Highly Compressed Multi-pattern String Matching on the Cell Broadband Engine

Xinyan Zha
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
Email: xzha@cise.ufl.edu

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

Sartaj Sahni
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
Email: sahni@cise.ufl.edu

***Abstract*—With its 9 cores per chip, the IBM Cell/Broadband Engine (Cell) can deliver an impressive amount of compute power and benefit the string-matching kernels of network security, business analytics and natural language processing applications. However, the available amount of main memory on the system limits the maximum size of the dictionary supported by the string matching solution.**

To counter that, we propose a technique that employs compressed Aho-Corasick automata to perform fast, exact multi-pattern string matching with very large dictionaries. Our technique achieves the remarkable compression factors of 1:34 and 1:58, respectively, on the memory representation of English-language dictionaries and random binary string dictionaries. We demonstrate a parallel implementation for the Cell processor that delivers a sustained throughput between 0.90 and 2.35 Gbps per Cell blade, while supporting dictionary sizes up to 9.2 Million average patterns per Gbyte of main memory, and exhibiting resilience to content-based attacks.

This high dictionary density enables natural language applications of an unprecedented scale to run on a single server blade.

I. Introduction

The evolution of "Web 2.0" applications and business analytics applications is showing a more and more prevalent production and use of unstructured data. For example, Natural Language Processing (NLP) applications can determine the language in which a document is written. E-mail web applications extract semantically tagged information (dates, places, delivery tracking numbers, etc.) from messages. Business analytics applications can automatically detect business events like the merger of two companies.

In these applications and many others, it is crucial to process huge amounts of sequential text to extract matches against a predetermined set of strings (the *dictionary*). Arguably, the most popular way to perform this exact, multi-pattern string matching task is the Aho-Corasick [1] (AC) algorithm. However, AC, especially in its optimized form based on a Deterministic Finite Automaton (DFA), is not space-efficient. In fact, the state-transition table that its DFAs use can be highly redundant. Uncompressed DFAs have a low transition cost (and therefore a high throughput) but also large footprint and, consequently, a low dictionary capacity per unit of memory. For example, a dictionary of 200,000 patterns with average length 15 bytes occupies 1 Gbyte of memory when encoded for an uncompressed AC DFA. Low space efficiency limits the algorithm's applicability to domains that require very large dictionaries like automatic language identification, which employ dictionaries with millions of entries, coming from hundreds of distinct natural languages.

In this paper, we address precisely this space inefficiency by exploring a variant of AC that employs compressed paths. Our algorithm is inspired by those proposed by Tuck et al. [12] and Zha and Sahni [19]. These algorithms are based on the Non-deterministic Finite Automaton (NFA) version of AC, and achieve significant memory reduction.

We choose an established multi-core architecture, the IBM Cell/Broadband Engine (Cell) for our work because it is a prominent architecture in the high-performance computing community, it has shown potential in string matching applications, and it presents software designers with non-trivial challenges that are representative of the next generations of multi-core architectures.

With our proposed algorithm, we achieve an average compression ratio of 1:34 for English words and 1:58 for random binary patterns. Our implementation provides a sustained throughput between 0.90 and 2.35 Gbps per Cell blade in different application scenarios, while supporting dictionary densities up to 9.26 million average patterns per Gbyte of main memory.

The remainder of this paper is organized as follows. Section II introduces the Cell architecture. Sections III and IV introduce the AC algorithm and the compression method of Tuck et al. Section V demonstrates a parallel, Cell-based implementation of our technique. Section VI discusses the experimental results. Section VII reviews the related work. Section VIII concludes the paper.

II. The Cell/Broadband Engine Architecture

The Cell processor [17] contains 9 heterogeneous cores on a silicon die. One of them is a traditional 64-bit processor with cache memories and 2-way simultaneous multi-threading, called Power Processor Element (PPE), and capable of running a full-featured operating system and traditional PowerPC applications. The other 8 cores are called Synergistic Processor Elements (SPEs). They have no caches, but rather

a small amount of scratch-pad memory (256 kbyte) that the programmer must manage explicitly, by issuing DMA transfer from and to the main memory. The cores are connected with each other via the Element Interconnect Bus (EIB), a fast double ring on-chip network.

The Cell delivers its best performance when the SPEs are kept highly utilized by streaming tasks that load data from main memory, process data locally and commit the results back to main memory. These tasks exhibit a regular, predictable memory access pattern that the programmer can exploit to implement double buffering, and overlap computation and data-transfer over time.

Achieving high performance on the Cell with non-streaming applications is all but trivial, and algorithms based on DFAs like ours are arguably the most difficult to port. In fact, these algorithms exhibit unpredictable memory access patterns and a complex latency interaction between compute code and data-transfer code. These circumstances make it difficult to determine what represents the critical path in the code, and how to optimize it.

