RC24488 (W0802-026) February 6, 2008
Computer Science

IBM Research Report

Too Many Words, Too Little Time: Accelerating Real-Time Keyword Scanning with Multi-Core Processors

Oreste Villa
Politecnico di Milano
Dipartimento di Elettronica e Informazione,
Via Ponzio 34/5
20133 Milano, Italy
and
Pacific Northwest National Laboratory
Computational & Information Science Directorate
MSIN K7-90
P.O. Box 999
Richland, WA 99352 USA

Daniele Paolo Scarpazza, Fabrizio Petrini
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Too Many Words, Too Little Time: Accelerating Real-Time Keyword Scanning with Multi-Core Processors

Oreste Villa [a,b], Daniele Paolo Scarpazza [c], Fabrizio Petrini [c,*]

[a] *Politecnico di Milano, Dipartimento di Elettronica e Informazione, Via Ponzio 34/5, 20133 Milano, Italy*

[b] *Pacific Northwest National Laboratory, Computational & Information Science Directorate, MSIN K7-90, P.O. Box 999, Richland WA 99352, USA*

[c] *IBM T.J. Watson Research Center, 1101 Kitchawan Road, Route 134, Yorktown Heights NY 10598, USA*

Abstract

Our digital universe is growing, creating exploding amounts of data which need to be searched, protected and filtered. String searching is at the core of the tools we use to curb this explosion, such as search engines, network intrusion detection systems, spam filters, and anti-virus programs. But as communication speed grows, our capability to perform string searching in real-time seems to fall behind. Multi-core architectures promise enough computational power to cope with the incoming challenge, but it is still unclear which algorithms and programming models to use to unleash this power.

We have parallelized a popular string searching algorithm, Aho-Corasick, on the IBM Cell/B.E. processor, with the goal of performing exact string matching against large dictionaries. In this article we propose a novel approach to fully exploit the DMA-based communication mechanisms of the Cell/B.E. to provide an unprecedented level of aggregate performance with irregular access patterns.

We have discovered that memory congestion plays a crucial role in determining the performance of this algorithm. We discuss three aspects of congestion: memory pressure, layout issues and hot spots, and we present a collection of algorithmic solutions to alleviate these problems and achieve quasi-optimal performance.

The implementation of our algorithm provides a worst-case throughput of 2.5 Gbps, and a typical throughput between 3.3 and 4.4 Gbps, measured on realistic scenarios with a two-processor Cell/B.E. system.

* Corresponding author.

Email addresses: ovilla@elet.polimi.it (Oreste Villa), dpscarpazza@us.ibm.com (Daniele Paolo Scarpazza), fpetrin@us.ibm.com (Fabrizio Petrini).

1 Introduction

We are experiencing an explosion in the amount of digital data we produce and exchange [9]. Accessing, searching and protecting our expanding digital universe against virii, malware, spyware, spam and deliberate intrusion attempts are becoming key challenges. Let us consider the case of Network Intrusion Detection Systems (NIDS): since undesired traffic cannot be filtered anymore on mere header information (because threats are often targeting the application layer), *deep packet inspection* is needed, where the payload of the incoming traffic is checked against a large database of threat signatures. Deep packet inspection is becoming more and more difficult to perform in real time without impacting the bandwidth and the latency of the communications under scrutiny, because of the increasing number of threats and the increasing link speeds (e.g., 10 Gbps Ethernet and beyond).

An important class of algorithms which addresses this need to search and filter information is string matching. String matching is the core of search engines, intrusion detection systems, virus scanners, spam filters and content monitoring filters. Fast string searching implementations have been traditionally based on specialized hardware like FPGAs and ASIPs. The advent of multi-core architectures, such as the Cell/B.E. processor, is adding an important player to the game.

Previous work in keyword scanning has been focused on speed: achieving the highest processing rate, typically with a small dictionary, in the order of a few thousands keywords. More recently, a second dimension is gaining importance: dictionary size. Implementations with dictionaries of hundreds of thousands –or even millions, of patterns are beyond the current state of the art. A major challenge for the scientific community is to seamlessly extend the scanning speed already obtained with small dictionaries to much larger data sets.

Figure 1 provides a visual, and admittedly cursory, overview of previous related work [17,6,16,2,15]. In [15], we explored the upper left corner of this bi-dimensional space, presenting a parallel algorithm for the Cell/B.E. based on a deterministic finite-state automaton which provides a search throughput in excess of 5 Gbps per Synergistic Processing Element (SPE). All the 8 SPEs in a Cell/B.E. jointly deliver 40 Gbps with a dictionary of about 200 patterns, or a throughput of 5 Gbps with a larger dictionary (1,500 patterns). While the throughput of this solution is very high, it can only handle a small dictionary, which makes it unsuitable for many applications.

In this work, we rather focus on the bottom right corner of Figure 1, the large dictionaries, by using the entire available main memory (1 Gbyte in our experimental setup) to store the dictionary. Since accessing main memory requires higher latency than the local store of each SPE, we devote great attention to minimizing and *orchestrating* the memory traffic.

We present an Aho-Corasick–based algorithm to perform string searching against large dictionaries which maps efficiently on the Cell/B.E. architecture. This paper provides three primary contributions.

(1) It demonstrates that multi-core processors are a viable alternative to special-purpose solutions or FPGAs. In our implementation we can achieve performance comparable to other results

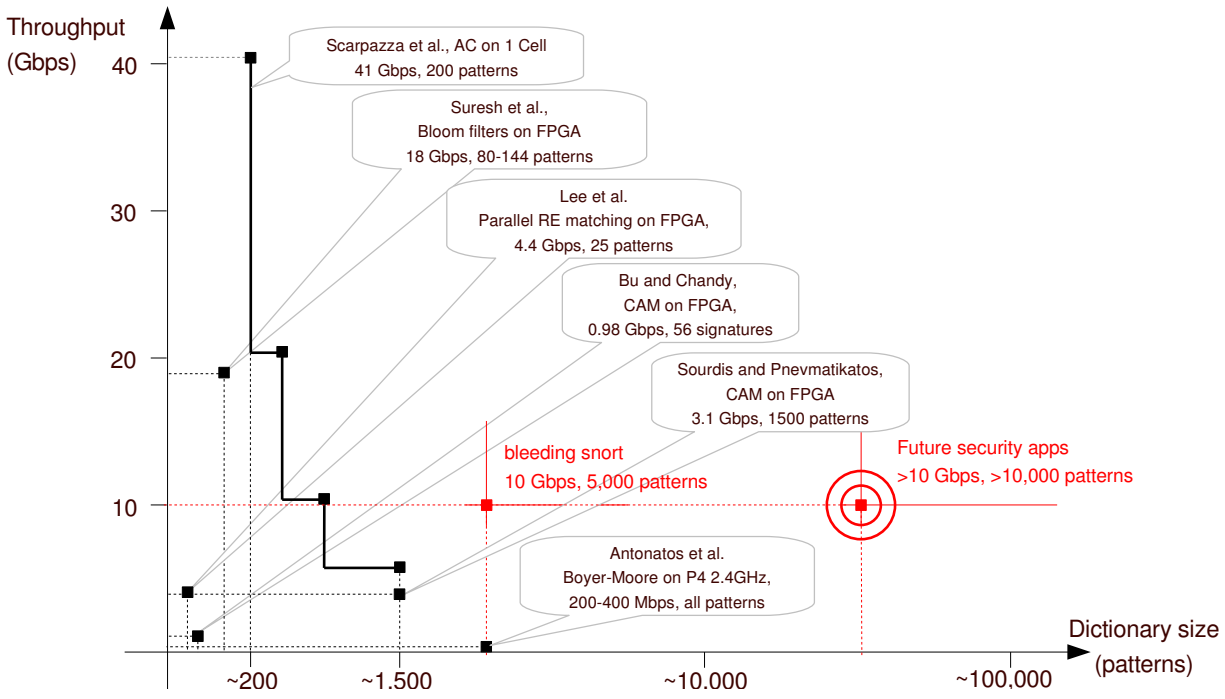

Fig. 1. Taxonomy of previous work on deep packet inspection. Out contribution attacks the lower right corner of this two-dimensional space.

---

presented in the literature with small data dictionaries, and take advantage of the sophisticated memory sub-system of the Cell/B.E. processor to effectively handle very large dictionaries.

(2) The proposed parallelization technique of the Aho-Corasick algorithm orchestrates the main memory requests to achieve high throughput and low response time (which we call memory gap), to feed several independent deterministic finite automata. We believe that this technique can be generalized to other algorithms and application domains, to handle large data sets.

