Cryptographic Instruction Set Processor Design

David Montgomery, Ali Akoglu

***Abstract*—Network processors utilizing general-purpose instruction-set architectures (ISA) limit network throughput due to latency incurred from cryptography and hashing applications (AES, DES, MD5, and SHA). This paper presents a methodology using computer-aided design, for development of high-performance application-specific instruction-set processor (ASIP) targeting applications saturated in repetitive sequential bitwise operations and data-flow dependencies, thus exposing both fine and coarse grain parallelism through a set of recurring pattern extraction tools. These specific instructions, in conjunction with a minimal set of general-purpose instructions, are then incorporated into a simplistic, single-cycle CPU architecture, for a comparison (in CPU cycles) with common general purpose instruction latencies (Intel P4). We show that the high-performance instruction set derived, based on the proposed methodology has the potential for facilitating dramatic improvements in performance (over the software kernel implementation) with substantially increased throughput. Results show that up to 93% of the code is utilized by the instructions derived through the developed toolset resulting with up to 2.7x cycle time improvement.**

***Index Terms*—Application Specific Instruction Set, Cryptography, Design Automation, Profiling**

I. Introduction

SECURE data processing (as in the case of IPSec) has always been a bottleneck towards achieving a high throughput for network processors since it involves encryption/decryption, hashing, and other operations. These operations are relatively compute-intensive when compared to the other functions of network processors such as forwarding of data packets. Today's network processors using general-purpose processors are a non-optimal solution for data integrity and security. Studies show that network routers spend a majority of their execution time performing check-sums and encryption on data packet. Since all the bits, including headers are encrypted; decryption and encryption must be done at every intermediate stage (router, gateway) of the network. This factor severely limits the throughput of a high-speed network such as Gigabit Ethernet; or rather IPSec acts as a bottleneck in sustaining the throughput of a high-speed network.

David Montgomery, Department of Electrical and Computer, Engineering, University of Arizona, Tucson, AZ 85721 USA, dtm@ece.arizona.edu
Ali Akoglu, Department of Electrical and Computer Engineering,University of Arizona, Tucson, AZ 85721 USA, akoglu@ece.arizona.edu

The redundancy of machine operations performed by network processors on an instruction-by-instruction basis fails to take advantage of the potential for parallel-code execution common for sequential bitwise operations. Hence, there is a need to streamline the design process for development of high performance and application-specific architectures such as network processors that perform secure data processing. Such a system needs to maintain a trade-off between specialization, performance and flexibility (characteristics required for IPSec), which requires an architecture whose design space falls in between the application-specific (performance oriented) and general purpose (flexible) systems.

Application-specific instruction-set processors (ASIPs)[1][2][12][13] allow designers to tailor the microprocessor by adding custom parallelized instructions and execution units within the processor, effectively increasing application performance. These ASIPs are usually designed by taking a general processor core and adding customized instructions specific to the application. The hardware implementing these extended instructions can be runtime reconfigurable units as in Chimaera [3] and Garp [4] or pre-synthesized circuits as in Xtensa [5] from Tensilica and Nios from Altera [6]. However, such extensions are similar to VLIW or SIMD approaches and are still limited to the data access through the processor's register file. This can severely limit the amount of parallelism potentially available to be exploited. Therefore, while these approaches provide significant advances in processor technologies, they are still far away from taking advantage of inherently parallel nature of the most embedded applications. We approach this challenge by identifying all possible levels of parallelism in the target application through extraction of recurring computation patterns at instruction level (fine grain) and at task level within the control flow structure (coarse grain). This is similar to the approaches taken by [7][8][9], and differs in that we are reverse engineering some of the explicit parallelism directives inserted into the code by the programmer.

First, a front-end compiler is used to retarget the application code into tri-operand intermediate representation code (IR-C). Pure data dependent sequential instructions form basic blocks of the control data flow graph in this representation. Then we expose instruction level parallelism within core basic blocks and coarse grain parallelism among basic blocks based on our novel two-phase sequence discovery tool. First the Sequence Discovery algorithm extracts repeating computation patterns (instruction sequences of any length) at the intermediate representation level. Then the Data Flow Discovery algorithm searches the set of dataflow graphs to recognize repeating

graph and sub-graph patterns. Finally the Instruction Generation step analyzes data-flow graph patterns against the patterns of sequence discovery step using run time profiling, data dependency within the repeating computation patterns and the executable cycle times of each in order to determine the set of high-performance custom instructions.