III. The Aho-Corasick Algorithm

Aho-Corasick (AC) [1] is a multi-pattern matching algorithm, commonly employed in NIDS applications. There are two versions of it: a deterministic and a non-deterministic one. Both versions use finite state machines. The version we adopt is the one based on a Non-deterministic Finite Automaton (NFA).

In this version, states are connected by *success* and *failure* transitions. Each state has one outgoing failure transition and one or more success transitions. A success transition is labeled with a symbol from the accepted alphabet. Each state has a set of matches (from the dictionary) that are matched when the NFA transitions into that state as a result of a success transition.

The NFA is initialized in its start state, with its read head on the first symbol of the input text string S. At each step, the NFA performs a state transition examining the current input symbol in S. If the current state has a success transition labeled as the current input symbol, the NFA follows that transition, and the read head moves one symbol ahead over S. When no such success transition exists, the NFA follows the failure transition without advancing its read head.

Whenever the NFA lands into a state that has a non-empty match set, the automaton reports that all the strings in the match set have just been matched. In NIDS applications, this is usually associated with the detection of malicious signatures, and it triggers appropriate alerts.

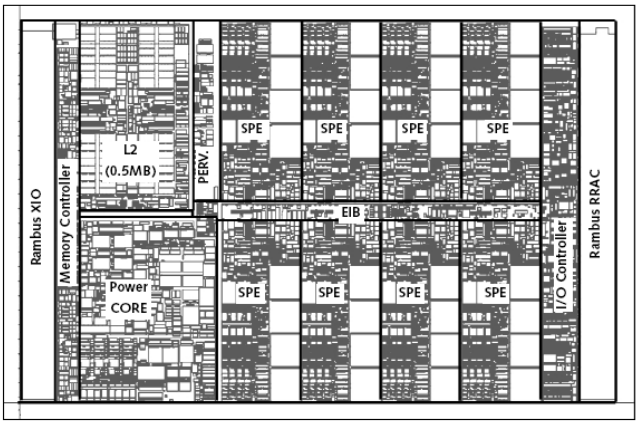

Fig. 1. Chip layout of the Cell/Broadband Engine Architecture.

IV. Tuck et al.'s compressed automaton

In this work, we focus on an adaptation of Tuck et al.'s [12] compressed NFA method. We choose to do so despite the fact that Zha and Sahni's [19] technique is 31% more space-efficient and it requires 90% fewer popcount[1], additions, because Tuck et al.'s method has a simpler control flow, and therefore it promises higher performance on an architecture like the Cell's SPEs, where branches are expensive. In detail, Zha and Sahni's compressed automaton uses three node types (bitmap, low degree, path compressed) whereas Tuck et al.'s uses only two. We expect that a port of Zha and Sahni's method will incur more frequent branch miss penalties. An optimized port of Zha and Sahni's automaton is expected to pose a harder design challenge and is left for future research.

We assume that the alphabet size is 256 (e.g., extended ASCII characters). A naïve way to store an AC NFA, for a given dictionary D, is to represent each state with a node having 256 success pointers, one failure pointer, and a list of matched strings, as follows:

1) *Success[0...255]* is an array where element *Success[i]* points to the state to transition to, when the input is with ASCII code i. *Success[i]* is null when such success transition is not defined.
2) *MatchList* is the list of strings matched when this state is reached via a success pointer.
3) *Failure* is a pointer to the state to transition, when no success transition matches the current input.

Assuming 32-bit pointers, this representation occupies 1032 bytes per node (including the MatchList pointer but not the size of the list itself). This representation can be very redundant: a large fraction of the *Success* array can be occupied by null or repeated values. We employ bitmaps and path compression to compact this representation, reducing the footprint of a node to 52 bytes.

Bitmap Compression. Briefly said, bitmap compression replaces each 1032-byte node of a NFA with a 45-byte node. Of these 45 bytes, 1 is for the node type and failure pointer offset, 8 are used for the failure and rule list pointers. 32 bytes contain a 256-bit bitmap with the property that bit i of this map is 1 iff *Success[i]* $\neq$ null. The nodes corresponding to the non-null success pointers are stored in contiguous memory and a pointer *firstChild* to the first of these stored in the 45-byte node.

[1]A population count (popcount) operation determines the number of '1' bits in a bit field.

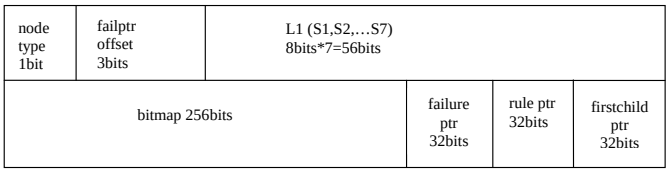

Fig. 2. Bitmap node layout.

