Peak-Performance DFA-based String Matching on the Cell Processor

Daniele Paolo Scarpazza[1], Oreste Villa[1,2] and Fabrizio Petrini[1]

[1]Pacific Northwest National Laboratory
Computational & Information Sciences Division
Richland, WA 99352 USA
{fabrizio.petrini, daniele.scarpazza}@pnl.gov

[2]Politecnico di Milano
Dipartimento di Elettronica e Informazione
Milano I-20133, Italy
ovilla@elet.polimi.it

Abstract

The security of your data and of your network is in the hands of intrusion detection systems, virus scanners and spam filters, which are all critically based on string matching. But network links are getting faster and faster, and string matching is getting more and more difficult to perform in real time. Traditional processors are not keeping up with the performance demands, whereas specialized hardware will never be able to compete with commodity hardware in terms of cost effectiveness, reusability and ease of programming.

Advanced multi-core architectures like the IBM Cell Broadband Engine promise unprecedented performance at a low cost, thanks to their popularity and production volume. Nevertheless, the suitability of the Cell processor to string matching has not been investigated so far.

In this paper we investigate the performance attainable by the Cell processor when employed for string matching algorithms based on Deterministic Finite-state Automata (DFA). Our findings show that the Cell is an ideal candidate to tackle modern security needs: two processing elements alone, out of the eight available on one Cell processor provide sufficient computational power to filter a network link with bit rates in excess of 10 Gbps.

1 Introduction

A Network Intrusion Detection System (NIDS) is an effective way to provide security to systems connected to the network. At the heart of a NIDS there is string matching algorithm, that allows the system to make decisions based not only on the packet headers, but on the actual content of the data flow.

Most payload scanning applications have a common requirement for string matching. For example, the presence of a specific string of bytes can identify the presence of an Internet worm or a malicious executable program. Because the location of such strings in the packet payload and their length is unknown, string matching algorithms must be able to detect strings of different lengths starting at arbitrary locations in the packet payload.

Packet inspection applications must be able to operate at wire speed: with network performance quickly increasing thanks to newer, more powerful version of Gigabit Ethernet, it is becoming increasingly difficult for software-based solutions to keep up with the line rates. In addition to the technological advances in network communication, we are also experiencing an explosion in the size of the data dictionaries that contain the strings that we need to match with the incoming stream of data.

Several hardware-based techniques have been employed for implementing packet inspection applications. In most cases, special algorithms have been developed on Field Programmable Gate Arrays (FPGAs)[8, 15], exploiting the potentially high level of parallelism available on these devices. A typical example is the implementation of Bloom filters on FPGAs [7, 13, 14]. A Bloom filter [2] is a data structure that stores a set of signatures compactly by computing multiple hash functions on each member of the input set. This technique queries a database of strings to check for the membership of a particular string. Another approach to implement string matching algorithms is through the use of Deterministic Finite State Automata (DFA). DFAs can efficiently implement algorithms (such as Aho-Corasick [1]) which allow to search for strings in a given dictionary. If the dictionary is expressed as a set of regular expressions rather than a set of exact strings, efficient techniques exist to generate a DFA which recognizes occurrences of all the regular

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.
1-4244-0910-1/07/$20.00 Copyright © 2007 IEEE.

expressions at the same time [4]. The literature presents numerous FPGA-based implementations of Aho-Corasick search algorithms [5, 16, 17, 10], with different degrees of performance and different dictionary sizes. Many other efficient string search algorithms exist, such as Knuth-Morris-Pratt [12], Boyer-Moore [3], Commentz-Walter [6], Wu-Manber [18] and their numerous derivatives, but they are not frequently adopted in security applications. In fact, they are based on heuristics and their workload depends on the input data, which makes them vulnerable to attacks based on malicious input streams specifically designed to overload them.

Meanwhile, the computing community is assisting to a shift of paradigm in processor design: gate densities, clock frequencies and heat dissipation are all reaching their physical limits, and the future promises of more computational power reside in the availability of more and more parallelism, through multiple cores on the same silicon die. The IBM Cell Broadband Engine (Cell BE, for short) is the most prominent member of the advanced multi-core processor family. This class of architectures promises unprecedented performance at a low cost, thanks to their popularity and production volume. Because of their relatively low cost and their significantly higher flexibility and ease of programming, multi-core processors could represent tough competitors to FPGA-based solutions in many application domains including network security, provided that they will be able to provide a level of performance which is comparable with the one provided by FPGAs.

