Memory Integrity Verification Speedup Using Separated L2 Hash Cache

Ying Xiong
Dept. of Electrical and Computer Engineering
Oklahoma State University
Stillwater, U. S. A.
ying.xiong@okstate.edu

Sohum Sohoni
Dept. of Electrical and Computer Engineering
Oklahoma State University
Stillwater, U. S. A.
sohum.sohoni@okstate.edu

***Abstract*—We propose a novel architectural technique in this paper in order to decrease the overhead of memory integrity verification using cached hash trees. We suggest using a separate L2 hash cache to store internal nodes of a hash tree since those nodes show different locality of access than regular application data. Our simulations results indicate that our simple but novel scheme can reduce the overhead by 8.6% for a heavily loaded APACHE server.**

Keywords—Secure processor;Memory integrity verification; Hash tree; Separated L2 hash cache; Performance

I. INRODUCTION AND MOTIVATION

Information security has become one of the most important responsibilities of modern computing systems because of expanding commercial privacy protection requirements. Wider acceptance of e-commerce, spreading copyright protection demands, and the threat of malware are driving these requirements further. These applications and the systems that execute them are facing challenges from rapidly growing confidentiality threats [1]. These threats can be mitigated by software solutions, such as antivirus software, firewall, software encryption/decryption, and spyware blockers. However, since the threats can spread widely through operating system security loopholes, software-based security solutions are not completely safe. Equally important is the performance slowdown that results from software solutions, especially when used for encryption/decryption. Hardware security measures are critically needed to provide more in-depth and stronger defense solution to protect critical information. Secure processors are a promising hardware-based measure [2]. A secure processor only trusts the computation results inside its boundary, for example, the processor core, and provides us two major security measures: memory data encryption/decryption and memory integrity verification.

The data transferred out to main memory will be encrypted by a cryptography algorithm using a secret key. The processor uses the corresponding key to decipher it. Even with data encryption, a spoofing attack may replace values in memory with spurious cipher text. If the processor cannot detect the attack and operates on the spoofing data, the behavior of the secure processor may be altered and confidential information may be revealed. An attacker may also use the cipher text in the main memory and re-input them into the processor to induce it to reveal critical information, even if he does not know the exact plaintext of the encrypted data. This is known as a replay attack [1]. In order to prevent spoofing and replay attack, the secure processor needs to guarantee that nobody changes the data without authorization. This guarantee is provided through memory integrity verification.

Memory integrity verification will cause performance loss due to the computation burden of hash. For example, Gassend et al. [3], report a performance overhead of less than 25% as the result of their improvement research work compared to 10X overhead of a naïve implementation of memory integrity verification. Though it is significant progress, 25% performance loss is still unacceptable to most users and it's focused on SPEC CPU benchmarks, which are not quite suitable to represent on-line transactions.

The goal of our work is to keep on improving the performance of the memory integrity verification. We do not focus on reducing the encryption and decryption delay in this paper. There is prior research dedicated to mitigate the impact of encryption and decryption delay [4].

The organization of the paper is as follows: Section II describes related work in the field of memory integrity verification; Section III describes cached hash tree in detail; Section IV introduces our contribution: separated level-2 hash cache; Section V explains the simulation configuration, including the use of SPECWeb 2005 as a more practical and relevant benchmark for secure architectures; Section VI presents and analyzes the simulation results; and section VII concludes our work, discusses its limitations, and proposes our future work.

II. RELATED WORK

Several memory integrity verification techniques have been proposed, such as Message Authentication Code (MAC) [5], cached hash tree [3], Message Authentication Code Tree (M-TREE) [6], log hash technique [7] and Bonsai Merkle Tree (BMT) [8].

In [5], Lie et al. use MAC to guarantee no data alteration is made without authorization. A MAC is a keyed, one-way hash of a data block. When there is data needed to be transferred outside processor chip, s*tore_secure* instruction

generates a MAC of the encrypted data and saves it along with the data in external memory. When that data is loaded back using *load_secure* instruction, the data's new MAC is computed and checked with that stored in the main memory before. If the two MACs match, the data is not altered. Otherwise the execution is halted. However, the use of MAC cannot eliminate replay attack. If an adversary captures a pair of data chunk and MAC, he can re-input them to the machine and the machine will not be able to detect any error. In this case, the adversary may trick the machine to leak some critical information.

