A Design Flow for Configurable Embedded Processors based on Optimized Instruction Set Extension Synthesis

R. Leupers, K. Karuri, S. Kraemer, M. Pandey
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany

Abstract

Design tools for application specific instruction set processors (ASIPs) are an important discipline in system-level design for wireless communications and other embedded application areas. Some ASIPs are still designed completely from scratch to meet extreme efficiency demands. However, there is also a trend towards use of partially predefined, configurable RISC-like embedded processor cores that can be quickly tuned to given applications by means of instruction set extension (ISE) techniques. While the problem of optimized ISE synthesis has been studied well from a theoretical perspective, there are still few approaches to an overall HW/SW design flow for configurable cores that take all real-life constraints into account. In this paper, we therefore present a novel procedure for automated ISE synthesis that accommodates both user-specified and processor-specific constraints in a flexible way and that produces valid, optimized ISE solutions in short time. Driven by an advanced application C code analysis/profiling frontend, the ISE synthesis core algorithm is embedded into a complete design flow, where the backend is formed by a state-of-the-art industrial tool for processor configuration, ISE HW synthesis, and SW tool retargeting. The proposed design flow, including ISE synthesis, is demonstrated via several benchmarks for the MIPS CorExtend configurable RISC processor platform.

1. Introduction

In the past decade, research on embedded processor design has emphasized ASIPs [1] as efficient and flexible programmable platforms for demanding applications such as telecommunication protocols and multimedia codecs. With the traditional RTL design entry level, ASIP design has frequently been considered too tedious and risky under tight project schedules. However, recent breakthroughs in *architecture description languages* (ADLs) [2, 4, 5] and the corresponding EDA tools have enabled the step to higher abstraction levels in processor modeling and made ASIP design feasible even for small design teams. Contemporary design tools support ASIP ISA and micro-architecture design from scratch, while generating synthesizable HW models as well as SW tools (C compiler, simulator, etc.) almost automatically.

Due to the entry barrier usually caused by new modeling languages and tools, though, and due to the observation that many ASIPs tend to have a RISC-like core architecture and ISA anyway, *configurable processors* have emerged as a special class of ASIPs. Their core architecture can be customized and optimized towards given applications by means of *instruction set extensions* (ISEs). An ISE consists of several *custom instructions* (CIs), i.e. application specific instructions with a usually higher complexity than generic machine instructions like ADD, SUB, etc. We assume that CIs are implemented on a *co-processor*[1] tightly coupled to the main core.

While partially sacrificing silicon efficiency (MIPS/Watt), configurable processors make ASIP design more incremental and less complex, since both the HW architecture and SW tools are largely predefined. Industrial examples of configurable processors include Tensilica Xtensa, ARC Tangent, and MIPS CorExtend. Customization of such platforms can be divided into the following major steps:

1. **Application code analysis:** The application specification, mostly given as C code, needs to be analyzed to identify major dynamic execution characteristics and *hot spots* (i.e. frequently executed code regions) that need optimization via CIs. This demands for extensive code profiling.
2. **CI identification:** Based on the analysis results, code regions need to be identified whose implementation as CIs leads to the desired speedup of hot spots and is feasible from an implementation perspective. This is a constrained optimization problem with a huge search space, which can be automated with sophisticated solving algorithms.
3. **CI implementation:** The selected CIs need to be integrated into the predefined configurable processor template. The interface specification between the processor core and the CFU must be met and efficient communication must be guaranteed. Furthermore, CIs need to meet latency and area constraints, e.g. by means of proper pipeline stage balancing. As the detailed effects of HW synthesis are not fully known at this stage, estimations have to be used.
4. **SW adaptation and tools generation:** CIs need to be recognized and supported by the C compiler, assembler, instruction set simulator etc. While retargeting of most of these SW tools can be performed automatically, the C compiler can automatically utilize the CIs only in special cases, due to limitations in the code generation techniques. Therefore, the original application C code usually needs

1 We refer to such co-processors as *Customized Functional Units* or *CFUs* for the rest of this paper.

to be modified, too, e.g. by means of inserting *compiler known functions* or *intrinsics* for CI calls.