Despite their performance potential, to the best of our knowledge, no works in literature have specifically tackled the problem of optimally mapping DFA-based string matching algorithms on advanced multi-core processors. In this paper, we analyze the performance attainable by the Cell BE processor when performing string matching through a DFA-based algorithm. Although we report experimental setup and data referring to the Cell BE architecture, the overall parallelization strategies we propose are general, and applicable to any other advanced multi-core architecture, e.g. like the one announced by Intel in its TeraScale initiative, which hosts 80 homogeneous processor cores on the same silicon die.

Our experiments show that the Cell BE architecture provides a considerable amount of performance: two of the eight processing elements which compose a Cell processor proved capable of performing a real-time filtering of a network link with a bitrate of 10 Gbps. The Cell also provides a remarkable degree of freedom in terms of the wide range of configurations in which multiple processing elements can be combined together to reach higher performance or to support larger dictionaries. In summary, the Cell architecture proves to be an ideal candidate for the implementation of DFA-based string search algorithms.

The remainder of this paper is organized as follows. In Section 2 we review the peculiar architectural characteristics of the Cell processor. Section 3 introduces the concept of DFA tile, which maps a DFA string acceptor to a processing element in the Cell BE, and Section 4 presents its implementation and discusses its performance. In Section 5 we show how tiles can be combined to increase performance and dictionary size. In Section 6 we present a technique which allows to employ arbitrarily large dictionaries at the price of a smooth degradation in performance. Section 7 concludes the paper.

2 Overview of the Cell BE Processor

This section discusses peculiar characteristics of the IBM Cell BE which are relevant in this context. These characteristics determine the way in which programmers must write software if they want to exploit the computational power available on the Cell.

The Cell BE [9] is a heterogeneous, multi-core chip capable of massive floating point processing, optimized for compute-intensive workloads and broadband, rich media applications. It is composed of one 64-bit Power Processor Element (PPE), 8 specialized co-processors called Synergistic Processing Elements (SPE), a high-speed memory controller and a high-bandwidth bus interface.

The PPE is responsible for running the operating system and coordinating the SPEs. It is a traditional 64-bit PowerPC processor core with a VMX unit, 32 KB Level 1 instruction cache, 32 KB Level 1 data cache, and 512 KB Level 2 cache. The PPE is a dual issue, in-order execution design, 2-way SMT, running at 3.2 GHz.

Each of the 8 SPEs contains a Synergistic Processing Unit (SPU), a memory flow controller, a memory management unit, a bus interface and an atomic unit for synchronization mechanisms. The SPU is a RISC-style processor with SIMD instructions and a large number (128) of wide

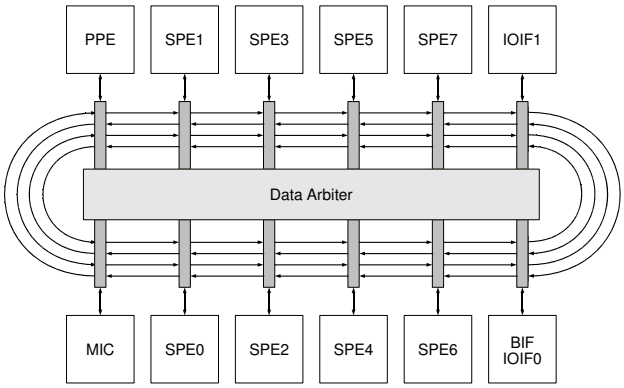

Figure 1. The Element Interconnect Bus.

registers (128 bit). The large number of registers facilitates efficient instruction scheduling and loop unrolling. To fully exploit the computational potential of the SPUs, the programmer must write explicit data-level parallel code by using an extended C syntax with intrinsics. Scalar code should be avoided: in fact a scalar operation is compiled as SIMD operation of which only a slice of the output is used. And even if a scalar operation produces a portion of the output of a SIMD one, it has the same cost.

SPU instructions cannot access the main memory directly. Rather, they access a 256 kbyte local store (LS) memory, which holds both instructions and data. The local store is in fact a software-controlled scratchpad, and it is the programmer's responsibility to manage its contents by transferring data from main memory to the LS and back via explicit DMA commands. Program execution continues unaffected in the SPU while a DMA transfer is in progress. This allows the programmer to overlap computation and data transfer steps in such a way that the transfer latency is completely hidden (e.g. via double-buffering techniques, as we detail in Section 6).