In [7], Suh et al. propose log hash technique to reduce the computation burden of memory integrity verification. Compared to verifying integrity for each memory operation, log hash postpones the verification to the end of a series of memory operations. Note that the verification is still done on all these memory operations. The advantage of log hash is to reduce the frequency of integrity verification. It should be mentioned that all the memory belonging to a process but not in cache needs to be loaded into the chip to finish the integrity verification. The machine may "freeze" doing all the loading and hashing if the process owns a lot of memory.

Gassend et al. suggest caching the hash tree (Merkle Tree, introduced in [9], and shown in Figure 1) in the level-2 cache to decrease the overhead of loading an internal hash tree node in [3]. Since level-2 cache is inside the chip, any internal hash tree node loaded into the level-2 cache is trusted and can be used to verify its child nodes' integrity. Therefore, it is not necessary to check the memory integrity all the way up to the root of the hash tree when there is a fetch of data.

Lu et at. [6] propose a novel tree-based memory integrity verification scheme which used 32-bit MAC rather than 128-bit hash to construct a tamper-evident environment. Because it becomes easy for an adversary to find a MAC collision between two cache lines, though the scheme provides substantial performance improvement, the security it provides is degraded. The author believes that an adversary cannot generate a MAC collision because MACs are computed using a secret key which is only known to the processor core. However, it is possible for an adversary to run his malicious program on this core to generate MACs. 32-bit MAC would make the key even more vulnerable.

Rogers et al. [8] propose a novel Bonsai Merkle Tree (BMT) scheme to reduce the size of a hash tree. Using 8 bit counters can decrease the size of protected memory from 4GB to 64MB (1:64). A security problem arises from using 8 bit counters to generate hashes. As discussed by Rogers et al., the difference between $H_k(C^{old}, ctr)$ and $H_k(C^{old}, ctr^{old})$ is the key to guarantee no replay attack can be carried out. However, it cannot guarantee that a Brute Force attack is not able to find the correct new ctr since it is 8 bit long, which means only 256 possibilities exist. This scheme could be employed in situations where a reduced-strength security model is acceptable, and where performance is critical. BMT scheme, as well as other hash tree schemes can also use our scheme to gain further improvement and flexibility, which will be explained in detail in the following section.

III. SEPARATED LEVEL-2 HASH CACHE

This section first covers some background on hash trees, and then presents the optimization.

A. Hash Tree Scheme

The cached hash tree technique is based on hash tree (Merkle Tree). In a hash tree, memory elements are stored in the leaf nodes. A node is usually of the size of a cache line except the root. Starting from the very left at each level, every *m* (*m* is the arity of the hash tree, meaning how many children a node could have at most) consecutive nodes have one same parent, which contains all the hash values of the *m* nodes. The root of the tree is stored in secure storage, which is inside the processor chip. Figure 1 illustrates the structure of such a tree.

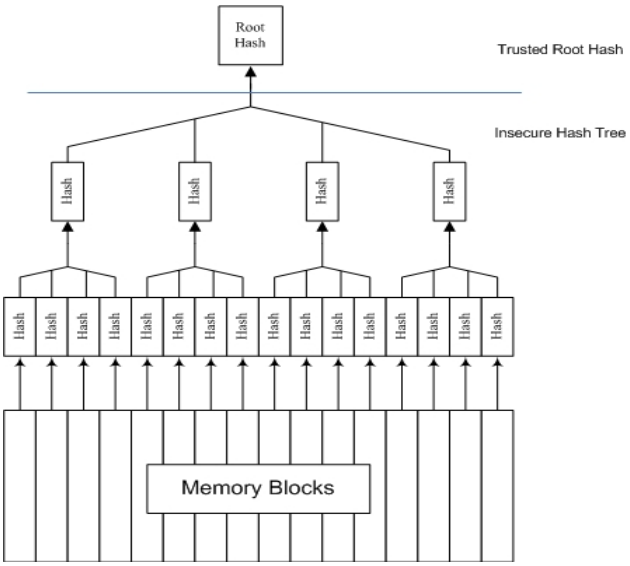

Figure 1. The structure of a hash tree.