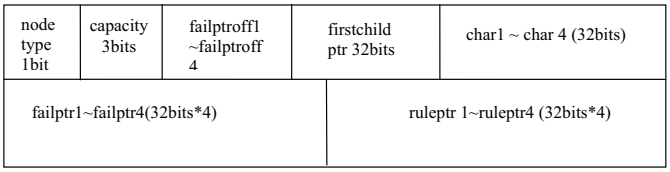

Fig. 3. Path-compressed node layout with packing factor equal to four.

A state transition for an input i works as follow. We first determine whether *Success[i]* is null by examining bit i of the bitmap. If this bit is zero, the next state is pointed by the failure pointer. Otherwise, we determine the *popcount* of all the bits in the bitmap having position $< i$. Then we transition to the state pointed by *firstChild*, offset by a state record size (45 bytes) as many times as the popcount.

To reduce the cost of popcount, Tuck et al. propose the use of precomputed summaries, that give the popcount for the first $32 \cdot j$, $1 \leq j < 8$ bits of the bitmap. Each summary is 8 bits long, and 7 summaries are needed. The size of a bit-compressed node with summaries is therefore 52 bytes.

Path Compression. Path compression is similar to end-node optimization [4], [7]. An *end-node sequence* is a sequence of states at the bottom of the automaton (the start state is at the top of the automaton) that comprises states having a single non-null success transition (except for the last state in the sequence, which has no non-null success transition). States in the same end-node sequence are packed together into one or more *path-compressed* nodes.

For each state s_i packed into a path-compressed node, we store one success 1-byte input character, the failure pointer and the match list.

Since several automaton states are packed into a single compressed node, a 32-bit failure pointer is not sufficient to address packed states within a compressed node. With an additional 3-bit offset, we handle nodes with capacity $c \leq 8$. Now, $\lceil 3c/8 \rceil$ bytes are needed for the offsets. A path-compressed node with capacity c needs $9c + \lceil 3c/8 \rceil$ bytes for the state information. 4 more bytes are needed to pointer to the next node (if any) in the sequence of path-compressed nodes. One more byte identifies the node type (bitmap and compressed) and its size (number of packed states). So, the size of a compressed node is $9c + \lceil 3c/8 \rceil + 5$ bytes.

Figure 3 shows a path-compressed node.

V. Cell-oriented algorithm design

This section describes the implementation choices we made to adapt our AC NFA algorithm to the Cell processor.

Dictionary	Original AC Size	Packing Factor	Compressed AC Size	Compression Ratio
(1) English	48.86 Mbytes	4	1.41 Mbytes	34.78
		8	1.83 Mbytes	26.65
(2) Binary	52.37 Mbytes	4	0.90 Mbytes	58.07
		8	0.85 Mbytes	61.53
		12	0.86 Mbytes	60.83

Fig. 4. Compression Ratios obtained by our technique on two sample dictionaries of comparable uncompressed size. Dictionary (1) contains the 20,000 most common words in the English language. Dictionary (2) contains 8,000 random binary patterns of same average length as in Dictionary (1).

To compute popcounts efficiently, we employ the CNTB and SUMB instructions (available at the C level via the spu_cntb() and spu_sumb() intrinsics). Also, we employ vector comparison instructions to get the longest match between the input and compressed paths.

For alignment reasons, we only consider path-compressed nodes with packing factors (c) of 4, 8 and 12. Figure 4 shows the corresponding compression ratio. Note that 4 is the best choice for the English dictionary and 8 is best for random binary patterns. For simplicity, we consider a packing factor of 4 in the experiments that follow. The difference in compression gain obtained with a packing factor of 8 is not significant enough to justify the increase in algorithm complexity. By using this compressed automata, we can compress dictionaries with an average compression ratio of 1:34 for English dictionaries and 1:58 for random binary patterns.

We now describe the optimizations we employed to map our compressed AC algorithm to Cell architecture and their impact. Results were obtained with the IBM Cell SDK 3.0 on IBM QS22 blades. Figure 5 shows the impact of the optimization steps on the performance and quality of code. We started from a naïve compressed AC implementation and we applied branch hinting, branch replacement with conditional expressions, vertical unrolling, data structure realignment, branch removal, arithmetic strength reduction and horizontal unrolling. The aggregate effect of these optimizations is to increase the throughput (by reducing the number of cycles absorbed per character), reducing the cycles per instruction (CPI), reducing stalls and increasing the dual issue rate (i.e. clock cycles in which both pipeline in an SPE issue a new instruction).

These techniques help to decrease the CPI, the branch stall cycles rate, the dependency stall cycles. They also decrease the single instruction issue rate and increase the dual instruction issue rate. Overall, the optimization effort results in a 16 to 25 times throughput speedup against the unoptimized PPE baseline implementation.

A. Step (1): Branch replacement and hinting

Whenever possible, we restructure the control flow so to replace `if` statements with conditional expressions. We inspect the assembly output to make sure that the compiler renders conditional expression with select bits instructions rather than branches.