The PPE and SPEs communicate through an internal high-speed Element Interconnect Bus (EIB) (see Figure 1), which is the heart of the Cell processor's communication architecture. It has separate communication paths for commands and data, and the data network consists of four 16 byte-wide data rings, two of which run clockwise and the other two counter-clockwise. The EIB operates at half the processor clock speed, and exhibits a peak bandwidth equal to 204.8 Gbyte/s [11].

This speed is attainable only by intra-chip data transfers. For transfers involving main memory, the peak bandwidth is

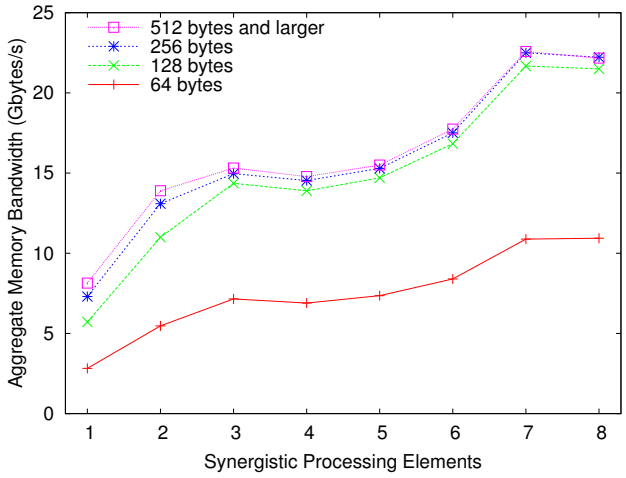

Figure 2. Available aggregate bandwidth from the main memory to the SPEs, at varying block size.

25.6 Gbyte/s. The actual achieved data bandwidth depends on a number of factors: mutual alignment of the source and destination addresses, interference with transfers already in progress, the number of Cell BE chips in the system, direction of the transfer, the efficiency of the data arbiter and, above all, the size of the transferred block. The programmer must transfer data in blocks large enough to amortize the bus negotiation overhead. Figure 2 shows that, in conditions of heavy traffic, bandwidth values close to the peak can be reached only when transferred block are at least 256 bytes or larger. In practice, programs should access the main memory at a medium-large granularities only.

3 Using Deterministic Finite Automata as String Acceptors

String matching is the core of network security systems like intrusion detectors, deep-inspection filters, spam filters and on-line virus scanners. A number of exact string matching algorithms are based on a Deterministic Finite Automaton (DFA) used as a language acceptor. DFA-based algorithms perform a state transition per individual input symbol consumed, and enter one of the final states when the input matches one of the words in the dictionary. DFA-based algorithms are very common in security applications, because their workload is content-independent, which makes them immune from overload attacks based on malicious contents.

A finite state acceptor is a quintuple $(\Sigma, S, s_0, \delta, F)$, where Σ is the input alphabet (a finite non empty set of symbols), S is a finite non empty set of states, $s_0 \in S$ is the *initial state*, δ is the state transition function and F is the set of final states, a (possibly empty) subset of S. Given a current state and an input symbol, the transition function yields a new state: $\delta : S \times \Sigma \rightarrow S$.

We consider each byte in the input stream as one input symbol. The DFA reads in the input one symbol at a time and performs a *state transition* according to the current state and the value of the input. If the destination state s is final ($s \in F$), then the current string is recognized. The acceptor enters a final state whenever a portion of the stream matches a word in the dictionary. In network security, this usually marks the detection of malicious contents. Therefore, DFAs should transit across non-final states the vast majority of time, whereas each final state should be associated with a type of malign content. Consequently, non-final to non-final state transitions should be considered the steady state of the system, and the case to optimize.

We express and compare the performance of different implementations of string matching algorithms in terms of throughput, i.e. the number of input symbols which are processed in the unit of time, which also measures the maximum number of state transitions possible in the unit of time. This quantity is consistent across implementations

Implementation Version	1	2	3	**4**	5
SIMD Vectorization	No	Yes	Yes	**Yes**	Yes
Loop Unroll Factor	–	–	2	**3**	4
Total cycles per action	311316	123976	90200	**82182**	91833
State transitions (= input block size)	16384	16384	16384	**16416**	16384
Clock cycles per DFA transition	19.00	7.57	5.51	**5.01**	5.61
Throughput (M transitions/s)	168.41	422.89	581.25	**639.21**	570.91
Throughput (Gbps)	1.35	3.38	4.65	**5.11**	4.57
Average CPI	2.60	0.67	0.63	**0.64**	0.62
Dual issue %	0.0	43.8	48.3	**48.7**	48.6
Stall %	63.2	7.4	0.0	**0.0**	0.6
Registers used	4	40	81	**124**	spill
Speedup	1.00	2.51	3.45	**3.79**	3.39

Table 1. The highest performance is obtained with SIMDization and accurate loop unrolling.

