Designing a CAM-based coprocessor for boosting performance of antivirus software

Technion

8th March 2004

Abstract

In this report we investigate the benefits of using a coprocessor coupled with content addrassible memory (CAM) for off-loading of a computation-intensive kernels of antivirus software.

Overview of antivirus technologies is presented, followed by performance analysis of real antivirus software to justify the application of coprocessor. High level architecture of the coprocessor and its interaction with main CPU is described. CAM usage is described and performance analysis is presented. A broader perspective of a using CAM-based coprocessor application for string pattern matching, various string operations, e.g. string comparisons, and regular expression matching is discussed.

Introduction

Modern applications are increasingly demanding high performance hardware. New processor architectures are constantly being developed, exhibiting previously unthinkable performance. However the performance comes at the price of dramatic increase of design complexity, leading to the growth of the time-to-market, increasing the power dissipation, and loosing the robustness.

The major design consideration of a general purpose CPUs is that they should satisfy the performance requirements of all possible applications to be executed on them. Sometimes it is beneficial to add specialized instruction set, e.g. MMX, for particular class of applications. However such additions complicate the design significantly, due to necessity to be integrated with the existing architecture.

In contrast to the general purpose processors, special purpose processors, such as DSPs, have much better chance to achieve better performance in their particular class of applications, being better suited for the appropriate application characteristics. The architecture design takes into account many application specific properties. In addition, many frequently used programming patterns are implemented in hardware, drastically improving the overall performance.

Home PCs are equipped with general purpose processors as they are used for the variety of different classes of applications - graphics, word processing, internet browsing, numeric computations, multimedia, streaming and etc. It is not feasible to optimize the processor architecture for all them. One possibility would be to implement the most frequently used programming patterns in hardware and extend the instruction set of the CPU. However this would significantly complicate the architecture of the CPU, potentially decreasing the performance. Another possibility would be to create special purpose coprocessors, which would offload some computations off the main processor.

The idea of using a special purpose coprocessor for the offloading of complex operations from the main CPU is not new. Modern personal computers are designed to utilize special purpose hardware for servicing peripheral devices, such as hard disk I/O (SCSI/IDE controller), multimedia (sound card), graphical (video controller) and network devices (network card). DMA architecture allows for efficient data transfer between the main memory and the device, without involving CPU, effectively turning the computer with single CPU into multi-processor.

While the use of special purpose CPUs for low level hardware operations is a common practice, coprocessors designed for specific application domains, such as streaming, security and etc, are still not found in modern PCs. It is too expensive to design, for instance, text editing coprocessor, since only word processing applications would benefit from it. In the same time such addition would increasing the overall price even for those who do not use their PCs for word processing. For the hi-end computers, though, the concept of application-specific coprocessors deserved more popularity [8, 10, 2].

In our report we are investigating the benefits of using a coprocessor for somewhat unusual purpose - for the hardware acceleration of the antivirus software. The motivation is clear: antivirus software runs on literally every computer in the world to protect it from being affected by malicious self-replicating programs, commonly referred as "viruses". Virus detection is a complex process with high computation requirements; the more efficient it should be, the more CPU resources it requires. Sometimes it is required to perform the scanning of all the files on the computer to ensure that none of them are “infected” by virus. However another contradicting requirement is that antivirus should not interfere with the normal execution of the running programs. Such software usually runs as a daemon background process, monitoring all the running programs and scanning the memory. Such requirement is usually

not met, since choosing between transparency and efficiency, the latter is always chosen in the case of antivirus - it is better to work slowly than to loose everything because of virus.

Contemporary antivirus software usually traces all accesses to file system and the most recent even to network, effectively placing the viral analysis in the **critical path** of any I/O operation.

It is necessary to analyze the existing techniques of virus detection in order to coin the compute-intensive operations to be further mapped for execution on a coprocessor [12]. Since much of the information about the existing commercial antivirus products is kept in secret, our analysis is based upon several open-source antivirus projects and publicly available research materials.

We conclude that sequence matching is the most common way of detecting malicious programs, and it is used in all antivirus products without exception. Our conclusion is confirmed by profiling the available antivirus software. The technique is very simple. In essence, a virus program is characterized by a unique sequence of characters, extracted from its binary representation. The file containing such sequence is considered as "infected". Thus an antivirus program scans all the suspicious files, attempting to match any of the signatures from the signature database.

Content Addressable Memory (CAM) suits perfectly for implementing fast searches required by antivirus software. It allows for the fastest search of the whole memory content at price of single memory access. CAMs are widely used for the longest prefix matching in network routers and switches[17, 18]. However we did not find any references of using CAMs in general purpose PCs.

While our research originated in antivirus software, signature matching is used in many other different classes of applications. For example, it is extensively used in computational biology (BLAST) for genome research. Hardware implementation of the algorithm would boost the performance of such applications in orders of magnitude.

The paper is structured as follows. In the next section we describe the target application domain, introducing the terminology and the algorithms. Then the profiling of the real antivirus product is shown. This allows us to estimate the maximum possible speed-up which would be achieved by mapping the compute-intensive operations to be executed by coprocessor. We then analyze the major performance factors in the software based antivirus, suggesting various optimization alternatives to be implemented in hardware. We analyze the possible solutions using trie and CAM as a building block. We then describe the high level architecture for the coprocessor as derived from the application domain requirements, and its interaction with the main CPU. We conclude with the discussion on other possible applications of such coprocessor.

Application domain overview

Computer viruses

Computer Virus - the words alone provoke images of vanishing data, crashing PC's and financial ruin. While maybe not as dramatic as this, viruses are a daily nuisance for both home and corporate computer users. With each new virus, a dozen antivirus vendors swing into action to find a cure. If the protection is installed in time, the virus spread can be stopped before the one caused any damage.