Optimization Step	Typical Throughput (Gbps)	Cycles/ char (1 SPE)	CPI	Insts per char	Used Regs	NOP Rate	Branch Stall Rate	Dep. Stall Rate	Single Issue Rate	Dual Issue Rate	Speedup
Scenario A: Full Text Search											
(0) Unoptimized PPE baseline implementation	0.082	—	—	—	—	—	—	—	—	—	=1.0×
(1) Naïve implementation on 8 SPEs	1.440	142.2	1.42	99.9	81	2.3%	10.4%	27.9%	47.3%	11.5%	17.1×
(2) 1 Engine, branch hints, conditional expr.	1.518	134.9	1.46	92.1	82	1.5%	23.7%	25.2%	38.0%	11.0%	18.5×
(3) 4 Engines, loops unrolling, alignment	1.616	126.8	1.46	86.7	92	1.9%	19.5%	29.0%	37.7%	11.7%	19.7×
(4) 1 Engine, branch removal	1.768	115.8	0.94	122.7	99	2.3%	2.7%	26.1%	44.9%	24.0%	21.5×
(5) 1 Engine, cheaper pointer arithmetics	1.771	115.6	0.97	118.9	99	1.8%	1.8%	26.0%	46.3%	23.4%	21.6×
(6) 4 Engines, horizontal unrolling	2.058	99.5	0.83	120.3	125	1.7%	3.2%	17.4%	43.8%	33.7%	25.1×
Scenario B: Network Content Monitoring											
(0) Unoptimized PPE baseline implementation	0.082	—	—	—	—	—	—	—	—	—	=1.0×
(1) Naïve implementation on 8 SPEs	0.655	312.8	1.23	307.2	84	1.8 %	15.0%	20.7%	52.0%	10.0%	8.0×
(2) 1 Engine, branch hints, conditional expr.	0.882	232.3	1.18	231.2	83	2.4%	8.6%	27.6%	48.7%	12.6%	10.8×
(3) 4 Engines, loops unrolling, alignment	0.992	206.4	1.16	225.18	86	2.7%	20.0%	20.0%	40.9%	16.3%	12.1×
(4) 1 Engine, branch removal	1.018	201.1	1.00	163.8	128	1.8%	1.0%	27.8%	48.5%	20.7%	12.4×
(5) 1 Engine, cheaper pointer arithmetics	1.464	139.9	0.92	120.22	128	2.3%	1.4%	23.2%	44.6%	28.4%	17.9×
(6) 4 Engines, horizontal unrolling	2.071	98.9	0.92	107.04	128	1.9%	2.4%	22.6%	44.9%	27.9%	25.3×
Scenario C: Network Intrusion Detection											
(0) Unoptimized PPE baseline implementation	0.082	—	—	—	—	—	—	—	—	—	=1.0×
(1) Naïve implementation on 8 SPEs	0.512	388.0	1.43	387.10	84	2.5%	15.4%	27.7%	49.1%	5.3%	6.2×
(2) 1 Engine, branch hints, conditional expr.	0.576	355.3	1.14	354.41	86	2.9%	14.1%	21.8%	43.4%	17.8%	7.0×
(3) 4 Engines, loops unrolling, alignment	0.636	321.9	1.28	343.40	83	1.7%	16.2%	25.3%	44.8%	11.6%	7.8×
(4) 1 Engine, branch removal	0.650	315.3	1.00	281.75	128	1.8%	1.2 %	27.6%	48.5%	20.8%	7.9×
(5) 1 Engine, cheaper pointer arithmetics	0.801	255.7	0.94	199.77	128	2.2%	3.4%	22.9%	42.9%	28.4%	9.8×
(6) 4 Engines, horizontal unrolling	1.318	155.3	0.92	165.73	128	1.9%	2.5%	22.6%	44.7%	28.0%	16.1×
Scenario D: Network Intrusion Detection											
(0) Unoptimized PPE baseline implementation	0.092	—	—	—	—	—	—	—	—	—	=1.0×
(1) Naïve implementation on 8 SPEs	0.451	453.6	1.30	441.3	84	2.1%	12.8%	26.0%	46.5%	12.2%	4.9×
(2) 1 Engine, branch hints, conditional expr.	0.560	365.4	1.16	314.57	86	2.8%	14.8%	22.0%	42.7%	17.4%	6.1×
(3) 4 Engines, loops unrolling, alignment	0.703	291.5	1.28	227.19	83	1.7%	16.4%	25.4%	44.5%	11.7%	7.6×
(4) 1 Engine, branch removal	1.588	129.0	1.06	121.14	128	1.5%	7.6%	24.7%	45.4%	19.9%	17.3×
(5) 1 Engine, cheaper pointer arithmetics	1.694	120.9	0.96	126.50	128	2.3%	1.2%	23.1%	44.9%	28.3%	18.4×
(6) 4 Engines, horizontal unrolling	2.204	92.9	0.92	100.65	128	1.7%	7.9%	20.6%	41.8%	27.3%	24.0×

Fig. 5. The impact of the optimization steps on the performance of our compressed AC NFA algorithm when evaluated in the four application scenarios presented in Section VI. Packing factor for compressed-path node is 4.

