Accelerating Cryptographic Applications Using Dynamically Reconfigurable Functional Units

Trouvé, Antoine
Department of Informatics, ISEE, Kyushu University

Gauthier, Lovic
Department of Informatics, ISEE, Kyushu University

Kando, Takayuki
Department of Informatics, ISEE, Kyushu University

Ryder, Benoît

他

Accelerating Cryptographic Applications Using Dynamically Reconfigurable Functional Units

Antoine Trouvé [†‡], Lovic Gauthier[†], Takayuki Kando[†‡], Benoît Ryder[‡], Sébastien Pouzols[‡],
Pradeep Rao[‡], Norifumi Yoshimatsu[†‡] and Kazuaki Murakami[†‡]
[†]*Department of Informatics, ISEE, Kyushu University, Japan.*
[‡]*Institute of Systems, Information Technologies and Nanotechnologies, Japan.*

***Abstract*—In this paper we propose and evaluate our platform to accelerate applications using custom instruction set extensions. We use a dynamically reconfigurable functional unit (DRFU) to execute the application specific custom instructions generated by our compiler framework. We explore two architectures with different computational granularities for the DRFU (look-up table and ALU based) and evaluate this framework using security and cryptographic applications as a case study. Our results indicate that the use of application specific instruction set extensions reduce code size by 10% and achieve a maximum speedup of 165% (41% on average).**

***Keywords*-Dynamic reconfiguration, accelerator, compiler**

I. Introduction

While application specific instruction set processors have demonstrated performance benefits over general purpose processors for a given application domain, they are associated with inherent risks. For instance, the constant evolution of software and low processor production volume requirements may not always justify the cost of introducing new instructions in the ISA. These concerns can be addressed using reconfigurable hardware which can evolve with the software and can be tuned to the application on demand [1]. Furthermore, increasing on-chip resources provide the means to achieve these objectives.

Our approach to the use of reconfigurable hardware for application acceleration uses a dynamically reconfigurable functional unit (DRFU). The DRFU is tightly coupled and integrated close to the processor core to reduce communication overheads [2]. This is in contrast with loosely coupled, reconfigurable application accelerators [1]. The DRFU is capable of executing application specific instruction extensions referred to as custom instructions (CI) in this paper. The CIs are automatically generated for each application (in C) using our compiler toolchain.

We introduce and evaluate our framework to automatically generate application specific instruction extensions and the architectural support required to support these using DRFUs. As a case study we consider several cryptographic applications for our experiments and we explore and evaluate two DRFU architectures – one based on hardware look-up tables (LUT) and the other based on arithmetic logic units (ALU).

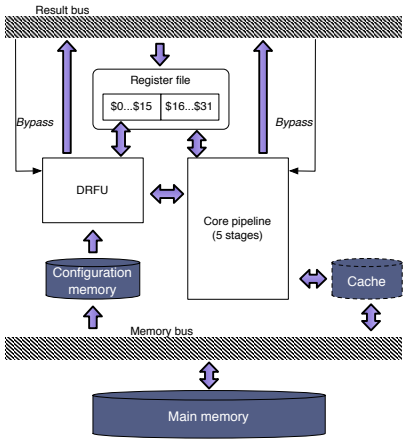

Figure 1. Architecture Overview

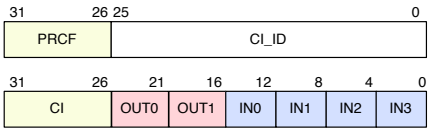

Figure 2. Additional instructions to support the DRFU

II. Hardware

A. Processor Architecture

The proposed platform (Fig. 1) for application acceleration consists of a core RISC processor with an integrated DRFU and a fully automatic toolchain used to generate CIs to exploit the DRFU. The core processor is a simple 5 stage pipelined processor and executes instructions in order. The processor instruction set architecture resembles the 32bit MIPS ISA. The processor implements two additional instructions to support the DRFU as shown in Fig. 2.