(3) Another important contribution is an in-depth analysis of the performance bottleneck of the algorithm: memory congestion. Whereas the evaluation with synthetic memory traffic provides almost optimal performance, the actual implementation of the algorithm experiences several significant degradation factors. We identify three aspects of memory congestion (memory pressure, layout issues and hot spots) and discuss quantitatively their impact on performance. We discuss algorithmic solutions to alleviate these phenomena and achieve quasi-optimal performance. To the best of our knowledge, there are no compilers, libraries or development frameworks for the Cell/B.E. architecture which are aware of these phenomena or employ the techniques we propose to counter them.

The remainder of this paper is organized as follows. Section 2 introduces the basics of Aho-Corasick, the algorithm we employ. Section 3 presents our parallelization strategy and determines its theoretical peak performance. Section 4 presents the impact of memory congestion and Section 5 discusses the techniques we employ to counter it. Section 6 presents our experimental results. Concluding remarks are presented in Section 8.

2 The Aho-Corasick Algorithm

The Aho-Corasick algorithm scans an input text and detects occurrences of each of the patterns of a given dictionary, including partially and completely overlapping occurrences. We call *pattern* a finite sequence of symbols from an alphabet, and *dictionary* a set of patterns $P = \{p_1, p_2, ..., p_k\}$. The algorithm constructs an automaton on the basis of the dictionary. The automaton takes as an input a given text and enters a final state every time a match is encountered. Each final state is associated with the set of matching patterns. There are two variants of the algorithm, called AC-fail and AC-opt. We first introduce AC-fail, and present AC-opt as an improvement of it.

The AC-fail automaton is based on the keyword tree (a.k.a. a *trie*) of the given dictionary. For each pattern, there is a path in the trie which starts from the root node and whose edges are labelled as the symbols in the pattern. Edges leaving a node have distinct labels. Figure 2 shows the trie corresponding to the dictionary {the, that, math}. We call *label* $L(v)$ of a node v the concatenation of the edge labels encountered on the path from the root to v. For each pattern $p_i \in P$ there is a node v with $L(v) = p_i$ (gray nodes in the figure), and the label $L(v)$ of any leaf v equals some $p_i \in P$.

A trie can be used to determine whether a given string belongs in the dictionary or not, as follows. To look up a string s, start at the root node and follow the path labeled as s, as long as possible. If the path leads to a node with an identifier i, then the string belongs in the dictionary, and it is pattern p_i.

The trie does not match multiple, possibly overlapping, occurrences. The AC-fail algorithm serves that purpose. AC-fail employs an automaton which is improperly called a Non-deterministic Finite Automaton (NFA) as discussed below. The NFA is derived from the trie as follows. First, nodes and edges of the trie become respectively states and transitions of the automaton. The root node becomes the initial state, and the nodes with identifiers associated to them (the gray nodes) become final states. Then, a transition is added from the root node to itself, for each symbol in the alphabet which has not already a transition leaving the root node "0" (in the example, all the symbols except 't' and 'm'). Finally, failure transitions must be added for each state. Figure 3 (left) represents the NFA, with failure transitions drawn as dashed arrows. The automaton takes a failure transition when the current input symbol does not match any regular transition leaving the current state.

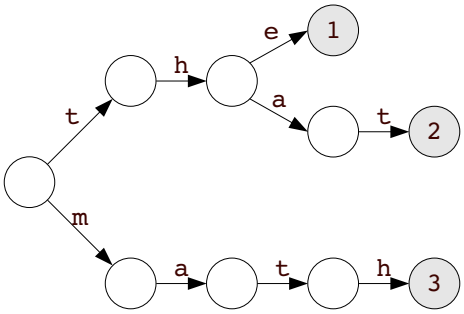

Fig. 2. The trie corresponding to example dictionary {the, that, math}.

Failure transitions reuse information associated with the last input symbols (suffix) to recognize patterns which begin with that suffix, without restarting from the initial state (for example, the input 'mathat' matches the patterns 'math' and 'that'). The first four symbols lead from state 0 to state 3 (matching pattern 'math'). Then, symbol 'a' does not correspond to any regular transition from node 3 (in fact, there are no transitions leaving node 3 at all). So, the automata takes the failure transition, reaching the node corresponding to path 'th'. From it, the remaining symbols 'at' cause the automaton to end up in node 2, completing the second match. Although AC-fail is deterministic, its automaton is called an NFA because of transitions which do not consume input (traditionally called ϵ-transitions).

A more formal description of AC-fail follows, in terms of a the transition function δ, a failure function f and an output function *out*.

Algorithm 1 The AC-fail variant of the Aho-Corasick algorithm.

```
1:  q ← 0; // initial state (root)
2:  for all i ∈ {1, 2, ...m} do
3:      while δ(q, T[i]) = {} do
4:          q := f(q); // failure transition
5:      end while
6:      q := δ(q, T[i]); // regular transition
7:      if out(q) ≠ {} then
8:          report i, out(q);
9:      end if
10: end for
```

Function $\delta(q, a)$ gives the state entered from current state q by matching symbol a. Function $f(q)$ gives the state entered at a mismatch: $f(q)$ is the node labeled by the longest proper suffix w of $L(q)$. Note that $f(q)$ is always defined, since the empty string is a prefix of any pattern, and is the label of the root. Finally, $out(q)$ gives the set of patterns recognized when entering state q.

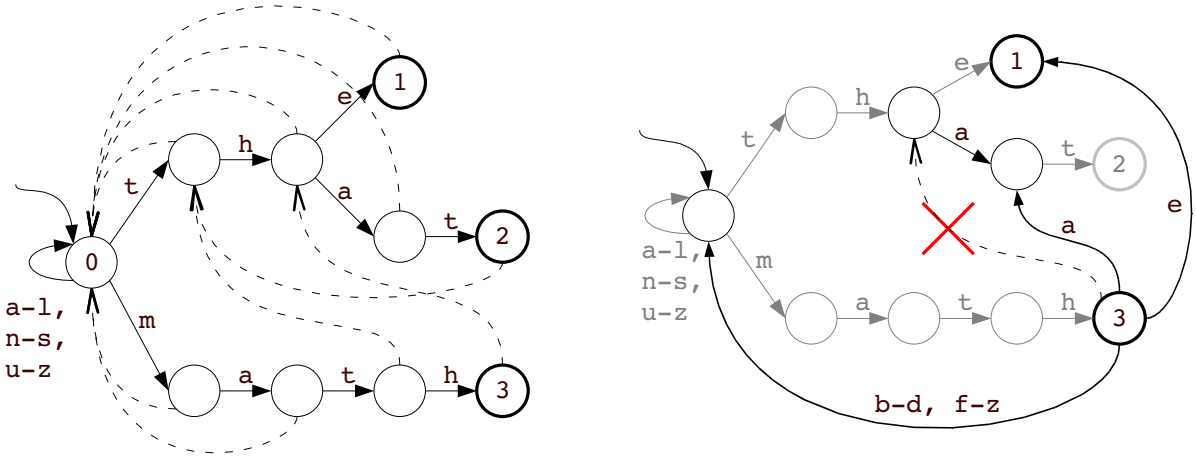

Fig. 3. The Aho-Corasick NFA automaton corresponding to the example dictionary (left). A failure transition is replaced with regular transitions (right).

AC-fail suffers from a potential performance drawback: on a failure, multiple state transitions can happen per single input symbol (within the *while* loop of Algorithm 1). When the size of the dictionary grows, the performance of AC-fail decreases quickly due to failure transitions [18]. An improved version of AC-fail, called AC-opt, solves this issue by employing a Deterministic Finite Automata (DFA) in place of the NFA, at the cost of increased memory requirements. The DFA is obtained from the NFA by replacing all the failure transitions with regular ones. The DFA has exactly one transition per each state and input symbol ($\forall q, a : |\delta(q, a)| = 1$), and each consumed input symbol causes exactly one transition. Failure transitions are replaced according to the following rule:

$$\forall (q, a) : \delta(q, a) = \{\} \ \wedge \ f(q) = r \qquad \text{add a new transition:} \qquad \delta(q, a) := \delta(r, a).$$

(to be repeated until fixed point is reached).

