Input-independent, Scalable and Fast String Matching on the Cray XMT*

Oreste Villa[1], Daniel Chavarría-Miranda[1], and Kristyn Maschhoff[2]

[1]High-Performance Computing
Pacific Northwest National Laboratory
{oreste.villa, daniel.chavarria}@pnl.gov
[2]Cray, Inc. kristyn@cray.com

Abstract

String searching is at the core of many security and network applications like search engines, intrusion detection systems, virus scanners and spam filters. The growing size of on-line content and the increasing wire speeds push the need for fast, and often real-time, string searching solutions. For these conditions, many software implementations (if not all) targeting conventional cache-based microprocessors do not perform well. They either exhibit overall low performance or exhibit highly variable performance depending on the types of inputs. For this reason, real-time state of the art solutions rely on the use of either custom hardware or Field-Programmable Gate Arrays (FPGAs) at the expense of overall system flexibility and programmability.

This paper presents a software based implementation of the Aho-Corasick string searching algorithm on the Cray XMT multithreaded shared memory machine. Our solution relies on the particular features of the XMT architecture and on several algorithmic strategies: it is fast, scalable and its performance is virtually content-independent. On a 128-processor Cray XMT, it reaches a scanning speed of $\approx$ 28 Gbps with a performance variability below 10 %. In the 10 Gbps performance range, variability is below 2.5%. By comparison, an Intel dual-socket, 8-core system running at 2.66 GHz achieves a peak performance which varies from 500 Mbps to 10 Gbps depending on the type of input and dictionary size.

1 Introduction

Network intrusion detection systems (NIDS) are an effective way to provide security to computers connected to a network. String searching algorithms are at the heart of NIDS allowing them to make decisions based not only on the packet headers, but on the actual content of the data flow. Modern, high-performance NIDS have to scan the entire inbound traffic against a large database of threat signatures (e.g. 100,000 and more) in real time

*This work was funded under the Center for Adaptive Supercomputing Software - Multithreaded Architectures (CASS-MT) at the Dept. of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

on fast links (e.g. 10 Gbps Ethernet and beyond) while trying to minimize their impact on latency and bandwidth.

In addition to the technological advances in network communication, we are also experiencing an explosion in the number of malicious threats. A recent report by Symantec, one of the leading firms in network security, shows that in the last years the number of malicious Internet threat signatures has grown at an exponential rate [17]. In total, Symantec detected 1,122,311 malicious code threats, of which almost two thirds were created during 2007.

Due to the advance in network speeds, it is becoming increasingly difficult for software-based NIDS to keep up with the line rates. Moreover, as the number of threats grows, conventional cache-based architectures exhibit a large performance variability depending on the content and size of both the searched input and matching patterns [12]. Intuitively, as the number of threat signatures increases (faster than architectural improvements in cache size and capabilities), the string matching engine becomes inefficient, achieving high performance only when the matching patterns (or their representation) are found in the caches and low performance when they have to be retrieved from main memory. In a searching process driven by unknown inputs (the input streams of a network), the string search engine has to access data in unpredictable locations of the main memory, leading to highly variable performance. This behavior is unacceptable for most real-time NIDS since it exposes the system to content-based attacks.

For these reasons, several hardware-based techniques have been employed for implementing real-time packet inspection applications. In most cases, special algorithms have been developed on Field-Programmable Gate Arrays (FPGAs) [7, 14], exploiting the potentially high level of parallelism available on these devices. Unfortunately, the flexibility and ease of programming is greatly compromised. Most importantly, the amount of matching patterns is limited by the available memory on the devices, which usually is not very large. Most state of the art NIDS indeed are composed of a fast and "not-exact" hardware-based front-end (FPGA on regular expressions) which selectively triggers a slow software backend on exact patterns [10].

A typical example is the implementation of Bloom filters or Deterministic Finite State Automata (DFAs) on FPGAs [5, 9, 11]. DFAs can efficiently implement algorithms such as Boyer-Moore [2] and Aho-Corasick [1], which allow exact string searches in a given dictionary. There are numerous FPGA-based implementations of Aho-Corasick search algorithms [4, 15, 18, 16, 8], with different degrees of performance and dictionary size.

NIDS designers are challenged at the same time along a multi-dimensional space: performance (throughput), performance variability, dictionary size and flexibility (system customization). Most FPGA solutions are very difficult to program / customize but are able to provide relatively high and stable performance on small dictionaries. Other solutions are highly customizable, can support large dictionaries, but have a limited and variable performance (Snort [12] on general purpose processors). Alternative approaches offer a relatively stable performance with medium-size dictionaries (Aho-Corasick on Cell B.E. processors [13, 19]), but require a significant programming effort. As far as we know, no single exact string matching solution is capable of simultaneously achieving content-independent, high

performance on very large dictionaries in a highly productive programmable environment.

