Fast Pattern Matching on the Cell Broadband Engine™

Francesco Iorio
IBM Systems & Technology Group,
Dublin Software Laboratory,
Dublin, Ireland
francesco_iorio@ie.ibm.com

Jan van Lunteren
IBM Research, Zurich Research Laboratory
Säumerstrasse 4, CH-8803
Rüschlikon, Switzerland
jvl@zurich.ibm.com
(contact author)

Abstract

Pattern-matching algorithms, which are essential to intrusion detection and virus scanning applications, typically only make limited use of the Single Instruction Multiple Data (SIMD) capabilities available in new generations of general-purpose processors. This paper presents the initial results of a study to increase the SIMD exploitation by pattern-matching schemes consisting of a novel vectorized state-machine implementation that is able to utilize the vector-processing units in the Cell Broadband Engine almost fully for all its processing steps by storing most of the data structure in the large internal register sets. The implementation provides an extremely deterministic aggregate processing rate of 6.7 Gb/s for a single vector unit, which can be scaled up to 50 Gb/s for one Cell Broadband Engine and up to 100 Gb/s for a blade for small pattern sets. It also supports configurations in which the scan rate can be partially traded off for increasing the number of patterns supported.

Keywords: Pattern matching, finite-state machine, parallel algorithms, Cell Broadband Engine

1 Introduction

After initially having been employed mainly in supercomputers, SIMD techniques have found a growing application in general-purpose processors targeted at the desktop in the past decade, with examples including IBM®'s VMX, Intel®'s streaming SIMD extensions (SSE) and HP's Multimedia Acceleration eXtensions (MAX). These extensions have typically been used to accelerate applications such as video and image processing, that lend themselves very well for vectorization.

Pattern-matching functions also have become increasingly important in recent years, in particular because of their usage for intrusion detection and virus scanning applications. Many pattern-matching algorithms are based on finite-state machines (FSMs). In parallel FSM implementations, several essential processing steps, such as branches and memory accesses, depend on multiple independent input streams and therefor typically can only be performed in a serial fashion. Thus it is much more difficult to exploit the available SIMD capabilities.

In view of the above observation, we started to investigate the question whether it is possible to devise an optimized state-machine implementation that is able to exploit SIMD capabilities to a much larger extent than conventional implementations do. This effort has focused on creating a parallel implementation of a novel type of programmable state machine, called B-FSM [1], on a Synergistic Processing Element abbreviated as SPE, which is the vector-processing unit in the Cell Broadband Engine jointly developed by Sony, Toshiba and IBM.

In this paper, we present the first result of our work, namely, a parallel B-FSM implementation that, to the best of our knowledge, is the first to achieve a full vectorization of all processing steps, including the memory accesses. This resulted in very high utilization of the available execution units enabling high scan rates of several tens of gigabits per second. We achieved this by letting the B-FSMs execute directly out of the large SPE register sets. This initial implementation therefor limits the size of the executed state diagrams to a maximum of a few thousand B-FSM transition rules, for which the corresponding data structures fit entirely into the SPE register sets. The B-FSM transition rules, unlike conventional state transitions, also support wildcards and priorities and therefor can express match functions in a much more compact way. This appeared sufficient to support applications involving pattern sets with up to a few hundred patterns, such as tokenizers, as well as applications in which the pattern collection is constructed out of smaller subsets, for which the corresponding data structures can be swapped efficiently between the register sets and memory.

The paper is organized in the following way. Section 2 discusses related work on pattern-matching algorithms that try to exploit SIMD capabilities. Section 3 provides a short introduction of the Cell Broadband Engine and the SPEs. Section 4 introduces the B-FSM algorithm, which forms the core of our work. Section 5 describes the vectorized implementation of the B-FSM algorithm. The performance of this implementation is then evaluated in Section 6. Section 7 concludes the paper.

2 Related Work

While pattern-matching algorithms have already been studied for several decades, research in this field has been intensified in recent years because of their application for intrusion detection and other (network) security-related applications that have rapidly gained importance. This has resulted in a large number of publications on a wide spectrum of both hardware- and software-oriented schemes, with capabilities ranging from basic string-matching to complex regular-expression support. A selection can be found in [2]-[11].

Despite the large number of publications, very few address the exploitation of SIMD capabilities. To our knowledge, no FSM-based pattern-matching scheme has been published to date that intends to or has been able to apply an efficient vectorization of all processing steps, including the memory accesses.

The work that is closest to ours, is probably the string-matching scheme published by Scarpazza et al., which also targets the Cell Broadband Engine [12]. Their scheme is based on a data structure comprised of

fully expanded state tables that contain entries for all possible state and input combinations and are stored in the SPE's local store. Their scheme has been limited to 5-bit-encoded input characters, which only require 32-entry state tables for each state, so that it can support a reasonable number of states, about 1500, in the 256 KB local store. Each SPE processes 16 input streams in parallel, with the input interleaving being performed by the PowerPC® core. The corresponding address generation and state update operations are implemented in a parallel fashion. The actual accesses to the state tables in the local store, however, are performed in a serial fashion. They report a processing rate of 5.11 Gb/s per SPE.

A key difference of our work compared with the method by Scarpazza et al., is the compression that we apply by exploiting the B-FSM algorithm, which has been reported to improve the storage efficiency by a factor of 15 to 500 compared with conventional schemes [1]. This compression allows the creation of a substantially more compact data structure, enabling an efficient exploitation of smaller but faster memories, such as the SPE register set in our initial implementation presented below. This compression comes, however, at the cost of additional instructions as the B-FSM algorithm involves slightly more complex processing steps than the simpler state-table lookup operation of the scheme by Scarpazza et al. These instructions, however, could be vectorized efficiently, requiring relatively few extra cycles. The performance evaluation, which will be presented in Section 6, revealed that the performance gain we achieved thanks to the lower access latency to the vector register set clearly outweighed the additional instruction cycles, resulting in an overall processing rate that is higher than the one reported by Scarpazza.