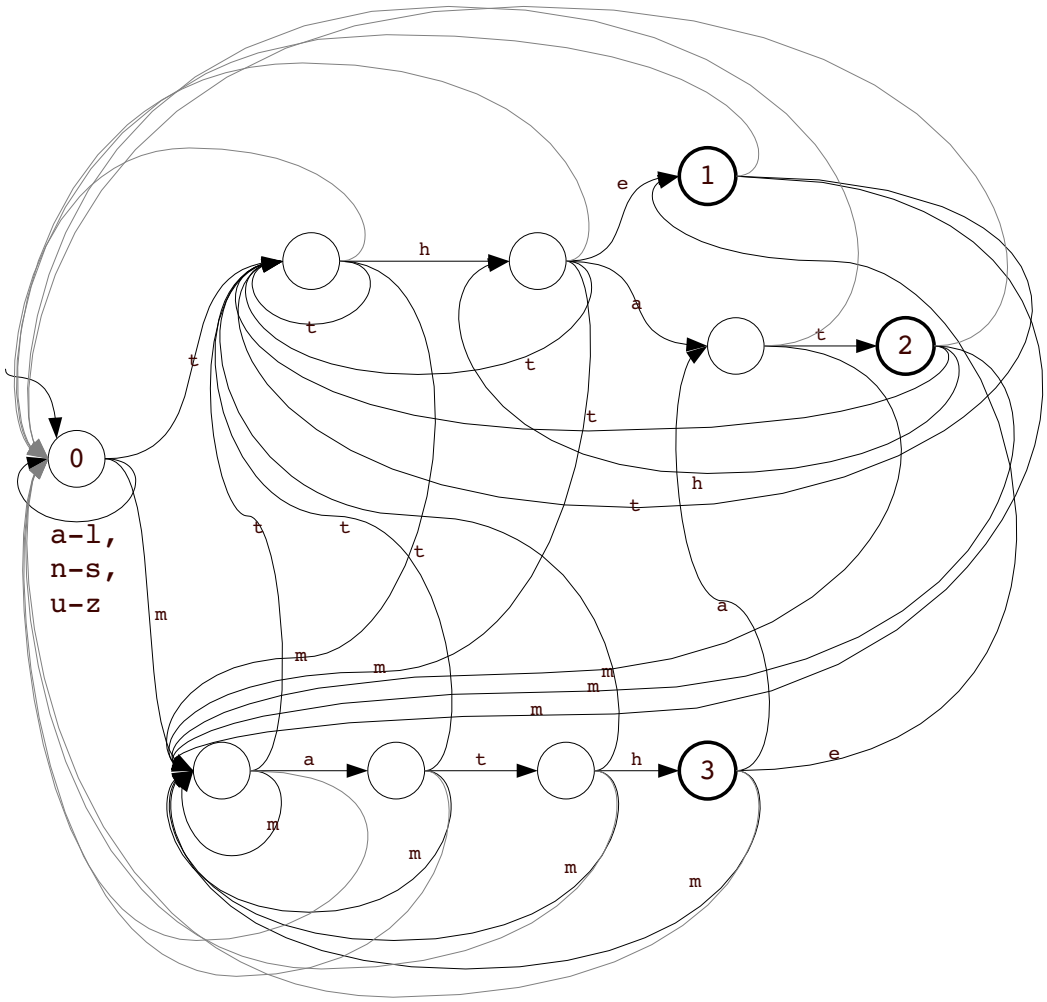

Fig. 4. The final Aho-Corasick DFA automaton.

Figure 3 (right) illustrates how a failure transition (the crossed out one) is replaced with regular transitions, while Figure 4 illustrates the final DFA.

3 Parallelizing the Aho-Corasick on a multi-core architecture

Our approach combines a simple parallelization, where multiple automata operate independently on different subsets of the input text, with a global orchestration of the main memory accesses, that effectively pipelines main memory requests.

We use the AC-opt variant of the Aho-Corasick algorithm which employs a Deterministic Finite Automaton (DFA) [18]. The main advantage of this algorithm is its dictionary-independent theoretical performance. In fact, it relies on a precomputed State Transition Table (STT) so that the processing of a new symbol from the input stream always involves a single state transition of the DFA, i.e. a fixed amount of computation. We exploit several levels of parallelism available in the Cell/B.E., as depicted in Figure 5. First, we split the input text in equal chunks which are delivered to each of the SPEs, so that they can all proceed to match it against the dictionary in parallel.

The life of an AC-opt automaton alternates between two tasks: determining the address of the next state in the STT (on the basis of the current state and the input symbol), and fetching the next state from main memory. These two tasks have very different latencies: the first task completes in a few nanoseconds (we call this interval *transition time*), but the memory transfer can take up to several hundreds of nanoseconds, depending on the traffic conditions and congestion. A single automaton utilizes poorly the hardware of the Cell/B.E.: it computes for a tiny fraction of

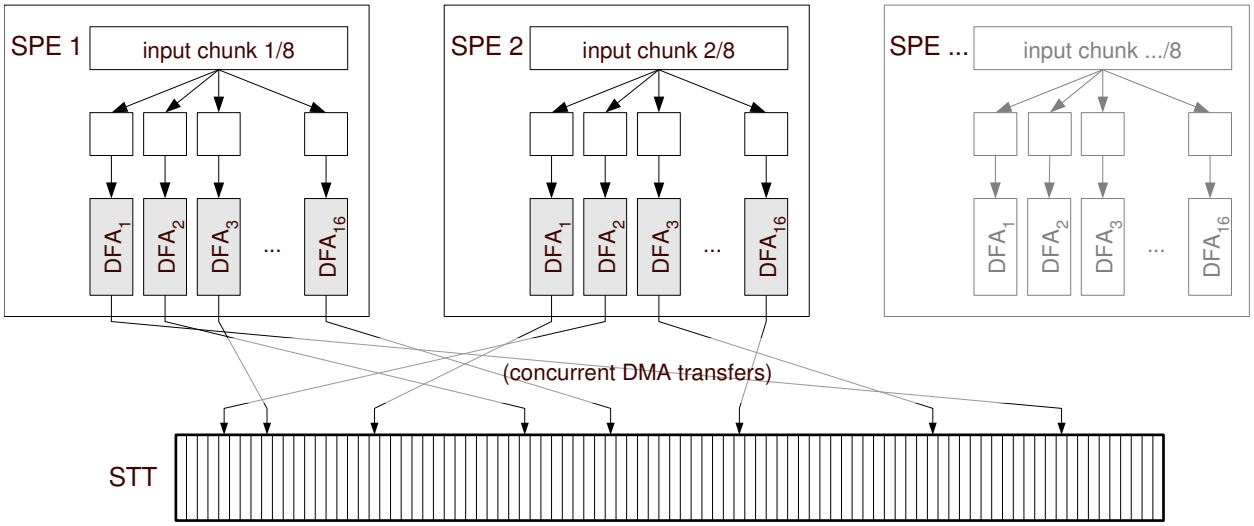

Fig. 5. Proposed parallelization strategy. Each SPE runs multiple automata (DFA_1, DFA_2, ...). Each automaton processes a separate chunk of the input text, but all the automata access the same State Transition Table (STT) in main memory.

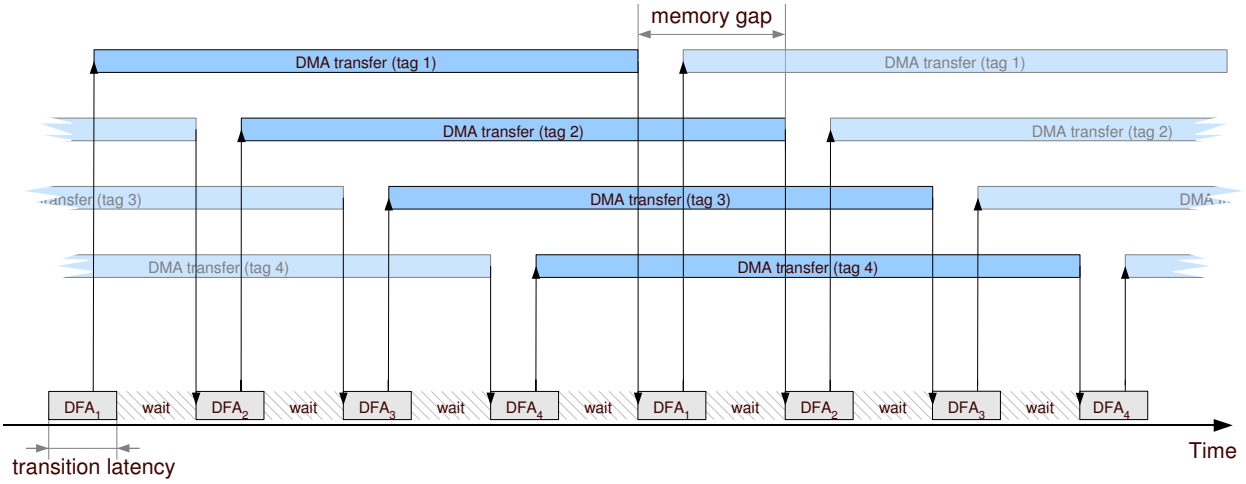

Fig. 6. How we schedule multiple automata (e.g. DFA_1 ... DFA_4) to overlap computation and data transfers. Grey rectangles represent transition times; upward arrows indicate DMA issue events; downward arrows indicate DMA completion events.

