An End-to-End Design Flow for Automated Instruction Set Extension and Complex Instruction Selection based on GCC

Oscar Almer, Richard Bennett, Igor Böhm, Alastair Murray, Xinhao Qu, Marcela Zuluaga, Björn Franke, and Nigel Topham

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB
United Kingdom

Abstract. Extensible processors are application-specific instruction set processors (ASIPs) that allow for customisation through user-defined instruction set extensions (ISE) implemented in an extended micro architecture. Traditional design flows for ISE typically involve a large number of different tools for processing of the target application written in C, ISE identification, generation, optimisation and synthesis of additional functional units. Furthermore, ISE exploitation is typically restricted to the specific application the new instructions have been derived from. This is due to the lack of instruction selection technology that is capable of generating code for complex, multiple-input multiple-output instructions. In this paper we present a complete tool-chain based on GCC for automated instruction set extension, micro-architecture optimisation and complex instruction selection. We demonstrate that our approach is capable of generating highly efficient ISEs, trading off area and performance constraints, and exploit complex custom instruction patterns in an extended GCC platform.

1 Introduction

Industry's demand for flexible embedded solutions providing high performance and short time-to-market has led to the development of configurable and extensible processors. These pre-verified application-specific processors build on proven baseline cores while allowing for some degree of customisation through user-defined instruction set extensions (ISE) implemented as functional units in an extended micro-architecture.

Existing design flows targeting ISE such as [1] are hardware-driven and only consider compilers and, in particular, the ability to exploit the newly generated ISEs, as an afterthought. This has led to a situation where code generation for complex instructions such as ISEs is either restricted to a single application from which the ISEs have been derived or is left at the responsibility of the

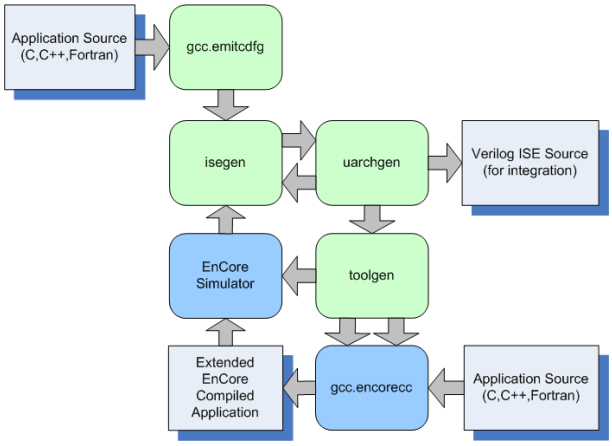


Fig. 1. Design flow of our framework for automated construction of ISEs.

(assembly) programmer. Clearly, this approach is not scalable with the growing size and complexity of embedded applications

In contrast, our integrated design flow treats the compiler, more specifically GCC, as a central component in the identification *and* exploitation of ISEs, thus, enabling reuse of complex instruction patterns *across* applications. Unlike previous work [2–8] that addresses isolated problems such as ISE identification, selection and implementation we cover the *entire* design flow from the source code level down to the physical processor implementation. We also introduce the EnCore, our extensible processor core, allowing us to synthesise full extensible designs from Verilog and standard cell libraries.

The diagram in figure 1 illustrates the high-level design and compilation flow of our automated ISE framework. Initially, GCC is used to process an application and construct a data flow graph (DFG) derived from its internal GIMPLE representation. Subsequently, the DFG is passed to the ISE generator for the analysis and extraction of candidate ISEs. The feedback-driven micro-architecture generator is in charge of selecting and implementing the most profitable ISEs based on heuristics that can be enhanced with performance, area and power information. This stage will also produce additional functional units using Verilog, suitable for integration in the baseline processor core. The *toolgen* program then extends the EnCore simulator and GCC compiler with the functionality to support the newly generated ISEs. The extended tools can then used to compile and simulate new applications, enabling the fully automated reuse of complex instruction patterns.