II. PROFILING METHODOLOGY

The design flow used for identifying new instructions (partial ASIP design flow) is shown in Fig. 1. The AES, DES, MD5 and SHA libraries were compiled into intermediate representation (IR) C-code via the Lance Retargetable C Compiler [10]. This provides fine-grained code and instrumentation for analyzing the application at higher than assembly level code. The Lance compiler generates directed control-flow graphs containing basic blocks of pure data dependent instruction sequences. We then expose the instruction level parallelism present within core basic blocks and coarse grain parallelism between the basic blocks by extracting recurring computation patterns. The parsing tool developed is designed to extract the computation patterns (operation sequences of any length) at the IR-C level. Determination of the optimal instruction-set, and CPU architecture to utilize is made via profiling analysis of the current version of OpenSSl (0.9.8c). After optimizing the code with the provided Lance optimization tools (IRopt), and applying further optimization tools developed by us (see figure 1.), all library test programs were compiled with the GNU GCC compiler to ensure an error-free code base. For example, the test program for the SHA hashing library is run via hashing each file within a file system that contains over 200 files of varying size. These are hashed and check-summed for errors to ensure an accurate compilation. Run-time profiling information is essential in order to identify the frequency of occurrence for each instruction sequence. This is then provided to the parsing tool in the form of an annotated IR-C code generated using GNU profiling tool GCOV. The parsing tool can take multiple files (annotated IR-C code) as input, and discover common IR-C sequences within the given code base. An IR-C parsing program was developed in C++ to discover and rank the most common IR-C operations that occur for each library. This program takes as parameters, the IR-C code generated by Lance and the profiling data of *GCOV*. It accurately returns the most frequently occurring IR-C patterns based upon estimated operation sequence latency (from machine instructions), and frequency of occurrence for the given sequence. The parsing program takes into account the latency in cycles of a given operation [11]. The latency of these operations were determined in 3 steps:

(1) Compile Lance IR-C code with GNU GCC,
(2) Perform an object-dump on the executable generated in the previous step via the GNU tool OBJDUMP form the BINUTILS package. This dump was annotated with the IR source-code as to make evident the corresponding ISA-32 Intel instructions.

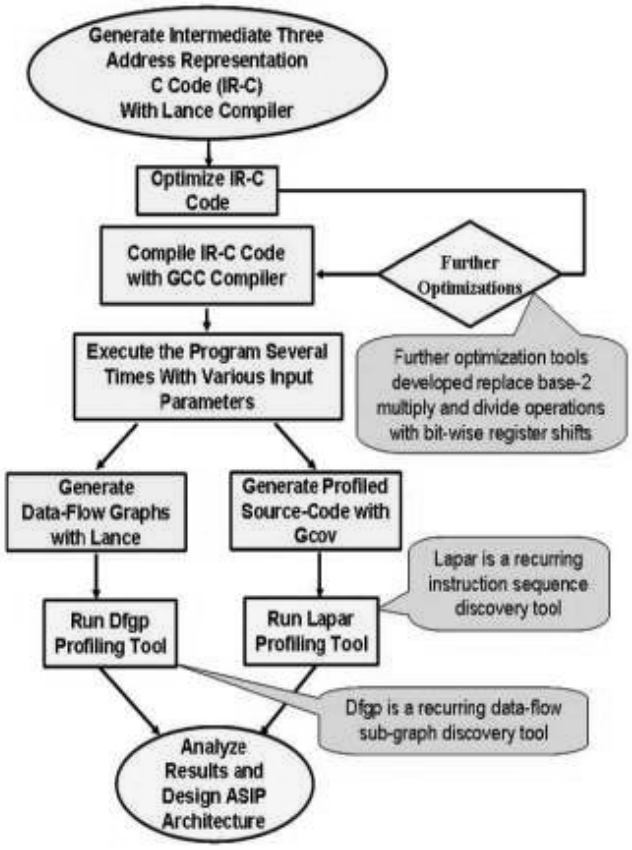

Fig. 1. Methodology utilizing recurring computation extraction toolset

(3) Determine the IR operation latency (in CPU cycles) using Intel Pentium-4 Processor Optimization Reference Manual

