Hardware Acceleration for Cryptography Algorithms by Hotspot Detection

Jed Kao-Tung Chang[1], Chen Liu[1], and Jean-Luc Gaudiot[2]

[1] Department of Electrical and Computer Engineering, Clarkson University, Potsdam, NY
jchang@clarkson.edu
cliu@clarkson.edu
[2] Department of Electrical Engineering and Computer Science, University of California, Irvine, Irvine, CA
gaudiot@uci.edu

Abstract. Data Encryption/Decryption has become an essential part of pervasive computing systems. However, executing these cryptographic algorithms often introduces a high overhead. In this paper, we select nine widely used cryptographic algorithms to improve their performance by providing hardware-assisted solutions. For each algorithm, we identify the software performance bottleneck, *i.e.,* those "hotspot functions" or "hot-blocks" which consume a substantial portion of the overall execution time. Then, based on the percentage of execution time of a specific function and its relationship with the overall algorithm, we select candidates for our hardware acceleration. We design our hardware accelerators of the chosen candidates. The results show that our implementations achieve speedups as high as 60 folds for specific functions and 5.4 for the overall algorithm compared with the performance of the software-only implementation. Through the associated hardware cost analysis, we point to an opportunity to perform these functions in an SIMD fashion.

Keywords: cryptography; hardware acceleration; performance analysis; hotspot function

1 Introduction

Data security is important in pervasive computing systems because the secrecy and integrity of the data should be retained when they are transferred among mobile devices and servers in this system. The cryptography algorithm is an essential part of the pervasive computing security mechanism. By performing encryption, decryption, and hash operations on transmitted data, intruders should not be able to identify the hacked cipher text without decrypting schemes, and even a minute modification of the data shall be detected with a good data integrity verification process.

However, performance is one concern regarding cryptography algorithms: these algorithms are extremely expensive in terms of execution time. To make cracking the code more difficult, many arithmetic and logical operations are bound to be executed during the encryption/decryption process. Furthermore, a huge amount of data needs to be transferred between the CPU and the memory. Using a general-purpose processor for this purpose would not be a cost-effective solution, and the performance is not that satisfying either. This paper attempts to address this issue. We first deployed *execution-based profiling* to identify the hotspot (performance-intensive) part of an application: in each benchmark, we select candidates for hardware acceleration based on certain aspects, such as the percentage of total execution time belonging to hotspot functions, the relationship between the hotspot points and the entire application, *etc*. We then implemented these hotspot points in hardware. We compared the hardware and software implementation from two perspectives: performance and hardware cost. Our ultimate goal is to provide hardware-assisted solutions along these lines, so as to improve the performance of the crypto-computations in pervasive computing systems.

The rest of the paper is organized as follows: We review related work in Section 2. We specify the cryptography algorithms in Section 3. We introduce the performance analyzer tool VTune, and describe how we employ it to identify the hotspot functions and hot-blocks across the set of benchmarks in Section 4. In Section 5, we describe the criteria for selecting the candidate algorithms for hardware acceleration. The implementation of the hardware accelerator for the chosen benchmarks is described in Section 6. In Section 7, we describe the effort to investigate the hardware cost of our hardware implementation. Finally, conclusions are drawn in Section 8.

2 Related Work

Many previous research efforts have improved different parts of the pervasive computing system. For example, Tang *et al.* [21-37] employed hardware–assisted approach to accelerate selected middleware execution, and used prefetching techniques in mobile systems. Different from them, ours is focused on addressing the issue of improving performance of cryptographic computations. With the multithreading features which have been recently added to many programming languages, the straightforward software solution is to increase the level of concurrency. For example, Bielecki *et al.* [6] has attempted to deal with the performance issue and focused on parallel programming approaches. If purely focusing on a software strategy, however, the performance improvement would be limited since a general-purpose processor is not as efficient as a dedicated cryptography coprocessor.

Many people have strived to implement cryptography operations on a single chip. Hodjat *et al.* [17] has used a dedicated cryptographic coprocessor to alleviate the load on the CPU. Bertoni *et al.* [20] had implementations at a finer granularity via instruction set extension: they implemented different stages of AES algorithm into customized instructions. The concept of applying SIMD architecture, such as graphic processing unit (GPU), in asymmetric and some modes of symmetric encryption algorithms was also employed to improve the performance of the cryptography algorithms. Harrison *et al.* [18] implemented the AES Encryption ECB mode on GPUs.