Compared with our work, the scheme by Scarpazza et al. is able to support a larger state transition diagram and, consequently, a larger number of patterns. This is, however, for a large part also due to the 5-bit input encoding applied in their scheme. If the encoding were increased to a 7-bit encoding similar to our implementation (see Section 5), then the state-table size would increase by a factor of four, allowing only one fourth of approximately 1500 states to be stored. This is less than 400 states and comes much closer to what the B-FSM implementation is able to store in the vector register set.

In summary, both schemes have their own specific merits and, based on the characteristics of the match function, have their own specific application domain in which they achieve favorable performance results.

3 Overview of the Cell Broadband Engine

The Cell Broadband Engine [13] is a processor architecture designed in a joint venture between Sony, Toshiba and IBM to overcome the traditional limitations in high-throughput processing capability and the memory subsystems' incapability of sustaining the ever increasing pace of the processors' data-access demands. The solution to these fundamental obstacles to scalability was implemented in an architecture that encompasses several strategies to increase the overall data-processing capabilities while obtaining a better power efficiency.

The architecture consists of one 64-bit PowerPC core (PowerPC Processor Element or PPE), which is the overall system controller and runs the operating system and applications, eight independent vector-processing units (Synergistic Processor Elements or SPE [14]), which are specialized in compute-intensive SIMD applications, a high-bandwidth internal communication and data transfer network (Element Interconnect Bus or EIB), and a high-throughput memory controller (Memory Interface Controller or MIC), plus additional I/O devices. This is illustrated in Fig. 1.

The PPE is an in-order-execution, dual-issue, dual-threaded 64-bit variant of the PowerPC RISC processor family. It features 32 KB of Level 1 instruction cache, 32 KB of Level 1 data cache, plus an additional 512 KB of Level 2 cache, and VMX SIMD vector-processing extensions.

The Synergistic Processor Elements are very efficient 128-bit RISC vector processors dedicated to running processing and data-intensive workloads. Each of the eight SPEs in the Cell Broadband Engine features 256 KB of embedded software-managed local memory store, and a large (128-element) 128-bit vector register set. In addition, each SPE embeds a powerful memory controller (Memory Flow Controller or MFC),

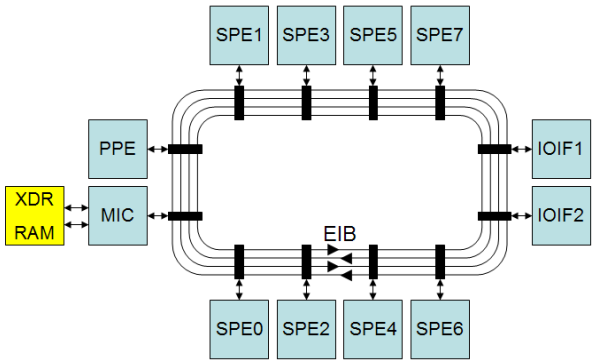


Figure 1: Cell Broadband Engine Overview.

whose main feature is the capability of performing asynchronous DMA data transfers between the main system memory and the SPE local store. The SPE processing core (SPU) has direct access only to its own local store for both instructions and data, thus a careful interleaving of data transfer and processing operations is crucial and is enabled by an appropriate choice of data buffering schemes.

To fully exploit the vast computing power of the Cell Broadband Engine, four levels of processing parallelism must be exploited when designing or porting algorithms:

1. Multi-core concurrent processing: each Cell Broadband Engine contains eight SPE cores, therefor algorithm-level parallelization is essential to distribute the computation on all resources available.

2. SIMD / vector processing: The SPEs are vector processors designed to operate on multiple operands with individual instructions. Accordingly, proper data structures organization and full usage of the instruction set are required to obtain optimal instruction level parallelism.

3. Pipeline optimization: Depending on the instruction type, the latencies of SPE instructions differ. Therefor, the best pipeline utilization depends on how data and results dependencies are masked with further computation to avoid empty instruction slots (pipeline stalls).

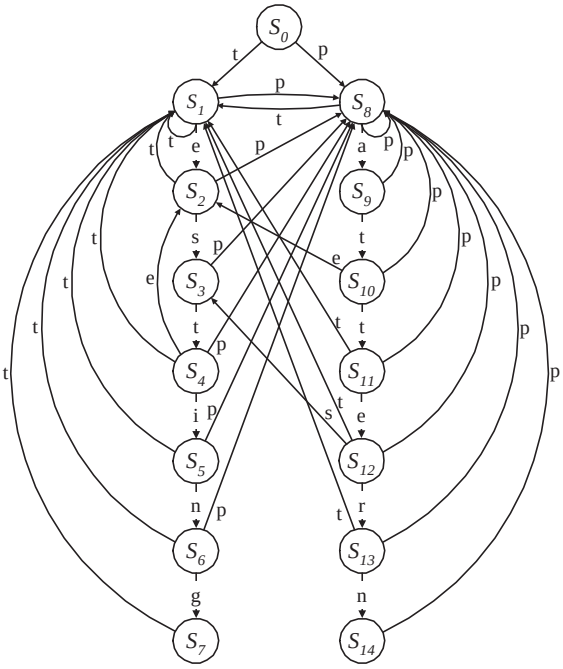
4. Dual-issue optimization: the SPEs are equipped with two execution pipelines, and are thus capable of executing two instructions per clock cycle, depending on the instruction types. Properly paired instructions can be executed simultaneously, so optimal cycle per instruction figures depend on appropriate use of instructions to exploit both pipelines.