---

its execution time, and keeps waiting for the rest of it. To improve this, we map multiple automata on the same SPE, and we overlap data transfer with computation. This is possible in the Cell/B.E. thanks to the Memory Flow Controller (MFC) integrated in each SPE, which can operate in parallel with the program control flow. We schedule the automata operations to maximize overlapping of computation and data transfer, and ensure optimal utilization of the memory subsystem (which is the bottleneck), as illustrated in Figure 6. The figure shows a possible schedule with four automata, which execute state transitions cyclically on a single SPE.

It is worth noting that the input text is split in chunks twice: when dispatching the input to different SPEs and within different automata in the same SPE. In both cases, chunks must partially overlap to allow pattern-matching across a boundary. The amount of necessary overlapping is equal to the length of the longest pattern in the dictionary minus 1 symbol.

The figure also illustrates the *memory gap*, the key performance parameter of our implementation: the memory gap is the time interval between the completion of two consecutive memory accesses that a single SPE is capable of sustaining in steady state, after the pipeline of memory requests is full. If the transition time is shorter than the memory gap (which happens in all our experiments), the memory gap determines the smallest time interval between the processing of two input symbols, which reflects the overall performance of the algorithm.

In order to obtain the highest level of performance, we have benchmarked the DMA transfers to determine how the memory gap is affected by the number of SPEs, the number of outstanding transfers per each SPE and the transfer block size. The benchmarks show that the memory subsystem is best utilized when the transferred blocks are smaller than 64 bytes, and a program control structure is adopted which fires DMA requests keeping at least 16 of them outstanding at any time (see Figure 7). This suggests a configuration with 16 automata per SPE, and to transfer portions of the STT in blocks of 64 bytes. With a DMA size of 32 and 64 bytes, the memory gap is practically

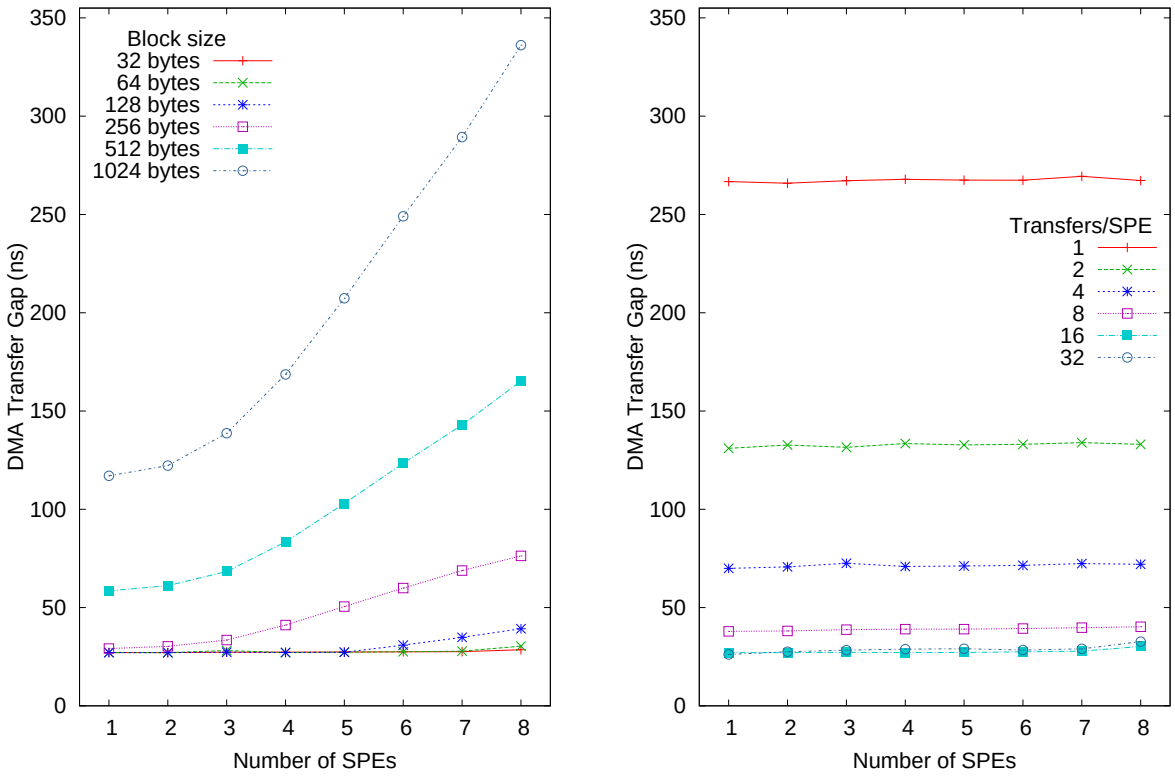

Fig. 7. The memory gap is a function of transfer size S (left) and number of concurrent DMA requests N (right). To optimize the memory gap, each SPE can use $N = 16$ and $S \leq 64$ bytes. In these conditions, scalability is ideal. In both cases, transfers hit random locations in a 864 Mbyte wide contiguous area. In the left plot $N = 16$; in the right plot $S = 64$ bytes.

constant. This implies that we can linearly increase the processing power up to 8 SPEs without performance penalty.

Using these results, we determine the performance bounds for a system composed of 1 and 2 Cell processors respectively. The reciprocal of the memory gap gives the maximum frequency at which each SPE accepts a new input symbol. We consider symbols of 1 byte, and we express aggregate performance values in Gbps, to ease comparison with network wire speeds.

	Memory gap (per SPE)	Symbol Frequency (per SPE)	Throughput (per SPE)	Throughput (Aggregate)
8 SPUs	29.34 ns	34.07 MHz	272.56 Mbps	2.18 Gbps
16 SPUs	40.68 ns	24.58 MHz	196.64 Mbps	3.15 Gbps

These throughput values represent the theoretical upper bound for any Aho-Corasick implementation which relies only on main memory to store its STT and uses no overlapping among input chunks. The objective of an efficient implementation is to approximate these boundaries as closely as possible.

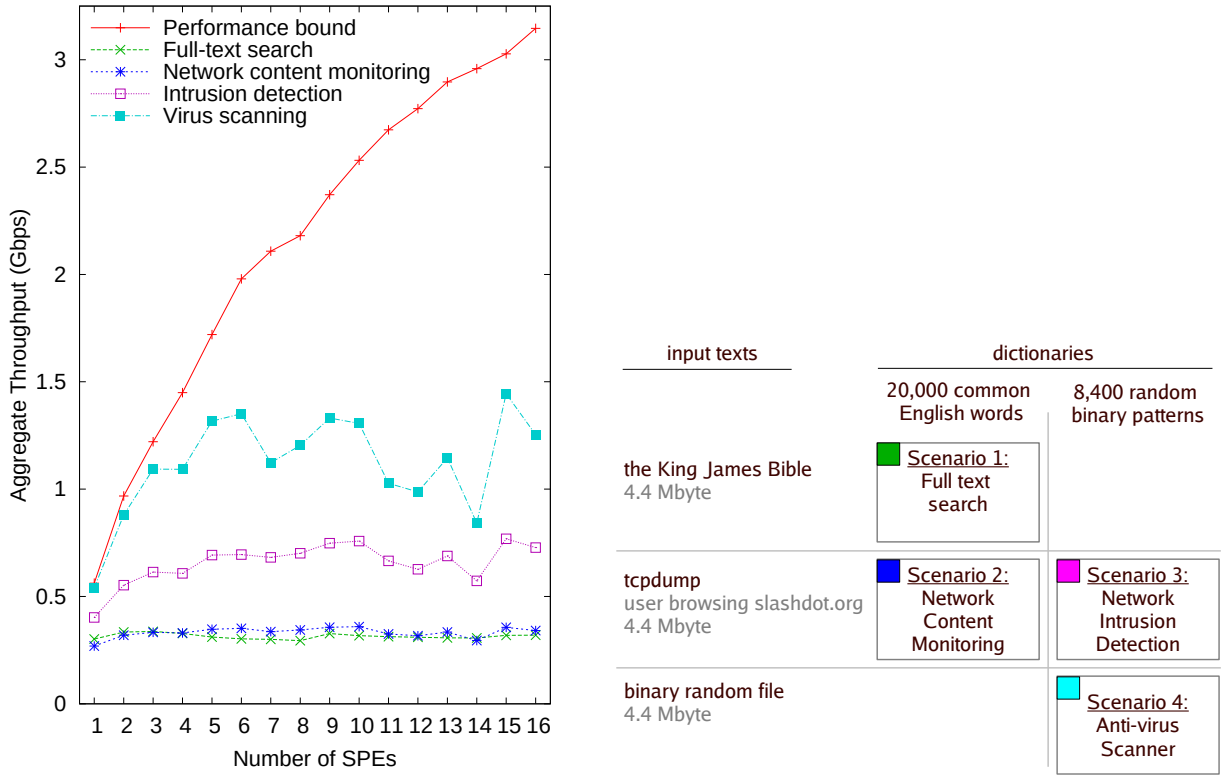

Fig. 8. Four representative experimental scenarios (left). Performance and scalability fall largely below the expected values, even if the implementation respects the requirements derived above (right).