The remainder of this paper is structured as follows. In section 2 we present the background on extensible processors and automated instruction set extension as well as our specific EnCore platform. The related work is discussed in section 3. This is followed by a presentation of our GCC based design and compilation flow in section 4. Finally, we summarise and conclude in section 5.

2 Background

In this section we briefly introduce extensible processors and automated instruction set extension in general, before we describe our specific target platform, the extensible *EnCore* processor.

2.1 Extensible and Reconfigurable Processors

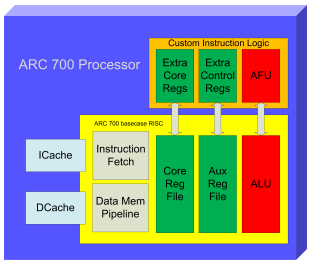


Fig. 2. A simplified system-level view of ARC700 family architecture, demonstrating the pre-verified baseline core and its connection to an ISE through custom registers and arithmetic units.

Extensible processors contain a number of variable components, essentially opening up design spaces inside the processor core for exploration by the designer of an ASIP-based system. Extensions to registers and supporting arithmetic logic are implemented outside of a prefab baseline core, the latter implementing all of the expected basic RISC functionality. In this manner users may make the best use of the degrees of freedom provided, with the knowledge that their extensions will not make unpredictable timing changes to the core as a whole. Architectures are extended by implementing extensions in SystemC or Verilog with respect to the architecture's extension interface. Examples of extensible processors include the ARC700 (see figure 2), Tensilica's XTensa [9], and Altera's NiosII.

Many extensible processors offer large degrees of flexibility through custom-specific instruction set extensions (ISE), which may help improve performance of compute-intensive kernels. As a result of this specialisation an optimised application-specific processor is derived from a generic processor template.

In order to explore different ISEs during the design stage and to trade off various, partially contradictory design goals (e.g. performance, power, chip area) tool support is indispensable. Existing commercial (e.g. [9]) and academic (e.g. [1]) tools analyse an application written in C, identify candidate instruction templates, modify the application's source code and insert handles to the newly created instructions. In general, the overall effectiveness of this approach depends on the designer's ability to generate complex instruction templates that (a) can replace a sufficiently large number of simple machine instructions, (b) are frequently executed and (c) can be efficiently implemented.

The automation of ISE exploration has been actively studied in recent years, leading to a number of algorithms [6, 2, 7, 8] which derive the partitioning of hardware and software under micro-architectural constraints. Most current approaches to automated ISE incorporate two phases:

1. **Identification**: whereby basic blocks are analysed to produce ISEs in the form of DFGs.
2. **Selection**: whereby a subset of the identified ISEs of the previous phase are chosen for implementation.

When extending an instruction set to cover a complete class of applications we can expect larger numbers of extension instructions to be identified, effectively representing the union of the extension instructions required by each application in the class. Even with a single complex application it is possible to find large numbers of potential extension instructions, each of which adds to the die area of the system. To avoid bloating the die area with large numbers of extension instructions it is important to identify and exploit such commonality between instructions and, where possible, to share hardware resources when this represents a good trade-off between die area and execution time. Brisk et al [10] have explored an approach based on finding the longest common sub-string, in order to determine which parts of a pipelined data path may be shared. This work was extended by Zuluaga et al in [11] in order to introduce parameters to control the process of merging data paths for resource sharing. The latter work focuses more on parameter exploration, allowing for integration into a design space exploratation framework.

2.2 Compiler Transformation for Instruction Set Extension

Early efforts to combine code transformation and ISE have been targeted at CDFG transformation towards a more efficient arithmetic structure [12]. This operates post automated ISE (AISE), so does not directly contribute to the design space search but improves upon the result.

In [13] it is shown that an exploration of *if-conversion* and *loop-unrolling* source-to-source transformations is successful in enabling better performing AISE. This work utilises control-flow transformations to move larger regions of the target application into the AISE algorithm at-once. The work in [13] demonstrates that new search methods and heuristics can be developed to control the application of transformations, with respect to the new set of goals inherent in ISE as compared to code generation.

