Automatic compilation framework for Bloom filter based intrusion detection

Dinesh C Suresh, Zhi Guo*, Betul Buyukkurt and Walid A. Najjar

Department of Computer Science and Engineering
*Department of Electrical Engineering
University of California, Riverside, CA 92521.
Email : {dinesh, zguo, najjar, betul}@cs.ucr.edu

Abstract. Virus detection at the router level is rapidly gaining in importance. Hardware-based implementations have the advantage of speed and hence can support a large throughput. In this paper we describe an FPGA-based implementation of the Bloom filter virus detection code that is compiled from the native C to VHDL and mapped onto a Virtex XC2V8000 FPGA. Our results show that a single engine tailored for handling virus signatures of length eight bytes can achieve a throughput of 18.6 Gbps while occupying only 8% of the FPGA area.

1. Introduction

Studies on economic impact of computer viruses have shown that global businesses incurred an estimated $55 billion in damages during the year of 2003 [12]. The report also estimates that the monetary losses due to viruses could further increase in the forthcoming years. Therefore, containing new virus outbreaks is one of the greatest challenges facing networks and organizations. One way to control virus outbreaks is to scan for viruses at the router/interconnection points. Packets generated from infected files contain *signatures*, which are strings that uniquely identify the presence of malicious code in an incoming packet. Signatures could be distributed anywhere within a packet or across packets. By accurately identifying signatures in incoming packets, malicious packets could be blocked at the router level, thereby making the networks more secure.

Speed is the greatest concern while handling packets at the routers and hence, any router-level signature detection mechanism should be capable of identifying signatures accurately at high throughputs. This could be accomplished by a dedicated hardware (ASIC or FPGA) that inspects packets in parallel to detect signatures. Advances in high density FPGAs have provided designers with a viable commercial alternative to ASICs. Unlike ASICs, FPGAs do not require a prohibitively high cost of mask production.

In this paper, we present ROCC, a C to native VHDL compiler framework. We demonstrate this tool by using it to generate hardware for Bloom-filter based virus detection. Our compiler framework can be easily adapted to accommodate new algorithms

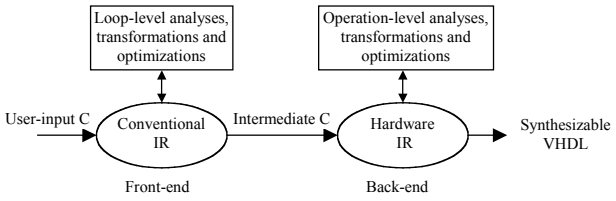

Figure 1 - ROCCC compiler framework

for virus detection and our generated hardware achieves multi-gigabit throughputs. Our contributions in this paper can be summarized as follows. This paper presents the first work in which a Bloom-filter based virus detection system is automatically generated from C code. We illustrate that automatic code generation is a feasible option in terms of the performance and area utilization of the FPGA. Our 8-byte Bloom filter code delivers a throughput of 18.6 Gbps while occupying a modest chip area of 8%.

2. Overview of the ROCCC C to VHDL Compiler

ROCCC [15] is built on the SUIF2 [13] and Machine-SUIF [14] platforms. Figure 1 shows ROCCC's system overview. It compiles code written in C/C++ or Fortran to VHDL code for mapping onto the FPGA fabric of a CSoC device. In the execution model underlying ROCCC, sequential computations are carried out on the microprocessor in the CSoC, while the compute intensive code segments are mapped onto the FPGA. These typically consist of loop nests, most often parallel loops, operating on large arrays or streams of data. Therefore, most loop level analysis and optimizations are done at this level. Most of the information needed to design high-level components, such as controllers and address generators, is extracted from this level's IRs.