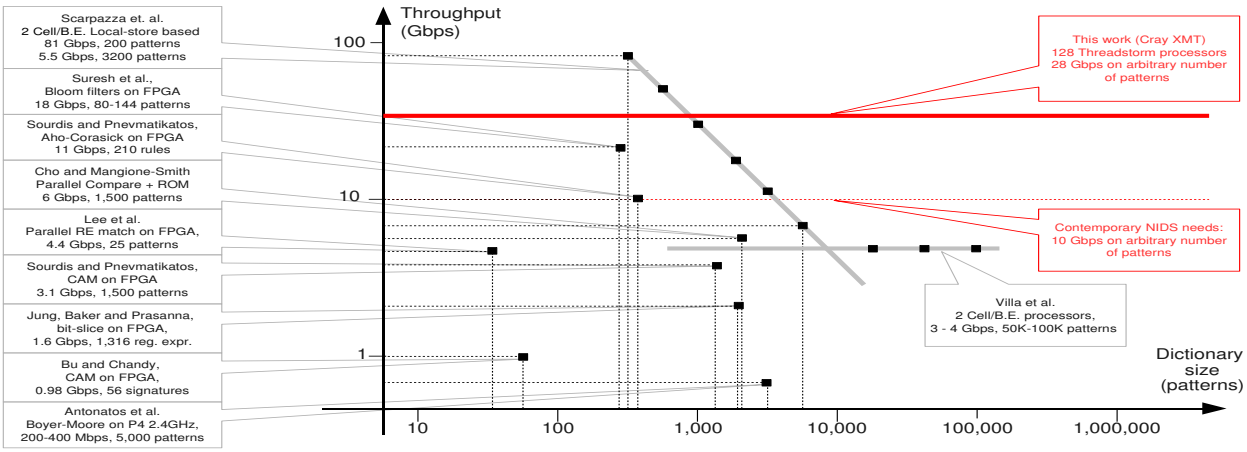


Figure 1: String matching solutions, represented in a throughput vs. dictionary size 2D space. Results are normalized (when possible) against an average pattern length of 8 symbols.

A promising solution, capable at the same time of handling very large pattern sets, and arbitrary input streams in a productive programmable environment, relies on the use of highly multithreaded processors. The Cray XMT is the new multithreaded shared memory multiprocessor machine developed by Cray (code name "Eldorado") [6]. Each *Threadstorm* multithreaded processor supports 128 hardware thread contexts which can be scheduled on a cycle-by-cycle basis to hide memory access latencies, common in irregular applications such as string matching.

This paper presents a software based implementation of the Aho-Corasick string matching algorithm on the Cray XMT multithreaded shared memory architecture. Our solution relies on the particular features of the XMT architecture and on several algorithmic strategies. It is flexible, fast and scalable. At the same time its performance is virtually content-independent. Our results show linear scaling accross a 128-processor XMT machine, with different types of input as well as with different amounts and types of searched patterns ($> 10^5$ patterns). On a 128-processor configuration, it reaches a scanning speed of almost 28 Gbps with a performance variability below 10 %. In the 10 Gbps performance range, variability is below 2.5%. Figure 1 shows, in a 2D space of performance (throughput) and dictionary size (number of patterns), some notable state of the art solutions compared to our Cray XMT solution.

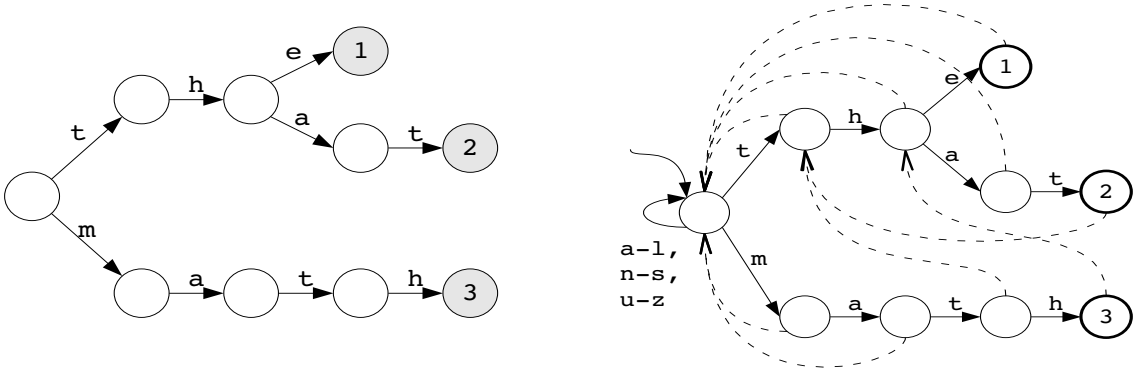
The paper is organized as follows. Section 2 presents background material on the details of the Aho-Corasick string searching algorithm and the Cray XMT system. Section 3 presents our algorithmic design on the Cray XMT while Section 4 discusses our experimental results and compares them to results obtained with other programmable architectures. Finally, Section 5 presents our conclusions.

2 Background

2.1 Aho-Corasick String Matching Algorithm

The Aho-Corasick algorithm [1] scans an input text T of length m and detects any exact occurrence of each of the patterns of a given dictionary, including partially and completely overlapping occurrences. The Aho-Corasick algorithm is particularly attractive in security applications because it can be implemented to deliver input- and content- independent performance, which can guarantee immunity from content-based attacks. It is employed in software solutions such as Snort [12] (although the performance is far from being content independent), in FPGA implementations [10, 4, 15, 18, 16, 8] and in Cell-based solutions [13, 19]. It is also the basis of our Cray XMT solution.

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



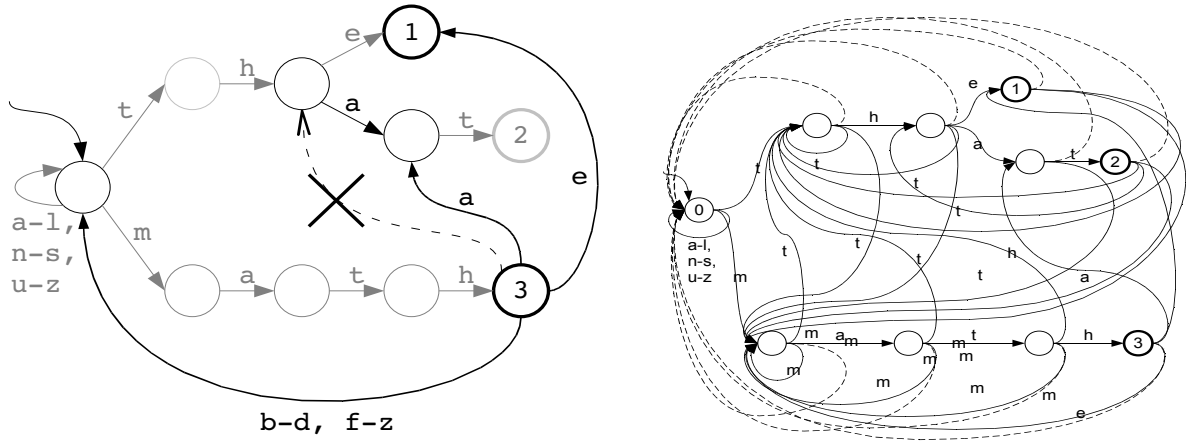
(a) The keyword tree (or trie) corresponding to an example dictionary containing the words { the, that, math }.

