HARDWARE-SOFTWARE CODESIGN FOR HIGH-SPEED SIGNATURE-BASED VIRUS SCANNING

HIGH-SPEED NETWORK CONTENT SECURITY APPLICATIONS OFTEN OFFLOAD SIGNATURE MATCHING TO HARDWARE. IN SUCH SYSTEMS, THE THROUGHPUT OF THE OVERALL SYSTEM, RATHER THAN THE HARDWARE ENGINE ALONE, IS SIGNIFICANT. THE AUTHORS OFFLOAD VIRUS SCANNING IN THE CLAMAV ANTIVIRUS PACKAGE TO THE BFAST* HARDWARE ENGINE. THEY FIND THAT THE DATA-PASSING PROCESSES SIGNIFICANTLY DEGRADE SYSTEM THROUGHPUT.

Ying-Dar Lin
National Chiao Tung University

Po-Ching Lin
National Chung Cheng University

Yuan-Cheng Lai
National Taiwan University of Science and Technology

Tai-Ying Liu
Avermedia

Network content-security applications—intrusion-detection systems, content-filtering systems, antivirus systems, and so on—use signature matching to inspect application payload for attacks or virus signatures. The efficiency of signature matching is thus critical to system performance. A promising trend for high-speed applications demanding multigigabit throughput is to offload inspection to a hardware engine.[1-5] This offloading requires efficient and elegant integration between software and hardware implementations. Most studies of this issue emphasize only the raw throughput of scanning engines. However, the entire process requires careful examination for other potential bottlenecks, or the overall system throughput could degrade, as limited by Amdahl's law.

The data path from software and hardware involves passing the data to be scanned:

- from the user space to the device driver in the kernel space,
- from the driver to the direct memory access (DMA) buffer (also in the main memory), and
- from the buffer to the TextRAM in the scanning engine.

This article examines the performance in each stage and identifies potential bottlenecks. We offload virus scanning from the ClamAV antivirus package, which uses the Wu-Manber and Aho-Corasick algorithms (see the "String-matching algorithms in ClamAV" sidebar), to the BFAST* hardware engine, the prototype of the Bloom Filter Accelerated Sublinear Time (BFAST) architecture (see the related sidebar). Like most scanning engines, the design's raw throughput is up to multigigabits per second. To achieve high throughput, the design features sublinear execution time on the average by skipping characters not in a match, and thus can effectively inspect multiple characters at a time. The scanning engine can quickly filter out the no-match cases, because most network traffic and files don't contain viruses.

Published by the IEEE Computer Society

String-matching algorithms in ClamAV

ClamAV (www.clamav.net) has signatures of nonpolymorphic viruses in simple strings and polymorphic viruses in regular expressions.[1] The Wu-Manber algorithm[2] is implemented in the Matcher-bm library, and is responsible for 225,000 nonpolymorphic signatures in ClamAV version 0.91.2. (In the library and function names of ClamAV, the Wu-Manber algorithm is incorrectly dubbed the Boyer-Moore algorithm, which also can skip characters not in a match, but scans only one pattern at a time.) The Wu-Manber algorithm features a search window of l_{min} characters sliding along the data, where l_{min} is the shortest pattern length, and can skip characters not in a match according to the following heuristic: If the rightmost block in the window appears in none of the patterns, we can shift the window by $l_{min} - B + 1$ characters without missing any pattern, where B is the block size. Otherwise, the shift distance is $l_{min} - j$, where the block's rightmost occurrence in the patterns ends at position j. ClamAV precomputes and stores every possible block's shift value (arbitrarily choosing $B = 3$) in a shift table of approximately 64 KBytes by filling in the smallest shift value in an entry if multiple blocks are mapped to that entry. The table compression might reduce the shift value, but no match will be missed in this way.

The Aho-Corasick algorithm,[3] implemented in the Matcher-ac library, is responsible for approximately 13,000 polymorphic signatures (in version 0.91.2). It builds a finite automaton that accepts all the patterns in preprocessing, and then tracks the data for a match during scanning. ClamAV scans for signatures by dividing each signature into individual parts, and builds the automaton to scan for these parts. ClamAV records those parts that have been matched, and verifies whether all the parts in a signature have appeared in a sequence or not. For example, the signature ''abcdefg{-10}hijkl'' contains two parts—''abcdefg'' and ''hijkl''—which the Aho-Corasick algorithm matches individually. When it finds the second part (''hijkl''), ClamAV also verifies that the gap between the two parts has no more than 10 characters, as specified in the signature.