A major `if` statement in the compressed AC NFA kernel does not benefit from this strategy, i.e., the one that branches depending on whether the node type is bitmap or path-compressed. The two branches are too different to reduce to conditional expressions. We reduce the misprediction penalty associated with this branch by hinting to mark the bitmap case as the more likely, as suggested by our profiling on realistic data.

B. Step (2): Loop Unrolling, Data alignment

We apply unrolling to a few relevant bounded innermost loops, and we apply data structure alignment. Our algorithm consists of two major parts: a compute part and a memory access part. Since the compressed AC is too large to fit entirely in the SPEs' local stores, we store it in main memory.

We safely ignore the impact of memory accesses required to load input text from main memory to local store and write back matches in the opposite direction. In fact, we implement both transfers in a double-buffered way, overlapping computation and data transfer in time.

When a single instance of an AC NFA runs, it computes its next-iteration node pointer and then fetches this node via a DMA transfer from main memory. DMA transfers have round-trip time of hundreds of clock cycles. To utilize these cycles, we run multiple concurrent automata, each checking matches in different segments of the input, unrolling their code together *vertically*. Multiple automata can pipeline memory accesses, overlapping the DMA transfer delays.

Figure 6 illustrates how different vertical unrolling factors affect the performance. We choose vertical unrolling factor 8 in our implementation as it gives the minimal DMA transfer delay.

We also performed an experiment to find out the best DMA transfer size to make full use of the bandwidth and minimize the DMA transfer delay. Figure 9 shows the optimal transfer size is 64 Bytes over the eight SPUs.

C. Step (3): Branch removal, select-bits intrinsics

After replacing if statements with conditional expressions, the branch miss stalls still account for about one fifth of the total compute cycles.

We use IBM asmvis [18] to inspect the static timing analysis of our code at the assembly level. It helps us to get a clear view of what the compiler is doing, instruction by instruction. The inspection reveals that conditional expressions are often translated by the compiler as expensive branch instructions. In this case, our code still suffers from expensive branch miss penalties, which can cost as much as 26 clock cycles

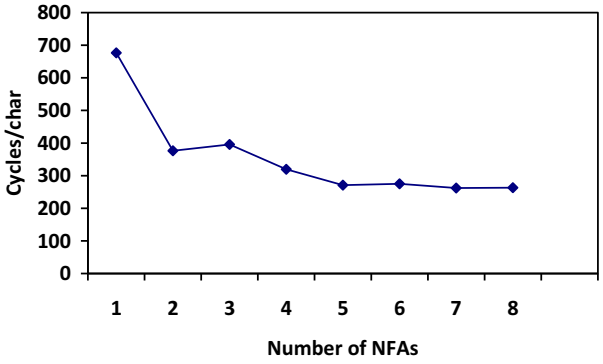

Fig. 6. The number of cycles processed per character with different vertical unrolling factors. (Full-text search scenario).

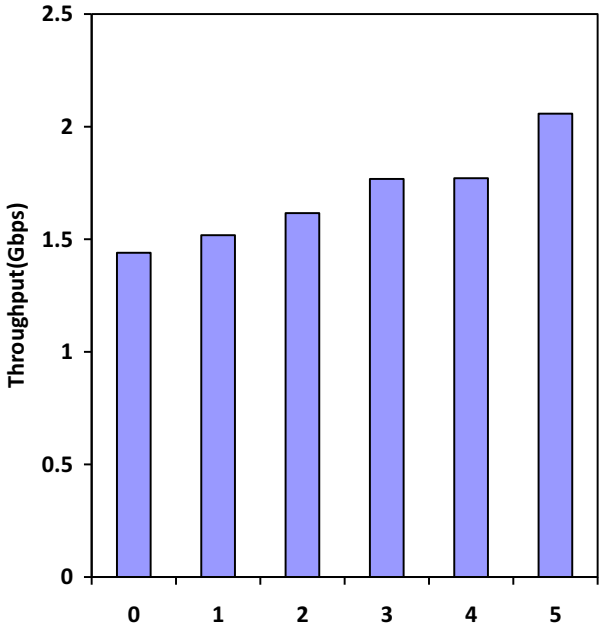

Fig. 7. How the throughput grows with each optimization step. (Full-text search scenario).

each. To eliminate branches, we manually replace conditional expressions with the spu_sel intrinsic [14]. The basic idea is to compute the two possible results for both branches and select one of the results using a select bit instruction. For example, the transformation reduces branch miss stalls from 19.5% to 2.7% of the cycle count for the full-text search scenario.

D. Step (4): Strength reduction

We manually apply operator strength reduction (i.e., replacing multiplication and divisions with shifts and additions) where the compiler did not. In addition, we use cheap pointer arithmetic to load four adjacent integer elements into a 128 bit vector. This reduces the load overhead. e.g. Manual strength reduction reduces the overall clock cycles 3% for the full text search scenario.