4 Performance analysis

We measured the performance of our algorithm, implemented as described in the previous section, in four representative experimental scenarios, designed to approximate the operating conditions of (1) a full-text search system, (2) a network content monitor, (3) a network intrusion detection system, and (4) an anti-virus scanner. In scenario (1), a text file (the King James Bible) is searched against a dictionary containing the 20,000 most used words in the English language, whose average length is 7.59 characters. In scenario (2), a capture of network traffic obtained at the transport control layer with `tcpdump` while a user is browsing a popular news website (Slashdot) is searched against the same English dictionary as before. In scenario (3), the same network capture is searched against a dictionary of 8,400 randomly generated binary patterns, whose length is uniformly distributed between 4 and 10 characters. In scenario (4), a randomly generated binary file is searched against a dictionary of 8,400 randomly generated binary patterns, with average length 7. For sake of uniformity, all inputs are truncated to the same length as the King James Bible (4.43 Mbyte), and both dictionaries yield STTs with approximately the same number of states (49,849 and 50,126 respectively). The performance values obtained in these four scenarios are represented in Figure 8.

To our surprise, an implementation merely based on the above considerations turns out to have poor performance (Figure 8). In all cases, scalability is poor and absolute performance is significantly

below the theoretical boundary. Since the computational part of the code operates identically in all the scenarios, the degradation must be due to the memory subsystem. Scenarios based on binary, randomly generated dictionaries outperform natural language, alphabetic ones: the best performance is achieved by the anti-virus scanner scenario, which searches a random input against a random dictionary. This scenario corresponds to a state-transition graph where the average branching factor is the highest among all scenarios, and the random input causes a uniform distribution of hits among all the states which can be reached with a transition from any given state. Therefore, uniformly distributed usage patterns cause less congestion in the memory subsystem. More precisely, our experiments show that memory congestion has three major components, that we call *memory pressure*, *memory layout issues*, and *hot spots*.

Memory pressure is the number of accesses that hit each block of memory of given, fixed size in the unit of time. Pressure is higher when many SPEs (and automata) are employed, and when the STT is allocated in a smaller memory area; pressure is lower when fewer SPEs are used, and when the STT is spread across a large memory area. Memory pressure strongly influences the memory performance. To show its impact, we consider accesses to an STT stored in a contiguous area of main memory, which is accessed concurrently by all the automata in each of the SPEs employed. We begin with a 64-Mbyte STT, and we increase its size up to the maximum available space on our system, 864 Mbyte. We present a benchmark where automata access uniformly distributed random

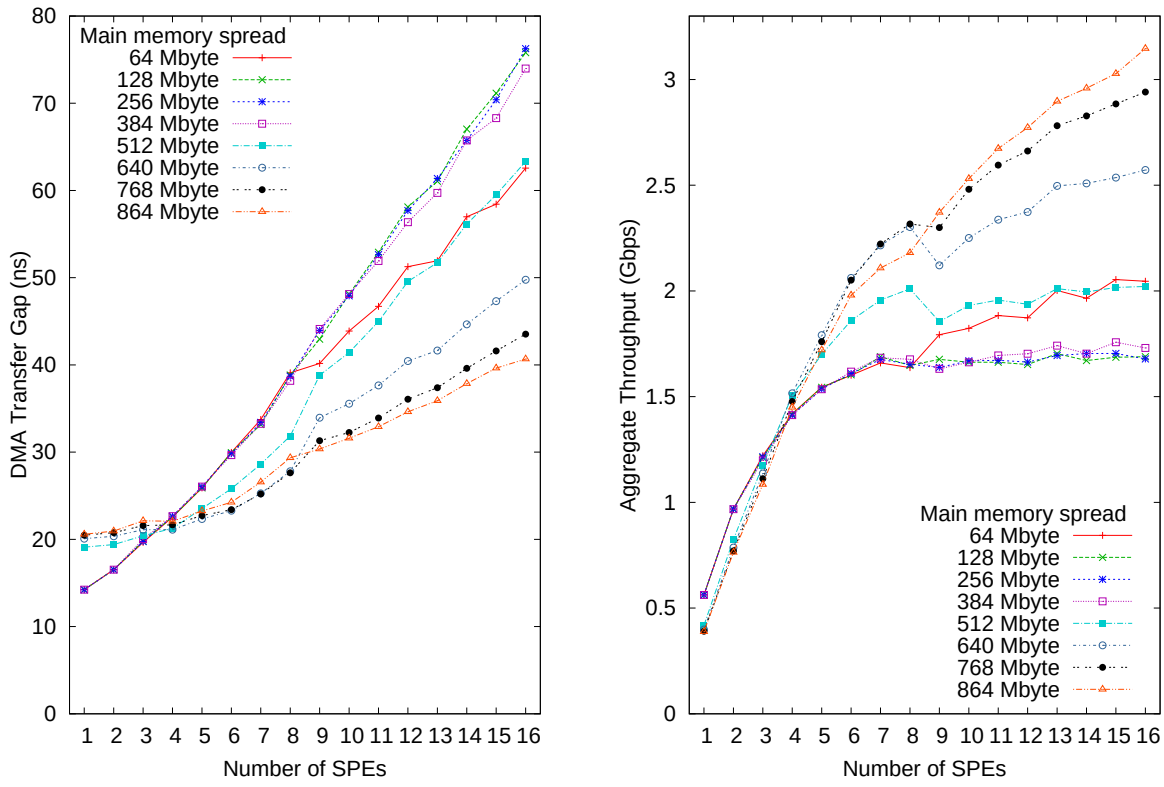

Fig. 9. When *memory pressure* grows, the memory gap increases (left) and, consequently, the throughput degrades (right). Spreading the STT across a larger area alleviates pressure and improves performance. Each SPE generates 16 concurrent transfers, 64 byte in size each.

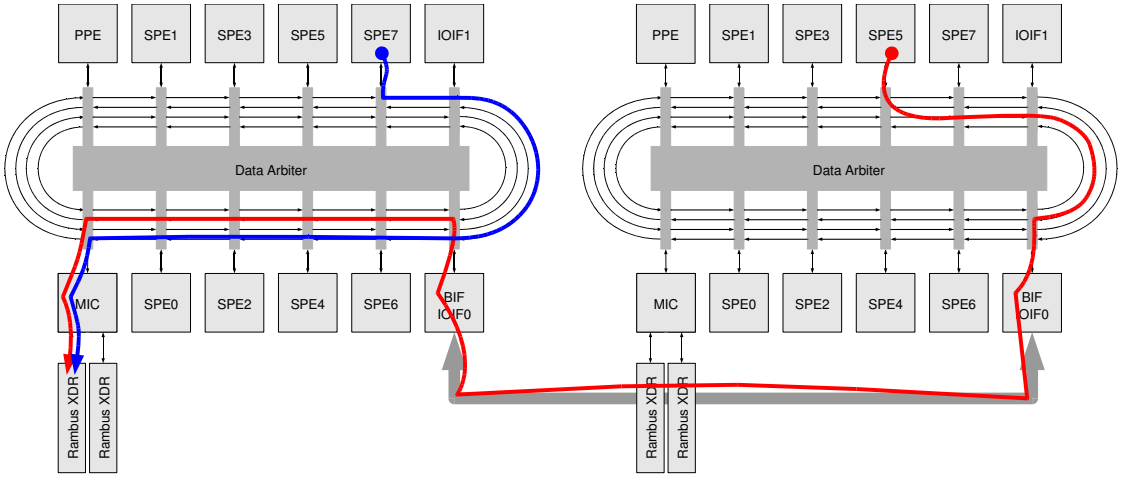

Fig. 10. Memory layout issues: if the STT resides entirely in memory connected to the MIC of first of two coupled Cell processors, all the SPEs will be congesting that MIC, while the other MIC is unused.

