RC24428 (W0711-152) November 19, 2007
Computer Science

IBM Research Report

Fast String Search on the Cell Processor

Daniele Paolo Scarpazza
Pacific Northwest National Laboratory

Oreste Villa
Politechnico di Milano

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

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

Last modification on 06/25/07 at 11:07:19 AM.

Fast string search on the Cell Processor

Your latest PC is probably a dual core or a quad core, but the one which will sit on your desk in a couple of years will be a “many-core”, like the Cell processor by Sony, Toshiba and IBM, which has 9 cores, or the Intel Terascale prototype, which has 80. Many-cores are the future, and since they are so different to program from traditional processors, programmers must get ready for the change now, e.g. learning how to map basic algorithms efficiently onto the new hardware.

String searching is one of these basic algorithms, and it has a host of applications: search engines, network intrusion detection systems, virus scanners, spam filters, DNA analysis and many others. The Cell processor, with its multiple cores, promises to speed it up a lot.

In this article we show how we mapped it efficiently on the Cell. We present two implementations. The “fast” one supports a small dictionary size (100 patterns approximately) and provides a throughput of 40 Gbps, which is 100 times faster than reference implementations on x86 architectures. The “heavy-duty” implementation is slower (3.3 - 4.3 Gbps) but it supports dictionaries with tens of thousands of strings –and beyond, depending on the available RAM.

This task is not trivial: we had to change our algorithm significantly to reach top performance. In particular, to exploit the memory subsystem at its best, we employ a pipelined parallelization strategy, and we shuffle the data layout to fight congestion: these techniques are quite unfamiliar to most programmers of traditional architectures. The details follow.

Why is string searching so important?

The Internet is a dirty place, with malware, spyware, spam and viruses trying to penetrate your systems through all possible vulnerabilities. Undesired traffic cannot be filtered anymore on the basis of header information. “Deep packet inspection” is needed, i.e. checking the payload against a database containing signatures of a large number of threats.

With network links getting faster and faster (e.g. 10 Gbps Ethernet), performing deep packet inspection on the fly is not child's play. For example, running Snort (a popular intrusion detection system) on a Pentium 4 PC only ensures a filtering

throughput between 200 and 400 Mbps. This is why network appliance manufacturers often employ specialized hardware like FPGAs (Field-Programmable Gate Arrays) or ASIPs (Application-Specific Instruction Processors). The Cell processor is a new player in this field, with the potential to offer a lot of computing power at low price. In fact, its large production volumes (it appears in the Sony PlayStation 3) contribute to keeping its price low.

Programming the Cell: rethinking "bottom-up"

The Cell is so different from traditional processors that you can't get the best out of it unless your code explicitly takes advantage of its hardware. The Cell contains 9 processors: one Power Processing Element (PPE), which is a 64-bit PowerPC-like processor, mainly used to run the operating system and load the tasks; and 8 worker elements, called Synergistic Processing Elements (SPEs).

SPEs have 128-bit wide registers and SIMD (Single Instruction Multiple Data) instructions. But they have a very simple branch prediction, so jumps are expensive and your code should avoid them, possibly replacing them with speculative execution. Additionally, SPEs cannot access the RAM directly, and they have no cache. Instead they have a small 256 kbyte memory called "local store" (LS), and they transfer data to and from main memory with asynchronous DMA transfers. By cleverly scheduling the DMA transfers in your code (e.g. with double buffering), you can hide part or all of the transfer latency. We gave a primer on these techniques (double buffering, software speculation, SIMDization) in our article "Programming the Cell", appeared in the April issue.

These hardware features revolutionize the way we write code. We have been told since "Programming 101" to develop code in a top-down fashion. Now, on the Cell, we are quite forced to look down the bottom and make both ends meet: hardware constraints strongly influence the algorithm.

Searching strings: one problem, too many solutions

Whether you are scanning files for viruses, performing a full-text search on a large collection of articles, or detecting intrusions on network traffic, you are doing string searching. Given a *text* and a *dictionary* (a set of patterns), your task is to find all occurrences of any of the patterns in the text.

String searching is a beaten track, with many proposed algorithms in the

literature. The one we employ, Aho-Corasick, is famous for appearing in the original "fgrep" UNIX utility. Despite its respectable age (it was presented in 1975), this algorithm is very simple and appears more suitable to the Cell than other algorithms (Knuth-Morris-Pratt, Boyer-Moore, Commentz-Walter, Wu-Manber, ...). In fact, Aho-Corasick can be expressed well as a branchless data flow that best exploits the SIMD capabilities of the SPEs. On the other hand, its evolutions have more complex control flows, which suffer from frequent and large branch misprediction penalties when run on an SPE.