ClamAV uses both algorithms for virus scanning for the two reasons. First, many parts in the polymorphic signatures are short, and they restrict the maximum shift distance allowed (bounded by the shortest pattern) in the Wu-Manber algorithm. Matching the polymorphic signatures in the Aho-Corasick algorithm can avoid this problem. Second, compared with the sparse automaton representation in the Aho-Corasick algorithm, the compressed shift table is a compact representation of a large number of nonpolymorphic signatures in simple strings, so the Wu-Manber algorithm is a memory-efficient solution.

References

1. O. Erdogan and P. Cao, ''Hash-av: Fast Virus Signature Scanning by Cache-Resident Filters,'' *Proc. IEEE Globecom Conference,* IEEE Press, 2005, pp. 1767-1772.
2. S. Wu and U. Manber, *A Fast Algorithm for Multi-pattern Searching,* tech. report TR94-17, Dept. of Computer Science, Univ. of Arizona, 1994.
3. A. Aho and M. Corasick, ''Efficient String Matching: An Aid to Bibliographic Search,'' *Comm. ACM,* vol. 18, no. 6, June 1975, pp. 333-343.

Implementation in the hardware-software codesign

Figure 1a shows the hardware and software partitioning in the hardware-software codesign. ClamAV offloads the major functions of signature matching to BFAST*, which quickly filters out the data segment without viruses. The Matcher-bfast library calls the driver to allow the DMA to transfer data into the memory on BFAST*, which then scans for the signatures in simple strings. If BFAST* detects a suspicious virus, ClamAV passes the suspicious data segment to the Matcher-bm library for further verification—for example, to check whether the signature occurs in the correct position because the occurrence is significant only at a certain position. Because most data contain no viruses, the filtering is efficient.

BFAST* could help scan for the long parts in the multipart signatures scanned in the Matcher-ac library, as it performs very fast for those. However, the Matcher-ac library must prevent the short parts from shortening the shift distance in BFAST* and slowing the performance because of the distance bound by the shortest pattern length. Although BFAST* doesn't handle the short parts efficiently, we can adapt several hardware designs based on tracking an automaton[4] to match them. BFAST* can work with those designs for matching multipart signatures for high-speed inspection. This scheme has an advantage. Because long strings could generate many sparse nodes if represented in an automaton and waste memory,[6] leaving them to BFAST* can reduce the memory requirement in those designs. Note that Figure 1a doesn't show the Matcher-ac library, because we don't tackle it here.

Hardware design

Figure 1b shows the five modules in BFAST*:

- *TextPoint* stores the search window's current position, which is initialized by two registers (EnableTextRam0