When there is a data read miss in the cache, we need to verify the integrity of the data block to be fetched into the chip. The hash value of the data block (actually one of the leaf nodes) will be calculated and compared with the value which was stored previously. This process will be repeated all the way up to the root node. If there is any mismatch during the procedure, the data is not trusted and the system will be halted. Data write misses are handled similarly in write-allocate and write back caches. There is a slight difference when there is a write-back. Updating a memory chunk will result in updating the corresponding hash values of that memory chunk.

Because each node is of the same size, for an *m-ary* hash tree, it needs an extra $\frac{1}{m-1}$ of the system memory to store all the internal nodes. Therefore the more memory a computer has, the more additional storage it needs to hold

the hash tree and the taller the hash tree will be. During the verification procedure, more levels will be traversed in order to reach the root. The run time is of the Log(N) order (N is number of the total nodes).

B. Cached Hash Tree Scheme

To mitigate the performance problem, Gassend et al. propose caching part of the hash tree in the L2 cache. Since the L2 cache is on-chip, the cached part can be used as a trusted base for verification, and we do not need to verify integrity all the way up to the root of the hash tree for fetch and write-back. Once we reach a parent node in the L2 cache and use the hash in this node to check the integrity, the checking procedures are finished. However, writing that memory chunk back again will require an update of its hash. Therefore we need to have its parent node in the cache and update the corresponding hash value. Then we can use the updated parent node as a trusted base to verify the memory chunk's integrity. The dataflow for fetch and write-back procedures for a write back cache are depicted in Figure 2 and Figure 3 respectively.

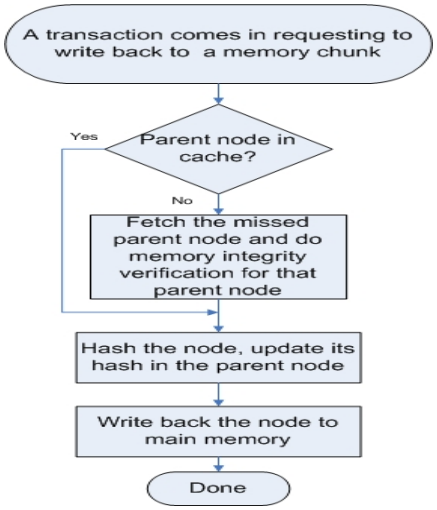

Figure 2. Flowchart of the write-back procedure

C. Separated Level-2 Hash Cache

Assume memory blocks A, B, C and D are all child nodes of memory block X. If they are fetched into L2 cache in turn, then cache lines occupied by A, B, C and D are accessed once respectively. However, their parent node X is read four times to verify each of them. From this access pattern, we can infer that the temporal locality of cache lines occupied by block A, B, C, and D is different than that of cache line occupied by block X. An LRU replacement policy will retain block X since it is accessed more recently than either A, B, C, or D. As a result, for a hash tree scheme, we observed that more than 50% (about 55% most of the time, sometimes about 60%) of L2 cache space is taken up by hash tree's internal nodes. However, the total memory space for storing the hash tree is 25% of the entire memory when the tree arity is 4.

Based on this, we believe it is not optimal to cache hash tree's internal nodes in L2 cache together with application data. We propose to split level-2 cache into 2 parts. One part is used to store application data, which are the hash tree's leaf nodes. The other part is used to hold the hash tree's internal nodes. We call it level-2 hash cache (L2HC).

The benefits of a separated level-2 hash are twofold. Firstly, it eliminates the contention between regular application data and hash tree's internal nodes. Therefore, existing pre-fetch strategies and speculative methods can be used directly on L2 cache. Otherwise, the hash tree's internal nodes will interfere with the correct operation of these techniques.