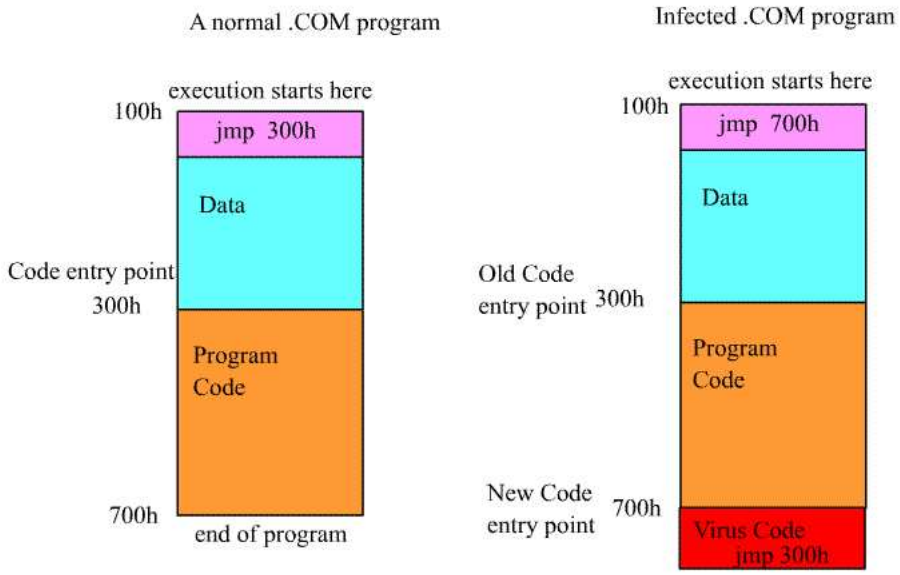
The term "computer virus" is defined as "a sequence of instructions that copies itself into other programs in such a way that executing the program also executes that sequence of instructions"[14]. However, the name is mistakenly applied to various types of malicious programs, usually classified as viruses, worms and Trojan horse. In fact, that classification is based on the way the malicious code spreads. Worms spread via networks, using bugs in existing software. "Trojan horses" show themselves as some useful program. Once invoked, along with the stated function, they perform another,

unstated, and usually undesirable one, infecting the executing computer and using it to spread further. Modern viruses often employ several aforementioned techniques all together, circumventing sophisticated antivirus defense. In the following we will refer to all these malicious executables as computer viruses[9].

A computer virus infects other entities during its *infection phase*, and then performs some additional (possibly null) actions during its *execution phase[7].* The infection phase starts with the virus executable attempting to reach another computer. Before the Internet age removable media was used for this purpose. Viruses used to replace the original boot sector of the floppy disk with the infected code. After using the infected floppy in the other PC, the user could mistakenly forget to remove it before the computer. During the boot process with the floppy inside, BIOS code by default would attempted to bootstrap from the infected floppy first. Then instead of the original bootstrap code, the virus code was activated, allowing for the virus replication to the new boot sector of the hard drive on the target host. Alternatively, viruses could infect executables on the floppy, so that further invocation of these executables in the other computer caused the virus to be invoked first, effectively replicating its code into the new computer. Replication usually involved copying of the viral code into the computer memory. Once there, the virus went to background, gradually infecting all the executables installed on the hard disk. In addition, it ensured its presence in the main memory for further infection of other floppy disks.

Internet has made the task of virus spread much easier. Modern worms and Trojans use the security vulnerabilities of the operating systems to propagate into the target system. Bugs in the network layers of the applications are used by virus writers to inject the virus code and start its execution, gaining remote control over the victim. Less sophisticated techniques are used by trojans, being spread as an e-mail attachments. A naive user, tempted by the curiosity to invoke a new application from an old friend, opens the Pandora

Figure 1: Virus infection of the executable



box, assisting in further spread of the virus and often rendering the computer unusable.

Executable infection techniques

The common practice of infection is to infect regular executables on the computer to trigger the execution phase of the viral code. In this case, executable file is edited during the infection phase, its code is changed to execute the virus code upon invocation, instead of the original code. This kind of infection was applied mostly in DOS, with executables in .COM format. Such files were relatively easy to infect, since the executable entry point was always at the beginning of the file. Viruses used to attach their code to the end of the file, changing the first instruction of the infected executable to jump to the virus start, jumping back after the virus execution. If well written, the virus could avoid detection by the user and go about its infecting ways.

Infection of the executables in .EXE format for MS Windows or .ELF format for Linux is much more complicated. It can be done by prepending, or appending code to a file, by splitting up the virus and hiding it in holes within the unused segments of the executable.

Not only executable are targeted by the modern viruses. So called macro viruses, implemented as a script programs and embedded in the data files, utilize the security vulnerabilities of the hosting applications, when the infected data files are being opened.

Virus detection techniques

There are numerous antivirus software packages, which claim to provide almost complete protection against computer virus. Unfortunately, none of them is perfect. It is theoretically proved [1, 14], that there is no algorithm that can perfectly detect all possible viruses. However, several techniques were developed, [6] and their combination is implemented in most antivirus software products to provide the best protection possible.

1. Infection Prevention - halt the virus replication and prevent the initial infection from occurring. The antivirus software monitors the critical system components and resources, such as boot sector of the hard disk, critical OS files, network access and etc. Unauthorized access to these components causes the system to be halted, preventing the malicious code from propagation. While being efficient in many cases, this method allows to withstand the viruses with the known propagation characteristics.