5. **HW architecture implementation:** The CFU and the interface specification must be converted into an RTL HDL model (VHDL or Verilog) that can be merged with the predefined code model for subsequent logic and layout synthesis with standard tool flows.

After a survey of related work in section 2, in this paper we propose a concrete design flow (section 3) for these steps. The key contribution is a new flexible optimization algorithm for the CI identification phase, described in section 4. This algorithm is embedded into a suite of state-of-the-art external tools for the remaining phases. We provide some experimental results in section 5, and conclusions are given in section 6.

2. Related work

A large body of research has already been done to investigate the different aspects of CI identification and implementation. This section briefly traces the related scientific literature along the lines of automatic instruction-set customization and associated design methodologies.

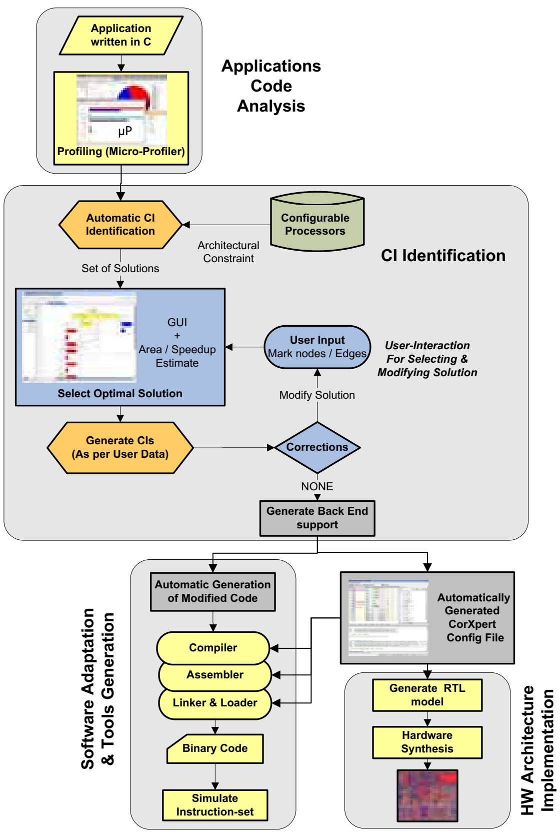
There exists two major competing approaches of *CI identification*. The first one, as presented in [7, 10], relies on incremental clustering of related data-flow graph nodes using heuristic approaches with the aim of identifying frequently occurring, and therefore reusable, CI patterns. Ienne et.al. [9], on the other hand, propose another approach where the identification algorithm tries to identify large, complex data-flow sub-graphs under I/O and convexity constraints. Reusability of the identified CIs is a secondary concern in this approach. Algorithmic improvements of the technique presented in [9] are reported in [14, 11]. None of these papers, however, present a complete methodology of processor customization using the CIs, and many parts of the design flow remain manual. A more automated instruction set customization methodology is presented in [8]. However, this work is mostly targeted towards a very specific architecture (Tensilica Xtensa).

There are two important recent works in the context of the current paper, which also use an Integer Linear Programming (ILP) technique for instruction-set extension. [12] proposes an ASIP design methodology using ILP, but it targets control dominated applications for VLIW machines, whereas our interest is mostly in data-flow oriented applications. The approach presented in [13] is closest to ours since it also uses an ILP based model for CI identification.

The key contributions of the current paper, with respect to the earlier works, are twofold. Firstly, we present a *complete*, *generic* processor customization flow which uses a novel CI identification algorithm. Our methodology is well scalable since it maps the CIs to an ADL based design flow. Secondly, from the algorithmic perspective, we use a novel technique of using local registers in the CFU to overcome the operand I/O constraints to a CI. This technique is described in detail in section 4.

3. Processor customization design flow

An overview of our processor customization design flow is presented in fig. 1. The following subsections provide a brief description of the different components of this design flow.



