Loop Level Analysis of Security and Network Applications

Dinesh C Suresh, Satya R. Mohanty, Walid A. Najjar, Laxmi N. Bhuyan and Frank Vahid

Department of Computer Science
University of California, Riverside
Riverside, California, 92521
{dinesh, satya, najjar, bhuyan, vahid}@cs.ucr.edu

Abstract--It has been known that loops constitute the most executed segments of programs and therefore are the best candidates for hardware implementation. We present a set of profiling tools that are specifically dedicated to loop profiling and do support combined function and loop profiling. One tool relies on an instruction set simulator and can therefore be augmented with architecture and micro-architecture features simulation while the other is based on compile-time instrumentation of gcc and therefore has no slow down compared to the original program

***Index Terms*—Loop analysis, profiling, hardware software partitioning**

I. INTRODUCTION

Software programs spend most of the time in a small fraction of code, a feature known as the "90-10 rule" – 90% of the execution time comes from 10% of the code. In order to speed up program execution, we need to identify the critical code that contributes to the bulk of the execution time. For embedded system applications, this frequently executed portion of the code is often made up of a few loops. Besides optimization, mapping the frequently executed portion to hardware would be an efficient way of speeding up program execution. For mapping an application to hardware, knowledge of the time spent in different portions of the application is necessary. Profilers like *gprof* are helpful to the extent of determining the time spent on function calls. However, to make judicious hardware/software partitioning decisions, knowledge of the program execution time at the granularity of loops is imperative. Instruction profiling tools can be tuned to provide useful information regarding the percentage of time spent in different parts of a program.

Instruction profiling tools can be broadly classified into two categories – instrumentation based instruction profilers and simulation based instruction profilers. An instrumentation based profiler instruments the compiler to add counters to various basic blocks of the program. During execution the counter values are written to a separate file. On the other hand, a simulation based instruction profiler uses an instruction set simulator to accomplish instruction profiling. Simulation based profilers can be further classified into static profilers and dynamic profilers. In dynamic profiling the instruction profile is obtained during the execution of the code on an instruction-set simulator while in static profiling the execution is written to a trace and the trace is processed to get instruction counts. For very large applications, the trace generated by a static profiler can grow to unmanageable proportions. Even though a dynamic profiling method is slow compared to the compiler-based instrumentation approach, a variety of architectural parameters can be tuned and studied while a program gets profiled on the simulator.

In this paper, we present a loop analysis tool set that is equipped with two different instruction profilers - an instrumentation based instruction profiler and a simulation based instruction profiler. We call our toolset as Frequent Loop Analysis Toolset (FLAT). FLAT consists of two major tools – FLATC, an instrumentation based loop analysis tool and FLATSIM, a simulation based loop analysis tool. We use FLAT to analyze well-known cryptographic algorithms and network applications and report the speedup that can be achieved by mapping time-consuming loops in these applications to hardware.