4 B-FSM Technology

This section provides an introduction to the B-FSM technology, which forms the core of our pattern-matching work. It will focus on the various processing steps that are part of the basic B-FSM operation and that are subject for vectorization. For a more general and detailed description of the B-FSM technology and its application to pattern matching, including a description of the compiler that converts patterns into a B-FSM data structure, the reader is referred to [1].

4.1 Transition Rules with Wildcards

The B-FSM engine is a fast programmable state machine originally designed for hardware implementation. At the core of the B-FSM technology is the concept of specifying state transitions using so-called transition



("default" transitions to state S0 are not shown).

(a) State-transition diagram.

rule	current state	input		next state	priority
R_0	*	*	$\rightarrow$	S_0	0
R_1	*	t [74h]	$\rightarrow$	S_1	1
R_2	S_1	e [65h]	$\rightarrow$	S_2	2
R_3	S_2	s [73h]	$\rightarrow$	S_3	2
R_4	S_3	t [74h]	$\rightarrow$	S_4	2
R_5	S_4	i [69h]	$\rightarrow$	S_5	2
R_6	S_5	n [6Eh]	$\rightarrow$	S_6	2
R_7	S_6	g [67h]	$\rightarrow$	S_7	2
R_8	*	p [70h]	$\rightarrow$	S_8	1
R_9	S_8	a [61h]	$\rightarrow$	S_9	2
R_{10}	S_9	t [74h]	$\rightarrow$	S_{10}	2
R_{11}	S_{10}	t [74h]	$\rightarrow$	S_{11}	2
R_{12}	S_{11}	e [65h]	$\rightarrow$	S_{12}	2
R_{13}	S_{12}	r [72h]	$\rightarrow$	S_{13}	2
R_{14}	S_{13}	n [6Eh]	$\rightarrow$	S_{14}	2
R_{15}	S_4	e [65h]	$\rightarrow$	S_2	2
R_{16}	S_{10}	e [65h]	$\rightarrow$	S_2	2
R_{17}	S_{12}	s [73h]	$\rightarrow$	S_3	2

(b) State-transition rules.

Figure 2: Example of a match function.

rules. Each transition rule consist of a test part, containing exact-match and/or wildcard conditions for the current state and input values, and a result part, containing a next state and an optional output vector.

The transition-rule concept is illustrated in Fig. 2 using an example involving the simultaneous scanning of an input stream for all occurrences of two character strings "testing" and "pattern". Fig. 2(a) shows a

conventional state transition diagram that can be constructed for this match function using existing methods. An arrival in state S_7 or state S_{14} corresponds to the detection of the first or the second pattern, respectively. Note that the diagram in Fig. 2(a) is slightly simplified for illustrative purposes; the "default" state transitions to state S_0, which are taken if no other transition shown in the diagram can be used, have been omitted for clarity.

Fig. 2(b) shows a set of transition rules that can be used to describe the same match function and was derived as described in [1]. In this example, the input is assumed to be encoded as ASCII and the corresponding numerical values are listed in hexadecimal notation after the characters. The B-FSM implementation discussed below directly 'executes' this specification by searching in each cycle for the highest-priority transition rule that matches the actual values of the state register and input. It then uses the result part of that rule to update the state register and, optionally, to generate output. As can be seen from the example, wildcards allow the use of a single transition rule to describe multiple state transitions in the original state transition diagram (e.g., rules R_0, R_1 and R_8), enabling a more compact and flexible definition of the match function.

4.2 Transition Rule Selection

The B-FSM engine searches for the highest-priority matching transition rule in each clock cycle using a data structure that is comprised of multiple equally-sized so-called transition-rule tables, with each table corresponding to a particular cluster of states and containing all the transition rules related to the states in that cluster. Within a given cluster, all states are encoded using 'local' state vectors that are only unique within that cluster. Consequently, a state is identified 'globally' by the cluster identifier upon which it is mapped (typically the address of the corresponding transition-rule table is used for this) in combination with its 'local' state vector within that cluster. This information is contained in the B-FSM state register.

For each state a separate hash function is used to select one of the transition rules that apply to that state, based on the current input value. This hash function has been derived from the Balanced Routing table (BART) search algorithm [15], hence the name BART-based Finite-State Machine (B-FSM). The hash function is defined by a mask vector that specifies how the hash index bits are extracted from the local state and input vectors according to the following function:

$$index = (state \ \mathbf{and\ not}\ mask)\ \mathbf{or}\ (input\ \mathbf{and}\ mask), \tag{1}$$

where **and**, **or**, and **not** are bit-wise operators, and $state$ and $input$ contain the current values of the local state and input vectors. According to (1), each mask vector bit specifies whether a corresponding hash index bit is extracted from the state vector (mask bit equals zero) or from the input vector (mask bit equals one), enabling a very efficient and fast implementation.

In addition to the local state vector of the next state, the result part of each transition rule also stores the address of the transition-rule table containing the transition rules for that next state and the mask that specifies which hash function has to be used for selecting one of these transition rules in the following cycle based on the next input value.

Despite the simplicity of the above hash function, the B-FSM compiler is able to achieve a very efficient construction of the hash tables so that each transition rule occurs only once and most tables are fully occupied. This results in near-optimal storage efficiency for a wide range of applications and pattern sets. For example as reported in [1], a collection of about 2,000 patterns involving a total of 32K characters can be compiled into less than 128 KB for a small array of B-FSMs. This represents one of the most compact structures reported in the literature.