Application written in C
Profiling (Micro-Profiler)
μP
Applications Code Analysis
Automatic CI Identification
Configurable Processors
Architectural Constraint
CI Identification
Set of Solutions
GUI + Area / Speedup Estimate
Select Optimal Solution
User Input Mark nodes / Edges
User-Interaction For Selecting & Modifying Solution
Modify Solution
Generate CIs (As per User Data)
Corrections
NONE
Generate Back End support
Automatic Generation of Modified Code
Compiler
Assembler
Linker & Loader
Binary Code
Simulate Instruction-set
Software Adaptation & Tools Generation
Automatically Generated CorXpert Config File
Generate RTL model
Hardware Synthesis
HW Architecture Implementation

Figure 1. Processor Customization Design Flow

3.1. Application code analysis

Provided the application is specified in terms of a C program, *C code profiling* is the key technique for the analysis phase. Traditional profilers, such as GNU gprof, are suitable tools for optimizing the application SW itself, but they work at a coarse granularity level that makes them less suitable for guiding ASIP ISA design and predicting low-level effects such as code optimizations in the compiler backend. Assembly-level profilers provide sufficient accuracy, but they require a detailed processor model that might not be at hand at early design stages. Furthermore, they are relatively slow. Therefore, we proposed a novel *micro-profiler* (μP) tool [6] that fills the gap in profiler technology for ASIP design. While the μP provides some functionalities beyond CI identification (e.g. performance estimation and memory access profiling), its major use here is the *determination of operation execution frequencies* in hot spots and *prediction of code optimization effects* in the C compiler.

3.2. CI identification

A program hot spot is graphically represented as a *data-flow graph* (DFG). CI identification is viewed as a problem of optimal *DFG covering*, such that a significant speedup is achieved by execution of some DFG fragments (or the entire DFG in an extreme case) by means of CIs. The detailed optimization algorithm is described in section 4. It takes into account the constraints of the core processor (e.g. maximal number of inputs and outputs of an instruction) and the presence of CFU internal state (e.g. local registers). The optimization is carried out under area and latency constraints imposed by the core processor and the human designer, based on estimations using a one-time pre-characterized CMOS target library. As the optimization algorithm cannot take into account the designer's expert knowledge nor the detailed HW synthesis effects, CI identification is intentionally conceived as an *interactive process*, where the user can select from alternative solutions and modify generated CIs within a graphical environment.

3.3. CI implementation

For CI implementation, we use CoWare's CorXpert tool. CorXpert is a graphical tool for CI capture for configurable processors. The use of CorXpert as a backend makes the design flow quite generic, since it uses the LISA ADL [2] as the specification formalism for the CIs. The user describes a CI by specifying the format, coding information, and the behavior of the CI through the CorXpert GUI. The CI behavior is specified in a subset of the C language. In our flow, the graphical CI entry is circumvented, and a CorXpert configuration file is generated directly from the CI identification phase. The CI behavior specification corresponds to the C code fragments selected as CIs in the previous phase, augmented with necessary core/CFU communication macros. At this point, the designer can perform fine-grained manual optimizations, such as balancing CIs over the desired number of available CFU pipeline stages, or rewriting the CI behavior code for optimized HW implementation.

3.4. SW adaptation and tools generation

Unless by chance the entire hot spot DFG has been covered by CIs, the general result of CI identification is a DFG that is partially implemented in SW (i.e. by means of core processor instructions) and HW (i.e. by means of CIs). The DFG portions moved to HW are automatically replaced by *compiler intrinsics* in the original C code, while the SW part is left virtually untouched. The result is a C program that after compiler retargeting and compilation makes use of the generated CIs. The CorXpert tool retargets the core processor C compiler as well as the instruction set simulator (ISS). The ISS can then be used for cycle-true core/CFU simulation to back-annotate the exact speedup in terms of cycle count achieved by the CIs.

3.5. HW architecture implementation

CorXpert also generates a synthesizable CFU HDL model and triggers the HW synthesis process via standard logic synthesis tools. As a result, an accurate estimation of the real speedup (including maximum clock rate) and area overhead is achieved for the given CMOS target library. In case of significant deviations from the estimated results, the designer can return to the CI identification phase to modify the original solution.