The variant of Aho-Corasick we employ is based on a Deterministic Finite Automaton (a DFA), and it is fast for two reasons. First, the DFA is based on a keyword tree (a.k.a., a *trie*): a trie has a path of labeled edges per each pattern in the dictionary. To process an input character, you just need to choose the outgoing edge which is labeled with that character, and jump to the following node along that edge. Secondly, in a DFA no special rollback actions are required in case of a mismatch: a non-matching input character triggers a simple state transition, like any other character. These claims are better illustrated in the box [Note to editor: please insert reference; contents of box are supplied separately in file "box.odt"].

Why Aho-Corasick and the Cell make a good match

The DFA-based Aho-Corasick suits the Cell very well. In fact, processors with SIMD instructions (like the SPE) offer a lot of data-level parallelism which you can exploit by running multiple DFAs together. Moreover, the many (128) registers available allow extensive loop unrolling and code replication.

A single instance of a DFA is not sufficient to satisfy this "hunger for parallelism". The most natural approach to parallelize the algorithm (thus multiplying throughput) is to adopt multiple instances of the automaton. All the instances will act as the same automaton (i.e. they share the same state transition table), but they will be provided with distinct chunks of the input text. This configuration is portrayed in Figure 5. Note that the chunks are partially overlapped, otherwise occurrences of a pattern which appear across a boundary would not be matched.

Since a DFA does exactly one state transition per consumed input character, multiple DFAs are naturally synchronized: they start and finish processing a given quantity of text at the same time. So, they can share a single program control structure. This is a great advantage: we can fuse multiple DFAs in a single, long basic block. Long basic blocks are crucial for performance on the SPEs: they mitigate branch overhead and give the compiler enough elbow room during instruction scheduling. This leads to

reduced stalls, hiding of load/store latencies, better dual pipeline exploitation and lower clock Cycles Per Instruction (CPI) values.

Our “fast” solution

The smaller the dictionary, the smaller the corresponding State Transition Table (STT). If the STT is small enough, we can keep it entirely in the local store rather than in main memory. This speeds up the algorithm a lot, because main memory accesses are slow (and we will see how much, in the next section). Our “fast” implementation is based on the simple idea of keeping the STT in the local store.

As we anticipated, a single automaton utilizes poorly the SPE hardware. Its implementation uses only 4 registers out of the 127 available, and no SIMD instructions. Moreover, an SPE has two pipelines and could issue two instructions at the same time if they do not conflict, but this implementation does not exploit this feature. As a result, the CPI is high, 2.6, and the throughput is low, 1.35 Gbps.

But if we fuse together 16 automata, things improve a lot, because with SIMD instructions our code can process data from 4 automata at the same time. This implementation exploits better the registers (40 out of 127) and the two pipelines (43.8% of the issues are dual), reaching a CPI of 0.67.

This is a large improvement, but the code still stalls 7.4% of the cycles. To get rid of this stalls, we unroll the code manually, creating longer basic blocks. This gives the compiler more freedom to reschedule instructions and fill the stalls. We found experimentally that the best conditions happen when we unroll the main loop 3 times. The resulting code uses almost all the registers (124 out of 127) without spilling, it has a high dual issue rate and no stalls. This leads to a remarkable throughput, 5.11 Gbps.

On a side, loading text from memory is not in the way: we adopt a double buffering scheme to load the next chunk while the current one is processed. This hides completely the load latency, because transfers happen asynchronously and take less time then processing. In fact, loading a 16-kbyte text chunk takes 5.94 microseconds, whereas processing it takes 25.64 microseconds.

All this happens on a single SPE. To unleash all of the Cell's power, the remaining 7 SPEs can be put to work as well. We can do this in two ways: in parallel or in series. When they operate in parallel, all the SPEs have identical STTs, but they process distinct chunks of input text. This multiplies by 8 the combined throughput, which is 40 Gbps. Alternatively we can put SPEs “in series”: the SPEs are all given the same chunk of input text, but they employ distinct STTs. Each SPE has an STT which represents a portion of the initial dictionary. This way, the available dictionary capacity is multiplied

by a factor 8.

Thanks to the regularity of the DFA and to the absence of variable-latency operations, the throughput values reported above are independent from the input. This is highly desired in security applications, because it means immunity to content-based attacks.

The main drawback is that the STT must be small enough to fit the local store. Ranging between 190 and 214 kb in size, each STT will contain between 1520 and 1712 states, assuming an alphabet of 32 characters. Such an STT may encode dictionaries of approximately 150 strings, of average length 10 (more, if many prefixes are common).