III. INSTRUCTION SET DEVELOPMENT

Our strategy in development an instruction-set specific to cryptography and hashing applications is to recognize reoccurring patterns of IR-C operations within the IR-C code base. To accomplish this task efficiently and accurately, a set of repeating pattern recognition algorithms were developed and implemented in the C++ programming language.

The Sequence discovery algorithm recognizes IR-C operation sequences of length 'n' within the scope of a function and further the source-code file for additional sequences that match this pattern. The program further searches the remainder of the code-base for additional occurrences of these sequences. The output of this program (see figure 2) contains the ordered-list of IR-C sequences discovered and ranked in descending order by executable cycle-time.

The determination of the sequence's cycle-time is computed by the multiplication of the number of occurrences by the cycle-time of each sequence (see equation 2). Sequences at the top of this list were then targeted for possible consolidation into single high-performance custom instructions for inclusion in our ASIP design architecture.

The Data-flow sub-graph discovery algorithm extracts repeating data-flow sub-graphs within a family of graphs generated by the Lance data-flow graph generation tool (showdfg). The data-flow graphs represent parallelism

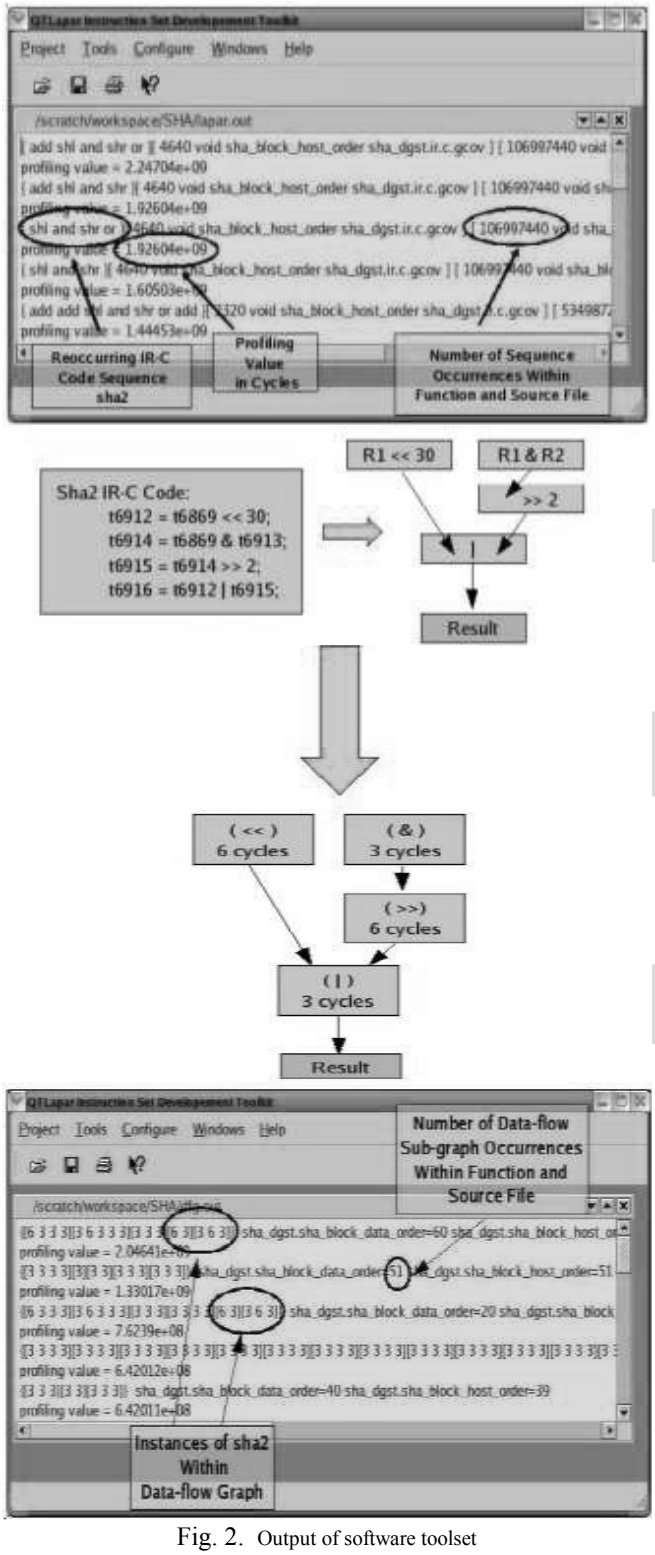

Fig. 2. Output of software toolset

occurring within the program, and are tree-structured by data-dependency. The algorithm, just as in the one for sequence discovery, searches the set of data-flow graphs to recognize repeating graph and sub-graph patterns. The implementation of this algorithm in C++ had an additional feature of pattern recognition by operation type (i.e. arithmetic, bitwise, data-movement), and thus recognizes the pattern based on the configured cycle latency of the sequence.

TABLE I
RECURRING INSTRUCTION SEQUENCE S AND THEIR PROFILING

Proposed Instruction	Operation Sequence	Profiling Value (CPU cycles)
sha1	{ add add shl and shr or add }	1.44453e+09