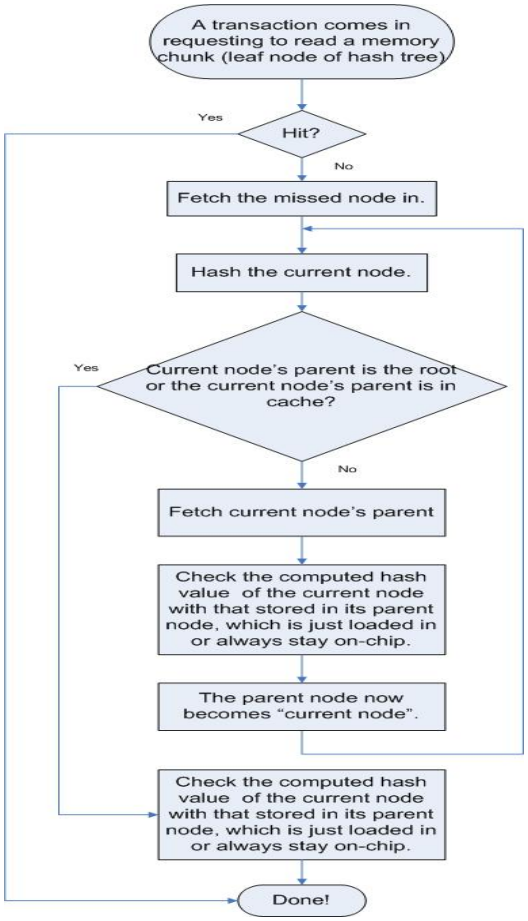

Figure 3. Flowchart of fetch procedure

Secondly, since locality of these two kinds of data is different, we could use different cache structures and replacement schemes to fit each of them. We name the scheme of employing another cache structure in L2 hash cache as heterogeneous level-2 hash cache. On the other hand, the scheme of using the same cache structure in L2 hash cache will be called as homogeneous level-2 hash cache.

With the separated L2 hash cache added, our hardware implementation is depicted in Figure 4.

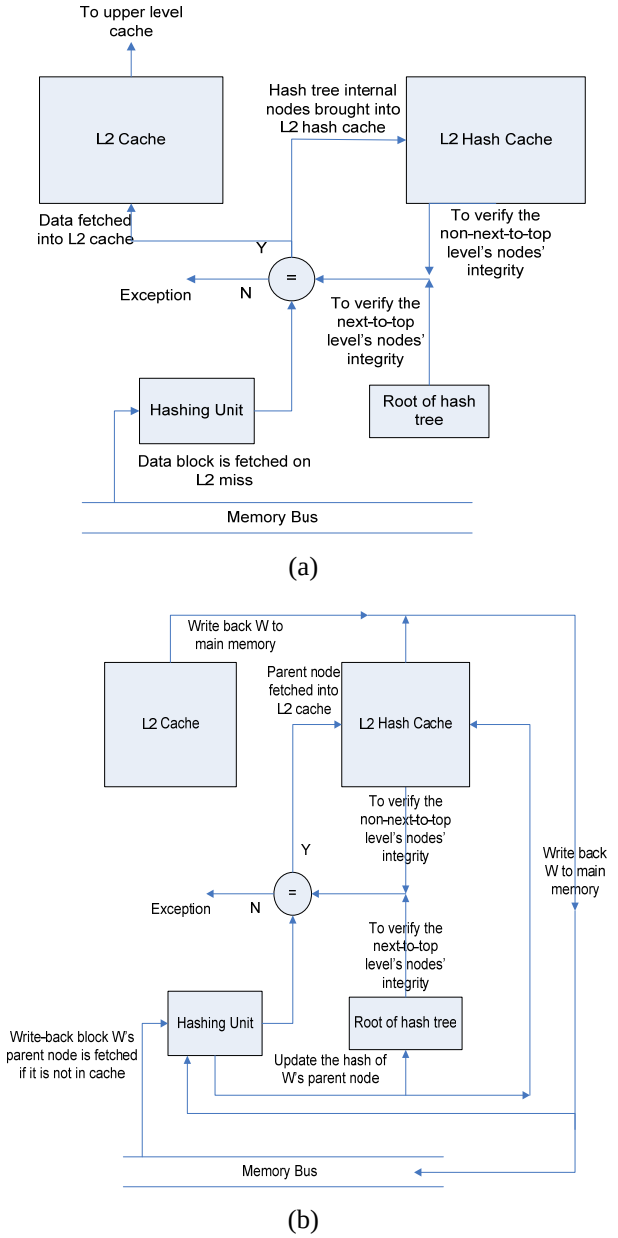

Figure 4. Hardware implementation of the cached hash tree scheme with separated level-2 hash cache. (a) Fetch from main memory. (b) Write back to main memory

IV. SIMULATION METHODOLOGY

Our secure architecture is simulated on Simics [10], which is a full system simulator. Currently, we are using an x86 core with Fedora Core 5 installed. The g-cache module (a module used in Simics to simulate cache structure) is employed to simulate our cache model.