2. Infection Detection - detect infection soon after it has occurred and mark specific components of system segments that have become infected. Such systems assume that executable files are never changed (read only) after being installed. Infection by a virus would necessarily result in the file content to be

changed, as virus requires a host executable to place its code and further be invoked. To verify the integrity of the files, such antivirus applications generate digital digests of these files and store them in the database, preferably backed up on the read-only media, and usually cryptologically secured. When during the periodic scans of the file system the difference is detected between the content of the current file and its original one, it signifies the presence of a virus. It should be emphasized, that such systems usually do not identify the exact virus strain, and can not recover the infected files, unless the original uninfected copy was backed up in advance. Another problem with such techniques is numerous false positive alerts, as sometimes the assumption of the unchanging executable is not true. Examples of applications of the infection detection techniques can be found in [5]

3. Infection Identification - identify specific viral strains on systems that are already infected and remove the virus. This kind of protection is the most popular one and is exploited in all the antivirus tools without exception. It is described in more details below.

The combination of these three essential techniques usually provides a defense against most malicious executables. However it is worth noticing that the first two of them do not allow to cure the infected system, and thus gained less popularity than the infection identification. Lately infection prevention is being promoted, such as monitoring of system files and dlls in Windows, various firewall products and etc. However contemporary operating systems come pre-configured with extremely insecure configuration, rendering the computer vulnerability to the virus attacks. Users usually recall that something should be done only when their computer is infected, and that is the point when the infection identification techniques are used.

Infection identification is based on the concept of the virus *fingerprint* [3], which allows to uniquely identify each virus. The fingerprint of a virus executable is typically a short series of machine code bytes (usually not larger than a few hundreds of bytes) - *aka signatures*, that the virus code contains. Specifically, a signature-based detector requires the virus' code length and the location of its 'contagious' segment, which is essential to its replication and transfer among storage media, computer memory and networks. These signatures are extracted by the antivirus researchers from the original virus code and then are packed into the database of the known viruses. For instance, Norton Antivirus 2003 contains more than 65000 such signatures of the known viruses [13], and the database is being extended on the daily basis. The detection process is performed by scanning the suspicious files, mostly executables, attempting to textually match the known virus signatures in the binary file. If the process results in successful match, it can be claimed with almost full confidence that the virus is present, though false positives occur. The next steps are to attempt to remove the viral code from the file, and if impossible, to restrict the further access to it. However if no viral signature was found, it only ensures that the known viruses are not present, but it might well be that the new one is.

In order to circumvent the signature matching process, virus writers employ various sophistication of the viral code [7]. They wrote viruses which pad their code with random code, used several ways to code the same functionality (for instance, `xor AX,AX` or `mov AX,#0`), and etc. There are also virus authoring toolkits, allowing for easy virus creation from the common code base. Such viruses still can be detected using signature matching with wildecards[6].

The most difficult to detect kind of viruses are polymorphic and metamorphic ones, which are able to obscure their entry points, have obfuscated code structures (that can shrink or expand themselves through their metamorphic engines). This is where heuristic

scanners are used. They also utilize the signature matching algorithms, but the signatures are the behavioral ones - clearing register, opening file, self modifying code and etc. Antivirus attempts to mimic the potential behavior of the executable and suspicious sequence of such behavioral signatures may raise the probability of virus.

Polymorphic viruses are often encrypted. While it is impossible to match the viral code itself, the first versions of such viruses were detected through matching their decryption engine, which had to be valid executable. To circumvent the decryption engine detection, virus writers obfuscated its code, making it untraceable for any signature matcher. The most sophisticated viruses are the metamorphic viruses, which encrypt not only their body, but the encryptor itself. For instance, W32.Simili virus uses a polymorphic decryptor, which changes size and location in its infections. It creates a metamorphic virus body, disassembling the virus to an intermediate form, compresses it by removing redundant and unused code, then mutating it by reordering functions and breaking up code. It then expands the intermediate form by adding random redundant code and unused instructions. It finally reassembles the intermediate code to a native code which is used to infect other hosts. To add insult to injury, the payload, a message box, is only displayed on certain days, depending on the virus variant. For such complicated cases, antivirus applications creates a simplified virtual machine and runs the code inside the controlled environment for the virus to decrypt itself. Once decrypted, a signature matcher is executed on the plain-text virus.

Antivirus software

Antivirus applications usually run in the background and intercept all the system calls, which attempt to open file. In particular, executing a file requires to open and read it; loading shared library

cannot skip the stage of reading it from the disk either. Modern antiviruses add network and mail support, filtering the incoming and outgoing network traffic as well.

This is the moment when antivirus software applies all its weapons and scans the data being read to detect the presence of virus. Such on-the-fly scan should be very efficient and fast, since long delays in starting a program would interfere with the user's normal activity. This restriction sometimes prevents the antivirus to come to the final conclusion about the executable within such short time, and it continues monitoring its actions even when the application has already been started. In such case its overhead should be minimized as well, restricting its functionality.

While the scan-on-execute technique is very popular in PC antivirus software, it is now a common practice to install antivirus scanners for analysis of all incoming and outgoing corporate internet traffic. With todays high speed networks, the huge amount of data should be processed with the real time restrictions. Of course, optimizing the speed of such antivirus gateways is one of the primary goals of the antivirus vendors, as the gateway performance directly influences the effective bandwidth of the outgoing and the incoming network traffic in the organization.

Signature matching - the bottleneck

From the review of the antivirus technologies it becomes clear how complicated and CPU intensive their implementation should be. Obviously, the common denominator among all of them is a signature matching. All the virus identification algorithms employ this technique at one stage or another during the execution [9].