The front-end of ROCCC performs a very extensive set of loop analysis and transformations aiming at maximizing parallelism and minimizing the area. The transformations include loop unrolling and strip-mining, loop fusion and common sub-expression elimination across multiple loop iterations. . The work reported in [7] shows that in less than one millisecond and within 5% accuracy compile time area estimation can be achieved. Information to generate high-level units, such as controllers and buffers, is also extracted from SUIF IRs. The restrictions on the C code that can be accepted by the ROCCC compiler, for mapping on an FPGA fabric, include no recursion, no usage of pointers that cannot be statically un-aliased. Function calls will either be in-lined or whenever feasible made into a lookup table. In the following section, we explain the operation of a Bloom filter for virus detection.

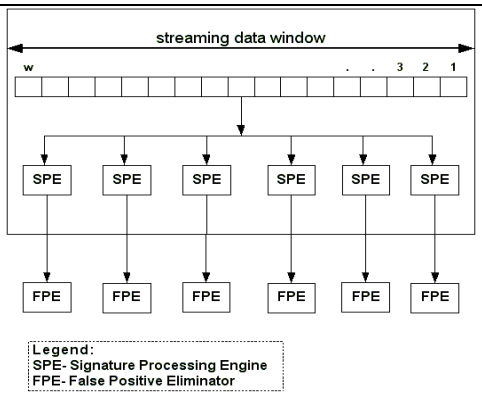

Figure 2 - The Bloom filter detects malicious packets at the network layer. The output of the Signature Processing Engine (SPE) is fed to the False Positive Eliminator (FPE)

```
for (k=0;k<224;k++)
{
  for  (j =0; j < 32; j ++)
   {
     for(i=0;i<8;i++)
     {
            temp = value & 0xff;
            result_location1 = temp ^ hash_function1[i];
            result_location2 = temp ^ hash_function2[i];
            result_location3 = temp ^ hash_function3[i];
            result_location4 = temp ^ hash_function4[i];

            value = value >> 8;

         found = bit_array[result_location1] & bit_array[result_location2]
          bit_array[result_location3] & bit_array[result_location4];
        return (found);
     }
   }
 }
```

Figure 3 - Bloom filter code for identifying signatures of width 8 bytes each in a stream of size 256 bytes. The packet size is assumed to be 64 bytes. The data structure bit array is a Bloom filter of size 256 entries

3. Bloom Filters

A Bloom filter [3] s a space-efficient data structure used to test the set membership of an element. An empty Bloom filter is described by an array of m bits, initially all set to 0. A Bloom filter uses K independent hash functions $h_1 \ldots h_k$ with range *{0… m-1}*. Each of these hash functions map an incoming item to a number in the range of *{0 …m-1}*.During insertion, hash functions $h_1 \ldots h_k$ are applied to the input item. Each return value from the hash function is used as an index to the Bloom filter (array of m bits) and the appropriate bit position is set to 1. A location can be set to 1 multiple times, but only the first change has an effect.

During a search operation, the locations returned by the hash functions are checked to see if they are already set to '1'. If bit values in all the return locations are set, then the Bloom filter is said to contain the pattern else it is a miss. An item x belongs to the set S with some probability if all $h_i(x)$ are set to 1 for 1<i<k. .If not, then x is not a member of S. A Bloom filter may yield a *false positive* when it suggests that an element x belongs to S even though it does not. The probability of a false positive is given by

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

Where, k,m and n denote the number of hash functions, number of bits in the bloom filter array and the number of elements currently inserted into the bloom filter, respectively. In the event of a match in the Bloom filter, a detailed string matching is performed using a RAM to ensure that the hit was not a false positive.

A functional prototype of the Bloom-filter based intrusion detection system has been implemented. We used a Bloom filter of length 256 bytes to detect patterns of length 8 bytes. Figure 2 shows the block diagram for a bloom-filter based virus detection system. Figure 3 shows a Bloom filter code that uses four hash functions on a 256 entry Bloom filter array. The hash functions are implemented as a simple XOR operation. The result of each hash operation sets a location in the Bloom filter. The compiler unrolls the inner most loop to compare the 8-byte patterns in parallel. We used the rule sets contained in bleeding snort database [11]. Each rule consists of two parts: a header and a rule option. The header is mainly used for packet classification and contains information like the protocol, source IP, source port, destination IP and the destination port. The rule option contains the signatures to be used in intrusion detection.