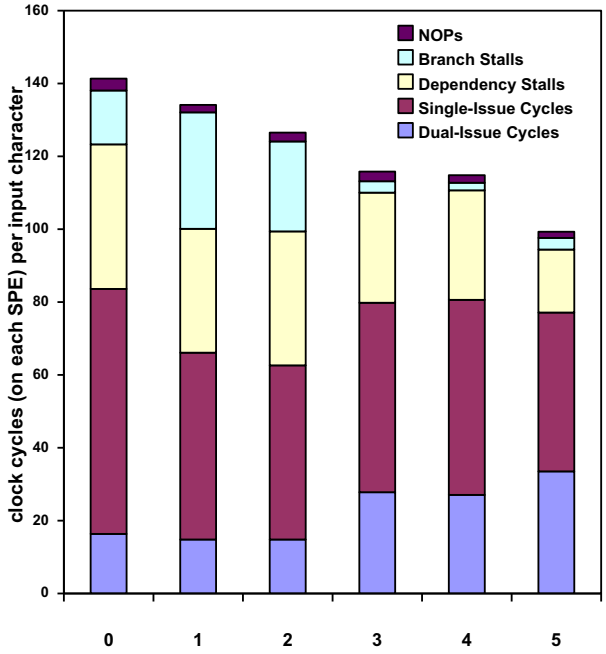

Fig. 8. Utilization of clock cycles following each optimization step (Full-text search scenario).

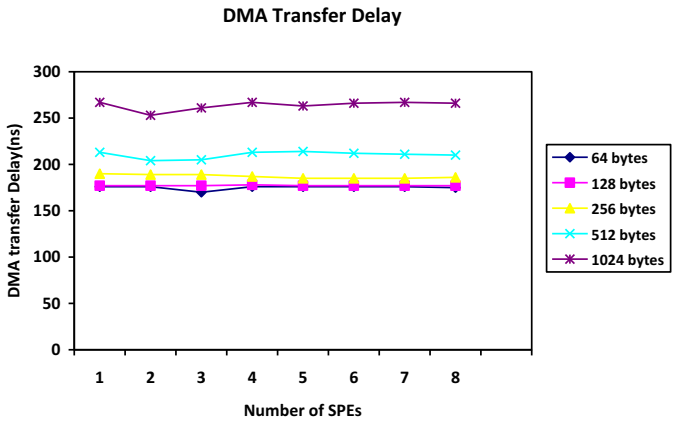

Fig. 9. DMA inter-arrival transfer delay from main memory to local store when 8 SPEs are used concurrently.

E. Step (5): Horizontal unrolling

After Steps 1–4, dependency stalls occupy about 25% of the computation time. Within the NFA compute code, one branch handles bitmap nodes, while the other one handles path-compressed nodes. In the code of both cases, there are frequent read-after-write data dependencies.

To reduce the dependency stalls, we interleave the codes of multiple, distinct automata; we call this operation *horizontal unrolling*. These multiple automata process independent input streams against the same dictionary. They have distinct states and input/output buffers, and they require multiple, distinct DMA operations to perform the associated streamed double buffering.

The horizontal unroll factor must be chosen accurately to reflect the trade-off between the decreased dependency stalls and the potentially increased branch stalls. Our experiments show that unrolling 2 NFAs achieves the highest performance improvement, 10%. For example, for the full text search scenario, dependency stalls decreased from 26.0% to 17.4%, while branch stalls increase from 1.8% to 3.2%.

VI. Experimental Results

In this section, we benchmark our software design in a set of representative scenarios.

We use two dictionaries to generate compressed AC automata: Dictionary 1 contains the 20,000 most common words in the English language, while Dictionary 2 contains 8000 random binary patterns. We benchmark the algorithm on three input files: the King James Bible, a tcpdump stream of captured network traffic and a randomly generated binary file.

Figure 10 shows the aggregate throughput of our algorithm on a dual-chip blade (16 SPEs) in the six scenarios described below. Scenario A (Dictionary 1 against the Bible) is representative of full-text search systems. Scenario B (Dictionary 1 against the network dump) is representative of content monitoring systems. Scenario C (Dictionary 2 against the network dump) is representative of Network Intrusion Detection Systems (NIDSs). Scenario D (Dictionary 2 against binary patterns) is representative of anti-virus scanners.

The last two scenarios in the figure are representative of systems (with Dictionary 1 and 2, respectively) under a malicious, content-based attack. In fact, a system whose performance degrades dramatically when the input exhibits frequent matches with the dictionary is subject to content-based attacks. An attacker that gains partial or full knowledge of the dictionary could provide the system with traffic specifically designed to overflow it. In scenarios five and six we provide our system with inputs entirely composed of words from the dictionary. Our experiments show a desirable property of our algorithm: its performance actually *increases* in case of frequent hitting.