However the question is whether the signature matching is the true bottleneck of an antivirus program execution, and if yes, to what extent. Does it really take most of the execution time? Though the everyday experience with the antivirus software proves it as

CPU-greedy, slowing down the overall performance and rendering the computer absolutely unresponsive when performing the scan, it should be clearly shown that the signature matching is the major factor for the antivirus CPU load. There are other potential time-consuming tasks, with the I/O being one of the most significant. Every piece of data is read from disk or from network, and it is not clear whether the improvement of the CPU intensive part of the antivirus process would improve the overall performance.

In order to estimate the portion of the signature matching in the overall antivirus execution, software profiling is required. However the fine-grained profiling of the leading antivirus products is impossible, since the vendors do not publish the sources of their products.

There are several open-source antivirus products, such as Clam Antivirus - ClamAV *[15]* and OpenAntiVirus *[16],* which make possible to analyze and change the source code.

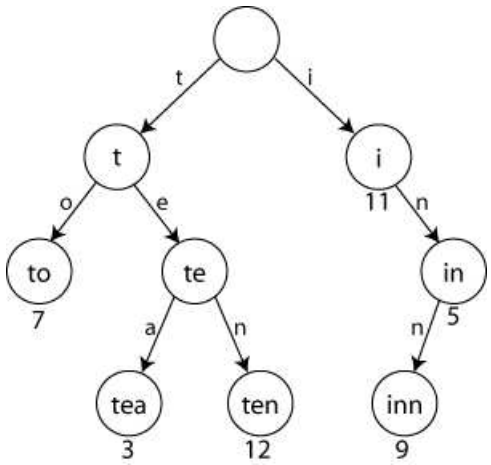
It is important to understand the algorithms and the performance bottlenecks of the existing antivirus software in order to be able to optimize them for implementation in hardware. The following subsections present the algorithm used for the pattern matching and analyze the performance of ClamAV antivirus.

ClamAV algorithm description

ClamAV employs very efficient Aho-Corasick pattern matching algorithm, based on finite state pattern matching automation represented by trie[19].

Trie is an ordered tree data structure, that is used to store an associative array where the keys are strings comprised of symbols from the given alphabet $|\Sigma|$. Any descendants of one node have common prefix of a string associated with that node. Leaves represent the end of string and may contain the information associated with that string. Search operation is performed by traversing the

Figure 2: Trie with the six words “to, tea, ten, inn,in,i”.



tree, following the pointers in the tree nodes. If there is no path for the given input string, that string is not present in the data structure. In the example on Figure 2 there are six strings in the trie - “to, i, in, ten, tea, inn”.

A naive implementation of trie requires allocation in every node of a pointer array, containing all possible symbols from $|\Sigma|$. Such implementation would allow to perform the search without comparison operations at all, as every next node to visit would be found using the next input symbol as index in the node array. However such implementation has extremely high memory complexity, exponential in the lenght of strings in trie. Thus for binary database with the alphabet of size 256, the one used by antivirus software, it would require 256^k, where k is the average number of symbols in signature. Typical size of signatures is from 40 to 100 symbols [15], and it is obvious that such implementation is not feasible.

Another solution would be to create variable length array and search it for the next symbol during the traversal. This however would greatly decrease the performance, requiring at best O(log(n))

comparisons at each level of the trie.

The approach taken by the ClamAV writers makes use of the fact that the number of signatures with the same prefix decreases exponentially with the increase of a prefix length. Thus they use very shallow trie with only 2 levels. These levels are implemented using the first approach. All the tails of the signatures with the common 2-symbols long prefixes are stored in the leaves of the trie and are searched sequentially.

ClamAV performance

In order to understand the distribution of the CPU load during the antivirus execution, function-level profiling was performed with GNU/Linux gprof. The profiling results revealed that the application indeed spends most of the CPU time (99%) in two signature matching functions. The first one (`cl_scanbuff` - 38% of CPU time) traverses the trie, while the second one (`cli_findpos` - 61%) is invoked to match the signature tail, found by the first function, with the input string in order to find whether there is a full match with the known virus signature.

These measurements, however, do not show the I/O portion of the execution. In order to estimate one, the signature matching functions, mentioned in the previous paragraph, were replaced by empty stubs, containing only `return` instruction.

Both versions were invoked on 250MB size dataset.

The system utilization during the invocations was measured by the '`time`' command. All values are in seconds. The following results were obtained:

Invocation with matching functions:

```
9.6user 0.63system 0:26.16elapsed
```

Invocation with matching functions disabled:

```
0.15user 0.31system 0:15.18elapsed
```

The results show that the program spent ~40% solely on signature matching. The rest of the time, disk I/O was performed.

Although the results demonstrated that large part of the antivirus run time is due to the disk I/O, the signature matching efficiency influences significantly the overall performance. In addition, the profiling clearly shown that signature matching is the major factor of CPU utilization. And finally, I/O technologies are being developed to deliver much higher I/O performance, e.g. Infiniband, applying which would make the CPU a real bottleneck.

In order to prove that, additional test was invoked on the same dataset, but this time the data resided on RAM disk and not on the hard disk. The following results were obtained:

Invocation with matching functions:

```
9.6user 0.35system 0:10.02elapsed
```

Invocation with matching functions disabled:

```
0.15user 0.32system 0:0.49elapsed
```

The results show that with the high speed I/O, the factor between the "fast" version and the regular version is almost **two orders of magnitude**.

Although these values pose the upper bound on the possible performance improvements, this is the goal of the hardware-assisted search.