This work was supported in part by NSF Award ITR 0083080

Figure 1: Tool flow for compiler based instruction profiling

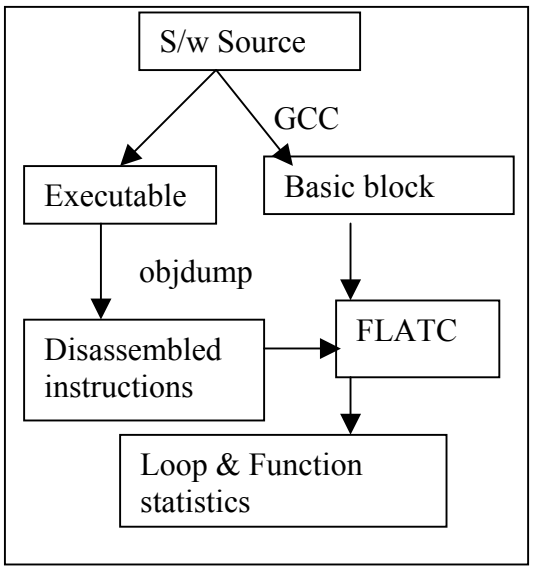
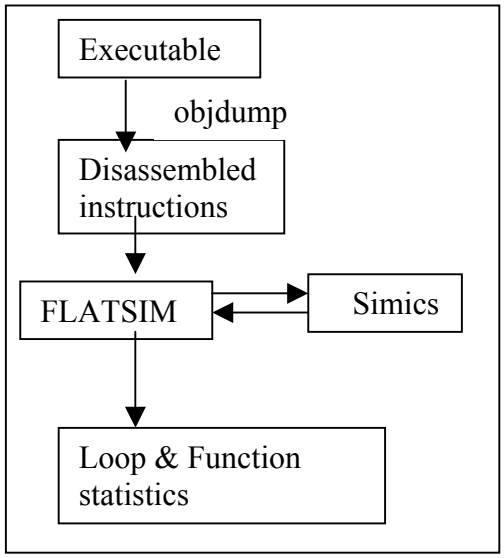


Figure 2 Tool flow for simulation based instruction profiling



II. RELATED WORK

Harvard Atom Like Tool (HALT [4]) provides a flexible way to add routines to program produced by the SUIF compiler. Users indicate interesting parts of the program by labeling them with SUIF annotations. Then Halt looks for these annotations, and inserts function calls to analysis routines that match the type of the annotation. Halt ships with a number of useful analysis routines; users may modify these or supply their own. Using different analysis routines, Halt provides a number of hardware simulators, performs branch-stream analysis, and records statistics for profile-driven optimizations. Halt and its associated libraries have been used in projects on branch prediction, code layout, instruction scheduling, and register allocation. It has been ported to MIPS and ALPHA processors

Optimally profiling and Tracing Programs [10] inserts counters at all nodes in the control flow graph in order to record the execution count of the basic blocks and the program. ProfileMe [6] samples instructions as they move through an out-of-order issue pipeline and reports statistics like cache miss rates. LooAn [1] is a profiling tool that gives loop and function level information. However, since it is a static profiler, trace files scale up to unmanageable proportions for very large programs.

Shade [12] combines instruction set simulation with trace generation capability. It uses a user-specified trace analyzer to control program execution and the extent of trace generation. The analyzer code is generated dynamically and is cached for reuse. ALTO [13] develops whole-program data flow analysis and code optimization techniques for link time program optimization and is targeted to the DEC Alpha architecture.

Intel's Vtune [14] performance analyzer collects, analyzes and displays software performance data from the program-level down to a specific function, module or instruction in a developer's source code. Vtune runs on windows and linux and is targeted for all Intel processors.

[2], [3] provide a detailed analysis of cryptographic algorithms. They propose addition of new instructions to the instruction set in order to speedup the execution of security applications. In this paper, rather than analyzing the applications at the instruction level, we study them at the granularity of loops and functions. We present a toolset that is dedicated to performing loop-level analysis of applications. We also examine the performance benefits associated with mapping the first four frequent loops or functions of each application to hardware.

SpixTool [15] is an instruction profiling toolset intended for the SPARC architecture and it consists of the following two tools – Spix and Spixstat. Spix generates basic block execution profile; while Spixstat generates statistics on instruction count, branch behavior, opcode

Table 1: FLAT_C's output for 3DES application

Loop Name	Frequency	Loop Size	Total Instructions	%
<Program>	1	13171	4394609445	100.00
<permute.1>	6051888	357	1792703712	40.79
<doencrypt.1.1>	3025944	51	98175072	2.23
<doencrypt.1>	2689728	262	90442126	2.06
<perminit.1.1.1>	33280	287	1967104	0.04
<perminit.1.1>	32768	331	404672	0.01
<spinit.2.1.1>	2560	136	199680	0.00
<setkey.2.2>	784	262	110176	0.00
<spinit.2.1>	2048	303	95200	0.00
<setkey.2.1>	912	161	86200	0.00
Functions				
Function Name	**Frequency**	**Loop Size**	**Total Instructions**	**%**
<f>	5379456	387	2001157632	45.54
<permute>	5379456	491	1944000912	44.24
<doencrypt>	336215	321	193996674	4.41
<endes>	336216	487	150288552	3.42
<perminit>	32	475	2379512	0.05
<spinit>	32	591	332845	0.01
<setkey>	56	813	238733	0.01
<main>	8	1158	4588	0.00
<desinit>	1	375	235	0.00
<_init>	0	48	0	0