More wide-ranging exploration for the space of compiler transformations in combination with instruction set extensions has been attempted in [14]. The work of [14] concluded that there is an unpredictable correlation between the set of transforms used prior to AISE and the speedup obtained overall. It is likely that further machine-learning approaches addressing this space will yield far better results than hand-crafted heuristics.

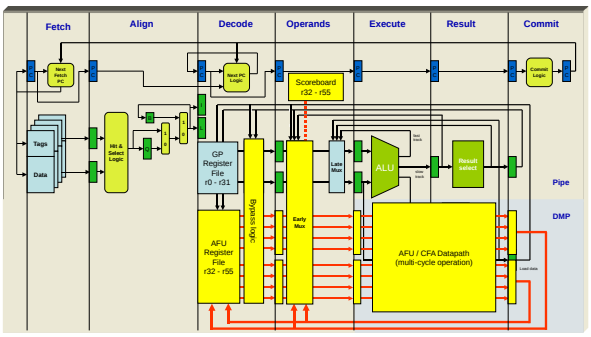


Fig. 3. Architecture of the 7-stage EnCore processor.

2.3 The EnCore Processor

The EnCore processor is developed entirely within the Institute for Computer Systems Architecture (ICSA) at the University of Edinburgh. The core is written in Verilog, and currently exists as two main variants; five and seven stage pipeline depth versions, each having varying cache configurations. Both feature a reasonably complete implementation of the ARCompact instruction set (as defined by the commercial ARC700 core), to the point that the arc-elf32 version of GCC can be used for compilation of C code for the processor. The EnCore does not currently feature an MMU, but this is in development and is likely to be incorporated into a later version of the EnCore.

The EnCore has been tested with various configurations in FPGA fabric, and has through this method been verified as correctly executing compiled C code obtained through the GCC arc-elf32 compiler. The five stage core closes timing at 25MHz using a Spartan-3 1600E FPGA as the implementation fabric, utilising approximately 50% of the available resources, and is expected to run two or three times faster using a newer FPGA family. The seven stage pipeline variant is expected to tolerate a roughly 40% higher clock frequency than this.

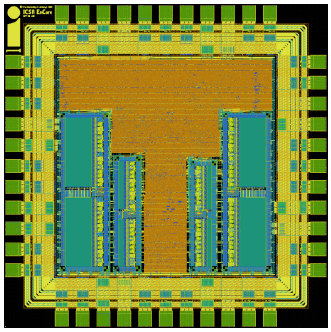
The core can be extended using modules written in Verilog derived from the ISEs. While it is expected that these will increase the logic in the core, in some cases by a large factor, we are not expecting it to impact the maximum operating frequency to a large degree. ISEs incorporated do not affect the critical path of the core itself, and are timed to fit with the clock frequency of the main core pipeline.

The mechanism by which the main core and the extension logic communicate is covered in figures 3 and 4(a). Essentially the extension registers of the ARCompact ISA are mapped to a set of 10 vectors, of which up to three can be selected in each extension instruction. There is also a permutation field, allowing for the elements in the vectors to be permuted before being sent to the extension logic.

We are currently in the process of finalising a tape-out of the five-stage variant of the EnCore for fabrication, configured with 4kB of instruction cache and 4kB of direct-mapped data cache. This is being attempted using a 130nm process, and it is expected to run at or close to 250MHz (ten times faster than the FPGA implementation). The total area for the EnCore using this process is roughly 1mm^2 including all the caches.

V0	0	0	0	0
V1	r32	r33	r34	r35
V2	r36	r37	r38	r39
V3	r40	r41	r42	r43
V4	r44	r45	r46	r47
V5	r48	r49	r50	r51
V6	r52	r53	r54	r55
V7	r32	r36	r40	r44
V8	r33	r37	r41	r45
V9	r34	r38	r42	r46
V10	r35	r39	r43	r47

(a) Extension register file organisation.



(b) Processor layout.

Fig. 4. Extension register file organisation and layout of the EnCore processor.