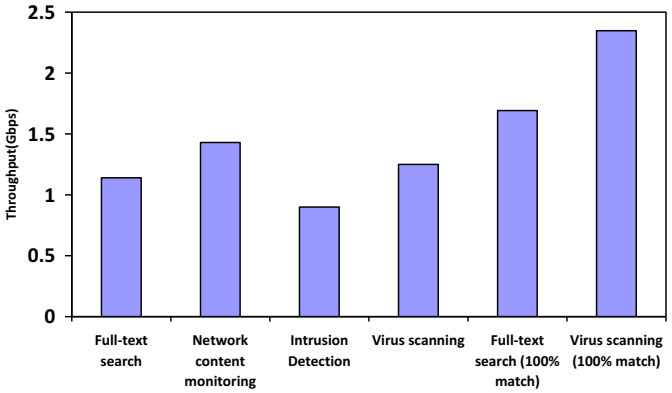

Fig. 10. Aggregate throughput of our algorithm on a IBM QS22 blade (16 SPEs).

Scenario	Throughput (Gbps)
Full-text search	1.14
Network content monitoring	1.43
Network intrusion detection	0.90
Anti-Virus scanning	1.25
Full-text search (100% match)	1.69
Anti-Virus scanning (100% match)	2.35

Fig. 11. Aggregate throughput on a IBM Cell chip with 8 SPUs (Gbps).

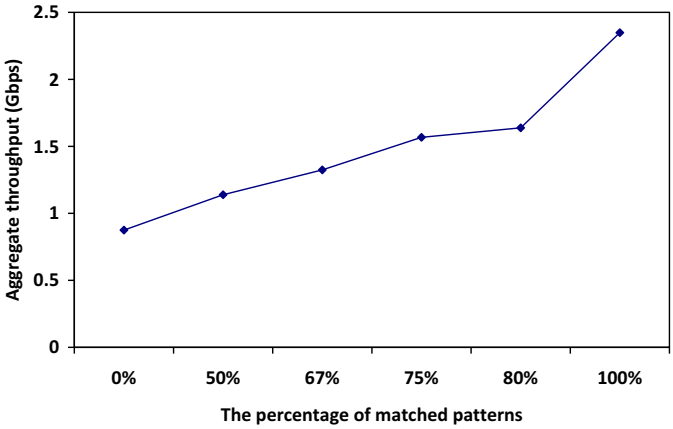

Fig. 12. How the percentage of matched patterns affects the aggregate throughput. The input here is on English input data, with English Dictionary.

The reason is that our NFA spends a similar amount of time to process a bitmap or a path-compressed node. For this reason, a mismatch takes a comparable amount of time to the match of an entire path.

For this reason, the cycles spent per input character decrease when more input characters match the dictionary. Path-compressed nodes pack as many as 4 or 8 original AC nodes, and allow multi-character match at one time. Figure 12 shows how the percentage of matched patterns affects the aggregate throughput on the IBM cell blade with 16 SPUs for the virus scanning scenario. As the percentage of the matched patterns increases, the aggregate throughput increases as well.

We explore the trade-offs between the AC compression ratio and the throughput in a Pareto space. We choose the English dictionary as the compression object and choose packing factors of 4, 8, 12 for path compressed nodes. As shown in Figure 13, the compression ratio decreases with increase in the packing factor. However, the throughput is better with a packing factor of 8 than with one of 4.

The reason for that is the input data is a English input which has 100% match against the dictionary. So instead of matching 4 nodes in the path compressed node at one time, matching 8 nodes at one time gives better performance. However, a packing factor of 12 has some throughput degradation compared to a packing factor of 8. One conclusion we draw from this Pareto chart is the compression ratio affects the throughput, in order to get a better compression ratio, we have to sacrifice throughput.

VII. Related Work

Snort [9] and Bro [8], [3], [5], [10], [2] are two of the more popular public domain Network Intrusion Detection

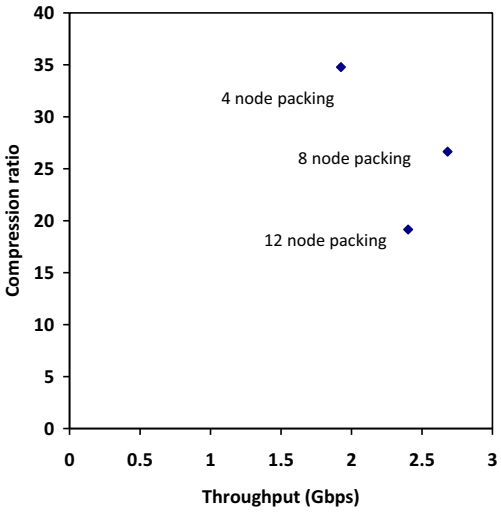

Fig. 13. The trade-off between the compression ratio and the throughput in a Pareto space.The input here is on English input data, with English Dictionary.