(b) Aho-Corasick NFA automaton corresponding to the example dictionary. Regular transitions are represented with solid lines; failure transitions with dashed lines.

Figure 2: Aho-Corasick trie and NFA for the example dictionary { the, that, math }.

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

A trie can be used to determine whether a given string belongs in the dictionary or



(a) A failure transition is replaced with regular transitions.

(b) DFA corresponding to the example dictionary (dashed lines represent default transitions for symbols not explicitly matched).

Figure 3: Aho-Corasick NFA (with failure transitions) and derived DFA for the example dictionary { the, that, math }.

not, as follows: to look up a string s, start at the root node and follow the path labeled with the characters of s, as long as possible. If the path leads to a node with an identifier i, then the string belongs in the dictionary, and it is pattern p_i. The trie does not match multiple, possibly overlapping, occurrences. The AC-fail algorithm serves that purpose. AC-fail employs an automaton which is improperly called a Non-deterministic Finite Automaton (NFA) as discussed below. The NFA (Figure 2(b)) is derived from the trie as follows. First, nodes and edges of the trie become respectively states and transitions of the automaton. The root node becomes the initial state, and the nodes with identifiers associated with them (the gray nodes) become final states. A transition is added from the root node to itself, for each symbol in the alphabet which does not have a transition leaving the root node "0" (in the example, all the symbols except 't' and 'm'). Finally, failure transitions must be added to each state.

Figure 2(b) represents the NFA, with failure transitions drawn as dashed arrows. The automaton takes a failure transition when the current input symbol does not match any regular transition leaving the current state. Failure transitions reuse information associated with the last input symbols (suffix) to recognize patterns which begin with that suffix, without restarting from the initial state (for example, the input 'mathat' matches the patterns 'math' and 'that'). The first four symbols lead from state 0 to state 3 (matching pattern 'math'). Then, symbol 'a' does not correspond to any regular transition from node 3 (in fact, there are no transitions leaving node 3 at all). So, the automata takes the failure transition, reaching the node corresponding to path 'th'. The remaining symbols 'at' cause the automaton to end up in node 2, completing the second match.

Although AC-fail is deterministic, its automaton is called an NFA because of transitions which do not consume input (traditionally called ϵ-transitions). For this reason, the AC-fail

suffers from a potential performance drawback: on a failure, multiple state transitions can happen per single input symbol.

When the size of the dictionary grows, the performance of AC-fail decreases quickly due to failure transitions [20]. An improved version of AC-fail, called AC-opt, solves this issue by employing a Deterministic Finite Automaton (DFA) in place of the NFA, at the cost of increased memory requirements. The DFA is obtained from the NFA by replacing all the failure transitions with regular ones. The equivalent DFA has exactly one transition per each state and input symbol.

Figure 3(a) illustrates how a failure transition (crossed-out) is replaced with regular transitions. The final DFA is shown in Figure 3(b) and it is the final dictionary representation used in our solution due to its one-symbol, one-transition property. A more detailed discussion of the derivation of the DFA and the AC-opt algorithm can be found in [20].

2.2 Cray XMT

The Cray XMT is the commercial name for the new shared-memory multithreaded machine developed by Cray under the code name "Eldorado" [6, 3]. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with *Threadstorm* processors. The entire system is connected using the Cray Seastar-2.2 high speed interconnect. The XMT system can scale up to 8,192 *Threadstorm* processors and 128 TB of shared memory.

Each Threadstorm processor is able to schedule 128 fine-grained hardware threads to avoid memory-access generated pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a hardware stream comprised of a program counter, a status word, a target register and 32 general purpose registers. Each Threadstorm processor has a VLIW (Very Long Instruction Word) pipeline containing operations for the Memory functional unit, the Arithmetic unit and the Control unit[1].

Each Threadstorm is associated with a memory system that can accommodate up to 8GB of 128-bit wide DDR memory. Each memory controller is complemented with a 128KB, 4-way associative data cache to reduce access latencies (this is the only data cache present in the entire memory hierarchy). Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grained thread synchronization with little overhead. The memory is hashed at a granularity of 64 bytes (see Figure 4) and fully accessible through load/store operations to any Threadstorm processor connected to the Seastar-2.2 network, which is configured in a 3D toroidal topology. While memory is completely shared among Threadstorm processors, it is decoupled from the main memory in the AMD Opteron service nodes. Communication between Threadstorm nodes and Opteron nodes is performed through a Lightweight Communication Library (LUC). Continuous random accesses to memory by the Threadstorm processor will top memory bandwidth at around 100M requests per second.

[1]The Arithmetic unit is capable of performing a floating-point multiply-add per cycle. In conjunction with the control unit doubling as arithmetic unit, a Threadstorm is capable of achieving 1.5 GFlops at a clock rate of 500MHz. A 64KB, 4-way associative instruction cache helps in exploiting code locality.