The compiler achieves this high storage efficiency by exploiting two optimization techniques. The first optimization involves the separation of transition rules that involve a wildcard condition for the current state (and hence are only dependent on the input value) and storing the result parts of these rules in a so-called default-rule table, which is being accessed based on the input value only. In this way, the transition-rule tables described above will only store transition rules involving exact-match conditions for both the current

state	vector	mask	table
S_0	00h	00h	0000h
S_1	08h	00h	0000h
S_2	01h	00h	0000h
S_3	09h	00h	0000h
S_4	04h	04h	0000h
S_5	0Dh	00h	0000h
S_6	0Ch	00h	0000h
S_7	06h	00h	0000h
S_8	0Bh	00h	0000h
S_9	0Ah	00h	0000h
S_{10}	03h	01h	0000h
S_{11}	05h	00h	0000h
S_{12}	07h	01h	0000h
S_{13}	0Eh	00h	0000h
S_{14}	02h	00h	0000h

(a) State encoding and masks.

input	result part
7Fh .. 75h	rule R_0
74h	rule R_1
73h .. 71h	rule R_0
70h	rule R_8
6Fh .. 00h	rule R_0

(b) Default-rule table.

index	transition rule
7Fh ..	
0Eh	rule R_{14}
0Dh	rule R_6
0Ch	rule R_7
0Bh	rule R_9
0Ah	rule R_{10}
09h	rule R_4
08h	rule R_2
07h	rule R_{17}
06h	rule R_{13}
05h	rule R_{12}
04h	rule R_{15}
03h	rule R_{16}
02h	rule R_{11}
01h	rule R_3
00h	rule R_5

(c) Transition-rule table.

Figure 3: Sample state encoding, default-rule table and transition-rule table contents.

state and input. Only when no matching rule can be found in the transition-rule memory, will a lookup be performed on the default-rule table. The second optimization involves an efficient approach for a combined state clustering, state encoding and hash function selection described in more detail in [1].

Fig. 3 illustrates an example of the compilation results for the transition rules shown in Fig. 2(b), while applying the two optimizations mentioned above. Fig. 3(a) shows the encoded state vectors and masks defining the hash functions for all states S_0 to S_{14}. Fig. 3(b) shows the contents of the default-rule table, assuming a 7-bit input value (ASCII encoding) which results in a total of 128 table entries, one for each input value. As can be verified in Fig. 2(b), the default-rule table implements the search for the highest-priority matching rule that has a wildcard condition for the current state, by a lookup on the input value. Fig. 3(c) shows the mapping of the remaining rules that involve exact-match conditions on both the current state and input on the transition-rule table.

The B-FSM operation based on the default-rule table and transition-rule table will now be illustrated using the following example, in which it is assumed that the B-FSM is in state S_4. As can be seen in Fig. 2, transitions can be made from state S_4 to five possible next states: with an input 'i' (69h) to state S_5 according to rule R_5; with an input 'e' (65h) to state S_2 according to rule R_{15}; with an input 't' (74h) to state S_1 according to rule R_1; with an input 'p' (70h) to state S_8 according to rule R_8, and with any other input to state S_0 according to rule R_0 (note that this transition to S_0 is not shown to keep the state diagram simple). These five cases are handled by the B-FSM in the following way. State S_4 is encoded using a local state vector '04h', and the transition-rule selection for this state is performed using a hash function defined by a mask '04h' (see Fig. 3(a)). If the input equals 'i' (69h), then according to (1) a hash index value '00' is calculated for the state vector '04h' and mask '04h'. At this index value, rule R_5 is retrieved from the current hash table, as shown in Fig. 3(c). The test part of this rule matches both the current state and input values, and consequently, the next state S_5 will be retrieved from its result part. Similarly, if the input equals 'e' (65h), then a hash index value '04h' will be calculated, and the matching rule R_{15} retrieved from the transition-rule table, providing a next state S_2. Any other input value will also result in a hash index value equal to either '00h' or '04h', but neither of the two rules will match the input value. Consequently, the default-rule table will be accessed. For input values equal to 't' (74h) or 'p' (70h), a transition to state S_1 or S_8, respectively, will be made. For all other input values, a transition to state S_0 will take place.

```
struct {
  unsigned int mask       : 7;
  unsigned int table_addr;
  unsigned int nxt_state  : 7;
  unsigned int res_flag   : 1;
} DefaultRuleMem[128];

struct {
  unsigned int cur_state  : 7;
  unsigned int input_val  : 7;
  unsigned int mask       : 7;
  unsigned int table_addr;
  unsigned int nxt_state  : 7;
  unsigned int res_flag   : 1;
} TransRuleMem[];

 unsigned int StateReg     = 0;
 unsigned int TableAddrReg = 0;
 unsigned int MaskReg      = 0;
 unsigned int InputVal;
```

(a) Variables.

```
/* address generation */
MemAddr  = ((StateReg & (MaskReg^0x7F)) |
            (InputVal & MaskReg));
MemAddr |= (TableAddrReg << 7);


/* rule selection */
if ((TransRuleMem[MemAddr].cur_state == StateReg) &&
    (TransRuleMem[MemAddr].input_val == InputVal)) {
  StateReg     = TransRuleMem[MemAddr].nxt_state;
  TableAddrReg = TransRuleMem[MemAddr].table_addr;
  MaskReg      = TransRuleMem[MemAddr].mask;
  ResultFlag   = TransRuleMem[MemAddr].res_flag;
}
else {
  StateReg     = DefaultRuleMem[InputVal].nxt_state;
  TableAddrReg = DefaultRuleMem[InputVal].table_addr;
  MaskReg      = DefaultRuleMem[InputVal].mask;
  ResultFlag   = DefaultRuleMem[InputVal].res_flag;
}
```

(b) B-FSM core loop.