sha2	*{ shl and shr or }	1.92604e+09
sha3	{ and or and or add }	2.00629e+08
sha4	{ xor and xor add }	1.60503e+08
sha5	{ xor xor xor }	3.85205e+08
sha6	*{ add add }	3.21006e+08
sha7	{ xor xor add }	6.07904e+08
md1	{ xor add shl and shr or add }	1.15562e+09
md2	{ add xor and }	1.92604e+08
md3	**{ add xor }	1.92604e+08
md4	{ add add xor }	9.0283e+07
md5	*{ add add }	3.85205e+08
aes1	*{ and shl add eq xor }	9.60697e+06
aes2	{ shr and shl add eq xor }	4.80166e+06
aes3	{ shr shl add eq }	3.38528e+06
aes4	**{ add eq xor }	6.11709e+06
aes5	{ add eq shr eq eq }	860640
aes6	{ add eq add eqe eq got }	443632
des1	**{ add eq xor }	756736
des2	{ eq and eq add eq xor }	570240
des3	{ shr shl add shr shr }	304640
des4	*{ eq and add eq xor }	494208
des5	{ shr eq eq eq and add eq xor }	200192
des6	{ shr xor eq and xor shl xor }	101816
des7	{ shr shl add and }	126792
des8	{ add shr and shr and or shl add eq xor }	369664
des9	{ and shl add eq }	102144

***Note that this is a subset of another sequence**

This made the program more powerful in being able to recognize patterns with some ambiguities that could be removed by control logic within the CPU architecture and allow for a more resource efficient and flexible CPU design. When a repeating pattern is found, its executable cycle time is computed and multiplied by the numbers of occurrences discovered, and ranked in descending order. Data-flow graph patterns at the top of this list were then analyzed against the sequence discoveries performed prior to data-flow graph generation to determine the optimal ASIP design.

An important discovery of the data-flow sub-graph pattern recognition tool is the potential for higher levels of parallelism within our newly created ASIP design. Many of the custom instructions developed for the cryptography and hashing applications are non data dependent sequences of instructions, and thus can be run in parallel thus reducing execution time much further. Future work extending from this paper will be in exploitation of this potential, and will show increased throughput for the quantity of data encrypted, decrypted, and check summed.

