Hashing + Memory = Low Cost, Exact Pattern Matching.

CONFERENCE PAPER · JANUARY 2005

DOI: 10.1109/FPL.2005.1515696 · Source: DBLP

2 AUTHORS, INCLUDING:

Dionisios N. Pnevmatikatos
Foundation for Research and Technology -…

95 PUBLICATIONS **1,392** CITATIONS

HASHING + MEMORY = LOW COST, EXACT PATTERN MATCHING

Giorgos Papadopoulos†

†Microprocessor and Hardware Laboratory,
Electronic and Computer Engineering Dept.,
Technical University of Crete,
Chania - Crete, GR 73 100, Greece
email: {gpap, pnevmati}@mhl.tuc.gr

Dionisios Pnevmatikatos‡†

‡Institute of Computer Science (ICS),
Foundation for Research and
Technology-Hellas (FORTH),
Heraklion - Crete, GR 71 110, Greece
email: pnevmati@ics.forth.gr

ABSTRACT

In this paper we propose the combination of hashing and use of memory to achieve low cost, exact matching of SNORT-like intrusion signatures. The basic idea is to use hashing to generate a distinct address for each candidate pattern, which is stored in memory. Our implementation, HashMem, uses simple CRC-style polynomials implemented with XOR gates, to achieve low cost hashing of the input patterns. We reduce the sparseness of the memory using an indirection memory that allows a compact storing of the search patterns and use a simple comparator to verify the match. Our implementation uses in the order of 0.15 Logic Cells per search pattern character, and a few tens of memory blocks, fitting comfortably in small or medium FPGA devices.

1. INTRODUCTION

The area of NIDS pattern matching has been very active recently. Several architectures have been proposed to implement SNORT-like pattern matching in FPGA. The architectures differ in the approach (finite automata or CAM-like), internal organizations, and of course in their cost-performance tradeoffs [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. The common factor of these efforts however is the continuous drive for lower cost, at the same or better performance. This work builds on two distinct ideas: (i) of the use of hashing of the input to retrieve approximate match information used in the Bloom filters [5, 10], and (ii) on the use of memories to provide exact match with fewer gates used by Cho and Magnione-Smith [8], and earlier by Burkowski [12].

Dharmapurikar et al. proposed the use of Bloom filters for low cost pattern matching [5]. Bloom filters are very elegant in representing set membership, but have two potential drawbacks: (i) they require multiple hash functions and memories, and (ii) they give an approximate match answer since they allow false positives. Attig et al. proposed the use of external SDRAM memory to augment the bloom filters with exact match, but at extra cost, and without offering worst case guaranteed bandwidth since the external memory is not pipelined [10].

Cho and Magnione-Smith used a CAM to match short patterns and to match unique prefixes of longer search patterns [8]. They choose the CAM width so as to provide unique prefix signals for each possible match. The match signals for all prefixes are then encoded to provide a memory address where the candidate suffixes are stored. The remaining input is compared against the expected suffix, and the result is the overall match for the pattern. Their approach offers very good memory density and low gate count. The cost of this approach however increases if the patterns have many and long common prefixes.

Our proposed architecture attempts to strike a different balance between memory and logic usage. We use simple hash functions to generate sparse but distinct addresses for each of the search patterns, giving us a hint of whether there is a possible match (with probability proportional to the density of the hash space). We then use an indirection table to "gather" the search patterns in a compact memory, and compare the input against the single possible search pattern to eliminate false positives. Our implementations achieve processing throughputs between 1.95-2.7 Gbps processing a single input character per cycle, for a cost of 0.15 logic cells per search pattern character and a few tens of memory blocks, fitting comfortably in small or medium FPGA devices.

The rest of the paper is organized as follows. In sections 2 and 3 we describe the proposed HashMem architecture, and extend it to achieve better cost and performance. In section 4 we present the implementations results and compare with other published results, and we conclude in section 5.