4. Optimized CI identification

4.1. Background

The primary goal of CI identification is to maximize speedup of the given application under different constraints. While the exact speedup is only known after HW architecture implementation, it can be reasonably estimated based on

1. The hot spot operation execution frequencies delivered by the application code analysis, and
2. An estimation of SW and HW latencies of different operations.

CI identification, in general, can be conceived as a problem of *optimally partitioning* the DFG of an application between different CIs (and possibly between CIs and core processor instructions) that maximizes application speedup. However, any arbitrary cluster of nodes in a DFG cannot qualify as a CI in general. A CI is architecturally feasible only when the constituent DFG fragment follows a certain set of *constraints*. Such constraints can be broadly classified into the following three categories:

- **Data-flow constraints:** A set of DFG operations grouped into the same CI must satisfy a *convexity constraint* [9] that prevents cyclic dependencies between one CI and other DFG operations. Similarly, *schedulability constraints* have to be met that prevent deadlocks between sets of CIs during the compiler's instruction scheduling phase.
- **Latency and area constraints:** As a rule of thumb, the speedup due to CIs is proportional to their complexity. Therefore, CI synthesis generally tends to put as many DFG operations as possible into a single CI. However, there are limitations on permissible CI size due to area, power consumption, and/or latency constraints. In particular, the CI combinational critical path must not exceed the core processor's clock period. Pipelining of CIs can soften this constraint, depending on the number of pipeline stages available for CI execution on the CFU. Since most embedded ASIPs need to meet very tight area budgets, a constraint can also be imposed by the designer on the maximum *silicon area* of a CI.
- **Architectural constraints:** Assuming the CFU is tightly coupled with the core processor, communication frequently takes place via the core's general purpose register (GPR) file. Due to the limited instruction word length, generally only a few GPRs are available for this purpose (e.g. 2 input GPRs and 1 output GPR per CI). Furthermore, a CI may or may not access the data memory, and other constraints may hold for immediate constants etc. Although these constraints impose tight limitations, the CFU may provide internal state to support more complex CIs via increased I/O capabilities.

4.2. CI synthesis algorithm

Optimized CI synthesis under the above-mentioned constraints can be precisely formulated as a mathematical optimization problem. A high degree of flexibility is required, though, in order to (1) enable CI optimization for a wide range of configurable processors and (2) allow for simple accommodation of additional, user-specified constraints. This makes *Integer Linear Programming* (ILP) an attractive solution vehicle. An optimal solution to the entire problem, however, cannot be

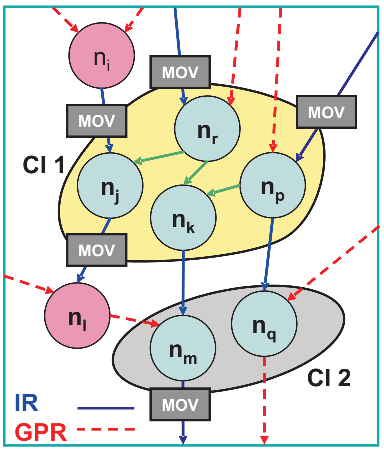


Figure 2. Communication through IRs

found within reasonable CPU time due to the high runtime requirements of ILP solvers. Therefore, we follow a divide-and-conquer approach where (1) CIs are constructed -locally optimal- one after another and (2) less important optimization subtasks are outsourced to fast heuristics[2].

Besides the selection of effective CIs, our algorithm also emphasizes minimization of *communication costs* between the core and the CFU, as well as within the CFU itself. We assume that each CI can access a small, fixed number of GPRs, while larger I/O bandwidth is only enabled via *internal registers* (IRs) of the CFU. IRs can be freely allocated during CI synthesis. However, they are subject to minimization due to their area overhead, and a *communication cost overhead* is assumed for moving data between GPRs and IRs. The use of such internal state registers in CI identification has been already proposed in [18]. However, consideration of the limited bandwidth between the core and the CFU, and the general possibilities of overcome the limits by IRs is the unique contribution of our algorithm.