The "heavy-duty" solution

The above solution is fast but lacks capacity. Even in the "series" configuration, it won't support a dictionary large than 1,200 patterns. Additionally, it supports only 32 input symbols, which practically limits us to case-insensitive English text matching only. Instead, popular rule sets available for Snort comprise about 5,000 rules, including both text and binary patterns, with a larger average length than 10 characters. And since the bad guys keep themselves busy, this rule set is expected to grow quickly in the future.

A solution with more dictionary capacity is needed, therefore we must move the STT in a roomier place: main memory. As a drawback, accessing the STT costs more: each state transition requires a DMA transfer. We can still cache some frequently hit states in the LS, but much optimization effort needs to be spent on accessing the main memory, where the vast majority of the STT resides. We must rethink the algorithm "bottom up" around this goal: making DMA accesses as fast as possible.

Automata spend their whole life performing state transitions. This breaks down in two phases: computation and data-transfer. Computation means determining the address of the next state in the STT, depending on current state and input. Data-transfer means getting data from that address. Since the STT is now in main memory, the data transfer takes much more than computation: 13 nanoseconds to compute and 250 to transfer.

Again, a single automaton yields poor utilization of both the processing power and of the memory subsystem (see Figure 6). And again, we employ multiple automata to improve this. In our code, we statically schedule these automata to run cyclically: the first automaton waits for its transfer to complete, computes the transition and start the next data transfer, then the second automaton will do the same. Then the

third, the fourth, and so on, up to the last one. At this point, another cycle begins, with the first automaton running again. Figure 7 illustrates this pipelining scheme. We put so much work between two runs of the same automaton, so that we can exploit the DMA latency time to do something useful.

In this approach, the bottleneck is the time elapsed between the completion of two subsequent DMA transfers, the "memory gap". A new state transition can be completed in no less time than the memory gap. We use 16 automata because benchmarks tell us that the best memory gap is experienced when at least 16 concurrent transfers happen at any time. Benchmarks also prescribe not to transfer blocks larger than 64 bytes, and to keep the distribution of accessed memory locations as uniform as possible.

When these conditions hold, the gap falls to 40.68 nanoseconds, when all the 16 SPEs in a double Cell platform are employed. Under these assumptions, transitions happen at a frequency of 24.58 MHz, yielding a maximum overall throughput of 3.15 Gbps.

We designed a first implementation respecting the above ideal conditions, and evaluated it in 4 realistic scenarios: full-text search, network content monitoring, network intrusion detection and virus scanning (see Figure 8). To our great disappointment, performance was largely below the theoretical bound (the curve in red in the figure), and scalability was poor. This degradation is due to congestion in the memory subsystem, caused by a non-uniform state hit distribution. In fact, a tiny percentage of the states is responsible for the last majority of the hits. For example, the states immediately reachable from the initial state are between 0.1% and 0.5% of all the states, but they receive between 46.8% and 89.8% of all the hits. Concurrent accesses to the same or adjacent states by multiple automata cause "hot spots" in the memory subsystem, which show up as longer gaps and lower throughput.

To fight congestion, we employ a combination of state caching, replication and shuffling. First, we cache in the LS those states which are closest to the initial one, thus completely avoiding memory access for the states which are accessed most frequently. Given the limited size of the LS, we can cache only 180 states out of the 50,000 of the examples considered.

Then, we shuffle the states by randomly renumbering them. This redistributes hot spots among the memory banks, ensuring that all of them are subject to the same *pressure*, but it does not relieve the impact of each single hot spot. To that purpose, we replicate the most frequently hit of the non cached states, such that different automata access different replica of the same STT entry. This way, each replica will receive only a fraction of the accesses of the original state. As a drawback, replication causes the STT to grow from 50 Mbyte to 800 Mbyte approximately.

All these optimization only solve half of the problem. In fact, if you see the

STT as a row/column table, where each row corresponds to a state and each column to an input character, the above optimizations balance pressure only on the rows but not on the columns. Pressure on the columns is very unbalanced especially with English text patterns, because most hits fall on lower case “a” - “z” characters, while almost none fall on characters 128 - 255. To counter this unbalance, we also shuffle the alphabet space: the offsets at which next states are written in an STT entry are hashed, with an hash function which depends on the stat number too.

With the above optimizations, the aggregate throughput reaches 3.3 - 4.3 Gbps depending on the scenario (see Figure 9). This suggests that a system composed of 3 double-Cell blades, each with 1 Gbyte of RAM, can be employed to match traffic at 10 Gbps, against a dictionary comprising at least 5,000 - 25,000 patterns with an average length of 16 bytes.