3 Related Work

ASIP design automation is a much studied area. Tensilica have the most notable automated effort [9] within commercial offerings, in that their approach is automated from source to structural extension of their XTensa processor [15]. For the purposes of research however, their tools are not open enough to explore the algorithms and alternative methodologies which are potentially beneficial to the overall process. In addition, there is at the time of writing no feedback apparent between the identification and selection phases of their approach. The user is expected to choose the design point from a pareto curve of area versus acceleration, as opposed to specifying constraints which the tool will meet automatically.

The combination of the LISATek and CoSy commercial packages has proven in the past to be a useful methodology for design space exploration of ASIPs including ISE. This approach has become known as "Compiler-in-Loop Design Space Exploration" [16]. Although the combination did not originally contain an automated component for the identification of new instructions later academic work has successfully combined a measure of design automation at this level [1]. It should be noted that the approach in [1] does not allow for mapping the new instructions to an application other than that originally analysed. This is because the new instructions are inserted as they are generated, treated as compiler intrinsics.

Other architectural approaches to this problem include ADRES [4], and Chameleon [17] amongst others. These depart from the standard RISC methodology to combine ISE with more eclectic hardware. The ADRES architecture [4] combined a VLIW processor with a coarse-grained reconfigurable array (CGRA), mapping from C-code to the ILP available on both the VLIW and CGRA units. The array is regularly arranged in a 2D grid, without specialisation towards any particular application other than via the width and height of the grid and the interconnect. The Chameleon architecture [17] and associated Montium tile processor are targeted very much towards streaming DSP applications, focusing on producing a low-power high-speed architecture able to be reconfigured for a variety of applications. A run-time mapping tool has been produced to dynamically map applications to the tiled architecture, making use of the fast reconfiguration time of the architecture. The architecture is largely fixed at the structural level, although new tiles could be produced with a more static ASIP bias, due to the heterogeneity of the tiled architecture.

Code generation utilising complex instructions is a topic which has been addressed widely by those wishing to target SIMD and DSP instructions (such as Multiply-Accumulate). Nuzman et al have produced extensions to GCC which are able to map SIMD instructions automatically to standard C code, for a range of platforms [18]. The SWARP [5] preprocessor on the other hand uses loop distribution, unrolling, and pattern matching to exploit complex multimedia instructions. Yet another graph-based approach of Leupers et al [3] is able to map simple MAC and SIMD instructions using a manually constructed model of each instruction. It would be difficult to extend this approach to accommodate an automated flow.

4 ISE Design and Compilation Flow

In this section we present our framework for automated processor specialisation and complex instruction selection.

4.1 Overview

Each tool in the framework is represented by a separate binary, with the majority of information passed between tools in human-readable serialised formats (largely XML). The overall DSE process is driven by a script of the user's choosing. The tools include:

- *GCC.emitcdfg*: a source (.c, .f, .cpp, etc) to XML DFG translator built into GCC.
- *GCC.encorecc*: GCC with a modified extensible *gas* (assembler), taking specification of the new instructions and mapping them in the code, producing a binary for the extended architecture.
- *isegen*: an implementation of a modified version of the ISEGEN [2] algorithm, including the capability to incorporate feedback on area, power, and actual hardware latency.

- *uarchgen*: an implementation of isomorphism-driven selection and resource sharing based on [11] for ISE; generates a specification in XML of the selected ISEs, along with Verilog extensions to the EnCore baseline. Information on instruction area, actual latency, power consumption, can be fed back to the isegen tool for inclusion in heuristic calculations.
- *toolgen*: generates extension object source for the ARCompact simulator, and encore.s which is used to extend the assembler used in GCC.encorecc.
- *Simulator*: partially cycle-accurate, ISA-extensible ARCompact simulator. Dynamically linked libraries can be passed to the simulator to describe the behaviour of new instructions.

See Figure 1 for detail of how these tools are expected to interact in the setting of AISE DSE. The following points give further details on the working and use of the tools outlined in Figure 1.