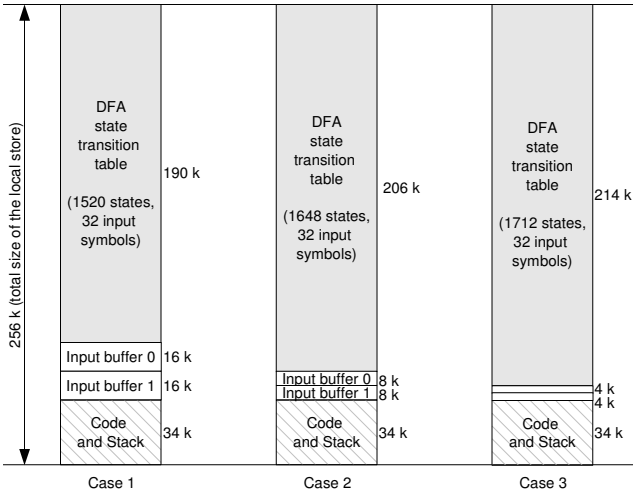

Figure 3. SPE Local store usage in our proposed implementation.

with different internal degrees of parallelism, which process a different number of streams concurrently. We express throughput values in terms of billion bits processed per second (Gbps) when we compare filtering throughput against the bitrate of a network link.

For the moment, we focus on a single SPE at a time. We call a *DFA tile* the implementation of a DFA acceptor realized on a single SPE, with a state transition table which fits the local store available in a SPE. A DFA tile alone can realize a string search against a dictionary compatible with the limit just introduced, at a given speed. If a larger dictionary is desired, or a higher performance is needed, multiple DFA tiles can be combined, as explained later.

4 Mapping DFA Acceptors onto the Cell BE: Implementation and Results

We now derive the maximum performance attainable by a single DFA tile. All the experiments and the measurements refer to implementations of the algorithm in C language, as described in this section, using the Cell BE intrinsics and language extensions, and compiled it with GNU GCC version 4.0.2. We have run the experiments on the a pre-production IBM DD3 Cell Blade. Profiling data come from the full-system simulator provided with the IBM Cell BE SDK version 1.1.

First, we describe the data structures we have chosen to represent the DFA. All our design choices are aimed at making the algorithm fit as much as possible the hardware characteristics of the SPE, thus maximizing its performance. The optimized representation of the DFA of our choice is as follows. We represent the State Transition Table (STT) as a data structure in the form of a complete table of words, having a row for each state and a column for each of the possible inputs. We represent the *current state* in the form of a pointer to the table row corresponding to that state. Experimental results show that a realistic upper bound for the number of states of a tile is between 1520 and 1712, see Figure 3.

In accordance with the requirements of most security applications whose filters do not need to be case-sensitive, we consider a restrained input range, containing 32 rather than 256 input symbol choices. For the sake of reduced memory footprint, we assume that an appropriate data-reduction strategy has been employed to fold the 0-255 character range into a smaller interval, e.g. the 32 values from 0x40 to 0x5F, which comprise the uppercase Latin alphabet plus other 6 characters. Such a data-reduction can be trivially implemented in an inexpensive way.

Appropriate data alignment allows us to reduce the computation associated with a state transition, and to compact the STT representation. In detail, we allocate the STT at an aligned location and choose an input set width which is a power of two. So, we can represent states by pointers to STT lines, and the last bits in these pointers are zero. Therefore, these last bits can be used to encode whether the next state is final, plus other frugal output values if needed.

We consider a bare-bone, purely sequential implementation of a DFA acceptor which complies with the above design choices, and counts the number of occurrences of dictionary entries in the given block of input data. This is sufficient in a large majority of application contexts, where a single match is sufficient to trigger further checks or to discard a network packet. This corresponds to "Implementation version 1" in Table 1. This code does not exploit efficiently the hardware features of the Cell BE: it does not use SIMD (single instruction multiple data) instructions and it uses only 4 out of the 127 available registers. An SPE contains two distinct pipelines which are able to issue two instructions at the same time, provided that they do not conflict. This implementation is unable to exploit this feature. As a result, the number of clock cycles per instruction (CPI) is high, 2.6, and the throughput is low, 1.35 Gbps.