The `pcrf` instruction configures the DRFU using the configuration information stored in the configuration memory indexed using `CI_ID`. The `ci` instruction corresponds to the execution of the generated custom instruction on the DRFU with `IN0..IN3` as input registers and `OUT0,OUT1` as the output registers.

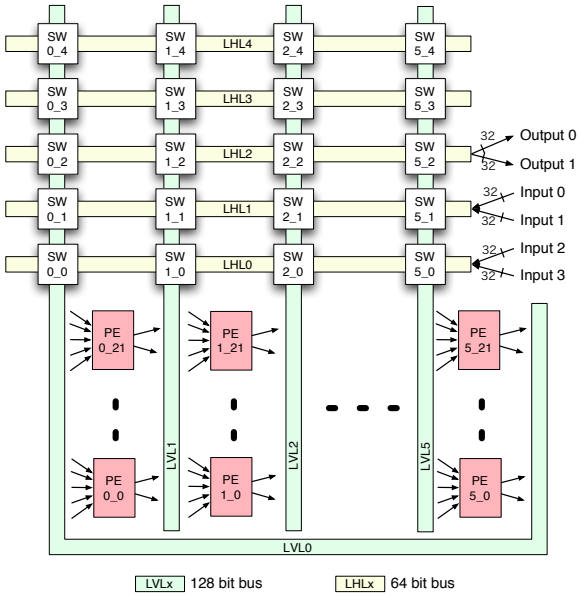

Figure 3. The DRFU-LUT

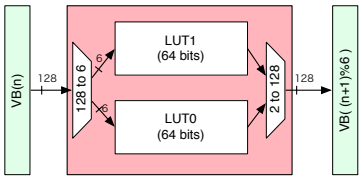

Figure 4. A processing element (PE) in the DRFU (LUT based)

B. DRFU

We consider two DRFU architectures – one based on hardware look-up tables (LUT) and the other based on arithmetic logic units (ALU). The DRFU-LUT contains 132 processing elements (PE) organized in six rows as shown in Fig. 3. The PE itself is implemented as a simple 6:2 lookup table (LUT) as shown in Fig 4.

The PEs communicate with each other using either the 128bit vertical bus (VB) or the 64bit horizontal bus (HB). Inter PE communication using the HB involve slower switches (SW) in the communication paths. The switch fabric is essentially implemented using diodes to connect the horizontal and vertical wires. The state of these diodes is a part of the configuration. We synthesized the DRFU using a 90nm technology and estimated the inter-PE propagation delay when using the VB and HB to be 1.15ns and 5ns respectively.

Each DRFU configuration corresponds to a CI and requires 4033bytes (<4kB) each and the entire set of CIs are stored in the configuration memory (Fig. 1). The memory required for each CI configuration is much lower than the hundreds of kilobytes of configuration required for typical

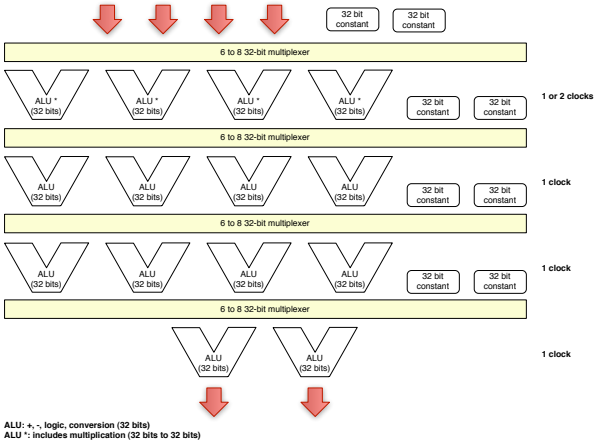

Figure 5. The DRFU-ALU

FPGAs. Large configurations prevent dynamic reconfiguration due to the large reconfiguration time required [2]. Our RTL level synthesis and simulation results show that it is possible to reconfigure the DRFU using the configuration stored in a 64 entry configuration memory in less than 10ns, thus enabling single cycle reconfiguration on a processor with a 100MHz clock.