Systems (NIDSs). The current implementation of Snort uses the optimized version of the AC automaton [1]. Snort also uses SFK search and the Wu-Manber [13] multi-string search algorithm.

To reduce the memory requirement of the AC automaton, Tuck et al. [12] have proposed starting with the non-deterministic AC automaton and using bitmaps and path compression.

In the network security domain, bitmaps have been used also in the tree bitmap scheme [4] and in shape shifting and hybrid shape-shifting *tries*[2] [11], [7]. Path compression has been used in several IP address lookup structures including tree bitmap [4] and hybrid shape-shifting tries [7]. These compression methods reduce the memory required to about 1/30—1/50 of that required by an AC DFA or a Wu-Manber structure, and to slightly less than what required by SFK search [12]. However, lookups on path-compressed data require more computation at search time, e.g., more additions at each node to compute popcounts, thus requiring hardware support to achieve competitive performance.

Zha and Sahni [19] have suggested a compressed AC trie inspired by the work of Tuck et al. [12]: they use bitmaps with multiple levels of summaries, as well as an aggressive path compaction. Zha and Sahni's technique requires 90% fewer additions to compute popcounts than Tuck et al [12]'s, and occupies 24%—31% less memory. Scarpazza et al. [15] propose a memory-based implementation of the deterministic AC algorithm that is capable of supporting dictionaries as large as the available main memory, and achieves a search performance of 1.5–2.2 Gbps per Cell chip. Scarpazza et al. [16] also propose regular expression matching against small rule sets (which suits the needs of the search engine tokenizers) delivering 8-14 Gbps per Cell chip.

[2]A trie is a tree-based data structure frequently used represents dictionaries and associative arrays that have strings as a key.

VIII. Conclusions

We present an optimized software design that exploits compressed AC automata to perform high-throughput multi-pattern string matching on the IBM Cell Broadband Engine.

We have presented a detailed overview of the algorithmic-level and implementation-level optimizations that we applied in order to improve the algorithm's performance.

Our solution delivers impressive compression ratios in experiment scenarios representative of natural language processing and network security applications: respectively, 1:34 on dictionaries containing English words, and 1:58 on dictionaries containing random binary patterns.

Also, our solution provide a remarkable throughput between 0.90 and 2.35 Gbps per Cell blade, depending on the statistical properties of dictionary and input.

References

[1] A. Aho and M. Corasick, Efficient string matching: An aid to bibliographic search, CACM, 18, 6, 1975, 333-340.
[2] H. Dreger, A. Feldmann, M. Mai, V. Paxson and R. Sommer, Dynamic application-layer protocol analysis for network intrusion detection, *USENIX Security Symposium*, 2006.
[3] H. Dreger, C. Kreibach, V. Paxson, and R. Sommer, Enhancing the accuracy of network-based intrusion detection with host-based context, *DIMVA*, 2005.
[4] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, *Computer Communication Review*, 34(2): 97-122, 2004.
[5] J. Gonzalez and V. Paxson, Enhancing network intrusion detection with integrated sampling and filtering, *RAID*, 2006.
[6] J. Lockwood, C. Neely, and C. Zuver, An extensible system-on-programmable-chip, content-aware Internet firewall. *In Field Programmable Logic and Applications*, 2003.
[7] W. Lu and S. Sahni, Succinct representation of static packet classifiers, *IEEE Symposium on Computers and Communications*, 2007.
[8] V. Paxson, Bro: A system for detecting network intruders in real-time, *Computer Networks*, 31, 1999, 2435–2463.
[9] Snort users manual 2.6.0, 2006.
[10] R. Sommer and V. Paxson, Exploiting independent state for network intrusion detection, *ACSAC*, 2005.
[11] H. Song, J. Turner, and J. Lockwood, Shape shifting tries for faster IP route lookup, *ICNP*, 2005.
[12] N. Tuck, T. Sherwood, B. Calder and G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, *INFOCOM*, 2004.
[13] S. Wu and U. Manber, Agrep–a fast algorithm for multi-pattern searching, Technical Report, Department of Computer Science, University of Arizona, 1994.
[14] D. Brokenshire, Maximizing the power of the cell Broadband Engine processor: 25 tips to optimal application performance *Technique Report, IBM STI Design Center 2006.*
[15] D. Scarpazza, O.Villa, F.Petrini, Accelerating Real-Time String Searching with Multicore Processors, *IEEE Computer Society 2008.*
[16] D. Scarpazza, G. Russell, High-performance Regular Expression Scanning on the Cell/B.E. processor. *23rd International Conference on Supercomputing 2009.*
[17] Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell
[18] IBM Assembly Visualizer for Cell Broadband Engine.http://www.alphaworks.ibm.com/tech/asmvis
[19] X. Zha and S. Sahni, Highly compressed Aho-Corasick automata for efficient intrusion detection. *IEEE Symposium on Computers and Communications 2008.*