We obtained a more efficient implementation by exploiting SIMD instructions to process in parallel the 16 bytes which compose a 128-bit word. A SIMD-ized implementation which processes 16 streams in parallel is 2.51 times faster than a sequential implementation which processes one stream (see Implementation Version 2 in Table 1). This implementation maintains 16 distinct DFAs, each working on a distinct input stream but sharing the same STT. The input streams are interleaved such that each quadword of the input (128 bit, 16 byte) contains at position i-th a byte from the i-th stream, which will be processed by the i-th DFA in the tile. Stream interleaving is a reasonably inexpensive operation, and can actually be mapped on the PPE, thus leaving all the 8 SPEs in a Cell BE available for other tasks.

The data-flow structure of this implementation is given in Figure 4. This implementation exploits better the registers (40 out of 127) and the two pipelines (43.8% of the issues are dual), reaching a lower CPI, 0.67. The compiler is still unable to reschedule instructions in such a way that no stalls occur (7.4% of the cycles are dependency stalls). To help the compiler in removing these stalls, we manual unroll instances of this data flow. We have unrolled it a different number of times to determine the optimal configuration (see Implementation versions 3–5 in Table 1). The optimal unroll factor is 3 (Implementation Version 4), which ensures almost complete register utilization (124 out of 127) without spills, a high dual issue rate and no stalls. To the best of our knowledge, this leads to the highest possible throughput

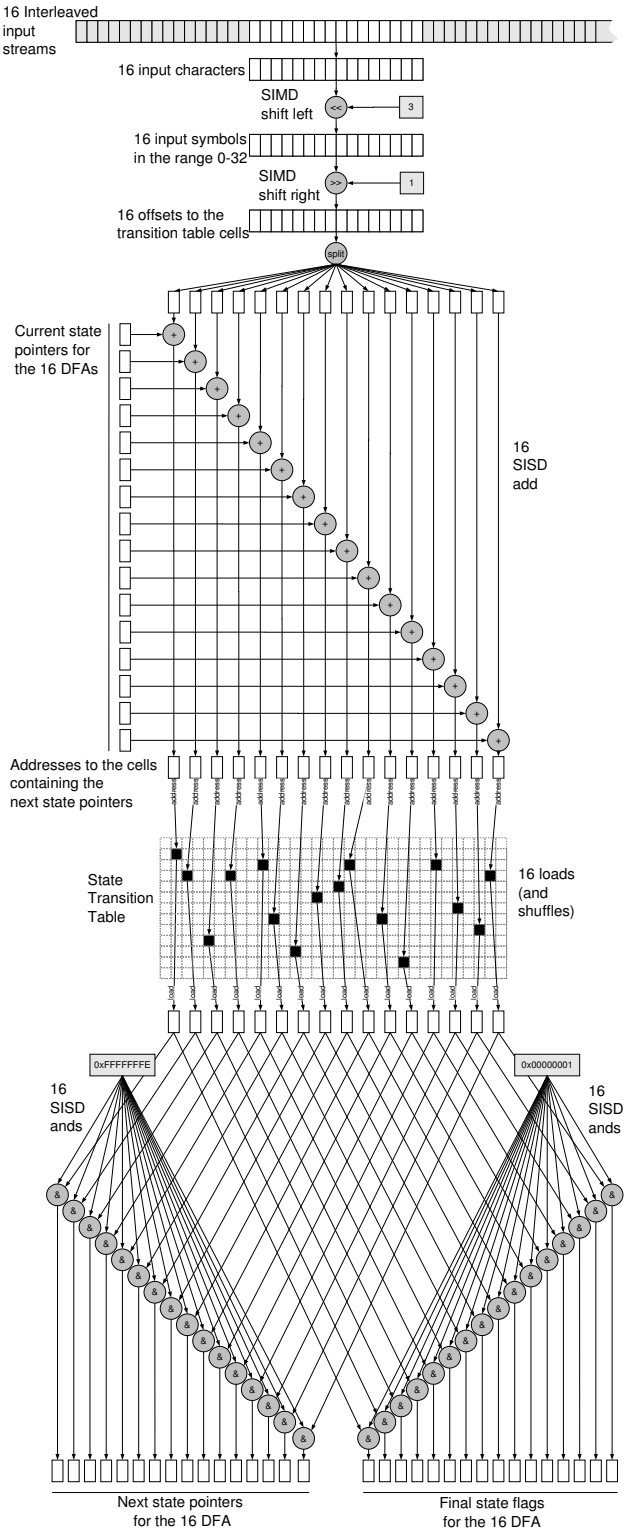

Figure 4. An optimal implementation of a DFA acceptor. The diagram shows which operations are SIMDized and which ones are not.