Figure 5 shows the frequency of the signature width of all rules in the bleeding snort database. As evident from the figure, most of the rules present in the bleeding snort database have a signature width of less than 30 bytes. In the following section, we present the generation of datapath and throughput evaluation for the Bloom filter code.

4. Datapath Generation and Throughput Analysis

Figure 4 shows the three-stage pipeline for the generated Bloom filter circuit. The XOR operation shown in the figure represents each byte of the input being XOR ed with one byte of the hashing function. The location returned by the hashing function is looked up and if all four hashing function lookups return a value of '1', then the circuit reports the current input pattern as malicious.

The compiler groups the instructions in each node into different execution levels to exploit instruction (operation) level parallelism. Instructions at the same level are executed simultaneously. Every level of the dataflow graph corresponds to the instantiation of one

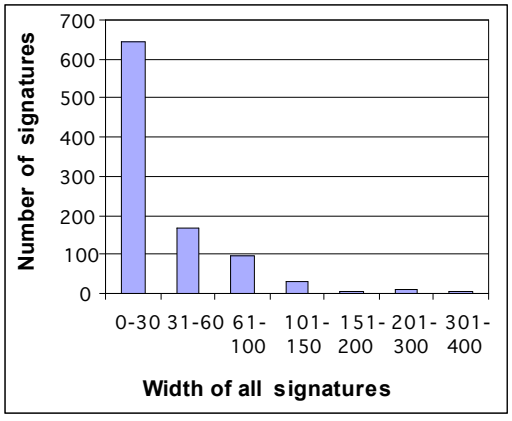

Figure 5 - Histogram of signature width for all rules in the snort database. The most frequently occurring signatures have a width of around 32 bytes

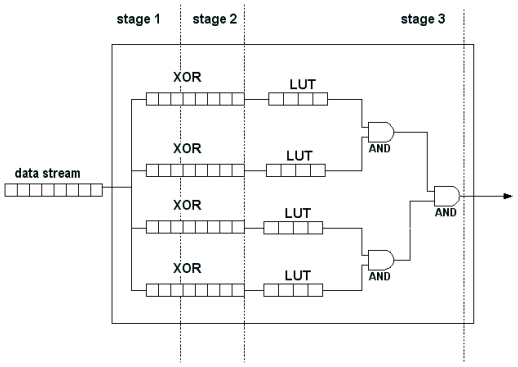

Figure 4 - Three-stage pipeline for the Bloom filter signature code. Each box in the XOR filed represents a byte of the input being XOR ed with one byte of the hashing function

loop iteration. ROCCC automatically places latches in the data-path for pipelining. Every latched level corresponds to one pipeline stage, and has a delay of one cycle.

Our Bloom filter code does not have loop-carried-dependency and the compiler fully pipelines the data-path. Therefore, the generated data-path can be fed with new set of input every clock cycle and the throughput is one iteration per cycle. We process 8 bytes each iteration. When we do loop unrolling, we assume that the memory-bus width also scales up with unrolling.

The clock frequency of the FPGA was found to be 73 MHz. The system uses a total of 4692 slices, which accounts for 8% of the total FPGA area. The BRAM on our target FPGA (XC2V8000) can process 256 bits (32 bytes) per cycle. Hence, the BRAM can support eight such hardware instances during each cycle. The total throughput of our hardware is given by

$$\textit{Throughput} = \textit{bits per cycle} * \textit{clock frequency}$$

$$T = 8 * 32 * 73 * 10^{6} \textit{ bits/ sec.} = 18.6 \textit{ Gbps.}$$

The throughput shown above is for a system that detects multiple signatures of a single width. When multiple instances for each signature width are instantiated, the overall circuit area would also increase proportionately. Synthesis tools tend to produce slower circuits when the design size increases. However, with increase in area, the compiler produces more parallel iterations and hence, the performance loss due to decrease in clock speed is overcome by the increase in parallelism. In order to provide a better insight into our estimated throughput values, we examine the throughput achieved by previously published works.