Figure 4: Serial B-FSM implementation.

5 Vectorized B-FSM implementation

This section will present a vectorized implementation of the B-FSM algorithm described above which is able to exploit the capabilities of the Cell Broadband Engine and of other processors with similar SIMD capabilities. First, a serial B-FSM implementation in C will be presented, which will then be converted into a parallel SPE implementation that can scan 16 independent input streams simultaneously against a single set of patterns that is compiled into one B-FSM data structure that completely fits into the SPE register sets. The key aspect of the implementation, which enables the vectorization of all B-FSM steps, is how the data structure is mapped onto the SPE vector registers.

5.1 Serial B-FSM implementation

Fig. 4 shows the B-FSM core loop in C that implements the concepts described in Section 4. This code fragment involves 7-bit state, mask, and input vectors. It covers the B-FSM core loop, which consists of the following processing steps: First a memory address is generated by calculating a hash index according to (1), which forms the lower part of the address, whereas the table address forms the upper part. Next, the current values of the state register and input vector are compared with the corresponding fields in the transition rule contained at the location in the transition-rule memory that corresponds to the memory address calculated. If these two values match, then the next state, table address and mask are taken from the result part of that transition rule otherwise they are obtained by a lookup on the default-rule table indexed by the current input value. This loop is repeated for each new state and input value.

In this example, each transition rule includes a so-called result flag, which is set if that transition involves a next state that corresponds to a matching pattern found in the input stream. Upon the detection of a set result flag, the match function will perform a lookup on the table address and state vector to determine the pattern identifier. This lookup, however, will not be discussed in this paper. For more details, the reader is referred to [1]).

5.2 Data Structure

Each SPE contains a total of 128 vector registers, each 128 bits wide, corresponding to a total of 2 KB of storage. Eighty of these registers will be used to store one default rule table and two transition-rule tables,

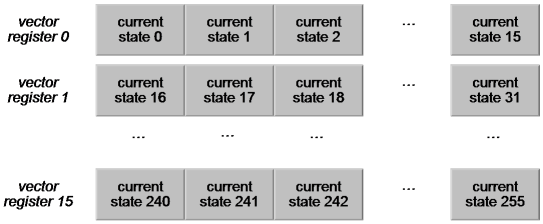


Figure 5: Sample data structure mapping.

```
#define SIMD_Lookup128(index, table, result)
lsb3_7 = spu_and(index, 0x1F);
tbl0_1 = spu_shuffle(table[0],table[1],lsb3_7);
tbl2_3 = spu_shuffle(table[2],table[3],lsb3_7);
tbl4_5 = spu_shuffle(table[4],table[5],lsb3_7);
tbl6_7 = spu_shuffle(table[6],table[7],lsb3_7);
bit2   = spu_cmpeq(spu_and(index,0x20),0x20);
tbl0_3 = spu_sel(tbl0_1,tbl2_3,bit2);
tbl4_7 = spu_sel(tbl4_5,tbl6_7,bit2);
bit1   = spu_cmpgt(index,0x3F);
result = spu_sel(tbl0_3,tbl4_7,bit1);
```

Figure 6: Selecting each of 16 bytes independently from any of 128 byte locations in 8 vector registers.

which can contain a maximum of $3 \times 128 = 384$ transition rules. The remaining 48 registers are sufficient for performing the various B-FSMs operations, including input interleaving. For this configuration, the table address field consists of a single bit.

Each transition rule vector has six fields, as shown in Fig. 4(a). Instead of storing the transition rules as an 'array of structures', they are stored as a 'structure of arrays', as is illustrated in Fig. 5 for the current state field: A block of 16 consecutive vector registers is used to store the 256 current-state fields of all the transition rules in the two transition-rule tables. The input, mask, and next-state fields are mapped in a similar way onto three other blocks of 16 consecutive vector registers, hereby the single-bit result-flag and table-address fields are packed at the most-significant bit positions together with the 7-bit mask and next-state fields, respectively. In this way, a total of 64 vector registers is used for storing the two transition-rule tables. The default-rule table is mapped in the same way. As shown in Fig. 4(a), each default-table entry contains only four fields, which can be combined into two packed fields as described above, so that the 128 default-table entries can be mapped onto two blocks of 8 vector registers. The mapping of the data structure on the SPE register set is illustrated in more detail in Fig. 7 in the Appendix.

5.3 Vectorized B-FSM Operation

Following the typical approach for vectorization, the current state, mask and input values for all 16 B-FSMs are mapped together onto three vector registers. Similarly, as done with the transition-rule tables discussed above, the single-bit table address of the current transition-rule table is packed with the 7-bit current state value at the most significant bit position.

The parallel implementation of the address generation function, described in Fig. 4(b), is realized using a single SPE instruction, `spu_sel`, which performs a mask-controlled bit selection from two vector registers, and thus directly implements the index calculation according to (1). By mapping the table-address bit at the most significant bit position with the current state vector and forcing the corresponding bit of the mask vector to be 'zero', the addition/concatenation of the table address (see Fig. 4(b)) is performed as part of the same instruction.