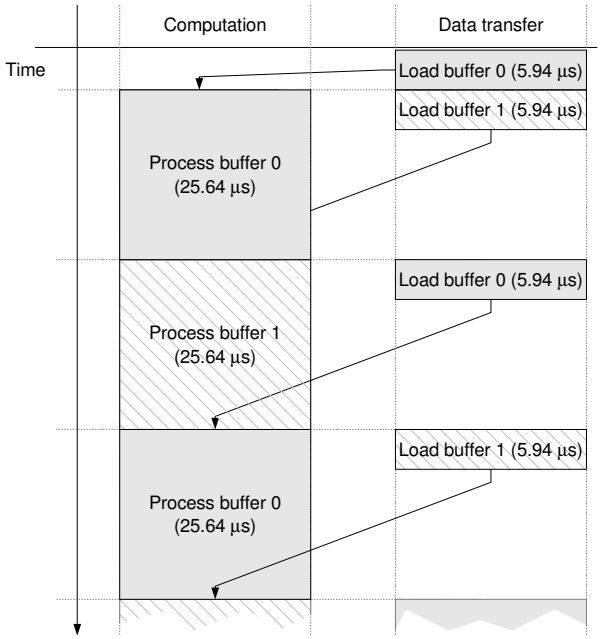

Figure 5. We hide the cost of data transfers by overlapping them with computation.

attainable by a single DFA tile, which is 5.11 Gbps.

For simplicity, so far we have not considered the overhead due to transferring new blocks of the input streams into the tile, i.e. into the SPE's local store. We now show that the latency associated with this transfer can be completely hidden by overlapping computation and communication. To do so, we adopt a double buffering technique for the input stream blocks, and transfer buffers with DMA commands. Computation can continue unaffected while DMA transfers are in progress. With the block sizes considered before (4–16 kbyte) computation always takes more time than data transfer. In the worst traffic conditions on the memory bus (all the SPEs accessing the memory at the same time), the available aggregate memory bandwidth is 22.05 Gbyte/s (see Figure 2), i.e. 2.76 Gbyte/s per SPE. Therefore, the time required to transfer a block of 16 kbyte is 5.94 μs, while the time required to process it is 25.64 μs. This leads to the schedule depicted in Figure 5: computation and data transfer can be overlapped in such a way that the cost of all data transfers (except the first one) is completely hidden. Values in the figure refer to an input data block size of 16 kbyte. The same considerations hold even when smaller block sizes are chosen, down to 512 bytes.

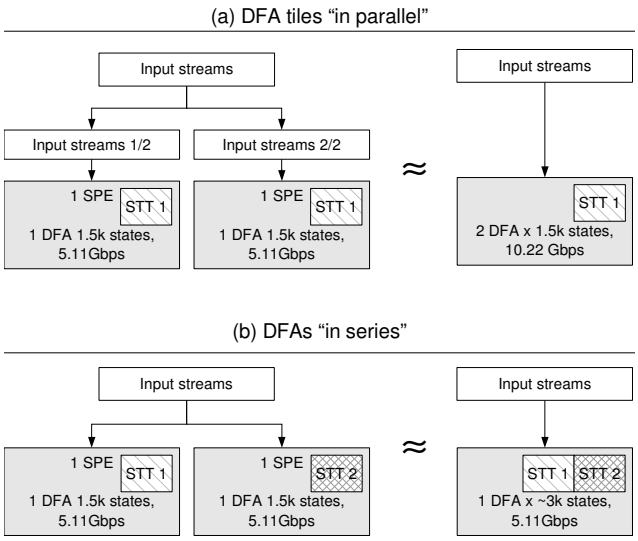

Figure 6. Composition "in series" and "in parallel" of DFA tiles.

5 Composing DFA Tiles to Increase Performance and Dictionary Size

In the previous section we have determined the maximum throughput attainable by a single DFA tile. But, depending on the application, the functionalities offered by a single tile may not be sufficient, either in terms of throughput, or in terms of dictionary size, or both. To address these needs, a system composed by multiple tiles can be implemented.