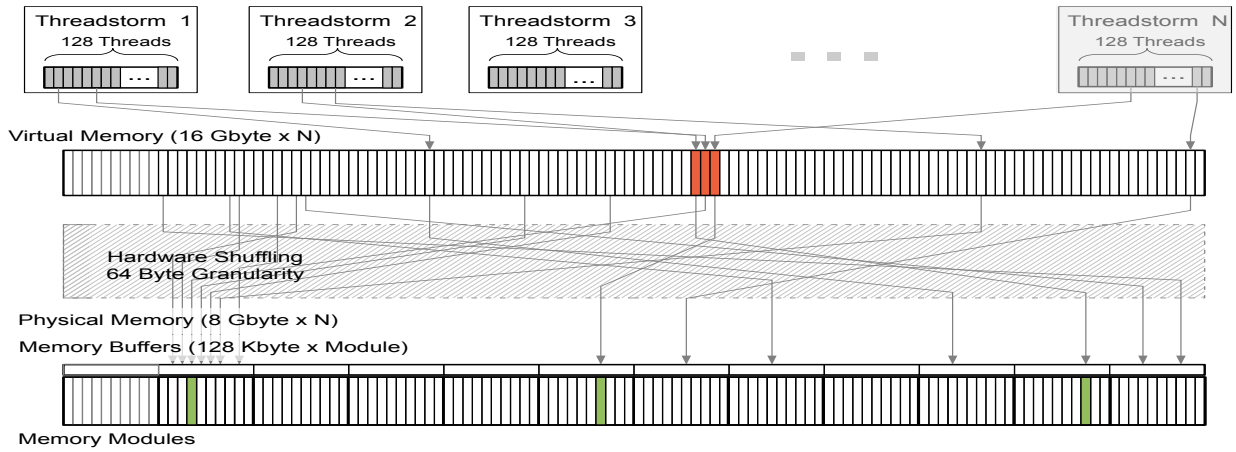


Figure 4: Cray XMT Threadstorm memory subsystem.

The software environment on the Cray XMT includes a custom, multithreaded operating system for the Threadstorm compute nodes (MTX), a parallelizing C/C++ cross-compiler targeting Threadstorm, a standard Linux 64-bit environment executing on the service and I/O nodes, as well as the necessary libraries to provide communication and interaction between the two parts of the XMT system. The parallelizing C/C++ compiler generates multithreaded code that is mapped to the threaded capabilities of the processors automatically. Parallelism discovery happens fully or semi- automatically by the addition of `pragmas` (directives) to the C/C++ source code. The compiler's parallelism discovery focuses on analyzing loop nests and mapping the loop's iterations in a data-parallel manner to threads.

3 Algorithmic Design & Implementation

Our algorithm design is based on the following cornerstones: a) minimize the number of memory references and b) reduce memory contention. As explained in Section 2.1, the searched patterns are represented as a Deterministic Finite Automaton (DFA) (see Figure 3(b)). As the input symbols are scanned, the search algorithm traverses the different nodes of the DFA. For each possible input symbol there is **always** a valid transition to another node in the graph. This key feature guarantees that for each input symbol there is always the same amount of work to perform. In detail, the Aho-Corasick string matching algorithm works as shown in Algorithm 1.

For a given dictionary the data structures in main memory (DFA and input symbols) are read-only. The parallelization strategy involves the use of multiple threads that concurrently execute the above algorithm. Each thread has a *current_node* and operates on a distinct section of the input. All threads access the same DFA structure. At run-time the input stream is split into chunks and assigned to the different threads. The chunks overlap partially to allow matching of those patterns that cross a boundary. The necessary overlapping is equal to the length of the longest pattern in the dictionary minus 1 symbol. The inefficiency of

Algorithm 1 Basic steps of the DFA-based Aho-Corasick string matching algorithm

1. **Load** node "0" (or root) from main memory (DFA) in *current_node*.
2. **Load** one input symbol from main memory in *symbol*.
3. **Load** the *next_node* from main memory (DFA) (following the link from the *current_node*, labeled by *symbol*).
4. **Check if** the transition to *next_node* is final (if it is, the last symbols are a matching pattern).
5. **Assign** *next_node* to *current_node*.
6. **Repeat** starting from step 2 **until** there are no more input symbols.

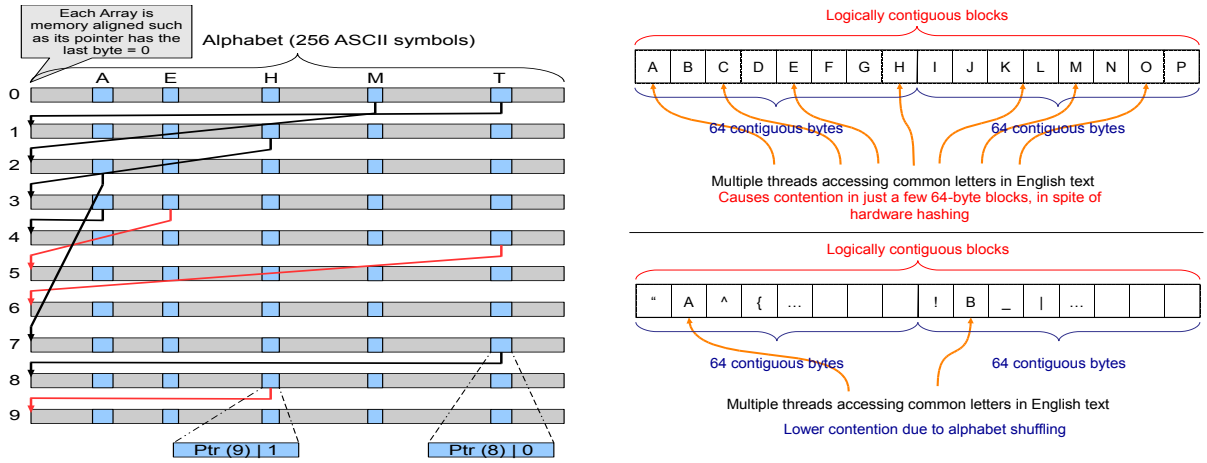
the overlapping (replicated work) is measured as (longest pattern − 1)/(size of the chunk). In our experiments, we chose chunks of 2 KBytes and longest patterns of 16 bytes resulting in a negligible inefficiency of $\approx 0.008\%$.