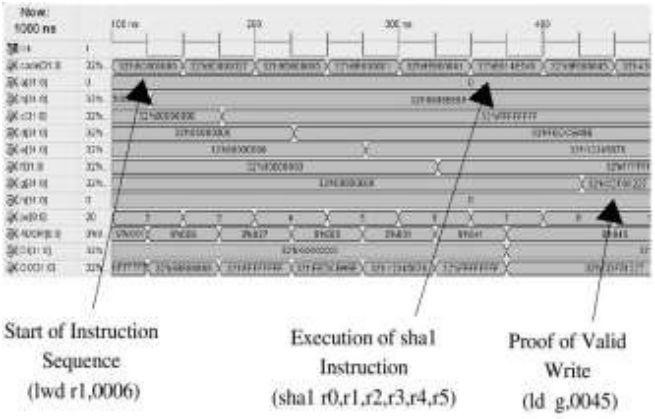

Diagram 1. Sha1 Instruction (6-cycles)

Fig. 3. Timing Diagram for sha1 ASIP Instruction

TABLE II
AVERAGE INSTRUCTION CYCLE LATENCY FOR PENTIUM 4 ARCHITECTURE

Machine Instruction	mov/ movb	add/ sub	and/ or/xor	shl/ shr	jmp/ bra	mul	div
Latency (cycles)	1	1	1	4	1	18	70

* Based on ***Intel Pentium-4 Processor Optimization Reference Manual***

TABLE III
IR-C SHA1 SEQUENCE WITH ATT-STYLE MACHINE-CODE AND LATENCIES

IR-C Operation	Machine-Language Instructions (ATT)	Latency (cycles)

t54 = t34 + t53;	mov 0xffffe208(%ebp),%eax
	add 0xffffffd0(%ebp),%eax
	mov %eax,0xffffe264(%ebp)
		3

t40 = t52 << 5;	mov 0xffffe170(%ebp),%eax
	shl $0x5,%eax
	mov %eax,0xffffe26c(%ebp)
		6

t42 = t52 & t41;	mov 0xffffe270(%ebp),%eax
	and 0xffffe170(%ebp),%eax
	mov %eax,0xffffe274(%ebp)
		3

After determination and analysis of the most common sequences and data-dependencies of the operations recognized via profiling, proposed machine instructions were then developed specific to each software library. These specific instructions, in collusion with many general-purpose instructions, were then designed within a single-cycle CPU architecture using the *Verilog* programming language. The programming and subsequent timing analysis was performed on the *Xilinx ISE Environment* to verify the instruction latency and accuracy. *Block-Ram* modules have been implemented to store the instruction code and static memory allocations made available to the processor. The ASIP machine code was generated by a special compiler, developed specifically for our purposes. This compiler parses for and recognizes the predetermined IR-C operations, and generates the corresponding library-specific instructions in the form of machine code. Other instructions not recognized within the constraints previously described are then compiled as general-purpose instructions (RISC) and subsequent machine code is generated.

TABLE IV
SPEED-UP USING PROPOSED ASIP SHA-INSTRUCTIONS

Proposed ASIP Instruction	Percentage of Occurrence (%)	ASIP Cycles	(Intel P4) Cycles	Speed-up Ratio
sha1	34.6644	6	27	4.5
sha2	23.4096	3	18	6
sha3	4.8145	5	15	3
sha4	3.8516	5	12	2.4
sha5	9.2438	5	9	1.8
sha6	NA	-	NA	-
sha7	14.5879	5	9	1.8
Total Code Utilization for SHA = 90.6		**Total Speedup for SHA = 2.66791**		
md1	63.4423	6	27	4.5
md2	10.5737	6	9	1.5
md3	.02203	5	6	1.2
md4	4.9564	6	9	1.5
md5	13.878	4	6	1.5
Total Code Utilization for MD5 = 92.9		**Total Speedup for MD5 = 2.44998**		
aes1	26.2752	6	18	3
aes2	20.8264	7	26	3.71429
aes3	14.6831	3	17	5.66667
aes4	10.1487	4	9	2.25
aes5	3.7329	4	16	4
aes6	1.9241	4	18	4.5
Total Code Utilization for AES = 77.6		**Total Speedup AES = .21055**		
des1	4.546	4	9	2.25
des2	17.4587	4	16	2.66667
des3	7.8564	5	27	5.4
des4	11.6666	4	14	3.5
des5	6.1292	5	24	4.8
des6	3.1172	5	23	4.6
des7	3.8819	2	17	8.5
des8	11.3178	9	41	4.55556
des9	3.1273	4	14	3.5
Total Code Utilization for DES = 69.7		**Total Speedup for DES = .99825**		

The data shown in table 1 represents the recurring operation sequences discovered within the IR-C code via the parsing tools. The profiling values were derived from the corresponding annotated object dump performed on the given binary executable file, and calculated using equation 2.