usage, etc. Loop information can be easily deduced from the tool's output.

Cacheprof [16] is an execution-driven memory simulator for the x86 architecture. It annotates each instruction that reads and writes memory and links a cache simulator into the resulting executable. Upon execution, the data references are trapped and sent to the simulator. Besides producing a procedure-level summary, Cacheprof reports number of memory references and the number of misses for each line of the source code.

FLAT is intended to provide loop/function level information for a wide variety of platforms. FLAT_C works for all platforms to which the GNU C Compiler (gcc) has been ported. FLAT_SIM is capable of producing loop level statistics for a variety of platforms like x86, Strong ARM, MIPS and SPARC. FLAT_SIM can produce loop information from the executable even when the source code is not available.

III. FREQUENT LOOP ANALYSIS TOOL SET (FLAT)

Instruction profiling tools provide information based on which useful hardware/software partitioning decisions can be made. Frequent Loop Analysis Tool set (FLAT) is a profiling tool written in python and it provides the execution time of a given application at the granularity of both loops and functions. Loop profiles can be obtained through two different ways. The first method is to instrument the compiler to output the frequency of a loop. The second method is to use an instruction set simulator to find the execution count of loops. Both methods have their own advantages and disadvantages. The instrumentation-based approach is a lot faster while the simulation-based approach is more effective in tuning the various architectural aspects of the application.

During hardware/software partitioning, frequently executed functions often prove to be the favorite candidates for hardware mapping. However, a frequently executed function could have lots of infrequently executed loops that contribute towards the total execution time of the function. Since loops perform the bulk of computation, returns for the silicon real estate would be maximized if a frequently executed loop of the program were chosen instead of the frequent function mentioned above. The output provided by FLAT is useful in deciding whether a loop or function needs to be mapped onto hardware. FLAT considers functions as loops that iterate once for each call. FLAT comprises of two profiling tools - FLATC and FLATSIM.

FLATC uses the GNU C Compiler to profile the application for basic block frequencies. The source program to be compiled is compiled with the "-a" option. This ensures that a file containing basic block frequencies is written after execution. FLATC uses the disassembled instructions to identify the presence of loops and functions. Every loop in the source program corresponds to a short backward branch instruction in the assembly program. Once the loops and function calls are identified, the percentage execution is determined from the execution percentage of basic blocks.

FLATSIM uses the Simics [11] instruction set simulator to do the instruction profiling. Simics™ is a full system simulation platform, capable of simulating high-end target systems. Simics can boot and run operating systems and commercial workloads. It provides a controlled, deterministic, and fully virtualized environment for a variety of hardware and software engineering tasks. Hence, we chose to instrument the Simics modules to get realistic loop profile estimates.

Simics is not an open source simulator. However, the source code for the add-on modules is included with the

distribution. The functionality of the simulator can be extended by modifying the existing modules or by creating custom modules. One such module that is supplied with the Simics distribution is the *id-splitter* module. The id-splitter module in Simics handles all cache accesses and redirects them to the instruction or data cache accordingly. FLAT_SIM relies on getting the instruction profile from a modified version of the id-splitter module. The suggested modification to the id-splitter module is as follows. A tree structure containing all the loop-addresses is introduced into the id-splitter module. During execution, if an instruction belongs to one of the loops, the counter associated with the loop is incremented. Finally information about loops and function calls are written to a file. FLAT_SIM analyses this file and prints out information regarding the loop execution.