When more throughput is needed, multiple identical tiles can be used concurrently on distinct (with a minor overlapping) portions of the input streams. We say that two DFAs combined in this way are "in parallel", by analogy with an electric circuit (see Figure 6(a)). The two tiles have an identical STT and therefore recognize the same dictionary. Since they operate on two portions of the input streams which are separate (except for a small overlapping region, to allow matching of strings which cross the boundary), the combined throughput is effectively doubled.

The use of multiple tiles in a parallel configuration allows designers to effectively multiply the throughput. This is possible because string matching is an "embarrassingly parallel" problem, which does not need communication among processing elements, thus allowing for the use of an arbitrary large number of processors without incurring congestion issues. Mapping a DFA tile to each of the 8 SPEs in a Cell BE leads to a performance limit of $5.11 \times 8 = 40.88$ Gbps attainable by a single Cell BE processor, under the assumption that stream interleaving is performed by the PPE, and the remaining computational power of the PPE is

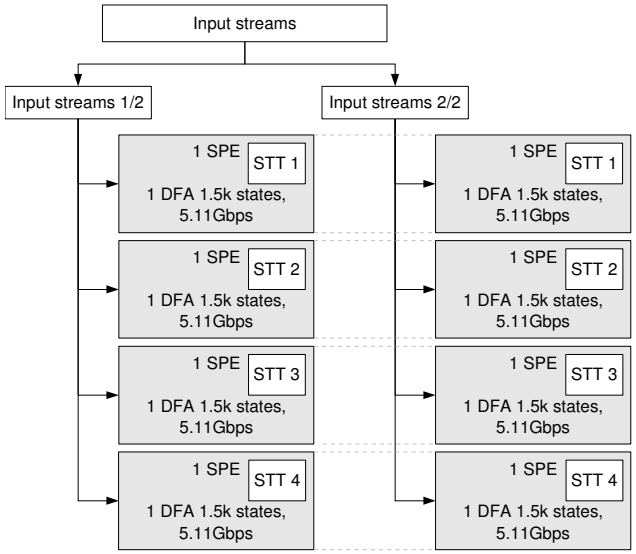

Figure 7. Example of a mixed series/parallel tile configuration.

sufficient to carry out the accessory tasks that the specific application demands. When an even higher performance is desired, multiple Cell processors can be used in parallel; e.g. a Cell Blade hosting two processors can reach 81.76 Gbps.

On the other hand, a state transition table of approximately 1500 states may not be sufficient to recognize the desired dictionary. To overcome this limitation, multiple DFA tiles can be combined in a series configuration, as in Figure 6(b). In this configuration, the tiles operate on the same input data at a given time, but each tile has a distinct STT. Each STT corresponds to just a portion of the dictionary.

To improve throughput and dictionary size at the same time, an application designer can adopt a mixed series/parallel configuration, in which groups of tiles operate on distinct portions of the input, while tiles share the same STT within a tile group, as depicted in Figure 7. The configuration in figure corresponds to an overall throughput equal to 10.22 Gbps, and a dictionary size which is roughly four times larger than the one which fits in a single tile.

6 Achieving Arbitrary Dictionary Size through Dynamic STT Replacement

If the space available for the state transition tables (STT) in the local storage of all the SPEs is not sufficient for the desired application, a different approach can be employed, which allows us to support virtually unlimited dictionary sizes, at the price of a smooth degradation in performance.

We call this approach dynamic STT replacement. The dictionary is partitioned into smaller subsets, each corresponding to an STT half as large as the one considered before, i.e. approximately 100 kbytes, which roughly correspond to 800 states. Each SPE contains now two STTs, which are managed in a double-buffering fashion. While one STT is used to filter the input, the other is used to load from main memory the next STT portion. The chain of DMA transfers required to load the STTs from main memory can be orchestrated to happen in the data-transfer idle time of the schedule of Figure 5. Each tile is assigned a number of STTs, and it will filter the input streams against each of those STTs, loading them cyclically one after the other.

In a configuration where an STT occupies 95 kbytes, a complete STT load can happen every two scheduling periods, as Figure 8 shows. We are assuming the steady-state condition in which the first STT is already loaded at the beginning of the periods shown. In the general case, if the dictionary required by the application requires n STTs, each SPE can now provide an effective bandwidth of $5.11/(2(n-1))$ Gbps. The $n-1$ term is due to the fact that, except for the very first period, an entire cycle through all the STTs requires $n-1$ transfers, rather than n. Figure 9 shows how the throughput provided by this approach varies when the number of STTs grows.