The parallel implementation of the rule selection function shown in Fig. 4(b) and described in Section 5.1, exploits the capabilities of the SPE `spu_shuffle` instruction to vectorize the 16 independent accesses to the data structure. This instruction allows each of the 16 bytes in the target vector register to be selected independently, from any of the 32 byte locations in two source vector registers, under control of a third vector register. By combining multiple `spu_shuffle` instructions with `spu_sel` and some compare instructions, it is possible to increase the number of source bytes from which the 16 target bytes can be selected. Fig. 6 illustrates a code fragment that allows the 16 bytes to be independently selected from 128 different byte locations in a total of 8 vector registers. A graphic representation is provided in Fig. 8 in the Appendix.

With the data structure being organized as described in Section 5.2, this flexible byte selection can now be used to fetch the various fields of the selected transition rules (also using a '16 out of 256 byte' selection function) under control of the calculated addresses for all 16 B-FSMs in parallel. In a similar way, the default rule table can be accessed in parallel for all 16 B-FSMs (using a '16 out of 128 byte' selection function, as shown in Fig. 6) under control of the current input value (see Fig. 4(b)).

The 16 state and input fields of the transition rules selected are grouped into two vector registers, and compared against the vector registers containing the actual state and input values by means of a simple compare instruction. The compare-result vector is then used to control a `spu_sel` instruction that will select whether the state, mask, and table address values will be updated from the transition rule selected if it matches or from the default rule table entry otherwise.

5.4 Input and Result Processing

Because of the parallel processing of 16 input streams in vector form (16 elements of 8 bits each, forming a 128-bit word) the input data must be interleaved, as each element in the vector word represents an input datum sourced from a different stream. If the PPE is used for the input-data stream interleaving task and thus simultaneously serves multiple SPEs, then there is a high probability for the interleaving process to become a throughput bottleneck, as the PPE would be required to perform data reads, shuffles, and writes at a data rate sufficient to feed all 8 SPEs data streams, while concurrently running OS facilities and network processing. For this reason we implemented two versions of the B-FSM loop, one that performs data interleaving directly on the SPE and one that assumes the input data is pre-interleaved by an external process, as described in relevant related literature [12]. Both implementations load input data from the main memory by means of DMA transfers to the SPE local store in blocks of 256 elements per stream for a total of $256 \times 16 = 4096$ bytes. They exploit a double buffering scheme to hide data transfer latency and all DMA transfers are SPE-initiated. The first implementation, which includes data interleaving, uses DMA lists to fetch blocks of 256 bytes from 16 different sources in the main memory, as in this case the input data streams are assumed to reside in separate memory locations. The second implementation, which does not include data interleaving, uses single DMA transfer commands to fetch blocks of 4096 bytes of pre-interleaved contiguous data from the main memory. The result flags generated by each B-FSM processing step are stored in an appropriate memory area in the SPE's local store, and can be transferred to the main memory for further use. This process incurs only in a very minimal penalty: Note that our current implementation does not move the results data to the main memory.

6 Performance Evaluation

6.1 Experimental Setup

The software was developed using the C language with specific language extensions and exploiting the IBM Cell SDK v2.1 gcc compiler and tools [16]. Experimentation and performance measurements were performed on a IBM BladeCenter® QS21 blade server running at 3.2 GHz [17]. Profiling information for tuning was collected using a combination of the IBM Cell Broadband Engine Full System Simulator and the IBM Assembly Visualizer for the Cell Broadband Engine [18].

6.2 Experimental Results

The version of the B-FSM implementation that includes input interleaving consists of two nested loops. The input stream interleaving process resides outside of the B-FSM core loop and processes 16 bytes from each plain input stream into an interleaved block of 256 bytes, which correspond to the 16 input elements for each of the 16 input streams. This operation was measured to require 105 clock cycles.

Table 1: Measured performance results for a single SPE.

	With data stream interleaving	Without data stream interleaving
Avg. clock cycles per state transition	4.22	3.82
Throughput (M state transitions/sec.)	756.55	836.89
Throughput (Gb/s)	6.1	6.7
B-FSM core loop CPI	0.59	0.57
B-FSM core loop dual issue	68.5	75.8
B-FSM core loop stalls	0.0	0.0
Registers used	126	110

By using static pipeline analysis we could infer that the B-FSM core loop, which operates on all 16 data streams in parallel, consists of 57 instructions in the even and 44 in the odd pipeline, requiring a total of 58 clock cycles, taking into account dual issued instructions. This corresponds to a theoretical peak performance of 3.65 (58/16) clock cycles per individual state transition made by each of the 16 B-FSMs.

Table 1 provides details on the performance measured for a single SPE: the version without input interleaving achieves a throughput of 6.7 Gb/s and has performance characteristics close to the theoretical peak, as it consists of the B-FSM core loop and only some minimal outer loop structures. The version with input interleaving needs to perform the input processing outside of the core loop, thus incurring a penalty, resulting in a slightly reduced throughput of 6.05 Gb/s. In both versions no loop unrolling of the B-FSM core loop was performed because of limitations in the current compiler register allocation policy that generates unwanted spills of values to memory, which degrades the overall performance.

The input stream interleaving code, which is outside of the core loop, runs entirely on the odd pipeline and takes a total of 105 cycles to execute. As there are 13 unused odd pipeline instruction slots per step in the core loop, the entire input interleaving process could be optimized manually to fit into the unused pipeline slots by employing a fully unrolled core loop (16 times), thus obtaining the full 6.7 Gb/s throughput including stream interleaving.

An interesting property of the B-FSM implementation is that all possible execution paths in the code take the same number of cycles, rendering the above performance numbers deterministic and completely independent of the characteristics of the input stream and the patterns.