4.2 ISE Generation

The conversion of source code to a format which can be analysed by the various ISE algorithms is not entirely trivial. For the purposes of this project, a modified version of GCC (*gcc.emitcdfg*) is used to convert the source code from whatever input language it utilises, through GIMPLE (the GCC tree-based SSA IR), finally to a DFG representation in memory. This DFG representation is then emitted as XML. The profile information is added through the use of the GCC profiling options, which are available in the GIMPLE IR when the appropriate command line flags are set. The XML is then passed to the *isegen* tool for analysis and extraction of candidate ISEs (templates).

An implementation of the ISEGEN identification algorithm is present in the *isegen* tool; the procedural nature of the algorithm allows for modification and/or extension of the heuristic function(s). The heuristic functions are made more flexible through the use of dynamic linking for additional functions over graphs processed, and command line parameters for changing weights and other internal parameters of the partitioning algorithm. It should be noted that the ISEGEN algorithm is the Kernighan-Lin partitioning algorithm with a specific (published) [2] heuristic. This tool represents the "Identification" phase of the standard AISE design-flow, but the framework blurs the line between this phase and the later "Selection" phase due to feedback. Feedback in this sense is in the form of graphs, and their associated performance information (latency, area, and power). This information can then be used when the tool is exploring potential templates (sub-graphs of application data-flow graphs) for nomination as candidates for core extension.

4.3 ISE Implementation

Implementation of ISEs requires that instruction selection from the list of available templates, instruction coding, micro-architectural structure, and latency information (for scheduling) are all defined or derived. In our framework, this is

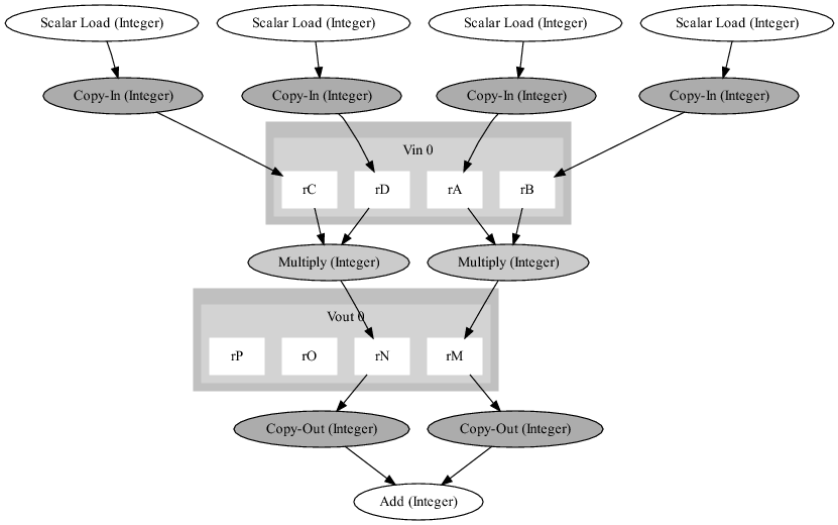


Fig. 5. A small example instruction that has been matched to a small example basic block. The white nodes represent parts of the IR to be implemented by normal scalar instructions, the light-gray nodes will be implemented by an extension instruction and the dark-grey nodes represent data movement in and out of the extension unit.