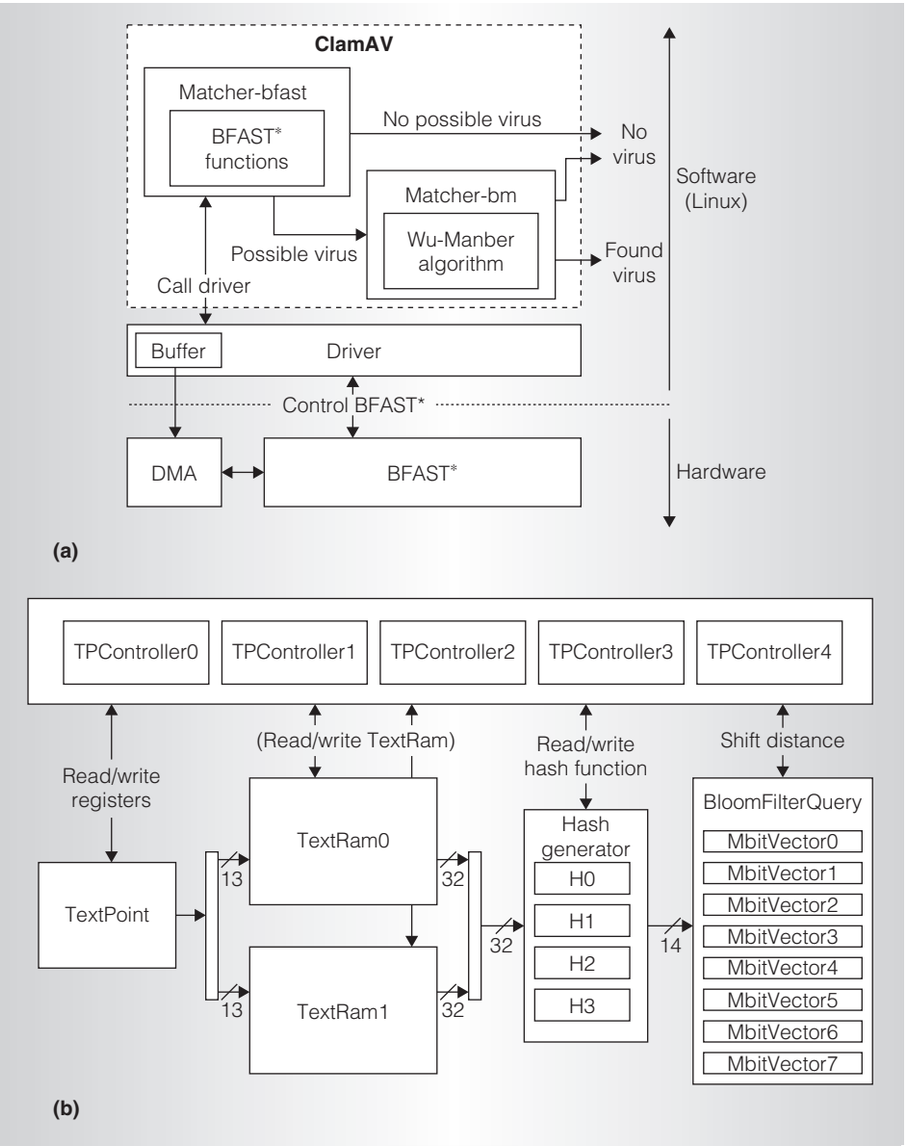

Figure 1. Hardware-software codesign of virus scanning in ClamAV: hardware-software partitioning in the codesign (a) and the five modules in BFAST* (b).

and EnableTextRam1) that specify the starting addresses and data lengths.

- Two 8-KByte *TextRam* blocks (TextRam0 and TextRam1) store the data to be scanned alternatively. While BFAST* scans one block, it can load data into the other.
- *HashGenerator* generates four 14-bit hash values of the block to be looked up in the Bloom filters from four hash functions: H0, H1, H2, and H3.
- *BloomFilterQuery* derives the shift distance by querying the Bloom filters. This module includes eight 4-KByte MbitVectors that store signatures in the Bloom filters.
- *TPController* controls the other four modules, shifts the search window, and stores the status in registers. We implemented five TPControllers for the five pipeline stages, and each scans one-fifth of the TextRam data buffer.

BFAST architecture

The BFAST architecture searches in sublinear time using Bloom filters.[1] It uses an algorithmic heuristic to derive the search window's shift distance by querying the Bloom filters. The heuristic avoids significantly reducing shift values in the Wu-Manber algorithm due to the compressed shift table (see the ''String-matching algorithms in ClamAV'' sidebar), and achieves high throughput by rapidly skipping characters that can't be a match. Figure A shows the architecture's three main components. The scanning module fetches data from the *text memory fetch* block in the search window located by the *text counter generator*, looks it up in the Bloom filters, and then shifts the window according to this query result. Next, the verification interface passes the location of a possible match to the *verification job buffer*, and if the buffer is nonempty, the verification module fetches the verification jobs and determines if a virus appears.

Reference

1. P.-C. Lin et al., ''Realizing a Sub-linear Time String-Matching Algorithm with a Hardware Accelerator Using Bloom Filters,'' *IEEE Trans. VLSI Systems,* vol. 17, no. 8, Aug. 2009, pp. 1008-1020.