---

locations within an STT of variable size (Figure 9). The benchmark shows that the performance degrades under high memory pressure. Spreading STT entries across a larger area helps relieving memory pressure and improves the performance. To enjoy best performance, an STT should be spread across all the main memory which is available for the application.

Memory layout issues are a second phenomenon which influences the curves in Figure 9. When two Cell processors are used together, half of the memory is connected to the Memory Interface Controller (MIC) of the first processor, and half to the MIC of the second one. Contiguous heap regions are allocated sequentially, starting from the first half. Therefore, a relatively small STT will reside entirely in the memory connected to the first Cell processor. The SPEs in the first Cell will access the STT directly (the red line in Figure 10), but the SPEs in the second processor will access the STT through the MIC of the first one (as illustrated by the blue line). This type of access experiences a longer latency due to the more transactions involved, and puts all the memory load on the MIC of the first Cell processor. Larger STTs allocated across the boundary between the two memory areas can still be unevenly distributed

Hot spots are memory regions which are frequently accessed by multiple transfers at the same time, typically because they contain states which are hit frequently. If a state is hit very frequently by each automaton, there is a significant probability that, at a given time, multiple automata access it, making it a hot spot. It is a common case that a few states are hit very frequently while the remaining ones, which are the vast majority, are almost never. Figure 11 illustrate this condition in the four considered scenarios. We plot the number of states contained in each *level*, and the relative frequency of hits per level (we call *level* of a state its distance from the initial state, determined with a breadth-first visit of the state transition graph). While the majority of states belong to the intermediate levels (left), the few states in the very first levels attract the vast majority of hits (see Table 1).

In summary, considerations inspired by memory pressure, memory layout and hot spots suggest to change the algorithm in order to distribute memory accesses as uniformly as possible across all the

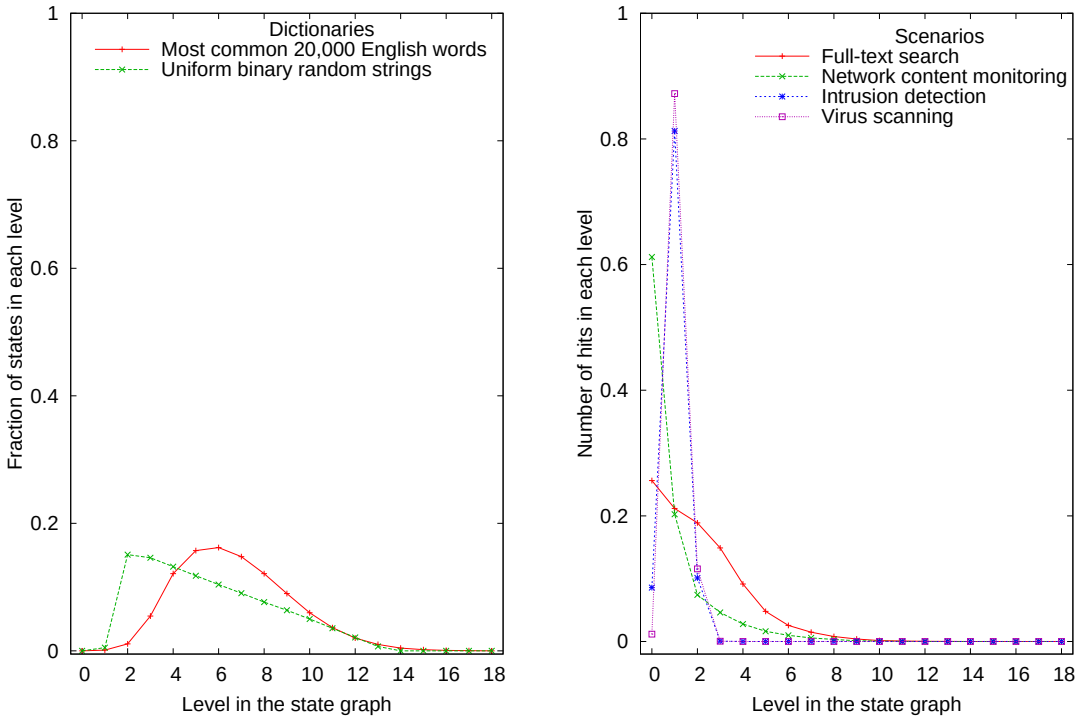

Fig. 11. State hits are very unevenly distributed. Only a small percentage of all the states are in levels 0 and 1 (left), but those states are hit very frequently (right). This causes hot-spots in the memory subsystem, and poor scalability.

	% States in levels 0,1	% Hits
Full-text search	0.114 %	46.79 %
Network content monitoring	0.114 %	81.46 %
Intrusion detection	0.506 %	89.84 %
Virus scanning	0.506 %	88.39 %

Table 1. A few states, typically in levels 0 and 1, are responsible for the vast majority of the hits.

available memory. To do so, we employ the strategies discussed and evaluated quantitatively in the next section.

5 Performance optimization

We have just showed that realistic string search scenarios do not enjoy the same uniformly distributed memory access patterns as in the benchmarks, and this causes congestion and performance degradation. To counter congestion, we transform the STT, by employing a combination of techniques: *state caching, state shuffling, state replication* and *alphabet shuffling*. In this section we

present these techniques and we provide a quantitative analysis of their effects.

By *state caching*, we mean that a copy of the most frequently used states is statically stored in each SPE's LS at program initialization. Since little space is available for this in each LS (around 180k), only a limited number of STT entries can be statically cached. The entire traffic generated by the cached states is now relieved from the memory subsystem.

By *state shuffling* we mean that the states are randomly renumbered and reordered. Every state receives a new, randomly selected state number. All the entries in the STT are rewritten according to the new numbering scheme (except for the initial state, which continues to be state number 0). This technique effectively uniforms memory pressure along all the available blocks, since all traffic is randomly redistributed. Depending on the available amount of main memory, we choose to shuffle state numbers to a target state space which is *larger* than the original one. This creates a larger STT which also includes unused entries, and allows to trade off space utilization and memory efficiency. The larger is the target space, the lower is the memory pressure.

State shuffling alone is not sufficient to guarantee uniform memory pressure, because concurrent accesses to similar offsets within separate STT entries also cause contention. To counter this contention, we employ *alphabet shuffling*, i.e. we shuffle the order of input symbols (from 0 to 255) and we reorder next state values within each STT entry according to the new scheme. The alphabet shuffling function also depends on the state number, so that different states enjoy different shuffles. We employed a shuffling function as simple as:

$$symbol' = (symbol + state) \ mod \ A$$

where A is the size of the alphabet (256 in our case). It proved to be very inexpensive, since it just involve a sum and a bitwise operation (the *mod* operator reduces to a bitwise *and*), and as effective in reducing congestion as all the more complex functions we have tried.

State and alphabet shuffling redistribute memory pressure evenly across all the available memory, including states which are hot spots. Nevertheless, they do not relieve contention at the hot spots. To do so, we employ *state replication*, i.e. we select frequently used states and we replicate them, so that different automata will access different replicas. Replicas are stored at regular intervals, so that automata can determine the address of the replica they need with inexpensive arithmetics. Replication proves to be very effective; for example, when a state is replicated four times, its memory pressure is actually reduced to one fourth, alleviating memory gap degradation when that state is accessed.