IV. PERFORMANCE ANALYSIS

The results of the timing analysis (fig. 3) performed for all application-specific instructions were then compared to the profiling data generated. A ratio of application specific to general-purpose cycle times was calculated by hand for each IR-C operation sequence where a corresponding special-purpose instruction had been developed. This showed us the potential increase in execution time for the given sequence.

Table 3 displays a portion of the IR-C code, the corresponding machine-code instructions (ATT), and the calculated instruction latency for the given sequence (see table 1). This type of calculation was performed on each and every proposed ASIP instruction and provides the base for the calculations for the "percentage of occurrence" column in table 1. Using our profiling tool, the total number of cycles for the entire program execution was calculated to be 4.16719e+09 cycles. Values for the proposed instruction sha6 were omitted from this table because it was determined that this sequence was a subset of sequence sha1, and had no other occurrences within the program code, unlike sha2. Analysis of the data from table 1 shows that the total percentage of code utilized by the proposed ASIP instructions for the SHA library is 90.6%, 92.9%, 77.6%, and 69.7% for MD5, SHA, AES and for DES respectively.

The processor-architecture used for testing our ASIP design is a simple 'single-cycle' design and provides a strong basis for the proof or our concept. Simulations were performed with processor clock speeds of up-to 500Mhz and restricted only by the limitations of the Xilinx ISE simulator. Using the values computed from tables 1, 4, 5, and applying the data to equation 5 (Amdahl's law), total performance improvements (speedup) of 2.66791, 2.44998, 2.21, and 1.99 were achieved for SHA, MD, AES and DES respectively (see table 4).

V. CALCULATION

The total number of cycles for the IR-C code was computed with the parsing tool developed. As each line of code is parsed, the IR-C operation is uniquely identified and mapped to its respective cycle latency within the profiling tool's configuration file. The profiling values for the individual sequences were computed and represented in table 1, by mapping IR-C operations found within the sequence to the corresponding cycle values.

$$T=\sum_{i=1}^{n} X_i \rightarrow Y$$

n = total lines of IR-C code
X_i ϵ {+, -, <<, ... , goto, =}
Y ϵ {3, 3, 6, ... 4, 2}

(1) Total IR-C Code Cycles

The values for percentage of occurrence present within table 4 were calculated with the equation 3 below using values from equations 1 and 2

$$PV=G*\sum_{i=1}^{m} X_i \rightarrow Y$$

m = lines of IR-C code in operation sequence
X_i ϵ {+, -, <<, ... , goto, =}
Y ϵ {3, 3, 6, ... 4, 2}
G = number of run-time executions derived from gcov annotated profiling output.

(2) Profiling Value for IR-C Code Sequence Cycles

For sub-sequence operations, where one sequence occurs within a larger one, the ratio of occurrence needs to be calculated using set theory. In equation 3a, the relationship and the resulting calculation is shown.

$$PO=\frac{PV}{T}*100=P*100$$

PV = profiling value
T = total execution cycles
P = ratio of occurrence

(3) Percentage of Occurrence for IR-C Operation Sequences

Values for the speed-up ratios shown in table 5 were calculated by in equation 4. The IR-C code sequence latency values used by the general-purpose processor were derived from the values shown in table 3.

$$P=\frac{A_2-A_1*\frac{A_1^*}{A_2^*}}{T}$$

A_1 = Super-Set Sequence (cycles)
A_2 = Sub-Set Sequence (cycles)
A^* = Individual Sequence length (cyc/seq)
T = Total Executable Cycles (cycles)