2. THE HASHMEM ARCHITECTURE

The basic idea of HashMem is to use the input pattern to generate a unique candidate pattern address. Lets assume we want to match a set of patterns of length L characters. We feed L input bytes into a hashing module to generate the

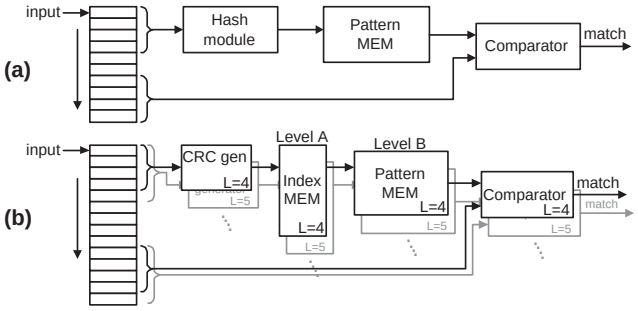

Fig. 1. **(a)** The basic HashMem idea. To search for a pattern of length N characters, we hash N input characters and produce a memory address. If the stored pattern matches the input pattern, we have a match, otherwise no. **(b)** Detailed HashMem Architecture. The Index memory rearranges the addresses of the possible match patterns into the smaller (but wider) Pattern Memory.

unique address of the candidate pattern. We then read the candidate pattern from the memory and compare it with the (delayed) input to verify the match.

This overall structure, shown in Figure 1(a), has been used by Cho et al. using prefix match logic, and an encoder to generate the unique address. Unlike this work, we use a CRC-type computation on the entire length of the match and avoid the use of encoders. CRCs have two advantages: first they are simple functions with small implementation cost. Second, they produce a "randomized" result, with a uniform distribution of all possible patterns of a specific width into each CRC value. Depending on the polynomial used, each CRC function will produce a different mapping of patterns to locations. This gives us a way of adapting to different pattern sets. Given a set of patterns, we can select a polynomial that produces distinct CRC values for every search pattern. Since a match can begin anywhere in the input stream, for our L character search, we have to check each of the L-character substrings starting at offsets 0, 1, ..., up to the end of the input packet. To achieve this, we pipeline the CRC generator.

Finding a polynomial that guarantees distinct addresses for each search pattern is easier when the density of the hash space is smaller. Experimentally we found that using 12 bits for Snort patterns, we can use simple polynomials that achieve this guarantee. However, a small memory density means that many memory locations are not used. This problem is exacerbated for longer patterns since the memory is as wide as the search patterns. To alleviate this overhead, we introduce a level of indirection, using 12 bits of CRC address, and packing the search patterns in a shorter memory to improve its utilization. The width of the indirection memory is related to the number of stored patterns, with 8 bits being more than enough for patterns of a given width

Until this point we discussed patterns of equal width (L). Dealing with multiple width patterns requires to (i) know the width of the possible matches, and (ii) the ability to read all these possible patterns. Since any given character of the input stream can be the last character of a pattern of arbitrary width, we use the simple approach of replicating the entire structure once for each of the different pattern widths. The resulting architecture is show in Figure 1(b).

Sourdis et al. [13] are working on improving this architecture, placing their emphasis on using a *perfect* hashing function, i.e. one that achieves a 1-to-1 mapping between search patterns and memory locations. While their work shares the use of indirection memory compared to ours, their contributions are different as they use a centralized, banked memory for efficient pattern storage, and they trade logic to reduce the memory size.

3. HASHMEM IMPLEMENTATION AND IMPROVEMENTS

For the construction of a HashMem system, we first group the search patterns according to their width L. Then, for each group, we identify a CRC polynomial that produces distinct addresses for each pattern in the group. The search patterns are packed in a pattern memory of width L without any restriction on their location. Finally the indirection memory is initialized. Initially the indirection memory is initialized to a special value "No-Match". Then for each search pattern P that is stored in location PMAddr(P) in the pattern memory, we set location CRC(P) of the indirection memory to PMAddr(P). This essentially creates a pointer to the stored search pattern, if there is one.

3.1. Efficient CRC Generation

The HashMem architecture uses one CRC generator for each search pattern width, and each generator has to produce the CRC of L characters in one cycle. To achieve a CRC implementation according to these requirements, we first implemented a fully parallel, unpipelined generator, and then we pipelined it to achieve (i) throughput of one full hash per cycle, and (ii) good cycle time.