The DRFU-ALU is organized as shown in Fig. 5 and consists of 32bit ALUs and can accommodate four inputs and two outputs (32bit each). The DRFU -ALU is fully pipelined as indicated and features 8x32bit registers to accommodate constants. We note that this work does not aim at introducing new architectures, but at comparing the efficiency of both ALU and LUT based approaches with our development flow. In consequence, both the DRFUs are over-provisioned to ease the place and route step in the compilation flow (Sec. III) and the resource usage results in Sec. V-B indeed indicate that the resources required are far lower than those provided.

In summary, we note that exploiting custom instructions requires changes to the instruction decoder in the processor pipeline as well as an additional configuration memory as indicated in Fig. 1. The register file also needs to accommodate the additional ports corresponding to the I/O ports of the DRFU. However, our experiments indicate that the performance benefit of additional I/O ports levels off at 4 input and 2 output ports for either DRFUs.

III. Compilation Flow and CI generation

Since most high-level programming languages do not support complex bit manipulation operations, either (1) the compiler has to recognize it from classic Boolean functions or (2) it should be programmed using assembly language. If (2) is not satisfactory, (1) is a NP-problem that a typical compiler cannot afford.

The solution presented in this paper addresses (2) by

providing a development flow from C language, and (1) with ISAcc, which solves a far larger problem of bit permutation recognition within a reasonable time (from 1 to 10 hours in the case of DES, depending of the implementation of the algorithm). We implement a fully automatic retargetable compiler to exploit the DRFU using CIs and is implemented in the COINS compiler framework [3]. The compilation flow is shown in Fig. 6 and takes the C program as input and automatically generates the assembly code as well as the CI configuration to be loaded into the configuration memory.

The compiler front end, parses the C program, applies several optimizations and finally outputs a dependence graph. CIs are formed by assembling hardware components that implement subsets of this dependence graph. Specifically, we implement the following stages.

The *annotater* identifies and evaluates hardware templates that can be used to implement subgraphs of the dependence graph. Each hardware template corresponds to a pattern available in the pattern library. A first pass does a pattern-match on the graph. When a match occurs the corresponding vertices in the graph are annotated with a reference to the instance of the hardware template. The second pass evaluates delay and size of for each instance of hardware template that implements a sub-graph.

The *CI partitioner* partitions the dependence graph into CI subgraphs and non-CI subgraphs to facilitate the generation of CI. The partitioning is guided by a cost function computed from the evaluations performed in the previous stage by the annotater.

The *synthesizer* creates the configuration for each CI from its corresponding subgraph. A first pass selects the template to implement the CI from among the hardware templates that cover the subgraph of the CI. In the case of identical CI subgraphs, only one configuration is generated. The selected hardware templates are then instantiated and assembled to form a netlist describing the unmapped hardware for the CI. This netlist is then optimized and mapped onto the target DRFU to produce the resulting configuration. During mapping, we use heuristics to determine a good initial solution and our technique is an adaptation of the algorithm proposed in [4] for routing. A failure at this stage updates the placement using simulated annealing. While this technique applied to FPGA is known to be slow, it has not been an issue with the small size of our DRFU. Finally, a traditional backend generates the assembly code.

Further details on each of the compiler stages, algorithms used and the optimizations applied in each stage is described elsewhere [5] to remain within the scope of this paper. The compiler build and test status will be available online at *http://ngarch.isit.or.jp*