An example of the use of such IRs and the corresponding communication overheads is illustrated in fig. 2 which shows two CIs, namely $CI1$ and $CI2$, that have more inputs/outputs than the maximum permissible number of GPR reads/writes (assumed to be 2 and 1, respectively). Therefore, nodes n_r, n_j and n_p inside $CI1$, and n_m and n_q inside $CI2$ require inputs from the IRs. Similarly, nodes n_j, n_p, n_k and n_m need to produce results into IRs. However, since nodes n_r, n_j and n_p have inputs coming from nodes outside CIs, these inputs need to be moved first into the IRs (indicated by the MOV boxes in fig. 2). Similarly, the outputs of nodes n_j and n_m are to be moved back from IRs to GPRs. Such extra moves introduce communication overheads that must be taken into account during the CI synthesis. Note that, however, the communication between nodes n_k and n_m (or, between n_p and n_q) does not incur any extra costs since IRs are available to both of these nodes. The possibility of such free communication between different CIs through IRs is also taken into account by the identification algorithm.

2 The detailed ILP formulation is omitted here for sake of brevity. It is available in a technical report [15].

Our CI identification algorithm works in two steps. Using ILP, the first step optimally partitions the DFG into different clusters of nodes that qualify as valid CIs. Although we do not consider the I/O constraints in this step, the ILP uses heuristics that take into account the possible communication overheads that might result due to such constraints. When this partitioning is done, the algorithm again uses another ILP model in the second step to decide about the means of communication between different nodes. These two steps are briefly described in the following two subsections.

4.3. Step 1: DFG Partitioning into CIs

In then first step, an optimization algorithm is iteratively applied on the DFG, $G = (N, E)$, of an application (or an application's hot-spot). In each iteration, the objective is to identify *one cluster* of DFG nodes that (1) maximizes the speedup function locally, and (2) obeys all the CI identification constraints except those on the number of GPR inputs and outputs. However, the objective function of the optimization problem does take the penalty of using IRs into account. This iterative process continues until either no more CIs can be identified (i.e. the remaining nodes in the DFG cannot be combined into a valid CI), or a user specified maximum permissible number of CIs is reached.

Each iteration starts with a set of DFG nodes, $S = \{n_1, n_2, \cdots, n_m\}$, which are still not part of any CI. Each node, $n_i \in S$, is associated with a unique binary variable U_i which is used to construct the ILP model for that iteration. If the solution of the ILP assigns 1 to U_i, then n_i becomes part of a new CI. Otherwise, it is left out. The objective function of the ILP is designed to maximize the speedup and therefore, tries to include nodes with high software latencies (i.e. number of cycles to execute that node in software). Moreover, it also tries to include nodes with low communication costs (e.g. nodes that are already connected to nodes inside other CIs and therefore, can communicate using IRs without any overhead). Therefore, the contribution of a single node, n_i to the objective function is defined as:

$$f(n_i) = (SW_i * U_i) - \sum_{neighbors} Comm_i(neighbor)$$

where SW_i is the software latency of n_i and $Comm_i(neighbor)$ is the communication cost for n_i with a neighbor in the DFG.

Any neighbor, n_p, of the node n_i falls in one of the following categories:

1. Nodes that can never be a part of a CI. Usually, nodes that make main memory accesses (loads and stores) fall into this category. Since these nodes require inputs/outputs through GPRs, communication with them *may come with an overhead if n_i is included in a CI.*

2. Nodes that are already part of earlier identified CIs. Communication with such a node (through a GPR or an IR) is *always free of any overhead if n_i is included in a CI.*

3. Nodes that are candidates for inclusion in a CI in the current iteration. Such a node do not have any communication cost *if it is included in the CI along with n_i.*