While each CRC generator is relatively simple, its cost is proportional to the input width. Some Snort search patterns exceed 50 characters or 400 input bits, for which even simple CRC generators use many gates. To reduce the cost of wide CRC generators, we produce the hash value for wide patterns reusing the narrower CRC values as partial hashes. We implement full CRC generators for the widths of 3, 4, 5 and 6 bytes. Then, to produce a CRC value for say 7 character wide pattern, we perform a CRC function on the CRC results for the first 4, and the next 3 input stream characters. The partial CRC values should be delayed appropriately so that they arrive at the same time to the second level CRC

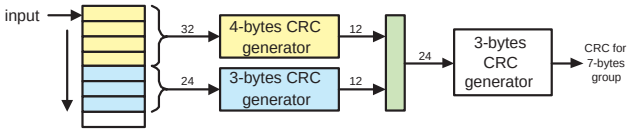

Fig. 2. Hierarchical CRC generation for a 7 character pattern using the partial CRC results for the first 4 and the next 3 input characters.

generator. Figure 2 illustrates this 7-character wide example. This hierarchical CRC calculation and the reuse of partial CRC values results in significant area savings reducing the input of the generators to a more manageable width.

Another parameter affecting the cost of the CRC computation is the width of the CRC value. Smaller widths reduce the size of indirection memory, but make it harder to guarantee distinct addresses for each search pattern. We experimented with various CRC widths, and found that a larger space density, increases the cost (in number of LUTs) of the CRC generators. In practice, we found that the best compromise between CRC implementation cost and memory size is at 12 bits (i.e. using polynomials of degree 12). For small pattern widths, we can use smaller degrees (10 and 11), but in general we used 12 bits.

3.2. Handling Very Short Patterns

Very short patterns (1-2 characters) offer very few input bits, making the CRC calculation an overkill. Furthermore, the total pattern characters are very few, underutilizing the indirection and pattern memories. To address this inefficiency, we use a simple lookup table of 256 entries to match the single character patterns.

For the two character patterns, we notice that the total distinct patterns characters are less than 64, allowing an encoding with 6 bits. Based on this observation, we added a recoding function in the unused bits of the single character lookup table, recoding the 8 bits input into a 7 bit code (6 bits for the encoded value and 1 bit for the single character match). Then, two input recoded characters amount to 12 bits, and we use a 4Kx1 lookup table to determine any two character matches. Each of these two lookup tables uses one memory block.

3.3. Reusing Logic and Memory for Wide Patterns

Very long strings on the other hand are few but use very wide pattern memories. The widest Xilinx memory block has 512 entries of 36 bits. The few wide patterns will leave almost the entire memory empty. In addition, most of the logic in the HashMem architecture comes from CRC generators, and the pattern comparators. Long patterns, due to their small number in the Snort rule set, offer us with this opportunity to reuse these pieces instead of replicating

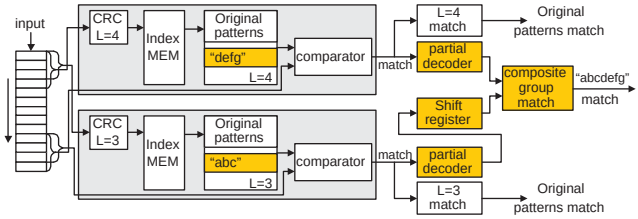

Fig. 3. A 7 character pattern ("abcdefg") is partially matched as two patterns of 3 ("abc") and 4 ("defg") characters. The partial matches are then combined to determine the overall match.

them. The idea is that instead of matching a wide pattern in a separate structure, we split it in smaller pieces, match each of them in existing narrower structures, and use extra glue-logic to combine the partial matches into the complete match.

Figure 3 depicts a short example where a 7 character pattern in matched as two sub-patterns of 3 and 4 characters. Note that we use this small pattern width only for reasons of clarity and brevity; we actually used this technique for widths 17 and up. The two sub-patterns are added into the existing structures for widths 3 and 4, reusing the CRC generators, memory, and comparators. There are two issues that we have to address in order to make this approach work.

First, in order to add a new pattern in an existing structure, the CRC of that pattern should not conflict with any preexisting pattern in the set. This restriction is addressed in two ways: first, the density of the hash space is small, so probabilistically our chances are good. Second, if we are indeed unlucky, we can always change the CRC polynomial and find one that removes the conflict. Should both of these options fail, we still have alternatives. We can partition the pattern in a different way (in our example perhaps in 4+3 or 2+5 characters instead of the 3+4 used in our example). The reuse of the resources can happen in many different ways increasing our chances of finding a convenient mapping. In our experience, the complicated options are not needed and it is straightforward to add the sub-patterns in existing structures.

The other issue is the glue logic that combines the partial matches into overall matches for the pattern. Since each of the two sub-patterns are detected at offset 0, we must delay the match signal for the first sub-pattern to AND it with the match of the second sub-pattern and determine the overall match. In our example we must delay the first 3 characters match by 4 clock cycles.