Compared with previous work targeting a specific computation-intensive part of an algorithm for hardware acceleration, we work on a set of algorithms which have certain program structures and crypto-computation operations in common. We employed *dynamic-based profiling* (also described in Chang *et al.* [28]): to profile the performance behavior when running the benchmark and identify the parts that take up the most of execution time as hotspot points. When implementing these hotspot points into hardware accelerators, not only did they have superior performance than running on general-purpose processors, but also in some cases, the hardware costs are extremely low. We can duplicate the hardware accelerators for the hotspot points so that multiple data elements can be processed on these accelerators in a parallel fashion.

3 Cryptography Algorithms

We choose nine popular cryptography algorithms as benchmarks for our study: AES [3], RSA [4], 3DES [8], RC5 [9], MD5 [10], IDEA [11], SHA1 [12], Blowfish [13], and ECC [14]. The reason for our choice is that we feel the program structures of these algorithms are quite representative of the contemporary cryptography algorithms. For example, some algorithms like 3DES, Blowfish, and RC5 use the Feistel cipher proposed by Luby *et al.* [1]; other algorithms, such as AES, IDEA, MD5, and SHA1 use the iterative cipher designed by Rijmen *et al.* [2]. ECC and RSA are public key (asymmetric) algorithms [16], which are neither Feistel cipher nor iterative cipher. Comparing with the other seven algorithms, ECC and RSA seem to be more complex. However, in these algorithms, each data can be encrypted or decrypted independently and the operations can be performed in parallel to improve their overall performance. Another common feature shared by these algorithms is the operation they use to encrypt / decrypt the data. Memory access operations are used by 3DES, AES, and Blowfish. This is because these three algorithms need to access lookup tables). Modular arithmetic operations are also heavy in modern cryptography algorithms, because they need to keep the plaintext as secure as possible.

4 Hotspot Function Identification

We profile the dynamic computation characteristics of the benchmarks and utilize VTune [5] from INTEL® as our performance analyzer to identify the hotspots. VTune analyzes the software performance on IA-32 and Intel64-based machines. It collects performance data on applications running on the host system, organizes and displays the data in an interactive way. VTune's call graph view provides a tree structure to show the call relationship among all functions along with their execution time. This would help us identify the "hotspot" functions and percentage of the hotspot functions occupying the total execution time of each benchmark.

We define *HF-Rate* as the percentage of the execution time belonging to hotspot functions of each algorithm. Figure 1 shows the *HF-Rate* for each algorithm. It should be noted that for the execution time, we only consider the crypto computation (key setup, encryption, and decryption) part of an application, excluding the file I/O or certain system calls within the dynamic-link library (DLL).

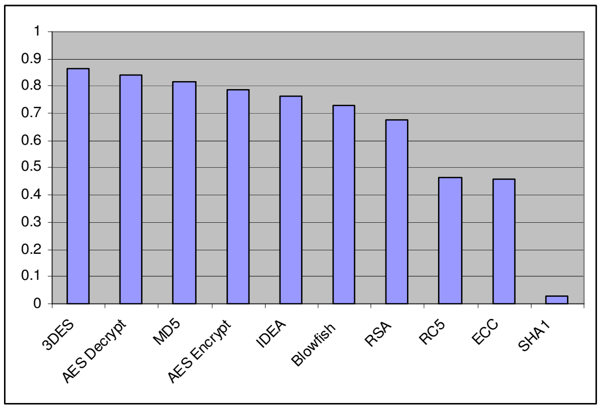

Fig. 1. HF-Rate for different algorithms

We can see that the execution time of the hotspot functions account for a majority of the total execution time for most of the benchmarks. In 3DES, MD5, and AES Decryption, the hotspot functions occupy more than 80% of the execution time of the entire algorithm. For AES Encryption, Blowfish, IDEA, MD5, and RSA, this number reaches or even exceeds 70%. However, SHA1 is an exception because most of its execution time is spent on I/O operations called by the crypto computation functions, such as reading from a file, which means that there is really no hotspot function to isolate. Overall, the function breakdown helps us identify the software bottleneck of the application.