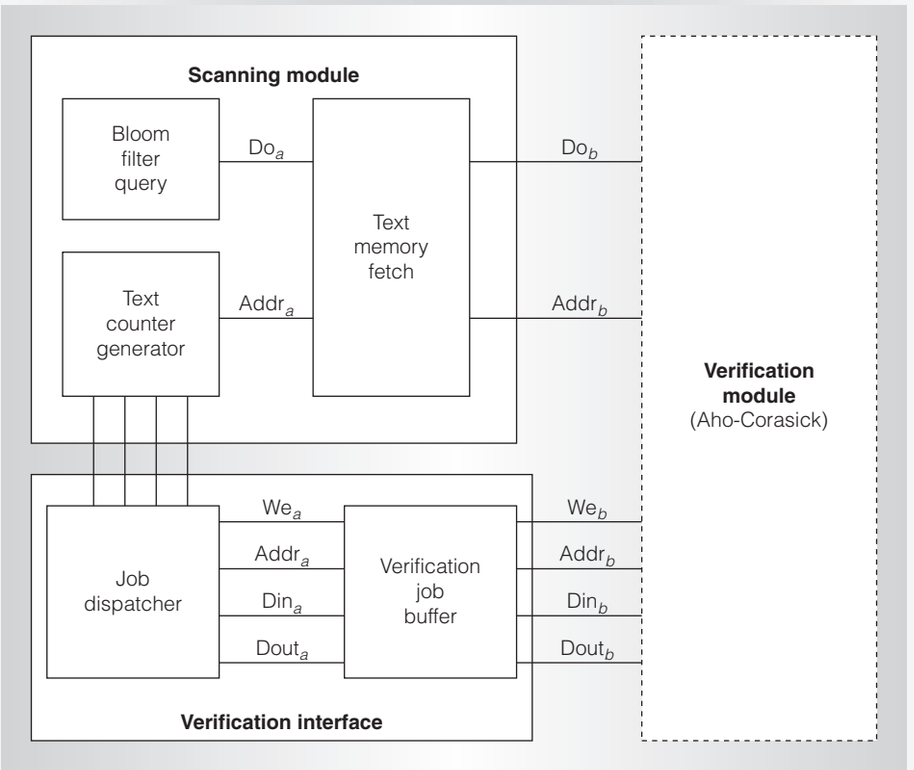

Figure A. The three main components in the BFAST architecture.

To increase efficiency, we implement a five-stage pipeline in BFAST* to overlap the operations of looking up the Bloom filters and shifting the search window. As noted, each TPController scans one-fifth of the 8-KByte TextRam buffer. Because a signature might span two contiguous data segments, each TPController should scan $l_{\min} - 1$ characters ahead of the segment it handles, where $l_{\min}$ is the search window size (and the shortest pattern length), so no signatures are missed. Scanning $l_{\min} - 1$ characters ahead is sufficient because the last segment must have a search window of $l_{\min}$ characters, and can be found by that segment's TPController.

Software design

ClamAV fetches information from BFAST* in two ways. First, if it finds a suspicious virus, BFAST* can *interrupt* the CPU and report to ClamAV that the virus should be verified. The CPU can perform other tasks while BFAST* scans for viruses; however, ClamAV's context switch following an

interrupt is overhead. If many suspicious viruses appear, this overhead can be large. Second, ClamAV can detect BFAST*'s status by *polling*. Because ClamAV can immediately recognize a suspicious virus without waiting for a context switch, polling increases the throughput of virus scanning but decreases system performance. We opt for polling for higher throughput in virus scanning.

To implement the Matcher-bfast library, we add three BFAST* scanning functions to Matcher-bm:

1. Initially, the *bfast_init* function opens the device and returns the BFAST* file descriptor.
2. When ClamAV reads the signatures from the ClamAV database, it calls the *bfast_addpatt* function to store the patterns into MbitVectors.
3. Finally, ClamAV invokes the *bfast_scanbuff* function to scan the data in the buffer loaded from files.

Hardware-software interface

ClamAV invokes the Matcher-bfast library, which in turn calls the driver to enable BFAST* to scan data in TextRam and to get the BFAST* status. The driver provides functions to access data in TextRam0, TextRam1, HashGenerator, and the MbitVectors. These functions include memory writing, virus scanning, and status fetching. The scanning process requires several data structures: the hash functions, blocks in the patterns, and the data to be scanned. After the data are well prepared, the TPControllers start to scan the data and write the status to registers.

The DMA transfers data from physical memory to the TextRam in BFAST*, but the buffer to be scanned in the user space might not be actually continuous in physical memory space; rather, it just appears continuously in virtual memory. Therefore, the driver allocates a piece of continuous physical memory space and copies the data from ClamAV into this space. This continuous space can't be cached in the CPU. If it was, the DMA might transfer stale data into the TextRam, while the latest data remains in the CPU cache.