3.4. Sharing Memory Structures between Different Pattern Widths

To further reduce the amount of memory needed, we can exploit two facts: (i) the low density of the indirection and data

memories, and (ii) that the Xilinx memories are dual ported. The idea for sharing the data memory is simple: we partition the memory in two independent portions. The "upper" portion is used for patterns of width X and the "lower" portion is used for patterns of width Y (usually $X + 1$). Consider for a moment the data memory for say width 3. The minimum dimension of a single Xilinx memory block is 512x36 bits (enough for 4 characters), while the number of patterns of widths 3 and 4 is 33 and 72 patterns respectively. It is clear that both these sets of patterns can coexist in the same memory block, *without* any overhead. However, since we use different CRC generators for 3 and 4 characters, the addresses for each of the widths will be different. Here we can use the two read ports of the memory: we statically assign each read port to a given size, and arrange the patterns in two separate portions of the memory space.

While sharing the data memories is fairly straightforward, sharing the indirection memories is a bit more involved. Consider a simple example where there is only one string for each of the widths 3 and 4. Each of the indirection memories for widths 3 and 4 will have a single non-empty entry at the location determined by the CRC function of the corresponding width. These non-empty locations will point to the data memory locations that the actual strings will be placed, say X and Y. If the locations of the two non-empty entries are distinct, we can merge the two memories, creating a simple indirection memory, with *two* non-empty locations, with contents X and Y at their original locations. Now we can use the two read port for each of the pattern widths.

To show that this scheme does work, we consider two cases, of a pattern match and of a pattern mismatch. First, if a pattern match input appears for say size 3, the hash value will point to the location containing X. The data memory string will be read for width 3, and a match will be reported. For an arbitrary non-match input, the CRC value will most probably point to an empty location, resulting in a mismatch. However, there is the possibility that for some 3 character input, the CRC value will point to the location initialized to the 4 character pattern. To avoid any false matches, we augment the comparator with an additional bit that indicates the portion of the memory that the pattern belongs to. In this way, when we read location Y of the data memory that contains in a 4 character pattern, it is impossible to report a 3 character match. Figure 4 outlines the sharing of both levels of memory in our architecture.

If the two sets have conflicts, then we have to change one or both of the CRC polynomials to avoid the conflict. In practice, in all the cases we found that the two sets did *not* conflict, and we used a simple merging of the memories.

3.5. Processing Two Characters per Cycle

The performance of a HashMem system can be doubled exploiting the fact that Xilinx memory blocks provide two read

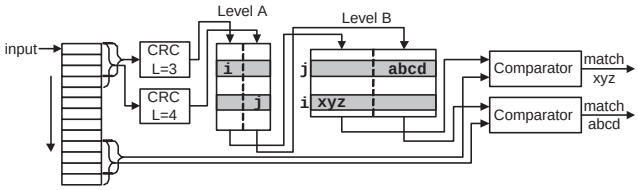

Fig. 4. Sharing of indirection and data memories for patterns of different widths. While conceptually the two ports point to the two original memories, the data is merged and the dashed separating lines indicate the separate read ports and not distinct data.

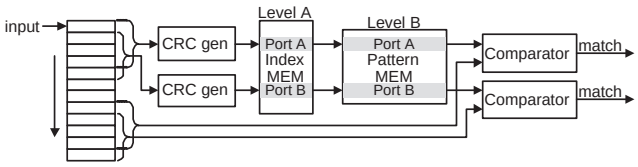

Fig. 5. Doubling the processing throughput requires replication of the CRC generators and comparators, but no new memory blocks or shift registers.

ports. Hence, we can double the processing throughput to 2 input characters per cycle, processing two patterns at offsets 0 and 1 per cycle. To achieve this, we need to provide two addresses into the memories per cycle. This means that the CRC generator logic has to be replicated, for offsets 0 and 1 in the input stream. The two addresses are fed to the two ports of the same memory (as they refer to the same pattern width). The indirection provides two pattern memory addresses, and the two possible search patterns must be match against the input for offsets 0 and 1, requiring two copies of the comparators. The overall structure for a given pattern width is shown in Figure 5. The advantage of this technique is that the throughput is doubled, without increasing the memory requirements, while the disadvantage is a 2x increase in the logic cost. Unfortunately, this technique cannot be used in conjunction with the memory sharing described in the previous subsection, since both techniques use both read ports of the Xilinx memories.