All eight SPEs in the Cell Broadband Engine were operating in parallel, each scanning a set of 16 input streams at a rate of 6.7 Gb/s against the transition diagram stored in its vector register set. As part of the experiments, various configurations were tested that involved different allocations of input streams to these eight SPEs, allowing the aggregate scan rate and number of transitions (and patterns) to be scaled in a flexible way. In one extreme configuration, all 8 SPEs were operating on 8 different sets of 16 input streams, corresponding to a total of 128 independent input streams, scanning each set against a state diagram consisting of up to 384 transition rules (256 regular rules and 128 default rules). This resulted in a total measured scan rate of over 50 Gb/s. In the other extreme configuration, all 8 SPEs were operating on the same set of 16 input streams, scanning these at a total scan rate of 6.7 Gb/s against a distributed state diagram that has up to $8 \times 384 = 3072$ transition rules (2048 regular rules and 1024 default rules). Other configurations allow other combinations of aggregate scan rate (in steps of 6.7 Gb/s) and number of transition rules between these extremes, e.g., a scan rate of 13 Gb/s against up to 1536 transition rules, a scan rate of 26 Gb/s against up to 768 transition rules, and so on.

Because the process is scalable, more than one processor can be used in parallel. The QS21 Blade used in the experiments has two Cell Broadband Engine processors, which allowed a further scaling of the scan rate up to 100 Gb/s (which was measured) or the supported number of transition rules to be increased up to a total of 6144.

An interesting feature is that the B-FSM data structure in each SPE only uses a total of 80 vector registers, which can be loaded from the SPE's local store in 85 cycles. This allows a rapid switching between different pattern sets for which the corresponding compiled B-FSM data structures have been pre-stored in the local store. With the local store in each SPE being 256 KB, it allows 200 different B-FSM data structures to be stored, which consist of up to 76K transitions rules per SPE and over 600K transition rules per Cell Broadband Engine. This is, of course, particularly useful for match applications for which the pattern set is organized into several smaller subsets, against which the input streams need to be scanned selectively.

Various pattern sets and input traces were used during the experiments to verify the correct operation of the match function. As already indicated in the introduction, it is important to note that the B-FSM transition rules mentioned here are different from conventional state transitions, because they support wildcard conditions and priorities, allowing a more compact representation of match functions. As a result, the B-FSM compiler was able to compile a few tens of patterns into the register set of each SPE, i.e., a few hundred patterns for all 8 SPEs. The actual number of patterns that fit into the register sets depends on the pattern characteristics which determine how well they can be mapped onto a set of B-FSM transition rules. This topic, however, is beyond the scope of this paper, but has been addressed in [1]. Note that by means of a selective pattern-distribution function as described in [1] the compiler is able to improve the storage efficiency further for configurations in which multiple SPEs operate on the same set of input streams.

7 Conclusion

This paper has presented a novel parallel implementation of the B-FSM algorithm on the Cell Broadband Engine, which, to our knowledge, is one of the first fully vectorized implementations of a state machine involving parallel memory accesses. This was achieved by storing the main data structures directly in the SPE register sets, and by a particular organization of these structures that made it possible to parallelize the accesses from the 16 B-FSMs executed by each SPE.

A key feature of this implementation is that the processing rate is extremely deterministic and independent of the input stream and the pattern characteristics. Each SPE achieved an aggregate scan rate of 6.7 Gb/s for match functions specified by up to a few hundred B-FSM transition rules, which support wildcards and priorities. The aggregate scan rate could be scaled to a measured value of over 50 Gb/s when using all 8 SPEs in the Cell Broadband Engine to simultaneously scan 128 independent streams, and to over 100 Gb/s for a blade containing two Cell Broadband Engines when scanning 256 independent streams. Alternatively, multiple SPEs could also be allocated to scan the same set of input streams, enabling one to scale the number of transition rules rather than the aggregate scan rate, which resulted in an increase of the number of patterns supported.

The results also indicate that the additional complexity the B-FSM algorithm incurs compared with conventional schemes based on a 'simple' next-state table lookup, only results in a relatively small number of extra instruction cycles because of an effective parallel implementation of the B-FSM core loop. Furthermore, these extra cycles are completely compensated by the higher storage efficiency obtained in this way, which allows the efficient exploitation of smaller and much faster memories (in this case the SPE register sets) to realize very high processing rates. A further advantage is that this also allows one to switch efficiently between multiple match functions for which the B-FSM data structures are stored in the local store.

The work presented here was the first result of an investigation into the efficient exploitation of SIMD capabilities for pattern matching. Ongoing work is targeted at scaling towards substantially larger pattern sets, thereby applying the experience gained from this first implementation. A second research topic is directed at extensions for regular expressions, in particular storage-efficient support for character classes.

References

[1] J. van Lunteren, "High-performance pattern-matching for intrusion detection," *Proc. IEEE INFOCOM*, Barcelona, Spain, April 2006.

[2] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.

[3] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, Oct. 1977.

[4] B. Commentz-Walter, "A string matching algorithm fast on the average," *Proc. of the 6th Colloquium, on Automata, Languages and Programming*, pp. 118-132, July 1979.

[5] 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.

[6] C. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection," *Proc. of the DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.

[7] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *Proc. IEEE Infocom*, vol. 4, pp. 2628-2639, March 2004.

[8] R. Sidhu and V.K. Prasanna, "Fast regular expression matching using FPGAs," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 227-238, 2001.

[9] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111-120, April 2002.

[10] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 249-257, April 2004.

[11] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 21-23, April 2004.