Experiment results

We implemented our system on a Xilinx ML310 VirtexII Pro Board with a soft CPU operating at 300 MHz. ClamAV runs on MontaVista Linux 2.4.20-8. We attach BFAST* on the process local bus (PLB). The on-chip peripheral bus (OPB) and the PLB both operate at 100 MHz. The PLB's maximum data width is 64 bits, twice the OPB's width.

ClamAV loads the signatures from the signature database into BFAST*'s BloomFilterQuery module with the Matcher-bfast library. After loading the signatures, ClamAV reads the file to be scanned in 128-KByte batches, and the Matcher-bfast library invokes the driver to sequentially load the file content into the driver's buffer and let the DMA transfer the data from the buffer into TextRam. BFAST* then scans the data in TextRam. If BFAST* finds no viruses, it reports that the file isn't infected. If BFAST* finds a virus, ClamAV passes the data segment to Matcher-bm for further verification.

We benchmark the performance in the following conditions: ClamAV on the ML310 board, BFAST* hardware simulation, BFAST* hardware implementation on the ML310 board, and ClamAV with BFAST*.

Pure software

First, we ran the original ClamAV on the Xilinx ML310 Vertex II Pro Board without offloading signature matching. In this experiment, ClamAV scans 1-KByte and 1-MByte data files with and without viruses for 42,108 signatures. If Matcher-bm reports that a file isn't infected, ClamAV passes the file to Matcher-ac for scanning multipart signatures. Here, we only consider the performance of matching nonpolymorphic signatures, as the hardware architecture doesn't accelerate Matcher-ac at all, and the simplification is sufficient for studying the performance issues in the hardware-software codesign. Table 1 lists ClamAV's execution times for scanning 1-KByte and 1-MByte files. If Matcher-bm finds a virus, it reports the finding and immediately stops scanning the file. As Table 1 shows, Matcher-bm's throughput for scanning a 1-MByte file is $(8 \times 1{,}024 \times 1{,}024 \text{ bits})/1{,}559{,}306 \text{ ms} = 5.38$ megabits per second.

Hardware simulation and synthesis

We also implemented and synthesized BFAST* in Verilog. In the hardware

simulation, BFAST* operates at 142.507 MHz. The average shift distance is 7.71, so the average throughput is 142.507 × 7.71 × 8 (8.79 gigabits per second), and the maximum throughput is 142.507 × 8 × 8 (9.12 Gbps) if TextRam is full and the shift distance is 8.

System on chip with no operating system

The Xilinx ML310 Vertex II Pro Board supports only four clock rates: 25 MHz, 33 MHz, 50 MHz, and 100 MHz. Our design can operate at 142 MHz, so we chose 100 MHz as the PLB clock rate. Figure 2 shows the scanning times for various data lengths. For lengths shorter than 1,600 bytes, BFAST* enables TPController0, and each shift takes five cycles. Otherwise, the scanning time is approximately 1,600/8 × 5 = 1,000 cycles (assuming skipping eight characters in the ideal case), because when TPController0 is scanning the window at address 0, TPController1 is scanning at address 1,600, TPController2 is scanning at address 3,200, and so on. In the pipelining architecture, the maximum throughput is 8,000 bytes/(1,004 × 10 ns) = 6.37 Gbps when the 8,000-byte data buffer is fully loaded.

System on chip with Linux

We used three configurations, depending on the location of the data to be scanned.

Data is in the user space. Table 2 shows the throughput of ClamAV when BFAST* scans 1-KByte and 1-MByte files. The execution time involves transferring the data from the user space into BFAST* and scanning for viruses. When ClamAV scans a file without finding a suspicious match, it only invokes Matcher-bfast; otherwise, it launches Matcher-bm for verification. ClamAV's total throughput of approximately (8 × 1,024 × 1,024)/55,552 = 151 Mbps for scanning a 1-MByte file and writing data from the user space into TextRam takes approximately 90 percent of the execution time (48,891/55,552 × 100 percent = 89.81 percent).

To investigate the significant disparity between the throughput of a pure hardware accelerator and the codesign, we look at the time spent moving data from the user space to TextRam in BFAST*, as Figure 3a shows. Although scanning on the field-programmable gate array (FPGA) is fast,

Table 1. Execution time of ClamAV using matcher-bm on ML310 (in microseconds).

File size	With virus	Without virus
1-Kbyte	539 (virus found at the 359th byte)	1,545
1-MByte	1,102,765 (virus found at the 738,663th byte)	1,559,306

Note: If we assign an address to each byte of data, with the virus signature starting at position *i*, ClamAV finds the virus at the *i*th byte.