4. EXPERIMENTAL EVALUATION RESULTS

We evaluate the HashMem architecture and our implementations using the official Snort rule set [14] dated early 2004 that consists of 1474 rules and a total of 18,636 characters. We implemented the HashMem architecture in VHDL using the Xilinx ISE 6.2 tools using Spartan 3, Virtex 2 and Virtex2Pro devices, all in the highest speed grades (-5, -6, and -7 respectively). We also used automated tools that given a set of patterns generated the VHDL code for the CRC generators and the corresponding mapping of patterns to mem-

Table 1. Comparison of the HashMem architecture variants.

Design	Input bits	Freq. MHZ	Throughput (Gbps)	Logic Cells	LC/ char	MEM (blocks)	PEM
HashMem (2 chars/cc)	16	285	4.560	10,160	0.55	181	8.4
HashMEM + Reuse	8	333	2.664	2,632	0.14	66	18.9
	16	322	5.152	5,219	0.28	66	18.4
HashMem + Reuse + Share	8	338	2.704	2,570	0.14	35	19.6
HashMem + Reuse + Share + Small CRCs	8	339	2.712	2,759	0.15	31	18.3

ories. In our results, we use the number of Logic Cells, i.e. $ReportedSlices \times 2$.

Table 1 compares the variants of the HashMem architecture for a Virtex2Pro device. The first is the basic architecture that uses a full structure for each of the search pattern widths. This variant is complete and scalable and can process two characters per cycle, but with a significant logic and an excessive memory cost (181 memory blocks). Still this designs fits in a medium Virtex2Pro device.

Adding memory reuse (+Reuse) for wide patterns reduces the number of memory blocks to 66, and requires only $\sim$2,400 logic cells of logic for processing of a single character per cycle, or 0.14 logic cells per character. At a frequency of $\sim$330MHz, the processing throughput exceeds 2.5 Gbps. Doubling the processing throughput doubles the logic cost, but without additional memory use. The Performance Efficiency Metric (PEM, i.e. ratio of performance over the area cost per character) for both designs is around 18.5.

Adding memory sharing between patterns of different widths (+Share) offers a big improvement in the memory use, reducing the number of required blocks to 35. The logic cost remains about the same, due to the additional glue logic for the overall match determination. The smaller design size pushes the frequency up to about 340 MHz, slightly increasing the processing throughput and the PEM value (19.6).

The last line of Table 1 uses smaller indirection tables for the smaller pattern widths (3-6). For these widths we can identify CRC polynomials of degree 10 and 11 that provide distinct addresses for all patterns. With fewer address bits the indirection memories use 1 instead of 2 memory blocks (2Kx9 instead of 4Kx9 memory), saving a total of 4 memory blocks. This is our best design, and achieves an operating frequency of $\sim$340MHz at a cost of 0.15 logic cells per character, and a PEM value of 18.3. The area cost of the design is divided as follows: CRC generators 32%, comparators 42%, Glue logic 13% and SRLs 13%.

Table 2 offers a comparison between the best HashMem variants and other published results. For reasons of comparison we also used Spartan3 and Virtex2 devices. Our HashMem implementations achieve similar operating frequencies and use significantly less logic cells per character compared to earlier published results. The efficiency of our designs is also clear considering the PEM metric, where HashMem achieves a value of 13-19 compared to $\sim$6 of previous works, and $\sim$9 that is reached by Sourdis et al. [13]. However it is also clear from Table 2 that the HashMem architecture uses more memory than other approaches. The occupied memory however offers the opportunity to accommodate future expansions in the pattern sets without any increase in the amount of memory or logic. Our memory use is slightly lower compared to the Bloom filter implementation [10], that also has potential to add more patterns without increasing the cost, and about 3.4 times larger than the memory used by Cho and Magnione-Smith [8]. Finally, the PHmem implementation [13] that has similar architecture with HashMem, achieves the half memory use than ours but with almost double cost in logic cells per character. Still, our designs fit in medium sized Spartan3 1500 and small Virtex2 500 and Virtex2Pro 7 devices.

5. CONCLUSIONS

The goal of this work was to combine hashing and memories in order to perform cost effective, exact pattern matching. The HashMem architecture is indeed very effective in achieving low logic cost for snort-like pattern matching, with about half the cost of logic that other approaches. The amount of memory used though is clearly larger that the minimum necessary (19Kbytes, or 152Kbits). The occupied memory however, is not necessarily wasted. It offers the opportunity to accommodate future expansions in the pattern sets without any increase in the amount of memory or logic. Evidence of this potential is the almost "for-free" inclusion of the wide patterns described in section 3.3. We plan to verify this hypothesis implementing the newest Snort rule set with $\sim$2,200 more characters than the one we used.