Approaches to hardware-assisted string matching

It is clear that in order to achieve the maximum performance boost the string matching should be optimized for hardware implementation. There is a lot of research activity in that field due to its applicability in network devices, such as routers and switches [20, 21, 22, 23]. However, there are several significant differences between the existing solutions and the one required for application in

antivirus software:

- Hardware acceleration algorithms are usually used for matching IP addresses, no more than 32 bits in IP V4 or 64 bits in IP V6. This restriction is utilized by the network packet matching algorithms to achieve high performance by using large memory. In our case the length of the string to be matched can be from 40 bytes to 100 bytes, which is one or two orders of magnitude larger than the former. Matching only the first two or three symbols is not enough to make the string matching sufficiently efficient.
- Routing tables managed by the routers are frequently updated, so that the network algorithms should balance between the lookup and update time. In contrast to the former, antivirus string matching is done mostly using read only signature database. Thus, write performance optimizations are irrelevant.

We considered several alternatives for string matching algorithm to be used:

1. Implementation of hash table
2. Implementation of the trie-based algorithm.
3. CAM-based approach

The first approach considers using some hash function to create a hash table of all signatures and to determine the successful match of the input strings based on the existence of the corresponding entry in the signature hash table. However since hash function is by definition is not injective, and since the active input domain size is extremely large, finding the matching entry in the hash table would not be sufficient to conclude that the input string is indeed a virus signature. There will be required additional comparison operations in order to ensure the correct match. Additional problem

is that the computation of the hash function is required for every input sequence, incurring significant overhead. In contrast to that, trie-based algorithms can filter most of the input strings only by comparison of two or three first symbols.

Hardware implementation of the trie-based algorithm

As we mentioned above, full trie has exponential memory requirements. The table below represents the memory requirements of the trie size from 2 till 5 levels:

#of levels	Size (B)
2	2^{16}
3	2^{24}
4	2^{32}
5	2^{40}

It is clear that the maximum full hash size possible for implementation is of up to three levels. If the match is found on these levels, then the rest of the input is matched with all the strings sharing the common prefix from the matched levels.

Trie performance estimation

In order to estimate the average time it takes to check one signature, the following formulae should be evaluated:

$\sum P(access_i) * T_{averageaccess}(i)$

where i is the level number, P is the probability,

$T_{averageaccess} = T_{no\,cache\,misses} + P(miss) * MissPenalty * NumOfMemAccesses$

The skeleton code for trie traversal is presented on Figure 3

Assuming that all the data resides in the L1 cache, it takes about 4 cycles to complete the traversal of one level of a trie. However such assumption is unrealistic due to poor spatial and temporal locality of memory access during trie traversal. Every next level requires access to another pointer array at the different location. Sequential

Figure 3: Program skeleton for trie traversal

```
r2=0;
trie_trace:
   r0=input_string[r2];
   r1=trie_level[r0];
   if (r1==null) goto failed;
   if (i==TRIE_MAX_LEVEL-1) goto continue_match;
   r2++;
   goto trie_trace;
```

accesses to the same array have purely random nature in terms of time of access.

In order to better understand the cache performance of trie traversal we ran the antivirus in cache simulator Cachegrind [26]. Cache configuration of the real Intel M-1.5 GHz CPU was used - 32K D1, 32K I1, 8-way, 64 B per line. The following results were obtained:

max trie levels	D1 cache miss rate
2	12.2%
3	10%

All the rest of the tests were performed with the trie depth equals to 3.

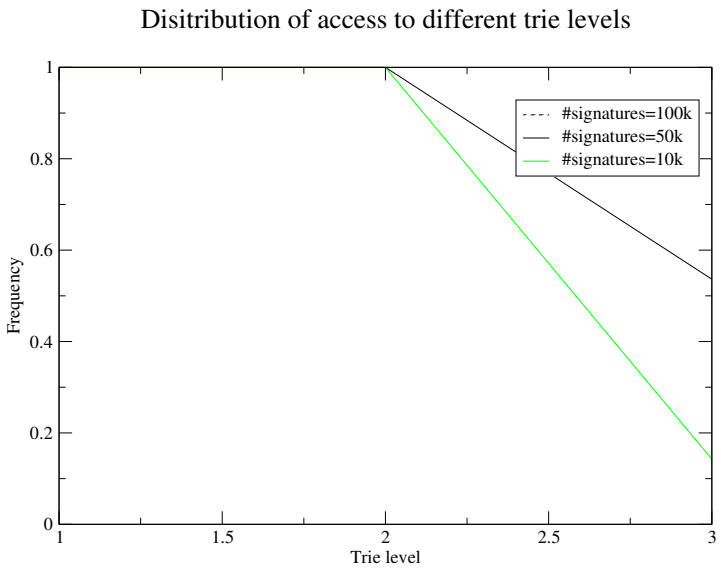
To count the average number of cycles required for completion of traversal it is necessary to obtain the statistics regarding the distribution of the search length, which also depends on the total number of viral signatures in the trie. For this purpose the code was instrumented to record this information.

The results are presented on the graph on the Figure 4

The results show that for any number of signatures the probability of access to the second level is 100%, that is the loop will always be executed at least twice. Assuming 100K signatures in the trie, as much as 72% of the accesses will require the third loop. This leads to the average of 1.9 traversals per input string.

Another algorithm is used to match strings which resulted in a full trie traversal. The trie leaves contain the list of the signatures with the same prefix, and the string is matched with each one se-

Figure 4: Frequency of accesses to the first three trie levels



quentially. The code skeleton on Figure 5 can be used.

There are two memory accesses and one comparison involved.