Let us assume that the original states of the AC-opt automaton (before the above transformations are applied) are ordered by increasing level: first the initial state, then level 1, level 2 and so on. We call this state numbering scheme the *logical state space*. Our transformations map the logical state space into a *physical state space*, which we use to physically store states in memory. Figure 12 illustrates how this mapping happens. The physical state space can be arbitrarily larger than the logical state space; the larger it is, the more the memory pressure is reduced. We partition the logical state space in three regions, and we apply state caching to Region 1, state shuffling and replication to Region 2, and state shuffling only to Region 3. We apply alphabet shuffling to all the

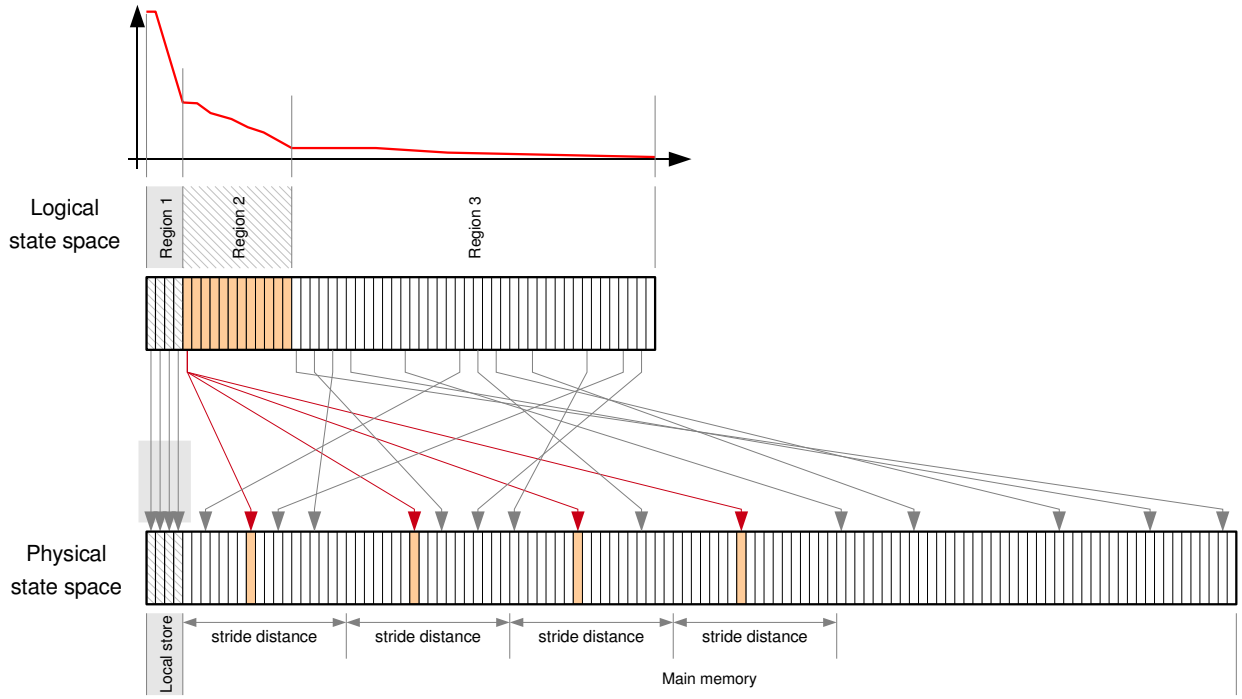

Fig. 12. To reduce congestion, we transform the *logical state space* into a *physical state space* through state shuffling, replication and caching.

---

regions. In detail:

- *Region 1* contains the initial state, and as many states from the first levels of the state transition graph as can fit in the LS of every SPE. In our configuration, Region 1 cannot be larger than 180k, allowing 180 states to be statically cached. Accesses to states in Region 1 will not impact the main memory anymore, reducing congestion significantly and increasing the overall throughput.
- *Region 2* contains states which are hit often, but could not be statically cached. To avoid hot spots, we replicate these states, so that different automata access distinct replicas of the same entry. While the position of the first replica of each state in Region 2 is selected randomly, the next replicas are stored at distances from the first one which are multiples of a stride distance, to simplify indexing. Correctness is not affected by replication, because replicas all contain the same values, while congestion is effectively reduced by the replication factor.
- *Region 3* contains all the remaining states, which are hit less frequently than Regions 1 and 2. These states are subject only to shuffling.

The border between Region 2 and 3 is arbitrary, and so is the replication factor. One may decide to replicate more states less times, or fewer states more times, and it is not clear *a priori* which of these strategies lead to the best results. Determining the optimal value for such parameters requires a solution space exploration strategy which is beyond the scope of this paper.

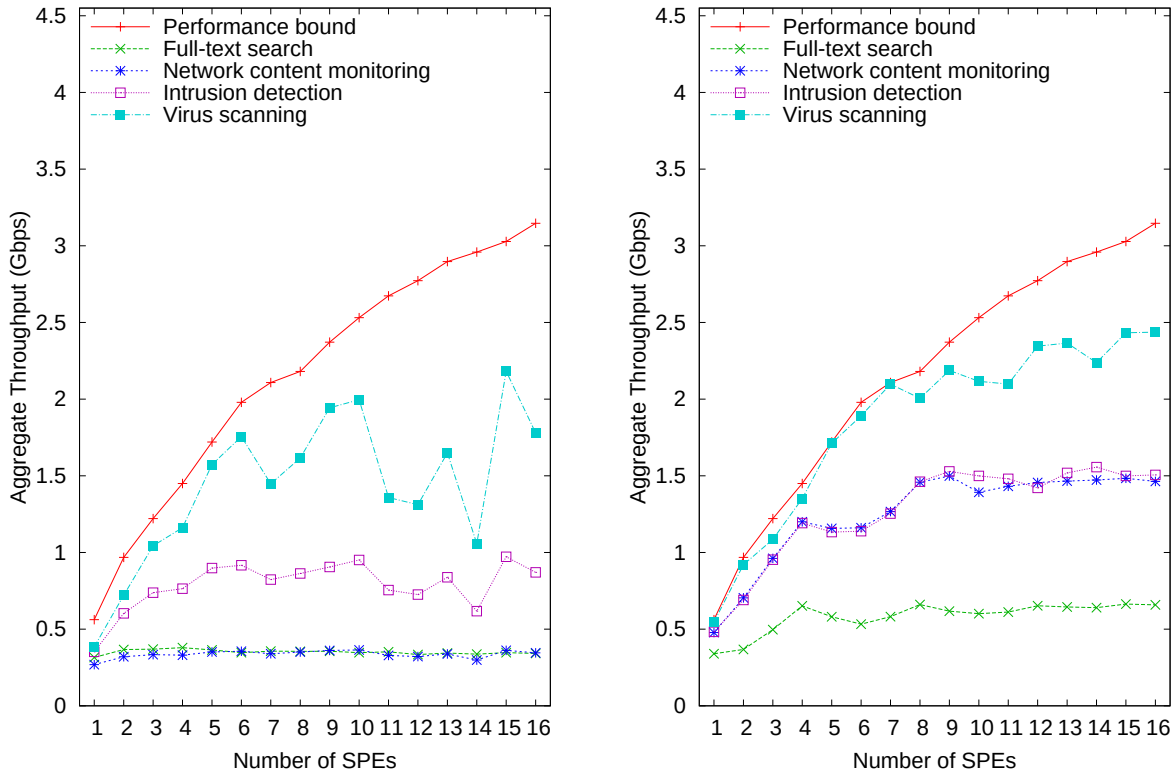

Fig. 13. Performance with state shuffling (left). Performance with state shuffling and replication (right).

6 Experimental results

In this section we experimentally evaluate the effectiveness of the techniques discussed above. For each technique, we compare its performance against the theoretical bound derived at the end of Section 3. In each of the graphs in this section, a red solid line indicates the performance corresponding to this bound.

We present state caching in the end because it is the only technique which also involves the local store; as a consequence, it achieves higher performance values than the theoretical bound, which assumes use of the main memory only.

All the experiments and the measurements refer to the implementation of the algorithms in C language using the CBE intrinsics and language extensions and compiled it with GNU GCC version 4.1.1 and the IBM Cell SDK version 2.1. We have run the experiments on an IBM DD3 blade with two Cell/B.E. processors running at 3.2 GHz, with 1 Gbyte of RAM and a Linux kernel version 2.6.16. All the experiments consider the same four scenarios introduced in Section 4. For sake of consistency, the source code of our implementation used in all the experiments (except for the one with caching enabled) is the same as the one used to derive the initial performance estimate in Figure 8. This allows us to claim that the performance improvement is due uniquely to the decrease in memory congestion caused by our transformations.