Table 1 shows the output of FLAT for *3DES* application. Each entry in the table contains a loop name, number of times the loop was called, the static loop size in bytes, total number of instructions executed in the loop and the percentage of instruction cycles contributed by the loop. FLAT maintains a Directed Acyclic Graph (DAG) like representation for handling loops and functions. Every loop and function is associated with a name. The loops and functions are named in a hierarchical fashion. For example, the loop name <*doencrypt.1*> in table 1 points to the first loop in the function called *doencrypt.* The first sub-loop of this loop would be named as <doencrypt.1.1>. The function statistic consists of the function name, number of times it was called, static size, total number of instructions executed inside the function and percentage of time spent in the function.

FLAT obtains loop information from the disassembled object code. Hence, if the compiler resolves the dependencies across threads and schedules the instructions accordingly, multithreaded applications can also be profiled effectively.

IV. BENCHMARKS

We analyze an extensive collection of security (AES, 3DES, rc4, rc6, idea, blowfish, seal [21] and sha1 [22]) and network applications (crc, dh, ipchains, drr, nat, route and md5 - all from Netbench [6]).
DES [20] was published in 1977 and it is based on IBM's work on Lucifer. It uses a 56-bit key to generate sixteen 48 bit per-round keys, by taking a different 48-bit subset of the 56 bits for each of the keys. The 64-bit input is subjected to a series of permutations and the encryption of a given block also depends on the previous encrypted block. 3-DES (pronounced Triple DES) is an extension to DES, it performs DES three times using three different unrelated keys and achieves a high level of security.

IDEA [18] is best known as a component of Pretty Good Privacy (PGP). It is a block cipher that uses a 128-bit length key to encrypt successive 64-bit blocks of plaintext. The procedure is quite complicated using sub keys generated from the key to carry out a series of modular arithmetic and XOR operations on segments of the 64-bit plaintext block. The encryption scheme uses a total of fifty-two 16-bit sub keys. The cipher Blowfish [19] is a symmetric block cipher that takes a variable length key from 32 to 448 bits and is considered an alternative to DES or IDEA. RC4 [23] is a stream cipher algorithm devised by Ron Rivest. It uses a variable length key from 1 to 256 bytes to initialize a 256-byte state table. This table is used for generating pseudo-random bytes and then a pseudo-random stream that is XORed with the plaintext to give the cipher text. The key is often limited to 40 bits but can be as much as 2048 bits. RC6 [24] is a cipher first introduced by the RSA Labs. Circular shifts and a quadratic function are some nonlinear elements that provide security to this cipher. It uses 44 keys each 32 bits long. Rijndael is a block cipher invented by Joan Daemen and Vincent Rijmen and it is the current AES [17] standard. It has a variable block and key length, both of which can be multiples of 32 bits.

In this paper we also focus attention on Netbench applications. The Netbench applications comprise of about 9 applications that are representative of commercial applications for network processors. There are two micro-level programs CRC: the CRC-32 checksum calculation program and TL, which is the Table Lookup algorithm for radix tree routing tables. There are four IP-Level programs, so named because they base decisions on source or destination IP of the packet. Route, NAT, DRR and IPCHAINS constitute the IP-Level programs. Route implements the table lookup along with the Internet checksum. DRR (Deficit Round Robin) is a scheduling method used in network switches and it is characterized by the presence of different queues for all the different connections through the routers. Network Address Translation (NAT) operates on a router and translates private addresses in a network to legal addresses before forwarding the packets onto another network. IPCHAINS is a firewall application that filters incoming IP packets according to some well-defined policies.

Finally there are some application level programs with intensive processing requirements. DH (Diffie-Hellman) is a public key based encryption/decryption algorithm and is widely deployed in several Virtual Private Networks.

