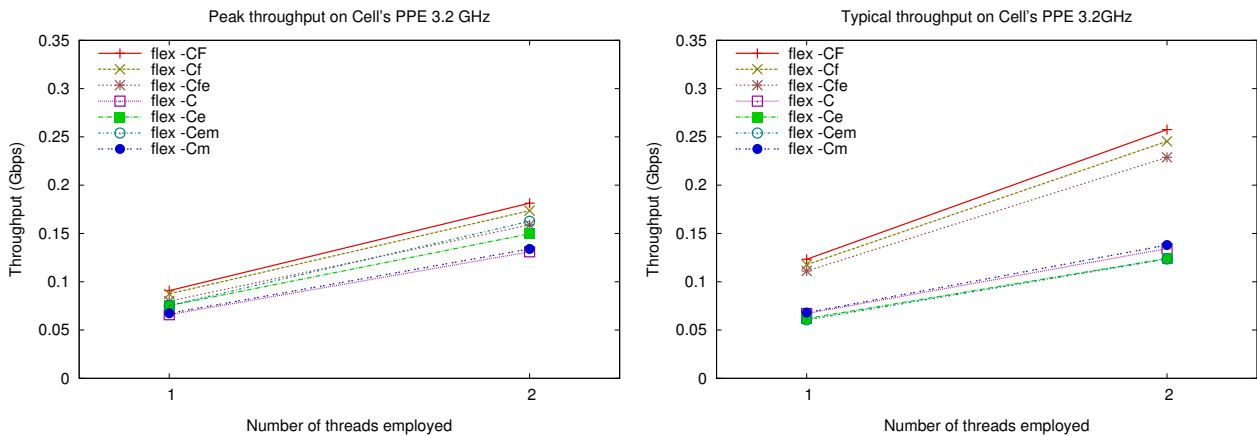


Figure 10: Peak (left) and typical (right) performance of the flex kernels (including token table update) on the Cell's PPE processor, compiled with different STT representations (options -CF, -Cf, ...). The processor is a 2-thread single-core running at 3.2 GHz. Unlike any other platform we have considered, the PPE runs in typical operating conditions faster than in peak conditions.

corresponding automata.

We applied optimizations including state-transition graph transformation, branch removal, software speculation, SIMDization, loop unrolling, manual load/store reordering, alignment, and padding. We show implementations on the Cell processor that deliver performances between 8 and 14 Gbps per chip under realistic conditions corresponding to the tokenizing rules of Lucene (a popular search engine) operating on Wikipedia pages.

Our results are the first on high-performance regular expression matching on the Cell processor. They indicate the strong potential of this platform for text processing and information retrieval, and provide a reference point for contemporary and future regexp matching solutions. Also, our approach offers a comprehensive list of optimization techniques that promise to be beneficial on other multi-core architectures.

Acknowledgments

We thank Dr. Alexandre E. Eichenberger, Prof. Sally A. McKee, and Prof. Robert D. Cameron for their help.

8. REFERENCES

- [1] comScore Releases 2007 U.S. Internet Year in Review, Press Release, Jan. 2008.
- [2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [3] S. Antonatos, K. Anagnostakis, M. Polychronakis, and E. Markatos. Performance analysis of content matching intrusion detection systems. In *4th IEEE/IPSJ Symp. on Applications and the Internet (SAINT 2004)*, 2004.
- [4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [5] L. Bu and J. A. Chandy. A CAM-based keyword match processor architecture. *Microelectronics Jnl.*, 37(8):828–836, 2006.
- [6] P. Bumbulis and D. D. Cowan. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, March–December 1993.