5. Related work

Hashmem [9] combines memory and hashing effectively to achieve exact matching of intrusion signatures at throughputs of up to 3.7Gbps while using nearly 0.15 logic cells per character. Baker and Prasanna [2] use automatic compilation to synthesize FPGA architectures that perform deep packet inspection at 10Gbps. Clark et al.[5] use NFAs with predecoded inputs to achieve excellent area and throughput performance. Lockwood et. al.[8] used the Field Programmable Port extender (FPX) platform for expression matching. Their synthesized circuit achieved clock speeds of 37MHz on a Virtex XCV2000E FPGA.

Gokhale et.al [6] used CAM to implement snort rules on a Virtex XCV1000E FPGA. Their hardware delivered a throughput of 2.2Gbps. Cho et. al [4] generated structural VHDL for deep packet filtering on an FPGA. Their design runs at 90MHz on an Altera EP20K device and achieves a throughput of 2.88Gbps. Attig et. al..[1] have implemented a Bloom filter circuit on a Virtex E2000 FPGA. Their circuit operates at 62.8MHz and provides a throughput of 502Mbps. This paper presents the first reported work that *automatically generates native VHDL for Bloom filter based intrusion detection code written in C.*

6. Conclusion

In this paper, we have described using ROCC, a C to VHDL compiler, to generate Bloom-filter based virus detection system on FPGAs. Ours is the first work that automatically generates VHDL for Bloom filter code written in C. We evaluate the performance and area of the synthesized hardware and prove that automatic compilation to hardware is a feasible design option. Our synthesized hardware runs at 73 MHz and delivers a throughput of 18.6 Gbps while occupying a modest FPGA real estate of 8%.

References

[1]. M. Attig, S. Dharmapurikar, J. Lockwood. "Implementation Results of Bloom Filters for String Matching,", In proceedings of the *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages. 322-323, 2004

[2]. Z. Baker and V.K. Prasanna, "High Throughput Linked-Pattern Matching for Intrusion Detection Systems, In Proceedings of *Symposium on Architectures for Networking and Communication Systems* (ANCS' 05) , Princeton, New Jersey, October 2005.

[3]. B.H. Bloom. "Space/time tradeoffs in hash coding with allowable errors", *Communications of the ACM,* 13(7): pages 422-426, July 1976.

[4]. Y. H. Cho, W. M. Smith, "Specialized Hardware for deep packet filtering", In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications* (2002), France.

[5]. C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, California, April 2004.

[6]. M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. "Granidt: Towards gigabit rate network intrusion detection technology", . In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 404–413, 2002.

[7]. D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi, Fast Area Estimation to Support Compiler Optimizations in FPGA-based Reconfigurable Systems, IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, April 2002.

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

[9]. G. Papadopoulos and D. Pnevmatikatos, "Hashing + Memory = Low Cost, Exact Pattern Matching," in Proceedings of 15th International Conference on Field Programmable Logic and Applications, Tampere, Finlad, August 2005

[10]. I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10Gbps FPGA-based network intrustion detection system",. In *Proceedings of International Conference on Field Programmable Logic and Applications*, 2003.

[11]. http://www.bleedingsnort.com

[12]. http://news.designtechnica.com/article2401.html.

[13]. SUIF Compiler System. http://suif.stanford.edu.

[14]. Machine-SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html .

[15]. Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. “Optimized Generation of Data-Path from C Codes”. In ACM/IEEE Design Automation and Test Europe (DATE), Munich, Germany, March 2005.