Table 2. Percentage Execution for the first four loops of security/network applications

Benchmark	Loop1	Loop2	Loop3	Loop4
AES	77.79	15.32	6.34	0.00
Blowfish	62.06	25.10	5.82	1.96
CRC	87.23	11.20	0.16	0.06
DES	45.54	40.79	4.41	3.45
DH	35.39	14.31	13.12	5.66
DRR	18.08	10.15	10.05	7.74
IDEA	48.10	33.68	6.36	4.92
Ipchains	56.41	21.88	4.63	2.7
MD5	32.17	24.47	19.77	3.38
NAT	34.24	19.32	19.18	8.30
RC4	94.47	0.01	0.01	0.00
RC6	86.19	1.37	0.00	0.00
Route	53.45	17.10	5.63	5.29
Seal	56.97	33.83	1.05	0.78
SHA1	75.31	18.61	4.04	1.62
Average	57.56	19.14	6.71	3.06

MD5 (Message Digest Algorithm) creates a secure signature for each outgoing packet. The signature is verified at the destination and packets without matching signature are discarded.

V. ANALYSIS OF PROFILE DATA

Table 2 shows the percentage execution time for the first four most frequent loops of security and network applications. We find that the first four frequent loops take up roughly 20% of the code size and contribute nearly 88% towards the total execution time. Compile-time optimizations, algorithmic improvements and hardware mapping are the different alternatives available to speedup the application. Normally, large code size often proves to be a hindrance in hardware mapping as it consumes a lot of programmer hours. Since the frequent loops take up a fraction of the code size and still provide a major contribution towards the execution time, they are ideal candidates for hardware mapping. Figure 3 shows the percentage of the total execution time spent in the first four loops of cryptographic and network applications.

To demonstrate the usefulness of our profiling tools we consider a System on a Configurable Platform (SoCP) that consists of an FPGA with an embedded CPU. Examples of such system include the Xilinx Virtex II Pro [8], the Altera Excalibur [7] and the Triscend A7 [9].

Table 3. Overall Speedup for the first four loops of security/network applications

Benchmark	Loop1	Loop2	Loop3	Loop4
AES	3.75	8.15	15.9	15.9
Blowfish	2.41	5.59	8.07	9.48
CRC	5.61	13.78	14.07	14.17
DES	1.75	5.36	6.9	8.89
DH	1.5	1.88	2.45	2.82
DRR	1.21	1.36	1.56	1.77
IDEA	1.83	4.36	5.9	8.12
Ipchains	2.13	3.81	4.57	5.17
MD5	1.43	2.14	3.57	4.03
NAT	1.48	2.02	3.18	4.23
RC4	9.1	9.1	9.11	9.11
RC6	5.32	5.72	5.72	5.72
Route	2.01	2.98	3.55	4.31
Seal	2.16	6.92	7.43	7.86
SHA1	3.44	8.69	12.98	16.21
Average	2.19	3.61	4.67	5.4

On such systems it is possible to migrate the most commonly executed code segment onto hardware by implementing it as a circuit on the FPGA. The obvious objective is the speed-up that can be achieved. In this section we describe an analysis of this speed-up based on the results obtained from the profiling tool. Note that in this analysis we will not assume any overlap in computation between the CPU and the FPGA on the SoCP. This is a pessimistic but fair assumption.

SoCP time = CPU time + FPGA time

CPU time = SW_only time – SW_Loop time

FPGA time = SW_Loop time/HW_speedup

where SW_only time is the time from a software only execution and the SW_Loop time is the time taken on a CPU by the loop that will be mapped to hardware. The HW_speedup is the speedup expected on the loop by mapping it to hardware. From past results [1] we have computed this speedup to be 17 in number of cycles.