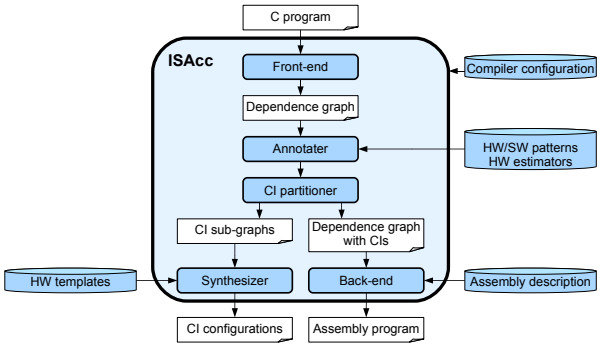

Figure 6. Compilation Flow

IV. Methodology

A. Workload

As a case study, our workload consists of commonly used checksum (CRC32, MD5 and SHA-1) algorithms as well as block ciphers (DES and AES). We also investigate the effectiveness of our approach on highly optimized code. This is evaluated by optimizing the source code to employ the following: (1) *Packing:* This refers to combining sub-word operations and accessing memory at the word level rather than bytes. (2) *Constant Integration:* The DRFU is able to include constants directly inside the configuration, thus reducing register pressure and the total number of instructions. Integrating constant tables in the DRFU also reduces memory accesses thus speeding up the application. For this purpose the LUT in the DRFU-LUT doubles up as memory with 6bit address (64 entries) while the DRFU-ALU integrates 8x32bit registers. (3) *Loop Unrolling/Rerolling:* Unrolling loops helps the algorithm identify larger CIs, resulting in program speedup. However, unrolling may result in increased register pressure leading to an increase in memory accesses.

Brief notes on each algorithm and the nature of optimizations employed are described next.

CRC32: CRC-32 is used as a tool to detect data corruption on storage or transmission. The common way of computing the CRC-32 in software uses a 8-bit pre-generated table, and almost boils down to a XOR of the accumulator with a value from this table depending on the next byte of data. The optimized version used a reduced 4-bit precomputed table (16x32bit words) which can fit into the DRFU. It also added a 32-bit packed read operation, which implied loop unrolling.

MD5: The MD5 algorithm operates on 64-byte data blocks to produce 128-bit hashes and does not use a substitution box [1]. The optimized version integrates the sine derived constant table in CIs by using C defines. The reference

[1]Substitution box (S-box) is a function associating an m-bit input to an n-bit output and are carefully chosen to resist known attacks. They are often implemented using table lookups.

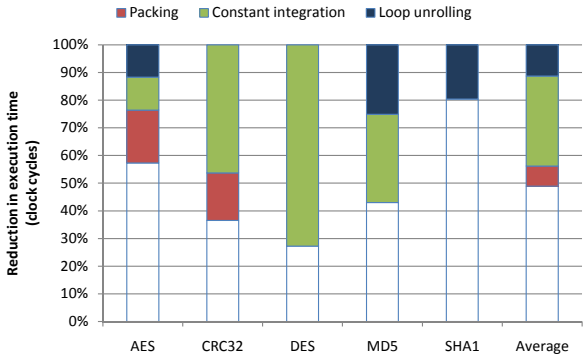

Figure 7. Iterative speedup of programs compiled with CIs due to source-level optimizations.

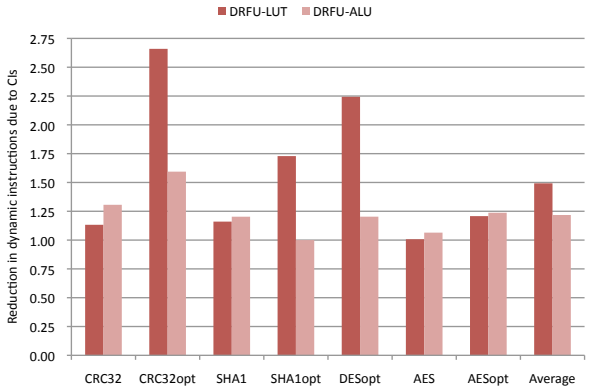

Figure 8. Reduction in the number of instructions executed

version used an array to store the values, which generated memory accesses and used one register.

SHA1: SHA1 is a popular cryptographic hashing function producing 160-bit hashes and is used in many security applications (IPSec, TLS, SSL, etc.). Like MD5, it operates on 64-byte blocks, features a message schedule (block expansion) and does not use any substitution box. The reference version implemented the first method described in the SHA-1 specification [6], computing the 80×32-bit message schedule at the beginning of each loop. The optimized version implements the second method, which computes the message schedule in-place between each operation.