For each input symbol there are conceptually 2 loads to perform, one for the *symbol* itself (Step 2) and one for the *next_node* in the DFA (Step 3). Although Step 4 could conceivably involve a load operation (as we need to check if the transition is final or not) it does not. Our implementation, as we describe later, indeed does not involve an extra load. Step 2 loads contiguous symbols of 8 bits each (ASCII alphabet in our experiments) from main memory. In a 64 bit architecture (as the Threadstorm), the frequency of this load can be reduced to only one load for every 8 symbols, shifting and masking 7 times to extract the right symbol. Furthermore, accessing the scanned symbols has very high spatial locality.

The second load (for *next_node*) is quite different. It is not predictable since it depends on the *symbol* just loaded and on the *current_node*. This load is the main cause of performance degradation on conventional cache-based architectures for this class of algorithms. If the input text contains a large number of patterns that are present in the dictionary, then the entire graph will likely be accessed during the matching process ("heavy" matching). For instance, matching the graph in Figure 3(b) against the string 'math that the' (the same words of the dictionary) we need to access every node in the graph. On the other hand, if there is "light" or no matching (i.e. a dictionary against a random pattern) most of the time the search algorithm will stay on node 0 (root) since the failure transitions jump again on node 0 itself.

Our first algorithmic decision was to implement a scanning engine that performs Step 3 with only 1 load per symbol. To achieve this goal we represent the DFA graph as a sparse State Transition Table (STT). The STT is large table composed of as many rows as there are nodes in the DFA and as many columns as there are symbols of the alphabet.

Each STT line represents a *node* in the original DFA. Each entry of a STT line (cell) (indexed by a *current_node* and a *symbol*) stores the address of the *beginning* of the STT line that represents the *next_node* for that transition in the DFA. Figure 5(a) shows the corresponding STT (with limited transitions) for the example dictionary we have been using ({*the, that, math*}) in Figure 3(b). The example STT is composed of 10 lines (10 nodes in the original DFA) and 256 columns (256 symbols of the ASCII alphabet). The STT lines are 256-byte aligned such that the least significant byte of the address is equal to zero. This property allows us to store in the least significant bit of each STT cell the boolean information



(a) STT structure used to represent the DFA of Figure 3(b). Only few transitions are represented. Transitions in red are final.

(b) Mapping of shuffled alphabetical ASCII codes on 64-byte blocks (top not hashed, bottom hashed)

Figure 5: STT structure and alphabet hashing

indicating if that transition is final or not (red transitions in Figure 5(a)) eliminating the need for an extra load in Step 4 (see Algorithm 1). Since we want to retrieve the *next_node* address dereferencing the *current_node + symbol* pointer, STT lines must have always the same size. Alphabet symbols not used in the dictionary have to be explicitly represented as failure transitions to the root node.

This sparse STT representation is expensive in terms of the amount of memory it uses, but it guarantees that Step 3 will only require one load operation. In our experiments, with 190,000 text patterns of average length 16 bytes and the 256-character ASCII alphabet, the STT size is 9.8 GBytes. However, on the Cray XMT this is not an issue since even on a 16-processor configuration, the total amount of shared memory is 128 GBytes. This is a typical example where memory space can be traded for access speed.

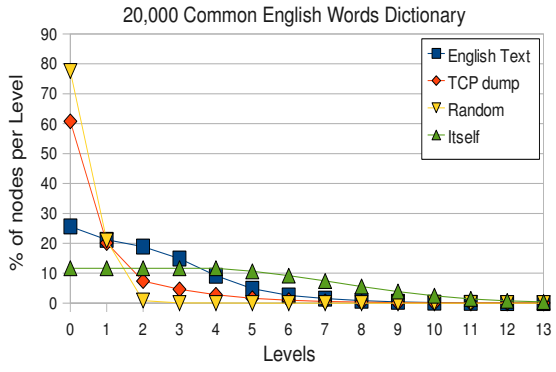
As discussed previously, the load in Step 3 (see Algorithm 1) is crucial since this operation could have the highest performance variability [2]. Because the objective of our solution is to provide content-independent performance, we need to obtain a relatively uniform latency for this load. In comparison to other solutions where absolute latency matters, our highly multithreaded approach focuses mostly on reducing latency variability. If the latency is constant or slowly variable, the system is able to schedule a sufficient number of threads to hide it.