5 Acceleration Candidate

When selecting our candidates for hardware acceleration, we need to consider two aspects of the hotspot function(s):

- Its *HF-Rate*
- The relationship between the hotspot function(s) and the overall algorithm.

The first aspect is evident: based on the Amdahl's Law, we would prefer to choose a hotspot function with a high *HF-Rate*, which is the percentage of total execution time dedicated to the hotspot function and is our target for acceleration. In the formula of Amdahl's law, *HF-Rate* corresponds to Fraction$_{\text{enhanced}}$, the percentage of the overall execution during which the performance enhancement was applied. If the *HF-Rate* is too low, the performance of the application cannot be significantly enhanced even if the performance of the hotspot point can be much improved.

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \tag{1}$$

Given the first criterion, as Fig. 1 indicates, SHA1 would not be taken into consideration due to its low *HF-Rate* of the hotspot function. The second aspect is equally important: if a hotspot function is the entire process of the overall algorithm, such as the encryption/decryption part, the hardware cost would be too high, because we would need to implement many hardware instructions and correspondingly many hardware components which would occupy too large a die area. Thus, a good candidate for hardware acceleration is a hotspot function with both a high *HF-Rate* and small size.

Next let us consider the hardware implementation for specific hotspot functions. The hotspot function of RSA is a function called *Power()*, which calculates two to the power of N, accounting for 67.6% of the RSA execution time. From the profiling we know that the maximum value of *N* is 31. It takes just one 32-bit barrel shifter to finish the operation.

The encryption/decryption of AES has four stages. The stages of encryption are *SubBytes()*, *ShiftRows()*, *MixColumns()*, and *RoundKey()*, while those of decryption are *InvSubBytes()*, *InvShiftRows()*, *InvMixColumns()*, and *RoundKey()*. The hotspot function of AES encryption is *SubBytes()*, the first stage of each round. In AES decryption, the three hotspot functions are *InvSubBytes()*, *InvMixColumns()*, and *SubBytes()*, corresponding to the first, third and fourth stages of AES decryption, respectively. Among them, *SubBytes()* and *InvSubBytes()* are memory access operations, replacing a byte with another one according to a prebuilt lookup table (SBox). In recent hardware implementations, the SBox is put into the cache so that the access time could be significantly reduced, according to Mourad *et al* [19]. For *InvMixColumns()*, its instructions can be implemented by a set of AND, shift, and table lookup operations according to [3]. *SubBytes()* in AES encryption contributes 78.5% of the total execution time, while the *InvSubBytes()*, *SubBytes()*, and *InvMixColumns()* in AES decryption contribute 55.8%, 17.5%, and 10.8% of the total execution time.

For Blowfish, the hotspot function is *F()* which contributes 73.13% of the total execution time. Blowfish has 16 rounds of data transformation and *F()* is executed once in each round. *F()* takes the 32-bit input and splits it into four eight-bit inputs and access four pre-built lookup tables (SBoxes) in parallel. The outputs are then XORed and added. Again, the four lookup tables can be placed in the cache and two adders, shifters, and one Exclusive-OR gate would be sufficient to implement the rest of the function. Thus, AES, RSA, and Blowfish would be good choices for us to perform the hardware acceleration.

As for 3DES, IDEA, MD5, and RC5, our analysis points to the hotspot function being the entire function of crypto computation. If we exclude this hotspot function, the rest of the algorithm is simply the initialization, the execution time of which is comparatively negligible. As mentioned previously, these algorithms are not good candidates for hotspot function acceleration since the hardware cost would be too high.

In this situation, we need to consider reducing the granularity for acceleration. We need to look inside each function and determine whether there is a "hot-line" or "hot-block" of code which contributes a significant amount of execution time and with low hardware overhead to qualify for acceleration. We employed the sampling wizard of

VTune to show the number of consumed clock cycles by each line of code. We go deeper into the functions and view the performance at a finer granularity.