DES: This paper uses the Triple DES implementation, which uses three original DES passes with two or three different keys. The algorithm consists of 16 applications of Feistel functions, each of them consisting of four steps: (1) key expansion, (2) key mixing (a XOR), (3) Substitution (using s-boxes) and (4) permutation. Implementation of such functions in a general purpose processor is highly inefficient due to the lack of fine-grain bit-manipulation instructions. For instance, a permutation of 32 bits requires around 128 instruction with a classic RISC ISA. The optimized version was obtained by enabling the constant table integration for the 8 substitution boxes of DES and the two bit-swap tables.

AES 128: This paper uses the 128bit key for the AES block cipher algorithm. The key is first expanded, then 16-byte blocks of data are ciphered using a substitution box, byte permutations and an XOR. The optimized version integrates the substitution boxes and packed XOR operations in the columns during key expansion and data ciphering, considering each column as a 4-byte integer. This packing reduces the number of instructions by a fourth. The optimized version also unrolled the operations occurring on the whole 4×4-byte state, which removed some instructions needed to process the loop.

The baseline speedup due to the various source level optimizations are shown in Fig. 7.

B. Processor Simulator

We have implemented a cycle accurate, execution driven instruction set simulator to drive performance evaluations. The simulator accepts the assembly code and the instruction description as input and enables the determination of the state of the DRFU and the processor pipeline at each clock cycle. The simulator assumes a simple memory model with each memory access requiring two cycles. This is a reasonable assumption for embedded processors driven at 100MHz and using SRAM memory.

V. Results

We first analyze the quality of the CIs generated and then evaluate the improvement in performance due to the use of custom instructions.

A. CI Generation and Application Speedup

We observe that the reduction in static code size averages 11% for the LUT based DRFU across the applications considered with the maximum code size reduction of 40% achieved for the source optimized version of `DES`. The static code size reduction for the ALU based DRFU averages 10% with the maximum reduction in code size of 22% for the optimized version of `CRC`.

Improvements to static code size reduces ROM requirements in embedded systems and consequently affects system costs. Reducing the dynamic number of instructions without increasing overall execution time is equally important as they impact the energy consumption in embedded systems and consequently impact battery life. The average reduction in the number of dynamic instructions, despite the addition of `prcf` and `ci` instructions required to support the DRFU, is shown in Fig. 8.

The dynamic number of instructions executed under each class of instruction (arithmetic, boolean, control, load/store and CI) is shown in Fig. 9. These plots indicate that the LUT based DRFU is better suited for cryptographic applications as they reduce the total instructions executed by 33% while