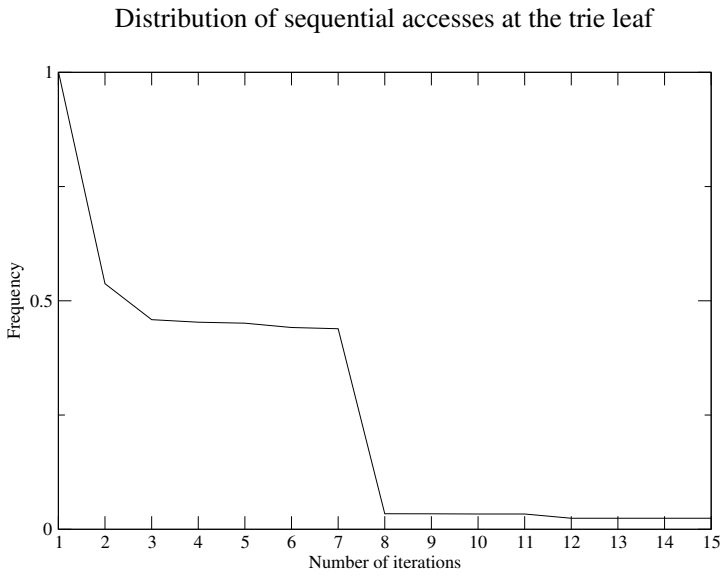
The graph on Figure 6 shows the distribution of the number of signatures at the trie leaf, normalized by the number of accesses to the leaf.

It follows from the graph that about 40% of all accesses require up to 7 iterations at the trie leaf to check all possible signatures with the same prefix. The mean value is 4.093 iterations per leaf. Such high rate is observed when the real antivirus signatures are

Figure 5: Program skeleton for string comparison

```
r0=trie_size;
start:
   r1=input[r0];
   r2=signature[r0];
   if (r1!=r2) goto failed;
   r0++;
   goto start
```

Figure 6: Distribution of how many times more than one string is compared, after all trie levels are matched



used, as opposed to the randomly generated signatures. The reason is in similarity in the prefixes of many viruses.

And finally the graph on Figure 7 depicts the distribution of the accesses which traversed the trie and arrived matched more than 3 symbols of the signature, that is, employing at least one comparison operation. The graph is normalized by the total number of accesses to the trie, i.e. is in the same units as the first one.

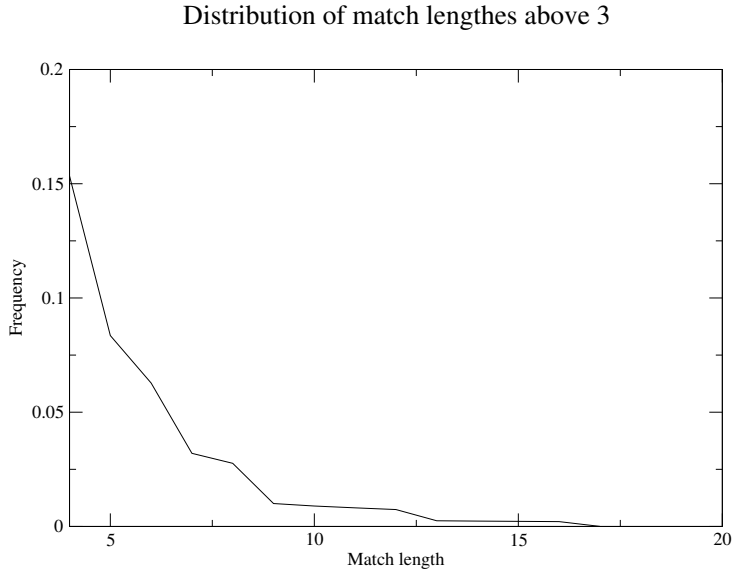
The results show that there are on average 2.8 comparisons per access. Together with the previous value it gives 2.8*4.093=13.8 times the comparison loop should be performed, once the trie traversal is complete.

The final computation is as follows:

Cache miss

Each execution of the first loop requires 2 memory accesses. Each execution of the second loop requires 2 memory accesses as

Figure 7: Frequency of search lengths where more than 3 symbols match



well. Thus, there are 13.8*2+1.9*2=31.4 memory accesses. Given miss rate of 10%, there are on average 3.1 **cache misses** per each input string.

Cache hit

Assuming all memory accesses to be cache hits (single cycle), and assuming that each compare and branch requires only one cylce, both loops would take about 4 cycles, two for each memory access and two for both compare and branch operations. It would result in total of 31.4 cycles for memory and the same number for compare and branch. Given cache hit of 90%, there will be on average 60 cycles per each input string.

Total

Matching strings using trie takes on average **3.1*(cache miss penalty)+60 cycles** to execute. Assuming miss penalty as an access time to L2 cache, i.e. about 10 cycles, the result is **91 cycles**

CAM-based string matching

Content Addressable Memory (CAM) allows for detecting string match within a single memory access. This unique feature of CAMs is widely used in network routers and network switches. Read access time is the same as the one of regular memories. Access rates as large as 800MHz are reported.[24, 25]

We propose using CAMs as a first stage for lookup of the matching signature, effectively substituting trie. All the signatures, more exactly, their first N symbols are loaded to the CAM at the boot time. Signatures with the same prefix are saved at the same address. CAM's output is used to index a standard memory array, serving as a repository for the signatures tails, exactly as in the case of trie-based implementation. Once CAM match is detected the comparison is continued sequentially. It is important to note, though, that the number of the signatures sharing the same prefix decreases exponentially with the length of that prefix. In order to improve the performance of the sequential comparison, small cache is sufficient for holding the string being checked and for prefetching the array pointed by the CAM, which contains the signature tail left to be matched. This cache reduces the sequential comparison penalty to only single cache miss.