For RC5, the hotspot function is the *rc5_key()* for the key expansion work. The hot-block is named *KEYXP_RC5*. *KEYXP_RC5* accounts 37.9% of the whole algorithm. For 3DES, the hotspot function is *des_crypt()*. The hot-block is named *ROUND_3DES*, which handles memory access and address translation. This hot-block *ROUND_3DES* accounts for 58.7% of the whole algorithm. The hot-block of MD5 is *P_MD5*, which consists of four rounds where each round is composed of sixteen function-based stages. The main operations for *P_MD5* are Modular additions and left rotations. *P_MD5* contributes 73.3% of the total MD5 algorithm. *MUL_IDEA* is the hot-block of IDEA. The input data is processed with the keys by the modular arithmetic and logic operations. The main operations for *MUL_IDEA* are modulo addition and multiplication operations. *MUL_IDEA* consumes 61.2% of the execution time of the entire algorithm.

6 Accelerator Implementation

Now we are ready to implement the hotspot function(s) of selected benchmarks (RSA, AES, and Blowfish) in hardware. We take the state transitions scheme for hardware implementation. First, we translate the function(s) into finite state machine. Note that although the instructions are executed in sequential order, if we find there is no data dependency among the instructions, we can execute them in parallel and put them into the same state in the finite state machine. For RSA, one 32-bit barrel shifter can shift left a number by 0 to 31 bits and we just need one state. For *InvSubBytes()* of AES Decryption and *SubBytes()* of AES Decryption and Encryption, it is only a replacement operation for a byte according to a pre-built lookup table. Some logic components would be sufficient as it is one state of memory access.

The implementation of *InvMixColumns()* using Rijindael mix columns requires six states. In the first state, the input array is read into the tentative storage used for the later states. The inverse mix column operates on the data by multiply numbers in Rijndael Galois Field, which takes four states because four dependent instructions need to be executed. In the final state, the results will be stored back to the array. Thus, a total of six states are required to implement this function.

The *F()* of Blowfish needs three states. In the first state, a 32-bit input is split into four 8-bit inputs as an index to access the lookup table. In the second state, the four lookup tables are accessed in parallel and they produce four outputs. In the third state, four outputs are combined into one using the modulo addition and XOR operations.

Next, we discuss the hardware implementation of the hot-blocks of RC5, 3DES, MD5, and IDEA. The hot-block *KEYXP_RC5* in RC5 is inside a for-loop structure. We need three states to implement the operations of mixing secret keys to the key table.

The 3DES's hot-block *ROUND_3DES* has three states. In the first state the input will be operated on using different calculations in parallel and two outputs will be generated. In the second state, the outputs are used as indices to access one of the pre-built lookup tables. Then, in the third state, the values from the second state will be used to access eight other pre-built lookup tables and to perform the XOR operation on these outputs to obtain the result.

The hot-block of MD5 is *P_MD5*. As we know, MD5 is composed of 64 function-based stages. The first, second, third, and final 16 stages are grouped together as one round with a slightly different translation functions. We thus just need to implement four stages as four hardware modules separately with different functions. During the first round, the first hardware module will be called; the second module is called in the second round, and so on. Each module just needs two states. The first state is to execute the function with the given input. The second state is to perform some simple operations based on the output of the first state.

Table 1. Hardware acceleration speedup

Hotspot Function / Benchmark	**FSM Stages**	**Function Speedup**	**Algorithm Speedup**	
Power() / RSA	1	26	2.9	
SubBytes() / AES Encryption	1	56	4.4	
InvSubBytes() / AES Decryption	1	48	1.2	5.4
SubBytes()/ AES Decryption	1	60	2.2	
InvMixColumns() / AES Decryption	6	9	1.1	
F() / Blowfish	3	2	1.7	
KEYXP_RC5 / RC5	3	4	1.4	
ROUND_3DES / 3DES	3	2	1.5	
P_MD5 / MD5	8	9	2.9	
MUL_IDEA / IDEA	4	5	2.0	

The hot-block *MUL_IDEA* of IDEA takes four states to complete. The first state is for initialization and the following three states are for set of shift, add, and multiplication operations on the input data and key.

We implemented the hardware-assisted cryptographic functions as the accelerator connected with Microblaze using the PLB bus on the Xilinx XUPV5-LX110T development system. We measured and compared the performance in terms of the number of clock cycles and of the hardware and software implementation.

Table 1 summarizes the number of stages of finite state machine needed by each hotspot function, the hardware acceleration speedups we achieved over specific functions and over the entire algorithm. The results show that for hotspot function acceleration we can achieve 2 to 60 folds of performance improvement; for hot-block acceleration, we can achieve 2 to 9 folds of performance improvement; as for the entire cryptographic algorithm, hardware acceleration can achieve performance improvements of 1.4 to 5.4 folds, depending on the program structure.