Keeping the above model of communication in mind, the communication overhead of n_i, for neighbor n_p, can be summarized as:

$$Comm_i(n_p) = \begin{cases} 1 & case(1) \\ 0 & case(2) \\ (1-U_p) & case(3) \end{cases}$$

The overall objective function, O_N, for the set S is:

$$O_N = \sum_{i=1}^{m} f(n_i)$$

The problem constraints mentioned above (generic, latency, architectural) are specified using a set of inequalities. For example, the convexity constraint requires that if any pair of nodes n_i and n_k are included in one CI, then any node n_j on any path between n_i and n_k must also be in the same CI. This fact is represented by the inequality:

$$U_i + U_k \leq U_j + 1$$

for all triplets of nodes n_i, n_j and n_k that are connected by a path in the DFG.

4.4. Step 2: Register Type Assignment

The second step of the algorithm is performed only once with the partitioned DFG obtained after the first step. This step enforces the GPR I/O constraints and optimally assigns register types (GPRs or IRs) to the incoming and outgoing edges of the different CIs. The objective here is to maximize the use of GPRs by the CIs (in other words, to minimize the use of the IRs so that the area of the IR file is kept small).

Let $S_E = \{e_1, e_2, \cdots, e_l\}$ be the set of I/O edges to the different CIs after the first step. The second step tries to maximize the following objective function:

$$O_E = \sum_{i=1}^{k} G_i$$

where G_i is a binary variable associated with edge $e_i \in S_E$. If the ILP solves G_i as 1, then the corresponding edge is assigned a GPR register type, otherwise, it is considered to go though the IRs. The I/O constraints on a CI (with input and output edge sets $S_{IN} = \{e_1, e_2, \cdots, e_k\}$ and $S_{OUT} = \{e_1, e_2, \cdots, e_j\}$, respectively) are enforced by forming the following inequalities:

$$\sum_{i=1}^{k} G_i \leq IN_{max}$$

$$\sum_{i=1}^{j} G_i \leq OUT_{max}$$

where IN_{max} and OUT_{max} are the maximum permissible inputs and outputs, respectively, through GPRs.

Fig. 3 shows an example run of the algorithm on a sample DFG. The first step is iterated twice over the original graph to identify two CIs (clusters of nodes enclosed in ellipses). The second step takes as a input the partitioned graph and assigns register types to different edges (the edges which are assigned GPR types are shown in light dotted lines, while the IR type edges are shown in dark solid lines) as shown in the figure.

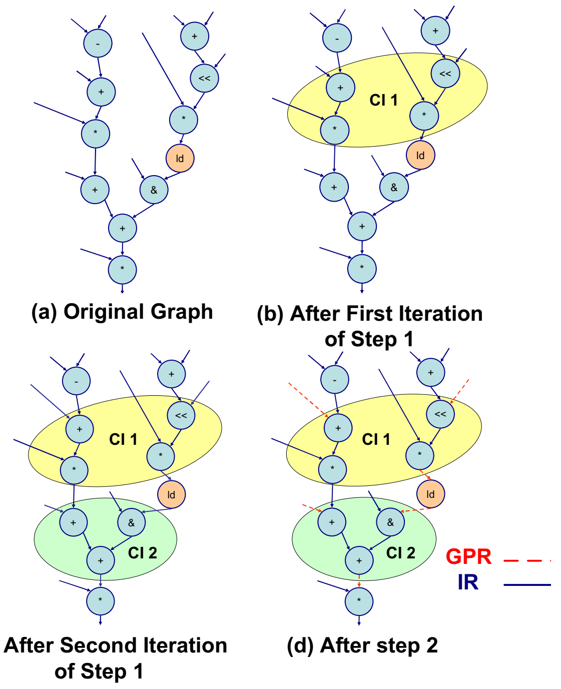

(a) Original Graph
(b) After First Iteration of Step 1
(c) After Second Iteration of Step 1
(d) After step 2
CI 1
CI 2
GPR
IR
ld

Figure 3. Example run of CI identification

4.5. CI scheduling and register allocation

As a preparation step for the SW adaption phase (section 3.4), the partial ordering of the selected CIs inside the hot spot DFG must be embedded into a sequential schedule. We use a simple *list scheduler* [20] for this purpose, where each group of DFG nodes assigned to the same CI is collapsed into a single node. The constraints obeyed during the CI identification phase ensure that a valid schedule always exists. According to the schedule obtained, C code fragments in the original source code are replaced by instances of compiler intrinsic, for sake of recognition by the C compiler.