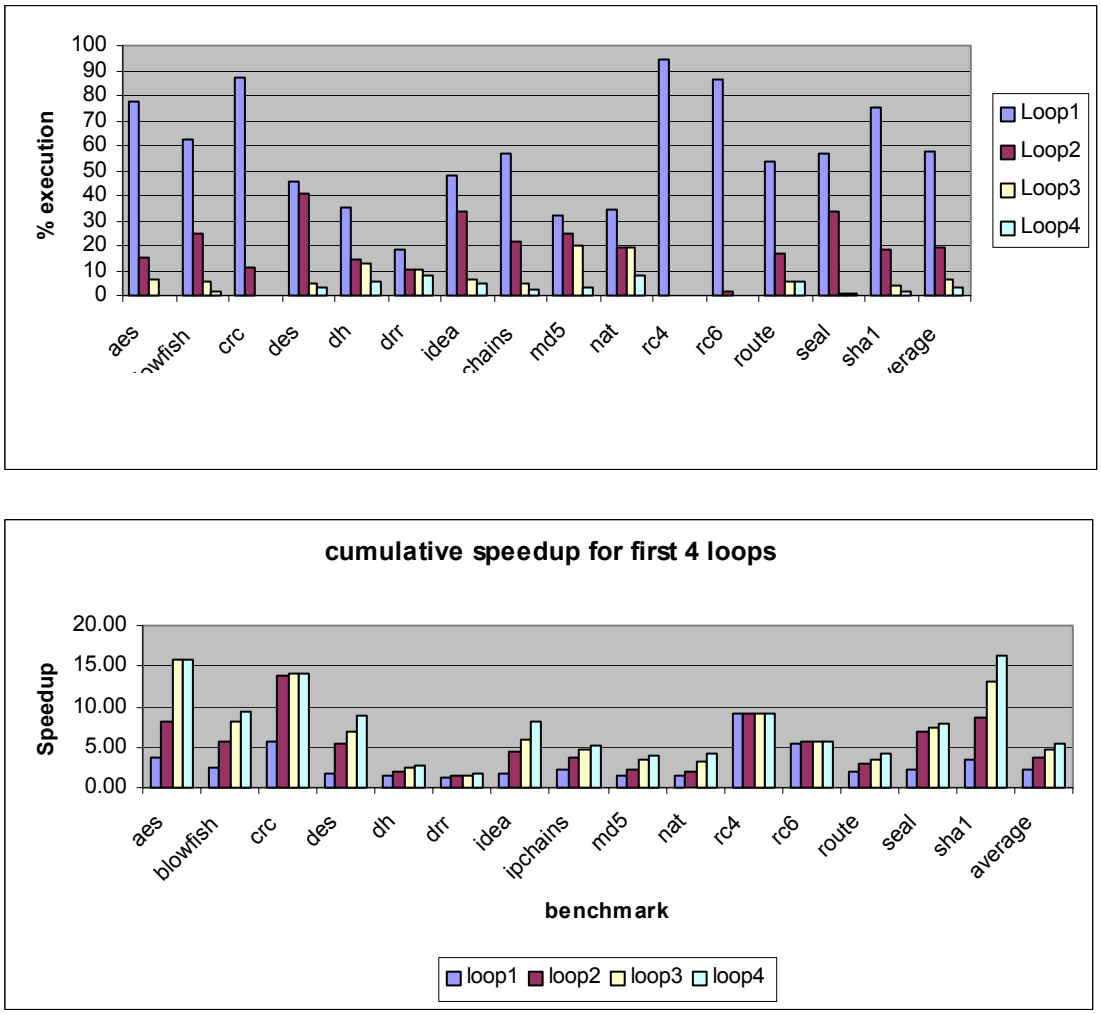


Figure 4. Cumulative speedups for the first 4 loops of each benchmark

However, our experience shows that the clock frequency that can be obtained on an FPGA is about 10 times lower than a CPU frequency.