7 Hardware Cost Analysis

Given the superior performance of the hardware accelerator implementation of hotspot functions and hot-blocks of the candidate benchmarks, we now look into implementing them in an ASIC design, hardware cost being our top-most concern.

We measured the hardware resource utilization based on the number of hardware slices (*#slices*), the number of flip-flops (*#FF*), and the number of look-up tables (*#LUT*). Table 2 summarizes the hardware resource utilization of these implementations.

Table 2. Hardware Resource Utilization

Hotspot Function / Benchmark	**#Slices**	**#FF**	**#LUT**
InvSubBytes() / AES Decryption	5	17	14
InvMixColumns() / AES Decryption	524	226	174
SubBytes()/ AES Decryption	5	17	14
SubBytes() / AES Encryption	5	17	14
Power() / RSA	5	17	14
F() / Blowfish	26	33	96
MUL_IDEA / IDEA	58	73	102
KEYXP_RC5 / RC5	137	175	423
ROUND_3DES / 3DES	6	19	19
P_MD5 / MD5	37	74	107

Based on the criteria we listed in Section 5 in the selection of the functions for hardware acceleration, the hardware overhead incurred with the accelerator is simply minimal, as shown in Table 2. For comparison, let us consider a very simple in-order MIPS processor design [7], the resource utilization of which requires 10,450 hardware slices, 10,400 flip-flops, and 19,500 look-up tables. Thus, if we implement the hotspot functions or hot-blocks in the form of a hardware accelerator, we need much fewer hardware resources compared to modern general-purpose processors as platforms on which we normally run cryptographic applications.

From the hardware cost, we observe that the hardware accelerator implementation is not only superior in performance but also matches the low cost of modern chip design. Given the several orders of magnitude difference between the hardware and software implementations, it is worthwhile for us to consider implementing the hotspot functions in the form of special functional units. We can achieve huge data parallelism by duplicating these function units. That is, we can extend our work to build a SIMD machine by grouping multiple data elements together and performing the operations using special processing units all at once.

8 Conclusions

In this study we have designed, implemented, and evaluated hardware acceleration approaches for cryptographic algorithms. We used the VTune performance analyzer to extract the hotspot functions and hot-blocks as identified by their high *HF-Rate* and low hardware overhead. We then translated the hotspot functions and hot-blocks into finite state machines and implemented them as hardware accelerators. Compared to previous research, we used a “hardware-assisted” method, i.e., we move the computation intensive part of the cryptography application into hardware so that the general-purpose computing resource can be released to perform other useful tasks. Our results indicate that for hotspot function acceleration we can achieve 2 to 60 folds of performance improvement; for hot-block acceleration, we can achieve 2 to 9 folds of performance improvement; for overall cryptographic algorithm execution, we can achieve 1.4 to 5.4 folds of speedups, compared with the traditional general-purpose processor. Through the hardware overhead investigation, we observe that the minimal hardware overhead is incurred during the accelerator implementation. This provides us an inspiration that we may build a SIMD machine to perform the hardware acceleration simultaneously, taking advantage of the data parallelism, which will be the focus of our future work.

ACKNOWLEDGEMENTS
This work is partly supported by the National Science Foundation under grants number CCF-1065448 and ECCS-1301953. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References:

1. Luby, Michael; Rackoff, Charles (April 1988), "How to Construct Pseudorandom Permutations from Pseudorandom Functions", *SIAM Journal on Computing* 17 (2): 373–386
2. V. Rijmen, "Cryptanalysis and design of iterated block ciphers," Doctoral Dissertation, October 1997, K.U.Leuven.
3. NIST (National Institute of Standards and Technology), “Advanced Encryption Standard (AES) – FIPS Pub. 197,” November 2001.
4. Rivest, R. L., A. Shamir, L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of ACM*, Vol.21, No. 2, Feb. 1978, pp. 120-126.
5. Intel VTune, http://software.intel.com/en-us/intel-vtune/
6. Wńodzimierz Bielecki and Dariusz Burak, “Parallelization Method of Encryption Algorithms”, *Advances in Information Processing and Protection*, 2008, pp. 191-204
7. The eMips project. http://research.microsoft.com/en-us/projects/emips/default.aspx
8. D.W. Davies and W.L. Price. *Security for Computer Networks*. Wiley, 1989.
9. R.L. Rivest. The RC5 encryption algorithm. In *Proceedings of the 2ndWorkshop on Fast Software Encryption*, pages 86-96, Springer, 1995.
10. R.L. Rivest. The MD5 message-digest algorithm, Request for Comments (RFC1320), Internet Activities Board, Internet Privacy Task Force, 1992.
11. Xuejia Lai. On the Design and Security of Block Ciphers. Hartung-Gorre Verlag, 1992.