In the XMT implementation, the main cause of variability in the memory access time is the presence of hot-spots. Hot-spots are memory regions frequently accessed by multiple threads simultaneously. To minimize hot-spots, the Cray XMT employs a hardware hashing mechanism which spreads data in all the system's memory banks with a granularity of 64 bytes (block) [6] (see Section 2). However, if different blocks corresponding to different

[2]Step 1 is executed only once, Step 2 loads contiguous symbols that most likely are in the cache, Steps 4-5-6 are arithmetic/logic operations on registers

memory banks have different access ratios, the "pressure" on the memory banks is not equally balanced, producing variability in the access time. In our implementation there are two reasons why this can happen:

- Each STT cell is large (8 bytes: address of the STT line + boolean information if the transition is final). Therefore each STT line (representing a DFA node) uses *alphabet size* $\times$ 8 bytes. If we consider the 256-symbol ASCII alphabet, each STT line requires 2048 bytes, or 32 blocks (64 bytes per block). If the scanned input has particular characteristics in terms of symbol frequency (i.e. English text, decimal numbers, etc.) the input symbols will only be a subset of the ASCII alphabet and the different concurrent threads will be accessing a (small) subset of the 32 blocks per STT line, producing hot-spots.
- Typically, a few states in the first levels of the DFA graph[3] are responsible for the majority of hits. Figure 6 shows, for an English dictionary with 20,000 patterns, the distribution of the accessed nodes in the DFA as function of the levels when scanned against English Text, a TCP traffic dump, random input and the dictionary itself. The table in Figure 6 shows the percentage of states in the first two levels and the percentage of the transitions to them from other levels, while scanning different inputs. In every situation, except when scanning a dictionary against itself, the first 2 levels are responsible for most of the accesses. As a result, the memory blocks containing those states form hot-spots. It is important to notice that for inputs that are similar to the dictionary, the transitions tend to be distributed more equally on the levels ("Itself" line in Figure 6) leading to reduced or absent hot-spots.



Dictionary Type	% of Nodes in Lev. 0,1	% of Transitions to Lev. 0,1			
		English Text	TCP Dump	Random	Itself
English	0.11435	46.79	81.47	99.16	23.29
Binary/Text	0.00345	47.26	81.47	98.54	12.53
Random	0.08896	57.32	54.05	48.43	25.03

Figure 6: Access distribution for an English dictionary with different inputs.

In order to alleviate the above problems we propose two solutions. Both of them are quantitatively evaluated in Section 4.

- **Alphabet Shuffling**: The alphabet symbols in a STT line can be shuffled using a relatively simple linear transformation, to ensure that symbols that are contiguous in the alphabet (i.e. standard English characters in ASCII) are spread out over multiple

[3] "Level" corresponds to a Breadth-First Search (BFS) exploration of the DFA graph

memory blocks. The shuffling function can be inexpensively and effectively computed as $symbol' = (symbol \times fixed\_offset) >> 8$. This transformation, in conjunction with the hardware hashing mechanism on the XMT's memory, can guarantee that accesses are spread out over distinct memory blocks for certain classes of inputs: English text, decimal numbers, or other inputs with characters located relatively close to each other in the ASCII ordering.

On the Cray XMT, an effective alphabet shuffling function does not need to depend on the state number (line in the STT table), in contrast to what would be necessary in a non-hardware hashed memory hierarchy. Different states will be effectively shuffled in different ways due to the underlying hardware hashing. Due to the combination of hardware and software hashing, a simple linear transformation is relatively secure against content based attacks. Figure 5(b) shows how this mechanism relieves contention in the accesses to two contiguous logical blocks.

- **State Replication**: We replicate the STT states corresponding to the first 2 levels of a Breadth-First Search (BFS) exploration of the DFA. Addresses of the different replicas of the same logic state (STT line) are randomly stored during the creation of the STT in the STT cells pointing to that state. This ensures that the memory pressure is equally balanced when the blocks for that state are accessed by different threads. This mechanism is greatly simplified by the underlying hardware hashing since different replicas are spread in different memory banks. We do not need to explicitly manage the replicas to position them in other memory banks.

4 Experimental Setup and Results

We have implemented the Aho-Corasick parallel algorithm as described in Section 3 for the Cray XMT multithreaded system. Our implementation has two main phases: building the State Transition Table (STT) and executing string matches against the built STT. The STT building phase is performed offline and stored in a file representation and we focus our experiments on the string matching phase, since this is the critical portion in the realistic use of this algorithm for network content analysis. Our implementation utilizes the hybrid capabilities of the XMT to offload expensive I/O operations from the Threadstorm processors to the Opteron processors on the service nodes.

The multithreaded parallelization of the string matching code, focuses on assigning a *chunk* of consecutive input symbols to each thread. Each thread then will try to match its assigned symbols by following the links in the STT. A per-thread counter is incremented on a match (reaching a final state in the STT traversal).

We have also built a standard `pthreads` version of our application that utilizes the same algorithm as the XMT version, as a way of comparing conventional cache-based systems against the XMT. The `pthreads` version shares the same code base as the XMT version and simple conditional compilation rules are used to produce either the x86-`pthreads` or the XMT versions. On the `pthreads` the input is divided into equal-size chunks and statically assigned to threads executing on the cores.

We have used a 128-processor Cray XMT with 1 TB of memory and a Intel Xeon workstation with dual-socket, quad-core processors running at 2.66 GHz and 16 GB of memory.

Our experiments utilize four different dictionaries: **Dict 1**: a $\approx 190,000$-pattern data set with mostly text entries with an average length of 16 bytes; **Dict 2**: a $\approx 190,000$-pattern data set with mixed text and binary entries with an average length of 16 bytes; **English**: a 20,000-pattern data set with the most common words from the English language, with an average length of 8.5 bytes; and **Random**: a 50,000-pattern data set with entries generated at random from the ASCII alphabet with an uniform distribution and an average length of 8 bytes. Figure 7 presents the distribution of ASCII symbols present in each of the experimental dictionaries. Dictionaries with more text-like entries have higher frequencies of alphabetical ASCII symbols. For both our XMT and x86 experiments, the data structures representing the dictionaries fit completely into the memories.