- [7] R. D. Cameron. A Case Study in SIMD Text Processing with Parallel Bit Streams - UTF-8 to UTF-16 Transcoding. In *2008 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 91–98, Salt Lake City, Utah, Feb. 2008.
- [8] R. D. Cameron. Method and apparatus for processing character streams, U.S. Patent 7400271, July 2008.
- [9] C. Chang and R. Paige. From regular expressions to DFAs using compressed NFAs. In *CPM '92*, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Lecture Notes in Computer Science, No. 644*, pages 88–108. Springer-Verlag, 1992.
- [10] J. Degener. ANSI C grammar, lex specification, <http://www.lysator.liu.se/c/ANSI-C-grammar-l.html>.
- [11] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.
- [12] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency, Mar. 2008.
- [13] T. A. S. Foundation. Lucene, <http://lucene.apache.org>.
- [14] N. Goyal, J. Ormont, R. Smith, K. Sankaralingam, and C. Estan. Signature matching in network processing using SIMD/GPU architectures. Technical Report 1628, University of Wisconsin at Madison, Jan. 2008.
- [15] J. Grosch. Efficient generation of lexical analysers. *Software – Practice and Experience*, 19(11):1089–1103, Nov. 1989.
- [16] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.
- [17] H. P. Hofstee. Efficient Processor Architecture and the Cell Processor. In *Conf. High Performance Computing Architectures*, February 2005.
- [18] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'02)*, page 111, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] IDC Corporation. The expanding digital universe. *White Paper*, March 2007.
- [20] Intel Corp. Tera-scale research prototype – connecting 80 simple cores on a single test chip, October 2006.
- [21] Intel Corp. Intel SSE4 Programming Reference, Reference Number: D91561-001, Apr. 2007.
- [22] F. Iorio and J. V. Lunteren. Fast pattern matching on the Cell Broadband Engine. In *2008 Workshop on Cell Systems and Applications (WCSA), affiliated with the 2008 Intl. Symp. on Computer Architecture (ISCA'08)*, Beijing, China, June 2008.
- [23] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, pages 589–604, July/September 2005.
- [24] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3), May/June 2006.
- [25] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A high performance NIDS using FPGA-based regular expression matching. In *2007 ACM Symp. on Applied Computing (SAC '07)*, pages 1187–1191. ACM, 2007.
- [26] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *ACM Intl. Symp. on Field Programmable Gate Arrays (FPGA 2001)*, pages 87–93, 2001.
- [27] J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38, Napa, CA, USA, Apr. 2003.
- [28] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.
- [29] M. Nicola and J. John. XML parsing: A threat to database performance. In *CIKM*. ACM, 2003.
- [30] V. Paxson. flex – a fast lexical analyzer generator, 1988.
- [31] E. Perkins, M. Kostoulas, A. Heifets, M. Matsa, and N. Mendelsohn. Performance analysis of XML APIs. In *XML 2005 Conf. and Exposition*, Nov. 2005.
- [32] D. P. Scarpazza, O. Villa, and F. Petrini. Peak-performance DFA-based string matching on the Cell processor. In *Third Intl. Workshop on System Management Techniques, Processes, and Services (SMTPS), held in conjunction with IPDPS*, Mar. 2007.
- [33] D. P. Scarpazza, O. Villa, and F. Petrini. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor. In *22nd IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS'08)*, Miami, Florida, Apr. 2008.
- [34] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerma, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008*, pages 1–15, New York, NY, USA, 2008. ACM.
- [35] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion. In *13th Conf. on Field Programmable Logic and Applications (FPL'03)*, September 2003.
- [36] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching, 2004.
- [37] D. C. Suresh, Z. Guo, B. Buyukkurt, and W. A. Najjar. Automatic compilation framework for bloom filter based intrusion detection. In *Second Intl. Workshop on Reconfigurable Computing: Architectures and Applications (ARC'06)*, pages 413–418, 2006.
- [38] A. D. Thurston. Parsing computer languages with an automaton compiled from a single regular expression. In O. H. Ibarra and H.-C. Yen, editors, *CIAA*, volume 4094 of *Lecture Notes in Computer Science*, pages 285–286. Springer, 2006.
- [39] J. van Lunteren. High-performance pattern-matching for intrusion detection. In *25th IEEE Intl. Conf. on Computer Communications (INFOCOM 2006)*, pages 1–13, Apr. 2006.
- [40] O. Villa, D. Chavarria, and K. Maschhoff. Input-independent, scalable and fast string matching on the Cray XMT. In *23rd IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS'09)*, 2009.