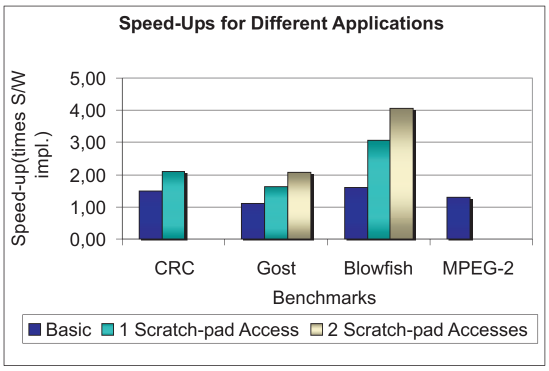
The DFG obtained after CI identification contains edges either assigned to GPRs or IRs. As the C compiler takes care of GPR allocation, we are only concerned with optimized IR allocation. We use a variant of the *left-edge algorithm* [19] for this purpose that efficiently achieves an allocation with a minimum amount of IRs based on live range analysis. This minimizes the area overhead due to IRs.

In the current implementation, scheduling and register allocation are separated. Due to the well-known mutual dependency between both, we expect that somewhat better results could be obtained with a *phase-coupled* approach.

5. Experimental results

This section presents some results to demonstrate the efficiency and applicability of our processor customization design flow. Our benchmarking process has been carried out with two objectives in mind:

1. Demonstration of the capabilities of the CI identification algorithm. A measure of this is the speedups



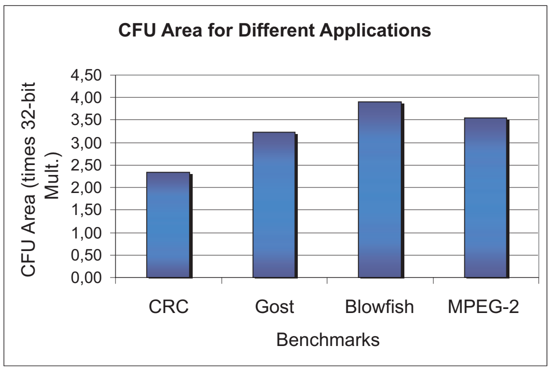


Figure 4. Different Results for the ISE synthesis methodology

obtained for various applications which are presented in fig. 4.(a), and the corresponding CFU area overheads as presented in fig. 4.(b).

2. The efficiency of the design flow. This is roughly proportional to the time and effort required for CI identification and CI implementation.

The speedup values for four different applications (Blowfish, gost, MPEG2 and CRC), presented in fig. 4.(a), have been obtained through cycle accurate *Instruction Set Simulators (ISS)* generated using the CorXpert tool chain for the MIPS CorExtend [3] processor extensions. As can be seen, the identified CIs result in an average performance improvement of around 1.4x, w.r.t. the pure software implementation on MIPS.

The speedup increases considerably if the possibility of memory accesses from CIs is considered. Since MIPS does not allow memory accesses from the CIs, the only way of making CIs capable of accessing memories is to put frequently used data objects into small *scratch-pad* memories [17] inside the CFU block itself. For the benchmarked applications, such objects include the *S-Boxes* for the two private-key cryptographic algorithms, Blowfish and gost, and a constant look-up table for the CRC calculation. MPEG2 did not have any such prominent data object. As can be seen from fig. 4.(a), allowing each CI to make one scratch-pad access results in around 2.3 times speedup on average, while the average speedup with two scratch-pad accesses per CI is around 3.1 times.