At first, we did not know what kind of structure of level-2 hash is the best. We simply split the level-2 cache into 2 even parts with the same configuration.

A separate L2 hash cache can make use of a more appropriate cache structure to gain further performance gain. We tried to change the ratio of the size of level-2 hash to that of level-2 cache and the associativity of both to find out what configuration would produce the best result.

In Section V, the overhead of the following three systems will be illustrated.

- The first one is the original cached hash tree scheme and with unified 1M L2 cache (Abbreviated as CHT).
- The second one is the original cached hash tree scheme with L2 cache divided into a 512KB 8-way L2 cache and a 512KB 8-way L2 hash cache (Abbreviated as CHT, Ho-L2HC).
- The third system is a cached hash tree scheme with heterogeneous L2 hash cache. (Abbreviated as CHT, He-L2HC)

Other simulated architectural parameters are listed in Table 1. We also need to mention, in order to keep it fair, the total amount of level-2 cache is the same (1MB) for all of the three systems.

TABLE I. ARCHITECTURAL PARAMETERS USED

Architectural Parameters	Specifications
Clock frequency	3.6GHz
Memory size	4GB
L1 instruction cache	64KB, 2 way, 32B line
L1 data cache	64KB, 2 way, 32B line
L1 hit latency	3 cycles
L2 cache and L2 hash cache hit latency	10 cycles
Memory latency	200 cycles
Hash latency	80 cycles
Hash input size	512 bits
Hash output size	128 bits
Hash tree arity	4

We employed the MD5 hashing algorithm [11], which takes 512-bit blocks as input, and outputs a 128-bit hash. MD5 has 4 rounds, which are very similar and run a different operation 16 times. So, with proper lay-out of the logic gates required, each operation could be finished in one cycle, which leads to a total of 64 cycles to hash a 512-bit block. However, we will use 80 cycles here for fair comparison, because that is the parameter used in our baseline cached hash-tree scheme [3].

We target a web server as our simulated platform and employ SPECWeb 2005 [12] as our benchmark program. We chose this as our workload because a web server of an on-line banking, shopping or other commercial system contains a large amount of critical financial and personal information and hence has a higher likelihood of being attacked than the applications represented by SPEC CPU. SPECWeb 2005 has three kinds of workloads: *Banking*, *E-commerce* and *Support*. The three workloads simulate how clients access a web server and how the server corresponds to the requests from these clients in 3 different situations. We use the *Support* workload in this research, since it is

known to tax the memory subsystem the most out of these three [13].

SPECWeb 2005 provides many parameters for users to configure, among which the most important one is: *concurrent_sessions*. This denotes the number of clients accessing the server simultaneously. Because a server's performance is measured mainly by the response time to certain number of concurrent clients' requests [14], and we care most when the server is under heavy load, which means it is a popular server and a valuable target, we run the benchmark with the *concurrent_sessions* set as 1000. Our intent is to observe the performance gain achieved by our scheme under this heavy load situations.

For each simulation, we run 100 million instructions after warming up the cache by 10 million instructions starting from the same checkpoint. We then calculate the execution overhead incurred by memory integrity verification. Each system's performance is denoted by (overhead execution time) divided by (total running time subtracted by the overhead execution time).

V. SIMULATION RESULTS AND ANALYSIS

Figure 2 indicates that we can reduce the overhead by about 4.5% (from CHT's 43.29% to CHT, Ho-L2HC's 38.77%) if we separate half of L2 cache as L2 hash cache to store internal hash tree nodes.

As mentioned earlier in Section IV, we tried various configurations of level-2 cache and level-2 hash cache. We found that it is better to make the ratio of the size of level-2 cache to that of level-2 hash cache less than 1. It is also better to use large associativity in level-2 cache. The reason is that because a level-2 cache miss would generate multiple loads of internal nodes fetched into the same set of level-2 hash cache due to the lay-out of the hash tree.

After several experiments, we chose a system with an 8-way 424KB L2 cache and a 12-way 600KB L2 hash cache, which is the best level-2 cache structure we found. This system reduces the overhead to 34.69%.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have shown the importance of memory integrity verification, which could provide a tamper-aware environment.