We also use four different input streams for each dictionary: **Text**, which corresponds to the English text of the King James Bible, **TCP**, which corresponds to captured TCP/IP traffic, **Random**, which corresponds to a random sample of characters from the ASCII alphabet, and **Itself**, which corresponds to feeding the dictionary itself as an input stream for string matching. As discussed previously, using the dictionary itself as an input will exhibit the "heaviest" matching behavior.

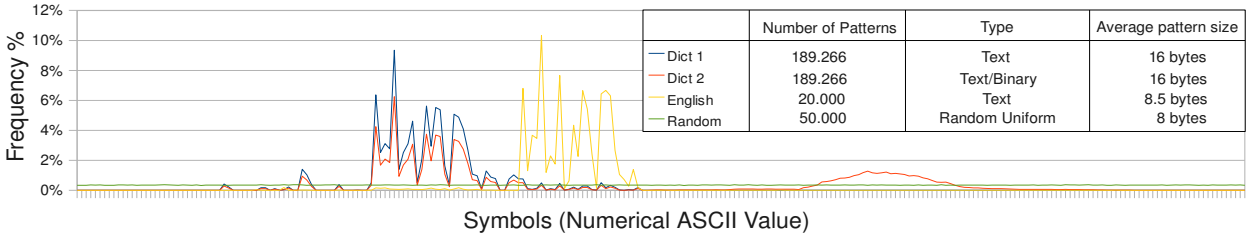


Figure 7: Distributions of the symbols in the dictionaries

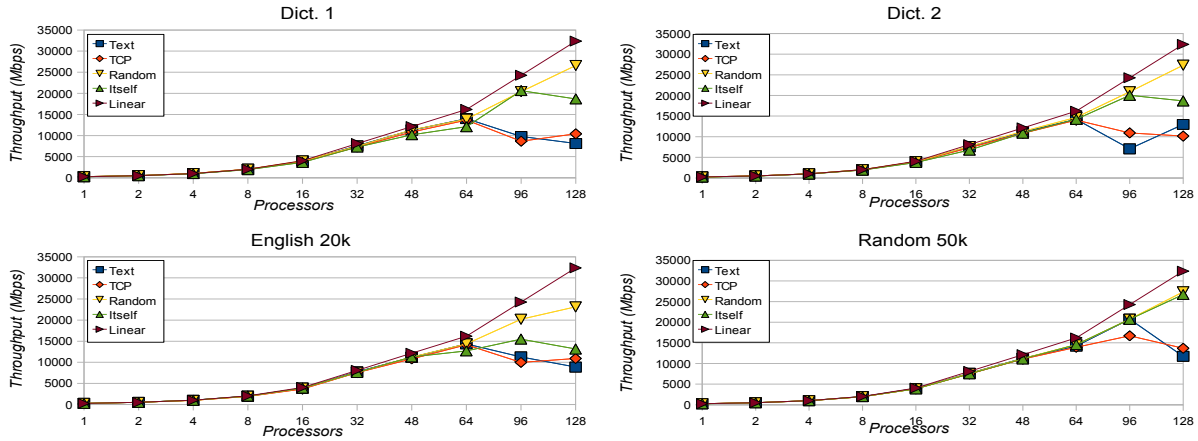


Figure 8: Performance of the original implementation on the 128p Cray XMT

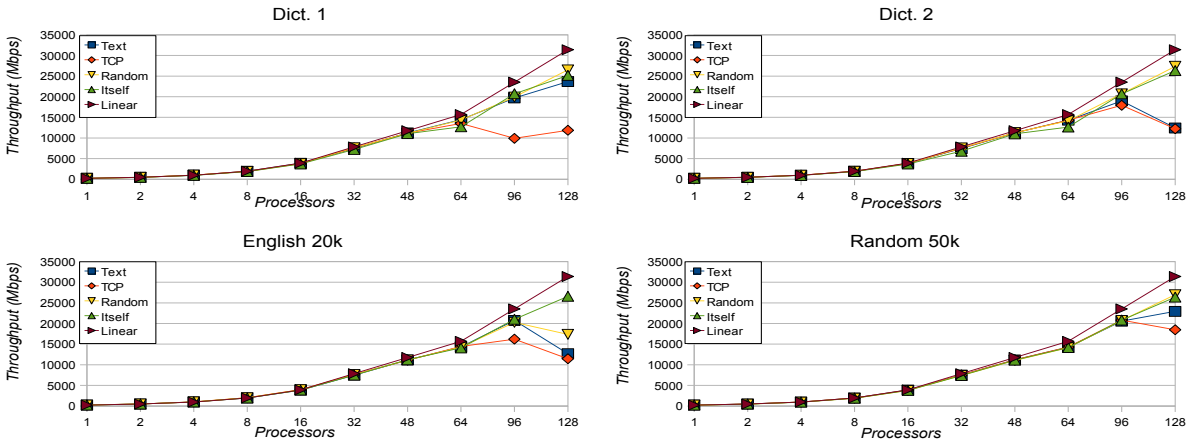


Figure 9: Performance of the implementation with alphabet shuffling on the 128p Cray XMT