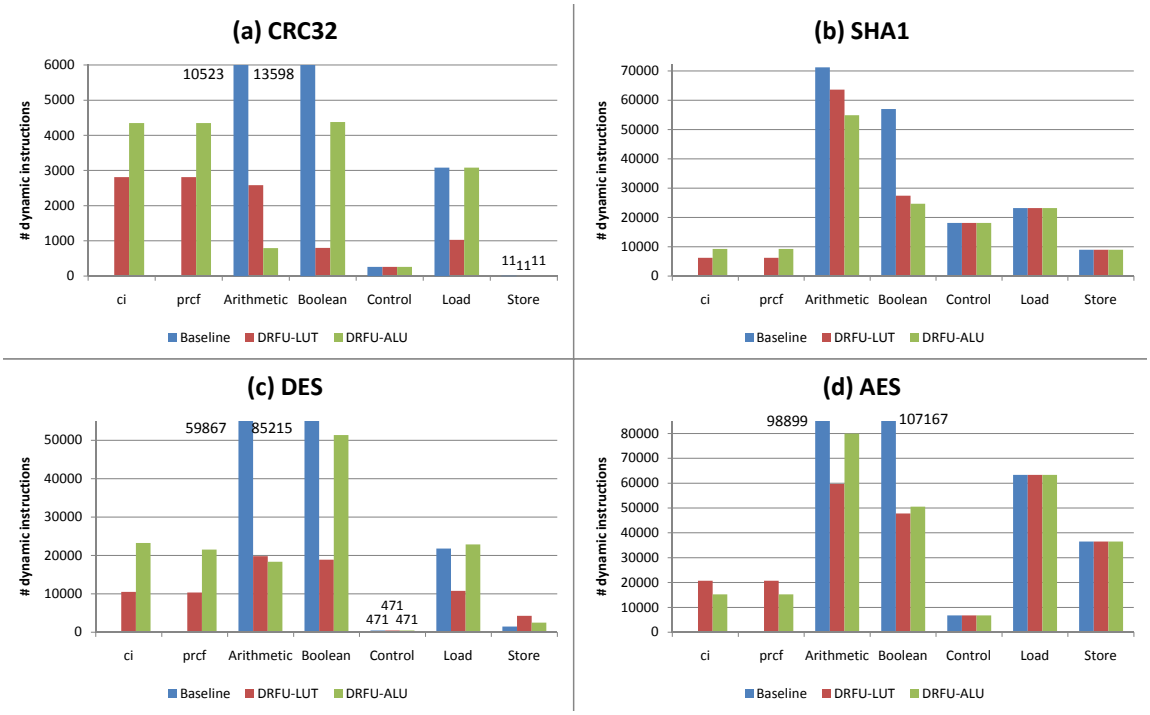

Figure 9. Reduction in the number of instructions executed classified by instruction type

achieving an average speedup of 1.41 as seen in the speedup plots in Fig. 10. This is expected since the DRFU-LUT is optimized for bit manipulation. Customizing boolean instructions is especially effective for CRC32 and DES as seen in Fig. 9 (a) and (c). The same plots also show an effective reduction in the number of `load` instructions and are attributed to hardcoding the lookup operations in the CI.

B. DRFU Resource Usage

Since we over-provision resources in the DRFU to ease place and route, we assess the use of resources in this section. Several hardware parameters may prevent code from being mapped onto the DRFU – (1) the number of PEs, since it is directly related the maximum size of the dataflow graph that can be mapped on the RDP, (2) communication between PEs, since the bus and switches may become saturated and restrict PE usage and (3) the number of input/output ports. If successive instructions depend on too many variables, the DRFU will not have enough I/Os to map all these instructions.

Fig. 11 shows the cumulative distribution of PEs usage against the number of CIs mapped on the DRFU-LUT for our workload. We observe that, on an average, over 90% of all CIs mapped use less than half the number of available PEs and only 1% of the the CIs require more than 60% of the PEs. Similar plots for the DRFU-ALU show that 85% of the CIs require less than 30% of the available PEs (ALU). Hence, it should be possible to reduce the

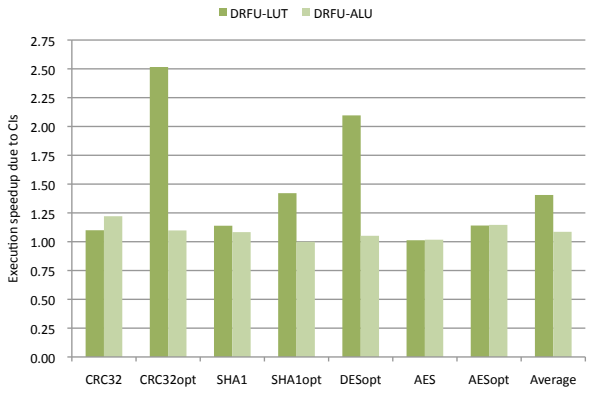

Figure 10. Speedup due to use of CI and DRFU

number of PEs without impacting the generation of CIs (and consequently without adversely affecting performance). These observations can be used to device mechanisms to split the resources into large virtual and few real resources. The virtual resources can then be used for place and route while the real resources only need to be implemented in hardware.