While in practice we found it is easy to find small and efficient CRC polynomials to achieve the desired property of unique addresses for the search patterns, this is a critical requirement of HashMem. We are investigating ways to further improve the flexibility of the structures and combine patterns of different widths in the same memory using a single CRC generator and memory port. Such an improvement would also allow the doubling of the processing throughput using the second read memory ports.

ACKNOWLEDGEMENTS

This work was supported in part by the "Scaleable Intelligent Video Server System (SIVSS)" European Union FP6 IST programme (contract 002075).

6. REFERENCES

[1] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *IEEE Symposium on Field-*

Table 2. Area and performance comparison of HashMem and other architectures.

Design	Input bits	Device	Freq. MHz	Throughput (Gbps)	Logic Cells	Chars	LC/ char	MEM (kbits)	PEM
HashMem + Reuse + Share + Small CRCs	8	Virtex2Pro 7	339	2.712	2,759	18,636	0.15	558	18.3
		Virtex2 500	251	2.008	2,760		0.15		13.5
		Spartan3 1500	244	1.952	2,766		0.15		13.1
HashMem + Reuse + Share	8	Virtex2Pro 7	338	2.704	2,570	18,636	0.14	630	19.6
		Virtex2 1000	250	2.000	2,570		0.14		14.5
		Spartan3 2000	244	1.952	2,570		0.14		14.2
HashMem + Reuse	8	Virtex2 3000	244	1.952	2,636	18,636	0.14	1,188	13.8
	16	Virtex2 3000	232	3.712	5,230		0.28		13.2
Sourdis et al. PHmem [13]	8	Virtex2 1000	263	2.108	6,832	20,911	0.32	288	6.45
			361	2.886	9,426		0.45	576	6.40
	16		358	5.734	13,544		0.65	612	8.85
Attig et al. Bloom Filters [10]	8	VirtexE 2000	63	0.502	36,720	420k	0.09	629	5.6
Cho et al. RDL+ROM [8][1]	8	Spartan3 400	200	1.600	$\sim 8,000$	20,800	~ 0.38	162	4.2
		Spartan3 1000	238	1.900					5.0
Baker et al. Tree-based [9]	8	Virtex2Pro 100	237	1.896	6,340	19,584	0.32	0	5.9
Sourdis DCAM [7]	8	Virtex2 3000	335	2.678	17,538	18,036	0.97	0	2.8
		Spartan3 1500	250	2.000	19,902		1.10		1.8
Clark et al. NFAs [6]	32	Virtex2 8000	219	7.004	54,890	17,537	3.13	0	2.2

Programmable Custom Computing Machines, April 2001.

[2] R. Franklin, D. Carver, and B. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[3] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," in *Proceedings of 12th Int. Conference on Field Programmable Logic and Applications*, 2002.

[4] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Spoull, and J. Lockwood, "Deep Packet Inspection using Bloom Filters," in *Hot Interconnects*, August 2003, stanford, CA.

[6] C. R. Clark and D. E. Schimmel, "Scalable parallel pattern-matching on high-speed networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[7] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed nids pattern matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[8] Y. H. Cho and W. H. Mangione-Smith, "Programmable hardware for deep packet filtering on a large signature set," in *First Watson Conference on Interaction between Architecture, Circuits, and Compilers(P=ac2)*, 2004.

[9] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficien intrusion detection systems on FPGAs," in *Proceedings of 14th International Conference on Field Programmable Logic and Applications*, August 2004.

[10] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[11] Z. K. Baker and V. K. Prasanna, "Time and area efficient reconfigurable pattern matching on FPGAs," in *Proceedings of FPGA '04*, 2004.

[12] F. J. Burkowski, "A hardware hashing scheme in the design of a multiterm string comparator." *IEEE Transactions on Computers*, vol. 31, no. 9, pp. 825–834, 1982.

[13] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *15th Int. Conference on Field Programmable Logic and Applications*, 2005.

[14] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of LISA'99: 13th Administration Conference*, November 7 -12 1999, seattle Washington, USA.

[1] We compute the number of logic cells based on the reported device utilization.