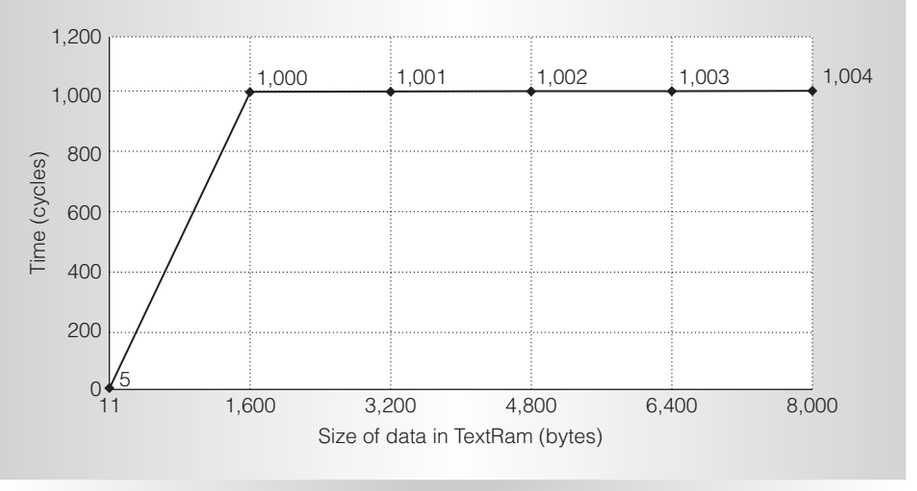

Figure 2. Scanning time for various data lengths.

Table 2. Execution time of ClamAV integrated with BFAST* on ML310 (in microseconds).

	With virus		Without virus	
File size	**Matcher-bfast**	**Matcher-bm**	**Matcher-bfast**	**Matcher-bm**
1-Kbyte	270 (time to write data into TextRam: 94)	17 (virus found at the 359th bytes)	273 (time to write data into TextRam: 95)	N/A
1-MByte	40,810 (time to write data into TextRam: 36,656)	26 (virus found at the 738,663th byte)	55,552 (time to write data into TextRam: 49,891)	N/A

Note: If we assign an address to each byte of data, with the virus signature starting at position *i*, ClamAV finds the virus at the *i*th byte.