We monitored simulations to check if the low PE usage was due to communication bottlenecks. Our results (not shown here) indicate that that the bus usage never exceeds 58% with the average bus usage at 19%. We also ran experiments to examine the impact of DRFU IO ports on

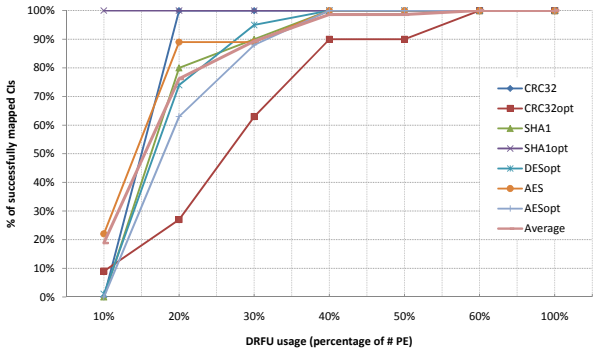

Figure 11. Cumulative distribution of DRFU usage (#PEs) against mapped CIs (as % of total CIs)

performance. We note that increasing the number of IO ports beyond 4 do not significantly increase PE usage. This observation is consistent with other studies [7] that arrive at similar conclusions regarding the impact of IO ports.

VI. Related Work

Several reconfigurable solutions exist [1], [8] and can be classified broadly as coarse-grained [9], [10], [11] or fine-grained [12], [13], [14] reconfigurable architectures. Since this study focuses on the DRFU architecture and its evaluation we defer the comparison of our compiler framework to CI synthesis studies to a separate paper [5].

Our work resembles fine-grained, instruction-level architectures [2] but seek to significantly improve the performance-energy ratio by addressing the DRFU complexity (for instance, we do away with large multi-ported shadow registers [15]). We also seek to improve the ability of the compiler to automatically find critical instructions that may be implemented as CIs directly from C code (as in [14], [16]), with minimal source level modifications.

Furthermore, we seek to explore and evaluate various design alternatives for the DRFU design to achieve the above stated objectives.

VII. Conclusions

We propose and describe a hardware/software framework to accelerate applications using application specific instruction set extensions. This paper evaluates our framework using cryptographic applications as a case study. We use a reconfigurable processor which includes a small fine-grained dynamically reconfigurable datapath in order to accelerate bitwise operations. This architecture is completely supported by a full software compilation flow, which automatically generates the DRFU configurations for the accelerator. Moreover, all CI synthesis tools and the processor simulator (with DRFU) were developed from scratch to acheive better integration of our toolset. This versatility enables the designer to quickly explore the DRFU design space for any given application domain.

We evaluate two designs for the DRFU with different computational granularities – based on LUTs and ALUs, to illustrate the variety of designs that can be explored within our framework. We show that our platform significantly improves code size (by 10%), performance (by 41%) and has significant potential to improve power since dynamic instruction count is reduced by 33%, all of which pose design constraints in embedded systems. As part of current research, we are investigating DRFU implementation overheads (die area) and looking at ways to improve DRFU utilization to reduce hardware costs.

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[2] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera reconfigurable functional unit," *IEEE Transactions on VLSI Systems*, vol. 12, no. 2, pp. 206–217, 2004.

[3] COINS, www.coins-project.org/international/index.html.

[4] A. Sharma, S. Hauck, and C. Ebeling, "Architecture-adaptive routability-driven placement for FPGAs," in *FPL*, 2005.

[5] Lovic Gauthier et.al., *Reconfigurable processor with near optimal custom instruction generation*. Technical Report, System LSI Laboratory, Kyushu University, 2008.

[6] *Official annoucment of sha1 by nist*, 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[7] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc., 2007.

[8] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007.

[9] J. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *FPGA*, 2001, pp. 141–150.

[10] *S6000*, http://www.stretchinc.com/products/s6000.php.

[11] IPFlex, *Architecture of dap/dna-2 processor*, http://www.ipflex.com/en/E1-products/dd2Arch.html.

[12] R. Razdan and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO*, 1994, pp. 172–180.

[13] M. Wirthlin and B. Hutchings, "A dynamic instruction set computer," in *IEEE FCCM*, 1995, pp. 99–107.

[14] Mei, B. et.al., "ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors," *Fine-and Coarse-Grain Reconfigurable Computing*, 2007.

[15] V. Zyuban and P. Kogge, "The energy complexity of register files," in *ISLPED*, 1998, pp. 305–310.

[16] Tensilica White Paper, "Rapid SOC Development Using Automatically Generated Processors."