Fig. 4.(b) presents the extra CFU area required for implementing the different CI configurations for the benchmarks. These values were obtained by synthesizing the CorXpert generated RTL code through Synopsys Design Compiler using a 130nm CMOS library. As can be seen, the average area overhead is around 3 times that of a 32-bit multiplier. This is only a fraction of the chip area occupied by a typical embedded RISC core.

6. Conclusions

This paper presents a new workbench approach and design flow for configurable processors with ISEs. As far as the area overhead and the speedup is concerned, our solutions are somewhere between pure SW implementations and full-custom designed ASIPs. A major advantage of our approach is the large degree of automation in identifying the CIs, and the completely automated generation of the modified C code and the ADL/HDL description. Using our design flow, it is possible to customize a base processor core with a CFU within a few hours design time. This is also enabled by the proposed fast optimization algorithm, which takes typically a few seconds, and up to some CPU minutes, for optimized CI identification in realistic benchmarks.

References

[1] M. Gries, K. Keutzer, H. Meyr, et al.: *Building ASIPs: The Mescal Methodology* Springer, 2005
[2] http://www.coware.com/products/lisatek.php
[3] MIPS Inc.: CorExtend Technology, http://www.mips.com
[4] Target Compiler Technologies: Chess compiler, www.retarget.com
[5] P. Mishra, N. Dutt, A. Nicolau: *Functional abstraction driven design space exploration of heterogenous programmable architectures*, Int. Symp. on System Synthesis (ISSS), 2001
[6] K. Karuri, M. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, H. Meyr: *Fine-grained Application Source Code Profiling for ASIP Design*, 42nd Design Automation Conference, June 2005
[7] M. Arnold, H. Corporaal: *Designing Domain-Specific Processors*, Int. Conference on Hardware - Software Codesign and System Synthesis (CODES), 2001
[8] F. Sun, S. Ravi, A. Raghunathan, N.K. Jha: *Synthesis of Custom Processors based on Extensible Platforms*, in Int. Conference on Computer Aided Design (ICCAD), 2001
[9] K. Atasu, L. Pozzi, P. Ienne: *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*, 40th Design Automation Conference (DAC), 2003
[10] N. Clark, H.Zong, S.Mahlke: *Processor Acceleration Through Automated Instruction Set Customization*, 36th Annual International Symposium on Microarchitecture, 2003
[11] P. Biswas, S.Banerjee, N. Dutt, L. Pozzi, P. Ienne: *ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement*, 42nd Design Automation Conference (DAC), 2005
[12] Gero Dittmann, Paul Hurley: *Instruction-set Synthesis for Reactive Real-Time Processors: An ILP Formulation*, IBM Research Report, RZ 3611, 2005
[13] K. Atasu, G. Dündar, C. Özturan: *An Integer Linear Programming Approach for Identifying Instruction-Set Extensions*, Int. Conference on Hardware - Software Codesign and System Synthesis (CODES), 2005
[14] P. Yu, T. Mitra: *Scalable Custom Instructions Identification for Instruction-Set Extensible Processors*, Int. Conference on Compiler, Architectures and Synthesis for Embedded Systems (CASES), 2004
[15] Manas Pandey: *Semi-Automatic Instruction-set Customization of Configurable Embedded RISC Processors*, Master's Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2005
[16] H. Scharwaechter, D. Kammler, A. Wieferink et. al.: *ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study*, Software and Compilers for Embedded Systems (SCOPES), 2004
[17] S. Steinke, L. Wehmeyer, B.S. Lee, P. Marwadel: *Assigning Program and Data Objects to Scratch-pad for Energy Reduction*, University of Dortmund
[18] P. Biswas, K. Atasu, V. Choudhary, L. Pozzi, N. Dutt, P. Ienne: *Introduction of local memory elements in instruction set extensions*, 41stDesign Automation Conference, 2004
[19] F. J. Kurdahi, A. C. Parker: *REAL: A Program for Register Allocations*, Design Automation Conference, 1987
[20] S. S. Gibons, A. Philips: *Efficient Instruction Scheduling for a Pipelined Processor*, SIGPLAN Symposium on Compiler Construction, 1986