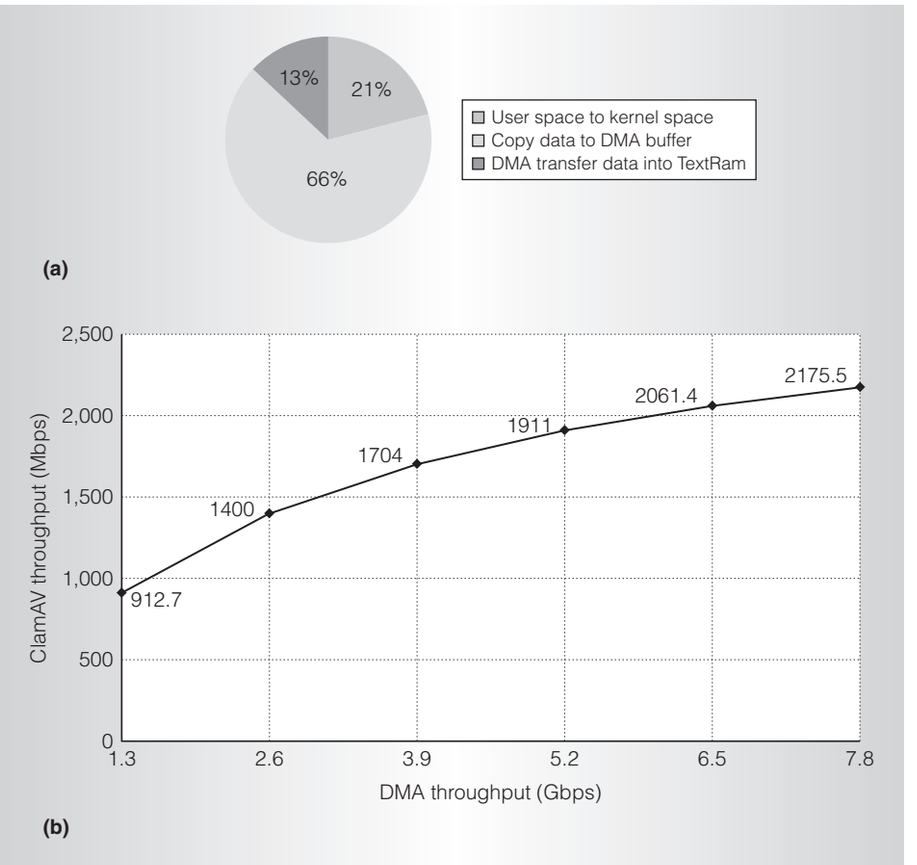

Figure 3. Performance analysis of data transfer: the percentages of the data transfer time in each stage of virus scanning (a), and the system throughput for various DMA throughputs (b).

ClamAV must pass the data to be scanned from the user space to the device driver in the kernel space. Next, the driver copies the data into the DMA buffer, and the DMA transfers the data in the buffer into TextRam. These three data-passing steps occupy almost 90 percent of the total processing time, significantly slowing the throughput. Copying

data into the DMA buffer is the most time-consuming step.

Data is in the DMA buffer. Given the above observation, merely pursuing a fast scanning engine hardware design is insufficient. Two steps are fundamental and inevitable in the design:

- having the data in the DMA buffer and
- transferring the data from the DMA buffer into the FPGA with DMA.

Even if the incoming data is already stored in the DMA buffer (thus eliminating the most time-consuming step), the data-transfer step isn't fast enough in our platform. It's only 1.3 Gbps, and the overall system throughput is 912.7 Mbps, so the DMA also restricts the throughput. Improving the DMA throughput would thus result in a faster design.

Data is in the DMA buffer with improved DMA throughput. Figure 3b shows the relationship between the DMA and the overall throughput. When ClamAV scans 1 MByte of data in main memory, the system takes 2,789 microseconds to monitor the BFAST* status until the scanning finishes, and to do such housekeeping work as copying $l_{\min} - 1$ characters ahead of the buffer and enabling the DMA. Given our platform's DMA transfer rate of 1.3 Gbps, loading data into TextRam takes 6,402 microseconds. ClamAV's throughput is therefore $8 \times 1{,}024 \times 1{,}024/(2{,}789 + 6{,}402)$, or 912.7 Mbps. As Figure 3b shows, if the DMA transfer is 7.8 Gbps, ClamAV can reach up to 2.176 Gbps. If the DMA could be infinitely fast, the throughput could be $8 \times 1{,}024 \times 1{,}024/2{,}789 = 3.01$ Gbps.

Related strategies

Previous work has sought to avoid the overheads of memory copying by using methods such as the zero-copy technology and streaming data into a hardware engine.[5,7,8] Although these solutions are effective, our work involves a different scenario—the application content to be scanned comes from ClamAV, not a network interface card (NIC).

There are subtle differences between the two sources. First, virus scanning shouldn't operate on a per-packet basis. Raw packets from the NIC must be handled by the TCP/IP protocol stack to restore the original application content, or factors such as packet loss, out-of-order packets, or an attacker's deliberate network evasion[9] will make the scanning results unreliable. Second, the application content can be processed before being scanned. The processing might include decoding from a mail message or decompressing from a compressed file. Therefore, raw packets (or raw file content, if read from a file system) must still be handled by ClamAV running in the user space, rather than directly pumped into the hardware scanning engine.

Our system flow—although not as fast as an in-line implementation—is necessary for reliable scanning and full-fledged functions. Zero-copy strategies are better for applications that work on a per-packet basis, such as routing a packet, or those that don't need to process application content in a user program.

This work implements the BFAST* virus-scanning engine on the ML310 board to increase ClamAV's throughput. The throughput of scanning a large file becomes approximately 151 Mbps—28.1 times faster than the original ClamAV's throughput. However, this throughput is still much lower than that in the hardware simulation—8.79 Gbps on the average and 9.12 Gbps in the best case.

Several factors contribute to the slower throughput. First, the ML310 board's fastest bus clock rate is 100 MHz, so the best throughput is $8 \times 8 \times 100 = 6.4$ Gbps, and the average throughput is $7.71 \times 8 \times 100 = 6.17$ Gbps. This bus clock lowers the throughput from 8.79 Gbps in the simulation. Second, even if the data to be scanned is in the DMA buffer and the DMA transfer is infinitely fast, the throughput is only 3.01 Gbps because of the housekeeping work between scanning each batch of data in the TextRAM. Third, our system's DMA doesn't perform fast enough, making the throughput only 912.7 Mbps, even though the data is in the DMA buffer. Finally, copying data from the user space to the DMA buffer is time consuming, making the codesign's overall throughput only 151 Mbps.