There is a clear tradeoff between CAM width and performance. The wider the CAM, the less is the chance to execute sequential string comparison. However it might be beneficial to make CAM as small as possible in order to shorten the access times by making on-chip implementation possible. Unfortunately such implementation seems to be unlikely, at least today, assuming that there is a need to save about 100K of signatures.

From the analysis of the trie performance above it follows that there are on average 6 symbols long comparisons required per one input string. That would be the minimum CAM width, resulting in 48 bit wide CAM, to overcome the high performance penalty of the

trie-based design, requiring only single memory access for 90% of the accesses. For other 10% it follows that on average 2 comparisons would have to be made, resulting in one additional memory access and several comparison cycles.

The general result:

Average CAM access time: 0.9*(1 CAM access)+0.1*(1 cache miss penalty + 2*4 cycles)

Cache miss penalty in this case should be assumed as a time to access main memory, since there is no L2 cache in our configuration. Assuming 100 cycles cache miss penalty and 10 cycles CAM access, the result is **~15.8** cycles per access, about **6 times faster** than in the case of trie-based implementation.

Increasing the CAM width to 10 symbols (80 bits) would allow to increase that number to 12 cycles per access. Total size of CAM in that case is about 1MB, which is the size of today's L2 caches, still allowing for just 10 cycles access penalty.

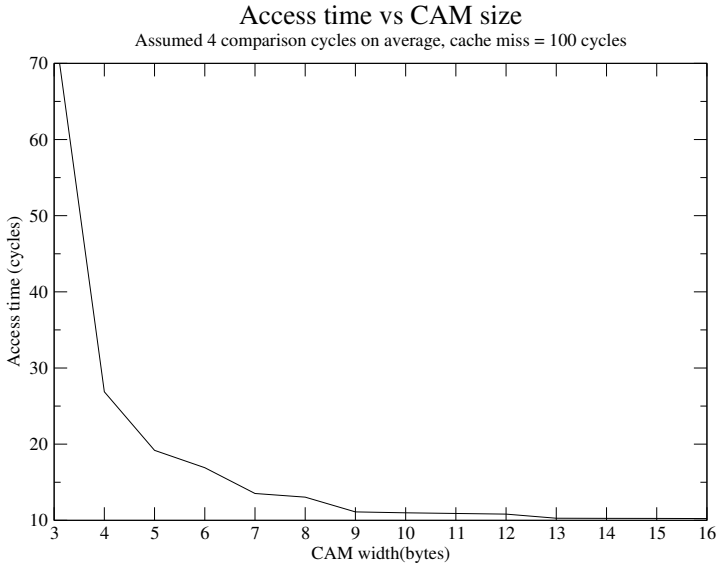
The graph on Figure 8 presents the estimated access time as a function of CAM width:

System architecture with the antivirus coprocessor

In the following, we propose the possible system design with the integration of the antivirus coprocessor.

The coprocessor offloads all the signature matching code from the main processor. However, it is likely that the coprocessor will be used not only for the simple signature matching, as we described in the previous sections, but for other more complicated tasks, like sequence matching and string search, and etc. It may implement various computational primitives for string search, string comparison and etc. For the purpose of the antivirus software, it should implement at least one operation:

Figure 8: Access time as a function of CAM size




```
find_position(start_addr, end_addr,
        result_register, end_offset_register)
```

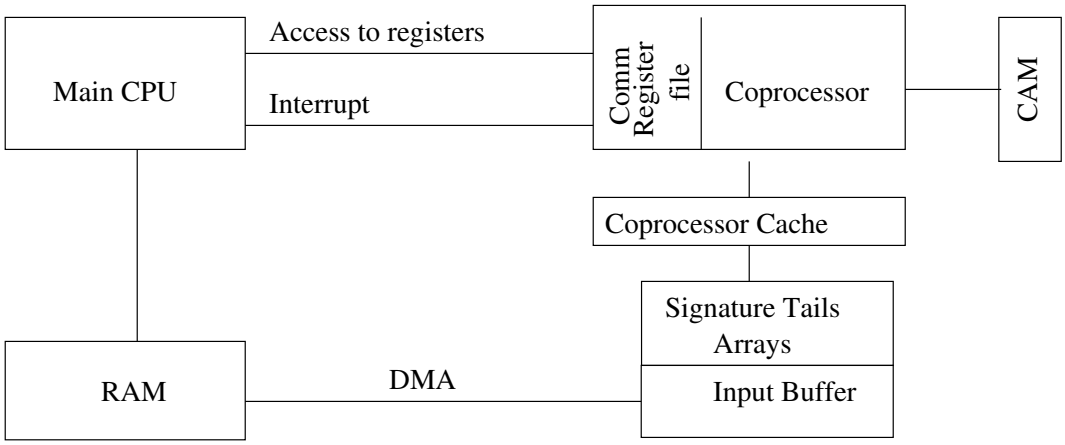
This operation matches any string from the database with the continuous memory area between `start_addr` and `end_addr`. These addresses point to the main-CPU memory. The result is the offset from the starting address, representing the first match found. The `end_offset_register` points to the end of the matched signature allowing to exactly identify the found sequence.

The coprocessor should be capable of asynchronous operations. It should support the pipelined mode of operation, so that while searching for the first match, the next addresses can be provided to perform the next search.

Our system architectural model is shown on Figure 9, and is comprised of the main CPU, copocessor and the RAM.