For the remainder of this analysis we will assume that *HW_speedup = 1.7.*

The overall speedup S is the ratio of the SW_only time over the SoCP time.

$$S = \frac{SW\_only\ time}{(CPU\ time + FPGA\ time)} = \frac{SW\_only\ time}{SW\_only\ time - SW\_Loop\ time + \frac{SW\_Loop\ time}{1.7}}$$

$$S = \frac{1}{1 - \frac{SW\_Loop\ time}{SW\_only\ time} + \frac{1}{1.7}} = \frac{1}{1.588 - \%Loop\_execution}$$

Note that the speedup is a direct function of the loop execution time. Figure 4 shows the speed-up values for different applications. Applications like RC4, CRC, RC6, AES and sha1 have significantly higher speedups than the rest of the applications. For these applications, the first frequent loop/functions contributes more than 75% of the execution time. Roughly 90% of the execution time is contributed by the first two loops of security applications and the first 3 loops of the network applications. On an average, the first loop contributes about 58% of the total execution time while the first 2 frequent loops contribute nearly 78% of the total execution time.

In our current implementation, the loop names in the output of FLAT correspond to the high-level source code only when the program is not optimized. Possible future work includes extending FLAT to provide loop-level analysis for optimized applications.

VI Conclusion

In this paper, we perform loop-level analysis of a few popular cryptographic and network applications. We propose a toolset that identifies the time consuming portions of these programs at the granularity of loops as well as functions. We report the possible speed-ups that can be realized by mapping the first four frequent loops to hardware. Our results support a strong case for hardware/software co-design of these applications.

References

[1] J. Villarreal, D. Suresh, G. Stitt, W. Najjar, F. Vahid "Improving software performance through configurable logic", Design Automation for Embedded systems, November 2002

[2] L. Wu, C. Weaver and T. Austin, "Cryptomaniac: A Fast flexible architecture for secure communication", International Symposium on Computer Architecture, June 2001

[3] J. Burke, J. McDonald and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography", Proceedings of ASPLOS-IX, October 2000

[4] C. Young, "The Harvard Atom like Tool Manual", http://citeseer.nj.nec.com/121315.html

[5] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors", International Symposium on Micro architecture, 1997

[6] G. Memik, W.H. Smith, and W. Hu, "NetBench: A Benchmarking Suite for Network Processors", In Proceedings of International Conference on Computer-Aided Design *(ICCAD),* pp. 39-42*,* Nov. 2001, San Jose / CA

[7] *Altera Corporation*, "Altera Excalibur", http://www.altera.com/products/devices/excalibur/exc-index.html

[8] Xilinx Inc., "Xilinx virtex II pro handbook", http://www.xilinx.com/publications/products/v2pro/handbook/

[9] *Triscend corp*., "Triscend A-7 Chip", http://www.triscend.com/products/a7.htm

[10] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", ACM Transactions on Programming Languages and Systems, 1994

[11] *Simics Simulator.* http://www.simics.net

[12] *SpixTools.* http://www.sun.com/microelectronics/shade/

[13] R. Muth, S. Debray, S. Watterson and K. Bosschere, ALTO : A Link-Time Optimizer for the Compaq Alpha, *Software Practice and Experience*, Jan. 2001

[14] *Intel's Vtune,* http://www.intel.com/software/products/vtune/

[15] *Shade Kit,* http://www.sunlabs.com/techrep/1993/abstract-12.html

[16] *cacheprof.* http://www.cacheprof.org

[17] AES algorithm, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[18] IDEA encryption, http://www.finecrypt.net/idea.html.

[19] Blowfish Encryption, http://www.counterpane.com/blowfish.html

[20] DES Algorithm, http://www.aci.net/kalliste/des.htm

[21] H. Handschuh and H. Gilbert, " Cryptanalysis of the SEAL Encryption Algorithm", Fast Software Encrytion, vol. 1267 of Lecture Notes in Computer Science, pp. 1-12, Springer-Verlag, 1997

[22] Sha1 Algorithm, http://www.faqs.org/rfcs/rfc3174.html

[23] RC4 Algorithm, http://burtleburtle.net/bob/rand/isaac.html

[24] RC6 Algorithm, http://www.rsasecurity.com/rsalabs/rc6/