Implementing ClamAV in the kernel could remove the need to pass data between the user space and the kernel space. Moreover, having the driver copy the data from the kernel space to the DMA buffer (also in the main memory) is necessary for two reasons. First, it ensures that the data is indeed written into the memory, rather than only the CPU cache, unless the cache is write-through. Second, it ensures that the data is stored contiguously in the physical addresses for the DMA to fetch. Note that the addresses of the data might not be contiguous in physical memory space. The DMA must somehow know the physical addresses to fetch the data—perhaps through the memory management unit (MMU). Therefore, copying the data into the noncacheable DMA buffer is a safe solution. If the data buffer could be made noncacheable, and the DMA can fetch the text in contiguous addresses or through the MMU, the heavy overhead could be eliminated. MICRO

Acknowledgments

This work was supported in part by the Taiwan National Science Council's Program of Excellence in Research, and in part by grants from Cisco and Intel.

References

1. P.-C. Lin et al., ''Using String Matching for Deep Packet Inspection,'' *Computer,* vol. 41, no. 4, Apr. 2008, pp. 23-28.
2. M. Aldwairi, T. Conte, and P. Franzon, ''Configurable String Matching Hardware for Speeding Up Intrusion Detection,'' *ACM SIGARCH Computer Architecture News,* vol. 33, no. 1, 2005, pp. 99-107.
3. Z.K. Baker and V.K. Prasanna, ''A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs,'' *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 04), IEEE CS Press, 2004, pp.135-144.
4. L. Tan, B. Brotherton, and T. Sherwood, ''Bit-Split String-Matching Engines for Intrusion Detection and Prevention,'' *ACM Trans. Architecture and Code Optimization,* vol. 3, no. 1, Mar. 2006, pp. 3-34.
5. S. Dharmapurikar, M. Attig, and J. Lockwood, ''Deep Packet Inspection Using Parallel Bloom Filters,'' *IEEE Micro,* vol. 24, no. 1, Jan./Feb. 2004, pp. 52-61.
6. N. Tuck et al., ''Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection,'' *Proc. IEEE Infocom Conf.,* IEEE Press, 2004, pp. 2628- 2639.
7. T. Liu et al., ''The Design and Implementation of Zero-Copy for Linux,'' *Proc. 8th Int'l Conf. Intelligent Systems Design and Application* (ISDA 08), IEEE CS Press, 2008, pp. 121-126.
8. P. Halvorsen et al., ''Performance Tradeoffs for Static Allocation of Zero-Copy Buffers,'' *Proc. Euromicro Conf.* (EuroMicro 02), IEEE CS Press, 2002, pp. 138-143.
9. M. Handley, C. Kreibich, and V. Paxson, ''Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics,'' *Proc. Usenix Security Symp.,* Usenix Assoc., 2001, pp. 115-131.

Ying-Dar Lin is a professor in the Department of Computer and Information Science at National Chiao Tung University, Hsinchu, Taiwan. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, wire-speed switching and routing, quality of service, deep packet inspection, network processors and systems on chip, and embedded hardware-software codesign. Lin has a PhD in computer science from the University of California, Los Angeles.

Po-Ching Lin is an assistant professor in the Department of Computer and Information Science at the National Chung Cheng University, Chiayi, Taiwan. His research interests include network security, content networking, and algorithm designing. Lin has a PhD in computer science from National Chiao Tung University, Hsinchu, Taiwan.

Yuan-Cheng Lai is a professor in the Department of Information Management at the National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include high-speed networking, wireless networks, network performance evaluation, and Internet applications. Lai has a PhD in computer and information science from National Chiao Tung University, Hsinchu, Taiwan.

Tai-Ying Liu is an engineer at Avermedia, Taipei, Taiwan. His research interests include embedded hardware-software codesign and content networking. Liu has an MS in computer science in National Chiao Tung University.

Direct questions and comments to Po-Ching Lin at Dept. of Computer Science and Information Engineering, Nat'l Chung Cheng Univ., 168 University Rd., Min-Hsiung, Chiayi, 621, Taiwan; pclin@cs.ccu.edu.tw.