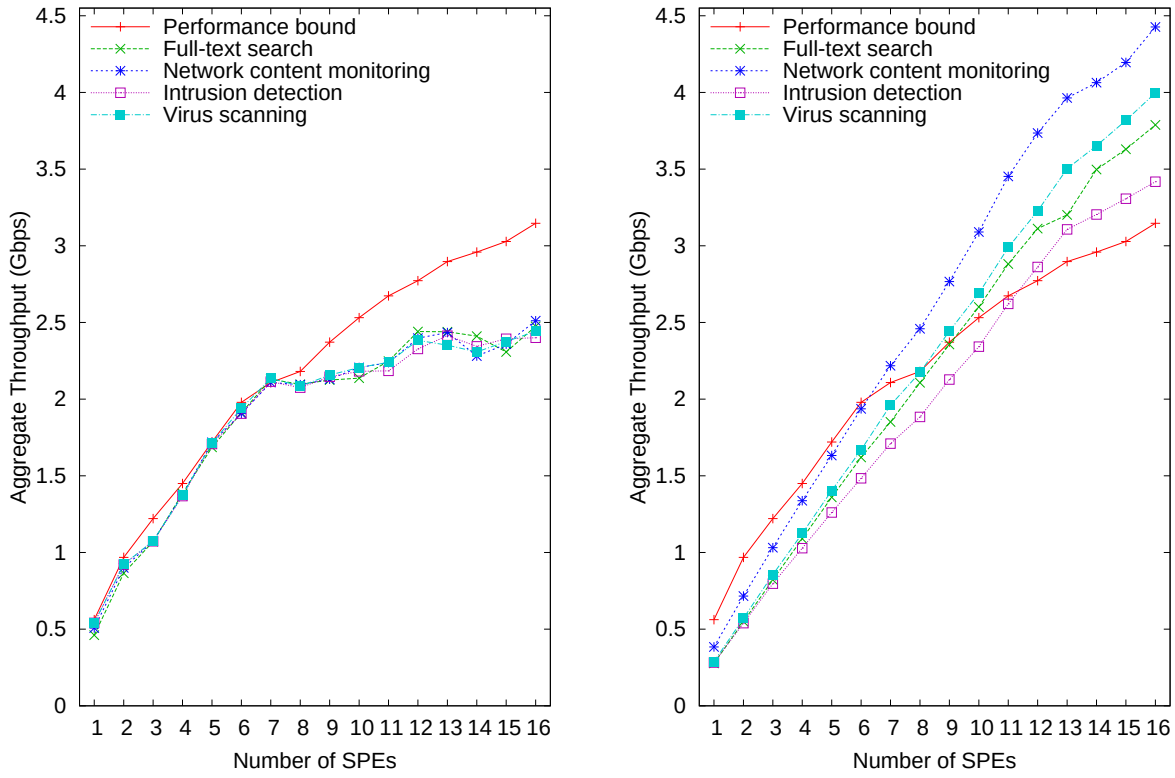

Fig. 14. Performance with state shuffling and replication, and alphabet shuffling (left). Performance with all of the above optimizations plus state caching (right).

Figure 13 (left) shows the effect of state shuffling. Comparing it against Figure 8, we can see a significant improvement, especially in the two scenarios which employ a binary dictionary. On the other hand, congestion is still compromising scalability. Figure 13 (right) shows the combined effect of state shuffling and replication. In our experiments, we have chosen a Region 2 as large as 10,000 states (approximately 20% of the states, in all the scenarios).

Figure 14 (left) shows the results when alphabet shuffling is used in addition to the previous case. Performance values are close to the optimal ones up to 7-8 SPEs, with sub-optimal performance when we add more SPEs. This is due to the memory layout issue discussed above, and to the fact that a portion of memory connected to the first Cell processor is used by the operating system, and therefore not available for use with our techniques.

Finally, Figure 14 (right) shows how the performance increases when we cache 180 states. Note that the performance with less than 6-7 SPEs has decreased with respect to the previous experiment, due to the additional cost of determining whether each state is cached in the local store or not, which causes a longer transition latency. On the other hand, when more SPEs are used, scalability is improved with respect to the previous experiments. As anticipated, performance is higher than the theoretical bound because the bound was determined assuming the use of main memory only.

7 Related work

String searching is a heavily beaten track. The literature presents many string searching algorithms: Bloom filters [3], Aho-Corasick [1], Boyer-Moore [4], Knuth-Morris-Pratt [11], Rabin-Karp [10], Commentz-Walter [7] and Wu-Manber [19] are just a few.

Bloom filters [3] are a space-efficient probabilistic data structures which can be used for generic membership tests, also frequently employed in network applications [5]. A Bloom filter can store a compact representation of a dictionary. The filter can be queried to determine whether a string belongs in the dictionary or not. The price for the reduced footprint is a predictable rate of false positives, and a relatively high query cost.

Designers can either choose design parameters (including the number and nature of the hash functions [14]) to keep the false positive rate negligible, or an exact search algorithm can be employed to perform the final decision.

Bloom filters exhibit a good amount of data-level parallelism that can be well exploited in specialized hardware, for example FPGAs [8,12,13]. On the other hand, they are difficult to implement efficiently on general-purpose instruction-set processors, and even more on the Cell/B.E.. In fact, a Bloom filter can store a dictionary where patterns have a fixed length. If the dictionary is composed by patterns of different lengths, a bank of Bloom filters must be adopted, each working on a window of different length of the input stream. Each time a new input character is considered, all the windows shift, and the entire bank of filters must be recomputed and verified.

A class of exact string searching algorithms is based on finite state machines or pre-computed tables. Knuth-Morris-Pratt [11] searches all the occurrences of a single pattern in a text, and it skips as many input characters as it is safe to do, when a mismatch occurs. To do so, it relies on a pre-computed partial match table. Boyer-Moore [4] is another single-pattern searching algorithm which speeds up the search via pre-computed tables. It starts matching from the last character in the pattern so that, upon mismatch, an entire span of the input is skipped, having the same length as the search pattern can be skipped. Commentz-Walter [7] can be regarded as a multiple-pattern extension of the Boyer-Moore algorithm. A match is attempted by scanning backwards through the input string. At the point of a mismatch, the characters which were matching immediately before the mismatch happened can be used to determine a distance to skip before the next match is attempted. These skip distances are also kept in a pre-computed table. Aho-Corasick [1], which we have discussed in detail in Section 2, also belongs in this class.

8 Conclusions

In this paper we have shown that the Cell/B.E. processor can be successfully utilized to perform tasks, such as high-performance keyword scanning, that once were the typical domain of specialized processors or FPGAs. Our prototype implementation of the Aho-Corasick algorithm has

achieved a remarkable deterministic throughput of 2.5 Gbits/second for several representative combinations of input texts and dictionaries.

At the base of this result there is the unique capability of the Cell/B.E. processor of supporting irregular memory communication using explicit DMA primitives. By properly orchestrating and pipelining memory requests we have been able to create at user level a virtual layer in the memory hierarchy with a *gap* latency of only 30 nanoseconds using a synthetic benchmark.

While implementing the Aho-Corasick algorithm, we discovered that this performance result can only be achieved under ideal conditions of the main memory traffic. We have developed a number of techniques to approximate these conditions and alleviate memory congestion –state shuffling, state replication, alphabet shuffling and state caching. With this analysis we have been able to close the loop and achieve the optimal performance. We believe that the techniques described in this article can be successfully applied to other data-intensive applications that display irregular access patterns across very large data sets.

Acknowledgements

The research described in this paper was conducted under the Laboratory Directed Research and Development Program for the Data Intensive Computing Initiative at Pacific Northwest National Laboratory, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy under Contract DEAC0576RL01830.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] S. Antonatos, K. Anagnostakis, M. Polychronakis, and E. Markatos. Performance analysis of content matching intrusion detection systems, 2004.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1977.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey, 2002.

[6] Long Bu and John A. Chandy. A cam-based keyword match processor architecture. *Microelectronics Journal*, 37(8):828–836, 2006.

[7] Beate Commentz-Walter. A string matching algorithm fast on the average. In H.A. Maurer, editor, *Proceedings of the Sixth International Colloquium on Automata, Languages and Programming*, pages 118–131. Springer-Verlag, 1979.

[8] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[9] IDC Corporation. The expanding digital universe. *White Paper*, March 2007.

[10] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[11] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[12] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *Proceedings of the ACM Intl. Symposium on Field Programmable Gate Arrays (FPGA 2001)*, pages 87–93, 2001.

[13] James Moscola, John Lockwood, Ronald Loui, and Michael Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38, Napa, CA, USA, April 2003.

[14] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput.*, 46(12):1378–1381, 1997.

[15] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Peak-performance DFA-based string matching on the Cell processor. In *In Proc. SMTPS '07*, 2007.

[16] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching, 2004.

[17] Dinesh C. Suresh, Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Automatic compilation framework for bloom filter based intrusion detection. In Koen Bertels, João M. P. Cardoso, and Stamatis Vassiliadis, editors, *ARC*, volume 3985 of *Lecture Notes in Computer Science*, pages 413–418. Springer, 2006.

[18] Bruce W. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. Technical Report 19, Eindhoven University of Technology, 1994.

[19] S. Wu and U. Manber. Fast text searching with errors. Technical Report TR-91-11, University of Arizona. Dept. of Computer Science, 1991.