Coprocessor has its private RAM, divided into two parts. The first one contains the string block to be checked, transferred via

Figure 9: High level system architecture



DMA channel between the main and coprocessor memories. The second part is initialized during the boot with the signature tails arrays. The coprocessor cache is big enough to hold the minimum block of input data.

Coprocessor has several registers to receive parameters from the main CPU. The registers are grouped in register files, each one containing two registers. These registers are used for the input by the main CPU to pass the memory range, and for the output by the coprocessor to pass the resulting offset and the pointer to the matched string. Additional register is used as a flag register to point to the active register file. This is useful for pipelining the string matching requests, so that the next address range is set by the time the coprocessor completes the current run. In addition, the interrupt line is set in both directions to support asynchronous operation: an interrupt is issued by the main CPU to the coprocessor, to indicate that the data is ready for the processing, and by the coprocessor to the CPU, to indicate the completion of the operation.

Summary and future work

In this report we analyzed the applicability of coprocessors in antivirus applications, which protect the computer from the computer virus attacks. This analysis included the overview of the modern antivirus tools and algorithms in order to better understand their computational requirements and detect the compute intensive kernels. The analysis revealed that the most frequently operation used during the virus scan is a signature matching, i.e. the search for the given string in the large database. Profiling of the real antivirus application proved that the signature matching is indeed very compute intensive, justifying its implementation in hardware. Based on this evaluation we proposed the high level system architecture with the integrated antivirus coprocessor based on CAM.

There are several possible directions to develop this interesting subject. There are a lot of computer applications which require string search. CAM -based coprocessor can be used to boost their performance

- **Personal Firewall.** More and more people are realizing the need to tighten access to their data, and install security toolkits with personal firewalls. These firewalls, or so-called port filters, trace all the network traffic, allowing or rejecting packets to get through according to their destination and source addresses. Application of fast string matching processor would significantly boost their performance.

- **Computational biology**. Adding write operation to the coprocessor would allow for using it in sequence matching applications in order to boost their performance.

- **Data mining.** Data mining applications in the area of plain text mining many times require searching over and over the same data trying to find specific string patters. For instance,

the operation of detecting the frequency of the word entries in the text is very common in the process of data clustering. Searching for specific words can be improved if the coprocessor allows to first initialize it with the file contents and then attempt to search for the required items.

- **Text search.** As in the previous case the text search speed can be increased if the coprocessor is used.

In general, it seems that the subject of using CAMs in personal computer is not sufficiently explored, despite of the high potential found in it.

Bibliography

[1] “An undetectable computer virus”, David M. Chess, Steve R. White, IBM T. J. Watson Research Center

[2] “S/390 CMOS Cryptographic Coprocessor Architecture: Overview and design considerations”, P.C. Yeh, R.M. Smith, Sr., Technical Report, IBM

[3] “Data Mining Methods for Detection of New Malicious Executables”,Matthew G. Schultz, Eleazar Eskin, Erez Zadok, Salvatore J. Stolfo

[4] “Dynamic detection and classification of computer viruses using general behavior patterns”, Baudouin Le Charlier, Morton Swimmer, Abdelaziz Mounji

[5] “Anti-Torjan and Trojan Detection with In-Kernel Digital Signature Testing of Executables”, Michael A Williams

[6] “Generic Virus Detection”, William Hsu, http://www.mactech.com

[7] “Antivirus Research and Detection Techniques”, Jay Munro, http://www.extremetech.com

[8] “The IBM 4758 Secure Cryptographic Coprocessor, Hardware Architecture and Physical Security”, Steve Weingart, IBM T.J. Watson Research Center

[9] “An Overview of Computer Viruses in a Research Environment”, Matt Bishop

[10] “An Evaluation Architecture for a Network Coprocessor”, Jason Hatashita, James Harris, Hugh Smith and Phillip L.Nico. Proceedings of IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, November, 2002

[11] “Coprocessor Co-design for Programmable Architectures”, Parbhat Mishra, Frederic Rousseau, Nikil Dutt, Alex Nicolau, Technical Report #01-13, Dept of Information and Computer Science, University of California

[12] “Reconfigurable Architecture for Multi-Standard Audio Codecs”, Vishal Choudhary, Antoine van Wel, Marco Bekooij, Jos Huisken, Philips Research Labs

[13] http://www.semantecs.com

[14] “Computer Viruses: Theory and Experiments”, F.Cohen, Seventh DOD/NBS Computer Security Conference Proceedings, Sep 1984

[15] Clam AntiVirus, http://clamav.elektrapro.com

[16] OpenAntiVirus, http://www.openantivirus.org

[17] “Content Addressable and Associative memory: Alternatives to the Ubiquitous RAM”, L Chisvin, R.James Duckworth, IEEE computer 1989

[18] “Using Content-Addressable memory for network Applications”, Sherri Azgomi http://www.commsdesign.com/main/1999/11/9911feat3.htm

[19] http://en.wikipedia.org

[20] “Efficient Hardware Architecture For Fast IP Address Lookup”,D. Pao et. al.

[21] “Routing Lookups in Hardware at Memory Access Speeds”, P. Gupta, S. LIn, N. McKeown, IEEE Network Magazine, Jan 1992

[22] “Putting Routing Tables in Silicon”, Tong Bi Pei and Charles Zukowski

[23] “Novel IP address lookup Algorithm for Inexpensive hardware implementation”, K. Seppanen

[24] ”Reducing routing table size using ternary-CAM”, Huan Liu

[25] “Content-Addressable Memories Speed Up Network Traffic And More”, Dave Bursky, ED Online ID #1087 January 24, 2000, http://www.elecdesign.com

[26] Valgrind - Linux profiling and monitoring environment, http://valgrind.kde.org/