(3a) Ratio of Occurrence for IR-C Operation Sub-Sequences

$$R=\frac{\sum_{j=1}^{n} L_i}{S}$$

n = number of IR-C operations
L_i = sequence latency
S = number of ASIP cycles

(4) Speed-Up Ratio for IR-C Operation Sequences

VI. CONCLUSION

Our toolset for pattern recognition used to develop this ASIP design has shown to be reliable and accurate in targeting the portions of an application with the greatest potential for performance increase. The results of this research conclusively show that computer-aided pattern analysis and profiling of applications provide an efficient method of developing application-specific architectures. Although our methodology has proven successful for cryptography and hashing applications, almost any application chosen for independent processing can be targeted using our toolset. The overall performance increase of the ASIP processor of that of the

general-purpose processor was determined using Amdahl's law (equation 5) using the values from tables 4 and 5 with equations 3 and 4.

$$Speedup = \frac{1}{1 - \sum_{i=1}^{n} P_i + \sum_{i=1}^{n} \frac{P_i}{R_i}}$$

(5) Amdahl's Law

The potential for performance improvement for the applications chosen in this work are not by any means limited to our results. The single-cycle platform was chosen for simplicity and to provide proof-of-concept for our toolset and methodology. Further performance increases extending from this work can easily be achieved via implementation of a multi-cycle CPU architecture with instruction pipelining, replication of ALU's to exploit the potential for parallel-code processing of non-data dependent ASIP instructions, and block-memory transfers specific for DES and AES applications

The toolset for pattern recognition used to develop our ASIP design has in addition to saving many hours of hand-analysis, shown to be reliable and accurate in targeting the portions of an application with the most potential for performance increase. The benefits of this toolset are by no means limited to ASIP design, but also provide a foundation for processing elements (PE's) for ASIC designs as well. Because the code-base of applications changes rather frequently, our application profiling toolset provides an efficient mechanism for keeping the architecture current with the latest software releases. An example would be the open source cryptography and hashing libraries used in this work, OpenSSL has released 2 revisions (0.9.8c-0.9.8e) in less than 6 months, and the importance of keeping a network CPU running IPSec related applications up-to date cannot be understated.

In the future, we expect that many other applications saturated with bitwise operations such as multimedia processing, streaming multimedia, and a host of scientific applications will be targeted for ASIP architectures. Because many of these applications contain large code bases, our robust toolset can realize the optimal instruction-set for a given application saving many hours of hand analysis.

REFERENCES

[1] Tensilica, Inc. http://www.tensilica.com, 2006.
[2] Stretch, Inc. http://www.stretchinc.com/, 2006
[3] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit," in Proceedings of Annual IEEESymposium on Field-Programmable Custom Computing Machines, pp. 87-96, Apr. 1997.
[4] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," IEEE Computer, vol. 33(4), pp. 62-69, Apr. 2000.
[5] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," IEEE Micro, vol. 20(2), pp. 60-70, Mar. 2000.
[6] Altera Corp., http://www.altera.com.
[7] B. So, H. Ziegler and M.W. Hall, "A Compiler Approach for Custom Data Layout", Proceedings of the Languages and Compilers for Parallel Computing Workshop, Jul. 2002
[8] S. Moon, B. So and M. Hall, "Evaluating Automatic Parallelization in SUIF", IEEE Transactions on Parallel Distributed Systems, 11(1) (Jan. 2000)
[9] S. Moon and M. Hall , "Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization", Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, Atlanta, Georgia, May, 1999
[10] http://www.icd.de/es/lance/lance.html
[11] Intel Corporation, "Intel Pentium-4 Processor Optimization Reference Manual.," http://developer.intel.com, 1999-2001, pp. C1–C16.
[12] J. Kong et al, "Application-Specific Instruction Generation for Configurable Processor Architectures", Proc. Intl'l Symp. Field Programmable Gate Arrays, 2004, pp.183-189.
[13] R. Kastner et al, "Instruction Generation for Hybrid Reconfigurable Systems", ACM Transactions on Design Automation of Electronic Systems, vol 7, Oct. 2002, pp. 605- 62