[12] D.P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the Cell processor," Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1-8, March 2007.

[13] 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*, pp. 589-604, July/September 2005.

[14] B. Flachs et al., "The Microarchitecture of the Streaming Processor for a CELL Processor," *Proc. IEEE International Solid-State Circuits Symposium*, pp. 184-185, February 2005.

[15] J. van Lunteren, "Searching very large routing tables in wide embedded memory," *Proc. IEEE Globecom*, vol. 3, pp. 1615-1619, November 2001.

[16] http://www.ibm.com/developerworks/power/cell/

[17] http://www-03.ibm.com/systems/bladecenter/
hardware/servers/qs21/index.html

[18] http://w3.alphaworks.ibm.com/tech/asmvis

A Appendix

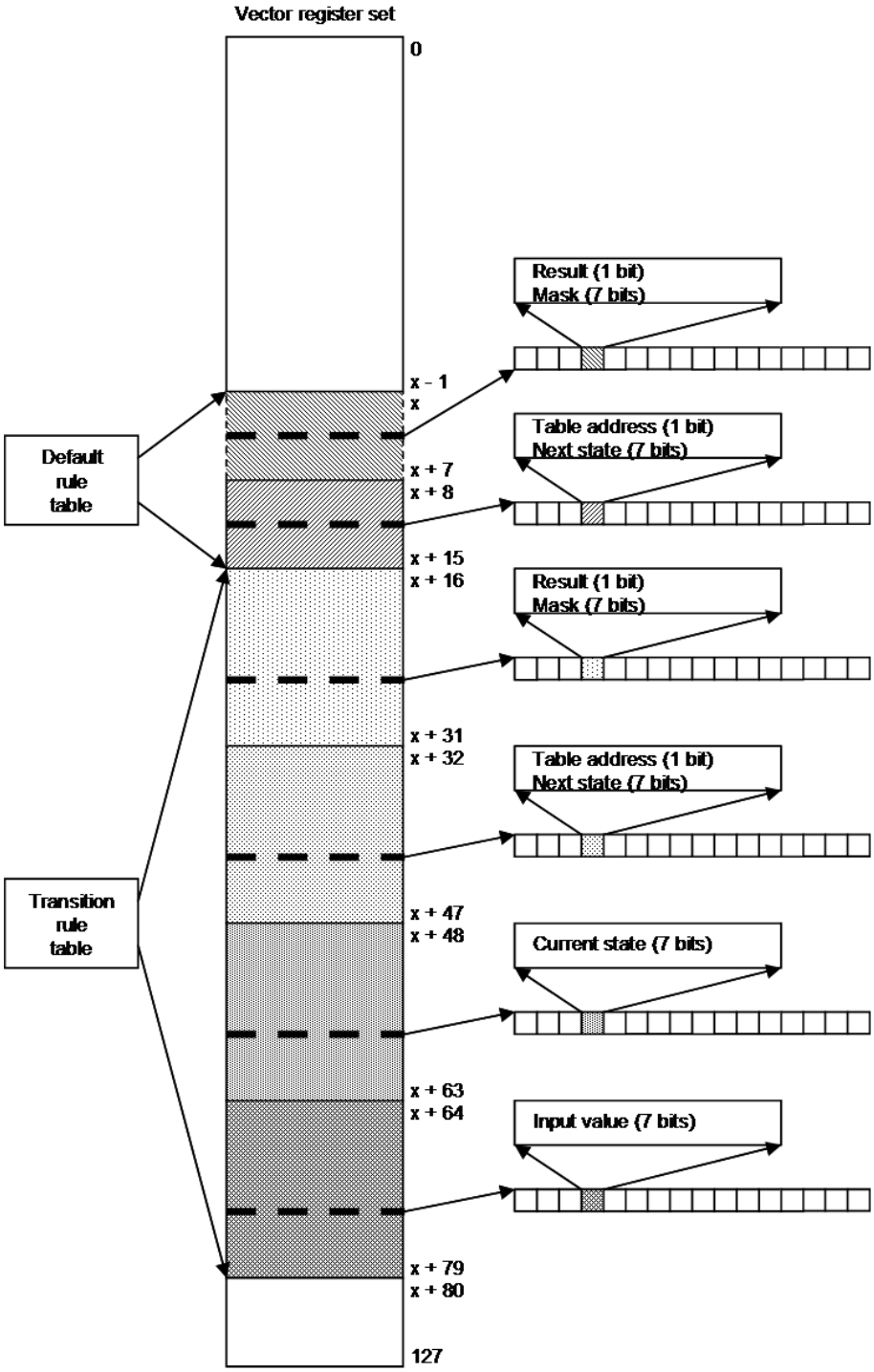


Figure 7: SPE register allocation.

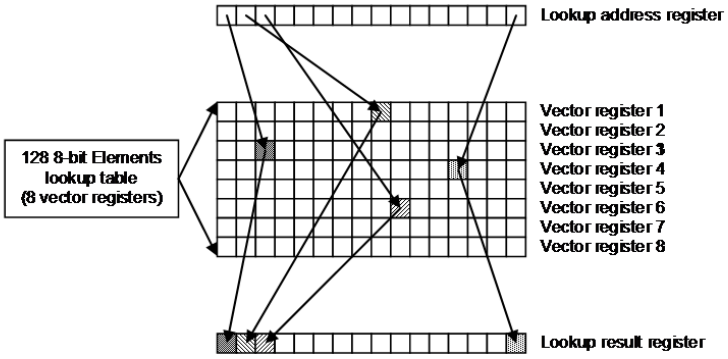


Figure 8: Flexible lookup.