The major contributions of our work are twofold:

- We observed the difference in locality of reference between application data and the hash tree's internal nodes, and proposed the use of a separate level-2 hash cache. From the analysis in previous sections, our scheme reduces the overhead by about 8.6% compared to the cached hash tree scheme.
- SPECWeb can simulate a heavily loaded server, which is a more relevant benchmark program for research on secure processors.

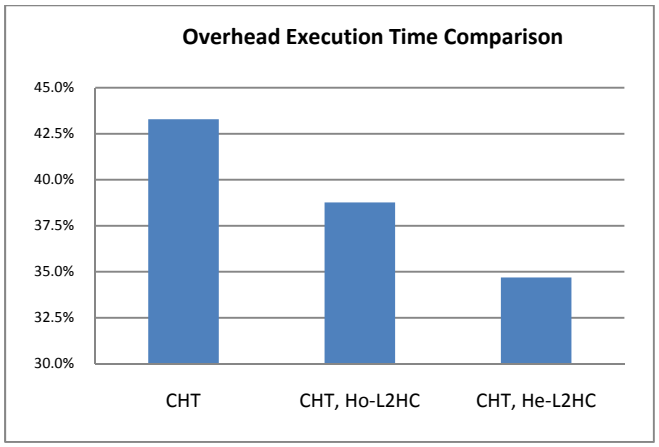

Figure 5. Overhead execution time comparison

We also found that during a smaller simulation, for example 100 thousand instructions, it is more profitable to cache hash tree's internal nodes of lower level in that hash tree. Because of the locality, programs will not jump too far in a short time. Lower level nodes could make integrity verification faster. If program jumps further enough, things may change. In that situation, a higher level node may be preferred. Therefore, replacement strategy of L2 hash cache needs to be studied in the future to further exploit this phenomenon. We will target an adaptive replacement strategy in our future work.

Because our simulations are carried out on a simulated scalar machine, the overhead is higher than that of the superscalar counterpart. The 43.29% overhead of our simulated machine is comparable to the 25% performance loss mentioned in [3]. Therefore, we will implement our scheme on a superscalar machine in the future, where we expect to achieve better results.

REFERENCES

[1] R. Panko, *Corporate Computer and Network Security*: Prentice Hall, 2003.

[2] R. Kannavara and N. G. Bourbakis, "Surveying secure processors," *Potentials, IEEE*, vol. 28, pp. 28-34, 2009.

[3] B. Gassend, *et al.*, "Caches and Hash Trees for Efficient Memory Integrity Verification," presented at the Proceedings of the 9th International Symposium on High-Performance Computer Architecture, 2003.

[4] J. Yang, *et al.*, "Improving memory encryption performance in secure processors," *Computers, IEEE Transactions on*, vol. 54, pp. 630-640, 2005.

[5] D. Lie, *et al.*, "Architectural support for copy and tamper resistant software," *SIGPLAN Not.*, vol. 35, pp. 168-177, 2000.

[6] C. Lu, *et al.*, "M-TREE: a high efficiency security architecture for protecting integrity and privacy of software," *J. Parallel Distrib. Comput.*, vol. 66, pp. 1116-1128, 2006.

[7] G. E. Suh, *et al.*, "Efficient memory integrity verification and encryption for secure processors," 2003, pp. 339-350.

[8] B. Rogers, *et al.*, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure

Processors OS- and Performance-Friendly," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, 2007, pp. 183-196.

[9] R. C. Merkle, "Protocols for Public Key Cryptosystems," presented at the Security and Privacy, IEEE Symposium on, 1980.

[10] Simics-3. (2007). *Simics User Guide for Unix*. Available: www.simics.net

[11] B. Schneier, *Applied Cryptography Second Edition: protocols, algorithms, and source code in C*: John Wiley & Sons, Inc, 1996.

[12] SPECWeb-2005. (2006, *SPECweb2005 Release 1.20 Benchmark Design Document*. Available: http://www.spec.org/web2005/docs/designdocument.html

[13] B. Ana, *et al.*, "Characterization of Apache web server with Specweb2005," presented at the Proceedings of the 2007 workshop on Memory performance: Dealing with Applications, systems and architecture, Brasov, Romania, 2007.

[14] M. Vipul, *et al.*, "MASTH proxy: an extensible platform for web overload control," presented at the Proceedings of the 18th international conference on World wide web, Madrid, Spain, 2009.