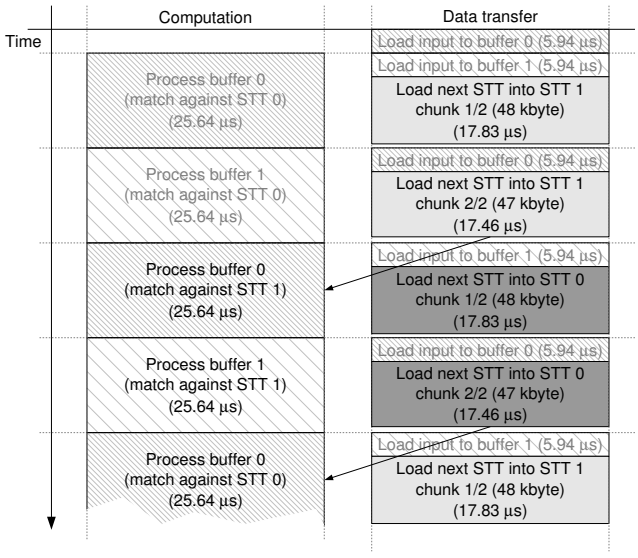

Figure 8. Schedule of a dynamic state transition table (STT) replacement.

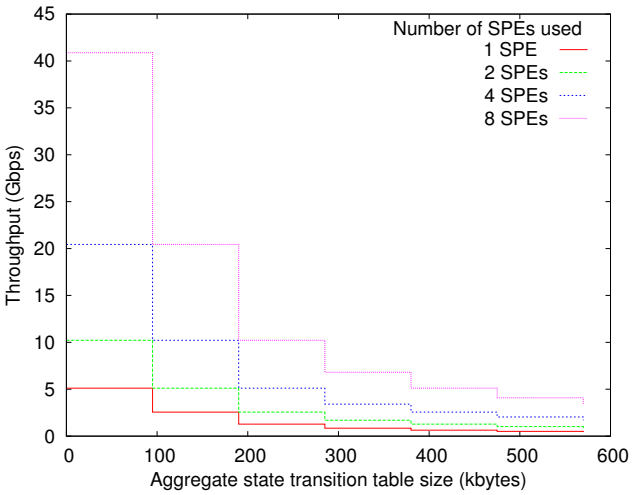

Figure 9. Throughput provided by dynamic STT replacement, when a variable number (1, 2, 4, 8) of tiles are employed.

7 Conclusions and Future Work

We have analyzed the suitability of the Cell BE architecture when employed to perform string matching of a stream of character against a given dictionary. We have identified a modular component, the *DFA tile*, which represents the optimal mapping of a finite state acceptor on a Cell SPE. A DFA tile can process input streams at a throughput up to 5.11 Gbps, with a state space comprising approximately 1500 states. We have shown how multiple tiles can be flexibly combined to achieve higher performance or to accept arbitrarily sized dictionaries. In summary, the Cell BE proves to be a fast and flexible architecture when applied in this domain.

Further work in this direction is being developed, also exploring the potentials of the Cell BE when implementing probabilistic string matching algorithms like Bloom filters.

References

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

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

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

[4] C. Chang and R. Paige. From regular expressions to DFAs using compressed NFAs. In *In Proc. 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.

[5] Y. H. Cho and W. H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 125–134, April 2004.

[6] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, London, UK, 1979. Springer-Verlag.

[7] 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.

[8] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 111, Washington, DC, USA, 2002. IEEE Computer Society.

[9] 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.

[10] T. Katashita, A. Maeda, K. Toda, and Y. Yamaguchi. Highly efficient string matching circuit for IDS with FPGA. *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, 0:285–286, 2006.

[11] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3), May/June 2006.

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

[13] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *Proceedings of the ACM Intl. Symposioum on Field Programmable Gate Arrays (FPGA 2001)*, pages 87–93, 2001.

[14] J. Moscola, J. Lockwood, R. Loui, and M. 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, Apr. 2003.

[15] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, April 2001.

[16] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion. In *Proceedings 13th Conference on Field Programmable Logic and Applications.*, September 2003.

[17] Y. Sugawara, M. Inaba, and K. Hiraki. Over 10 Gbps string matching mechanism for multi-stream packet scanning systems. *Lecture Notes in Computer Science, Field Programmable Logic and Application*, 3203/2004:484–493, 2004.

[18] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, May 1994.