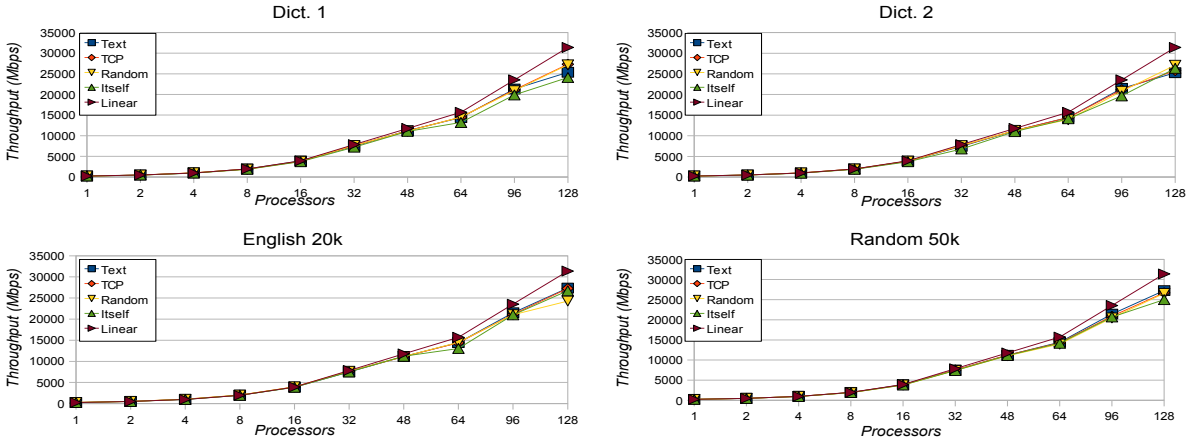


Figure 10: Performance of the implementation with alphabet shuffling and state replication on the 128p Cray XMT

Figure 8 presents the performance of our original implementation of Aho-Corasick string matching on a 128-processor Cray XMT for the four previously discussed dictionaries. The performance in many cases does not come close to the linear scaling curve due to memory hot-spots present in accessing the STT representation by large numbers of threads.

Figure 9 presents the performance of our string matching code using alphabet shuffling as discussed in Section 3. The scalability and absolute performance have improved for dictionaries and input streams that exhibit large numbers of text-like (alphabetical) patterns in them (**English** and **Text**).

Finally, Figure 10 presents the performance of our code using both alphabet shuffling and replication of the first two levels in the STT graph (the levels have been replicated 8 times). This transformation relieves all the remaining memory contention pressure and enables very good scalability and absolute performance, independent of the input stream

and the matching dictionary.

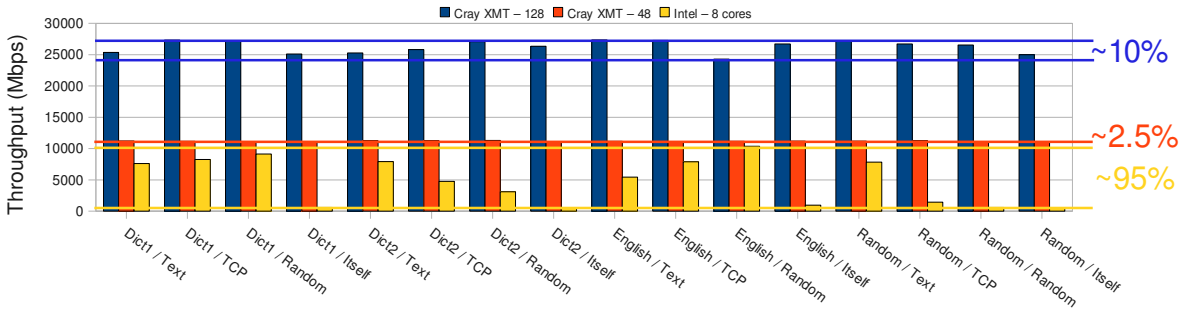


Figure 11: Performance variability of the Cray XMT vs. an 8-core x86 Intel Xeon.

Figure 11 compares the performance variability for the Cray XMT version of the code and for the x86-`pthreads` version of the code running on an 8-core Intel Xeon workstation (2 sockets, 4 cores each). The performance variability of the Intel system can be as high as 95% while the performance variability of the XMT is below 10%. For the 10Gbps range, the performance variability of the XMT implementation is 2.5% for different combinations of dictionaries and input streams. We believe that using the same source code base for both XMT and x86 platforms is appropriate since our implementation minimizes loads and data structure overhead for both, given the absence of locality. For the x86-`pthreads` experiments, we did not use alphabet shuffling or replication since they degraded the performance slightly due to their control overhead.

5 Conclusions

We have presented the design and performance of a Cray XMT-based implementation of the Aho-Corasick string searching algorithm. By utilizing the particular features of the XMT multithreaded architecture and several algorithmic strategies (data structure selection, alphabet shuffling, state replication), we have achieved scalable high performance, which is independent from the analyzed input stream, as well as the matching pattern set. The implementation of our sophisticated algorithmic strategies, was greatly simplified by the underlying hardware and software support on the XMT.

Our absolute performance is one of the highest reported in the literature, at ≈ 28 Gbps, for a software solution with very large dictionaries. The performance variability of our solution is very low, compared to other software implementations, which makes it amenable to real-time usage. Our implementation has achieved this level of performance with a moderate programming effort and a much simpler code structure, compared to custom solutions on FPGAs and multimedia processors such as the IBM Cell/B.E.

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

[3] D. Chavarría-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer. Early Experience with Out-of-Core Applications on the Cray XMT. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[4] 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.

[5] 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.

[6] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[7] 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.

[8] 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.

[9] 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.

[10] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, New York, NY, USA, 2007. ACM.

[11] 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.

[12] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *LISA*, pages 229–238, 1999.

[13] D. P. Scarpazza, O. Villa, and F. Petrini. Exact multi-pattern string matching on the Cell/B.E. processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 33–42, New York, NY, USA, 2008. ACM.

[14] 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.

[15] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion. In *Proceedings 13th Conference on Field Programmable Logic and Applications.*, September 2003.

[16] 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.

[17] Symantec Global Internet Security Threat Report. *White Paper*, April 2008.

[18] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE Infocom Conference*, pages 333–340, 2004.

[19] O. Villa, D. P. Scarpazza, and F. Petrini. Accelerating Real-Time String Searching with Multicore Processors. *Computer*, 41(4):42–50, 2008.

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