12. FIPS 180-1. Secure hash standard, NIST, US Department of Commerce, Washington D.C., Springer-Verlag, 1996.
13. *Counterpane Systems*. http://www.counterpane.com.
14. Koblitz, N.: Elliptic curve cryptosystems. *Mathematics of Computation* 48 (1987)203–209
15. IEEE 1363: Standard Specifications for Public-Key Cryptography http://grouper.ieee.org/groups/1363/
16. Bruce Schneier, Applied Cryptography. John Wiley & Sons, 1996
17. Alireza Hodjat, "Interfacing a high speed crypto accelerator to an embedded CPU", *Proceedings of the 38th Asilomar Conference on Signals, Systems, and Computers*, 2004, pp. 488-492
18. Owen Harrison and John Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units", *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, 2007, pp. 209-226
19. Ould-cheikh Mourad, Si-Mohamed Lotfy, Mehallegue Noureddine, Bouridane Ahmed, Tanougast Camel, "AES Embedded Hardware Implementation," ahs, pp.103-109, *Second NASA/ESA Conference on Adaptive Hardware and Systems* (AHS 2007), 20002
20. Guido Marco Bertoni, Luca Breveglieri, Farina Roberto, Francesco Regazzoni , "Speeding up AES by extending a 32 bit processor instruction set", in *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, (ASAP 2006), pp. 275-282
21. Jie Tang, Shaoshan Liu, Zhimin Gu, Xiao-Feng Li and Jean-Luc Gaudiot, "Hardware-Assisted Middleware: Acceleration of Garbage Collection Operations", Proceedings of the 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2010),2010
22. Jie Tang, Shaoshan Liu, Zhimin Gu, Chen Liu, and Jean-Luc Gaudiot, "Prefetching in Embedded Mobile Systems Can Be Energy-Efficient," Computer Architecture Letters, DOI: http://doi.ieeecomputersociety.org/10.1109/L-CA.2011.2, February, 2011
23. Jie Tang, Pollawat Thanarungroj, Chen Liu, Shaoshan Liu, Zhimin Gu, and Jean-Luc Gaudiot, :Pinned OS/Services: A Case Study of XML Parsing on Intel SCC", Journal of Computer Science and Technology, in press
24. Jie Tang, Shaoshan Liu, Chen Liu, Zhimin Gu and Jean-Luc Gaudiot, "Acceleration of XML Parsing Through Prefetching", IEEE Transactions on Computers, in press.
25. Jie Tang, Shaoshan Liu, Zhimin Gu, Chen Liu and Jean-Luc Gaudiot, "Memory-Side Acceleration for XML Parsing", The 8th IFIP International Conference on Networking and Parallel Computing (NPC 2011), Changsha, Hunan, China, October 21-23, 2011
26. Jie Tang, Shaoshan Liu, Zhimin Gu, Xiao-Feng Li and Jean-Luc Gaudiot, "Achieving Middleware Execution Efficiency: Hardware-Assisted Garbage Collection Operations", Journal of Supercomputing, DOI: 10.1007/s11227-010-0493-0, November, 2010
27. Shaoshan Liu, Richard Neil Pittman, Alessandro Forin, and Jean-Luc Gaudiot, "Minimizing the Runtime Partial Reconfiguration Overheads in Reconfigurable Systems", Journal of Supercomputing, DOI: 10.1007/s11227-011-0657-6, July 22, 2011
28. Jed Kao-Tung Chang, Chen Liu, Shaoshan Liu, and Jean-Luc Gaudiot, "Workload Characterization of Cryptography Algorithms for Hardware Acceleration", Proceedings of the 2nd ACM International Conference on Performance Engineering (ICPE 2011), Karlsruhe, Germany, March 14-16, 2011