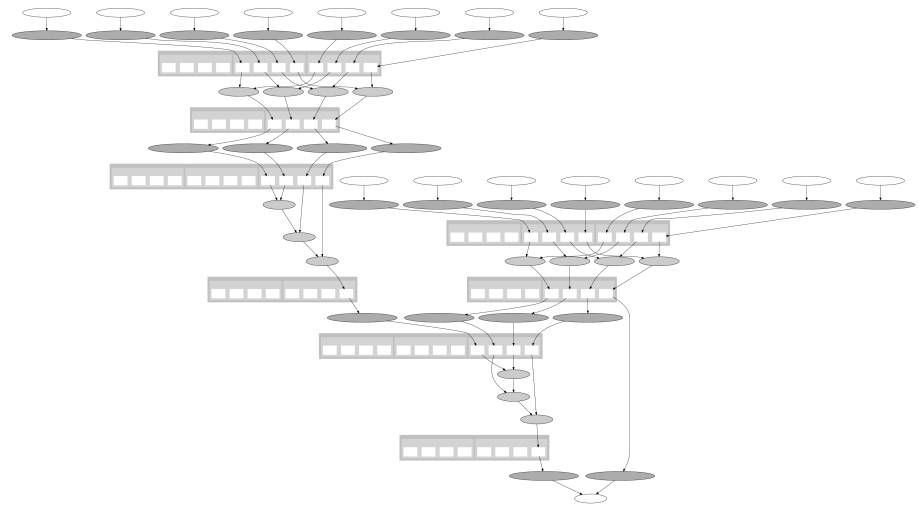


Fig. 6. The shape of a basic block that has been mapped to four separate extension instructions, two of which are entirely parallel and another two have internal dependencies. Colours are as in figure 5.

the job of the *uarchgen* tool. In addition, the tool provides specific information on the performance of templates passed to it (area, latency, power consumption) in order that this data can be passed back to the *isegen* tool to update and complement its heuristics. DesignCompiler and ModelSim are used in order to perform the relevant synthesis and analysis of generated structural Verilog.

It has been shown by members of our research group that an appropriate level of resource sharing between functional units implementing each a single ISE results in a significantly simpler hardware design whilst only modestly increasing the average instruction latency [11].

Based on existing resource-sharing techniques, the new heuristic controls the degree of resource sharing between a given set of custom instructions, given that design objectives are not always extremes as minimum execution time, or minimum die area. On the contrary, there are many possible intermediate points in the area-delay relationship, any one of which may be ideal for a given system.

Each instruction is represented as a data flow graph represented by a set of vertices V and a set of edges E, where vertices are operators, inputs or outputs, and edges indicate the data dependencies between them. Resource sharing is induced by the search for maximum common sub-strings between two paths. A maximum common sub-string is a sub-sequence of vertices that maximises area reduction. The area of a sub-string is given by the sum of the areas of each operation within the sub-string.

4.4 ISE Exploitation

Current ISE methodologies are typically limited to single applications, i.e. there is no compiler support available for complex instructions patterns generated by previous ISE identification and synthesis stages. Instead, the ISE identification tool specifies where the instructions should be used.

To be able to attempt to re-use instructions in additional programs to the ones they were generated for we implemented a complex instruction selector within GCC. Unusually, this performs instruction selection at the GIMPLE level. The reason for this is that it identifies matches based on the same internal representation as the ISE identification tool and this representation is constructed from GIMPLE, it also avoids the well-known difficulties in extending the *expand* GENERIC-to-RTL pass with machine-specific higher level instructions. Only complex instructions are matched by this pass, so many GIMPLE nodes will not be covered, these are handled by standard instruction selection in the back-end. The standard back-end is not used with the complex instructions as its pattern matching capabilities are not able to handle the large graphs that describe the instructions being considered here.

This pass operates by constructing the data flow graph of the instructions found during the ISE identification stage (the structure is described in an input XML file) and constructing the same internal representation as used by the ISE identification tool for each basic block. A sub-graph isomorphism library based on the VF2 algorithm [19] is used to find where each instruction may be used in each basic block. Once all matches have been found the original GIMPLE is

modified, the GIMPLE nodes that are to be covered by an extension instruction are removed and an extended asm node is inserted in their place. This node is constructed in the exactly the same way as the C front-end constructs this type of node when an extended asm statement is used in a C program. The reason for using extended asm nodes is that they provide the level of flexibility that we require due to having to be able to handle arbitrary instruction specifications that are described in the input XML, including unusual register allocation constraints that are not specified in the back-end, i.e. the vectors and permutation units described in section 2.3. Additionally, as it is expected that this tool will be used with-in an iterative design space exploration loop any solution that required recompiling the compiler for each new instruction specification was unacceptable.

A trivial hand-written example of an instruction matched to a basic block is shown in figure 5. The graph shows a basic block that contains two non-dependent integer multiplies and an addition, the multiplies are implemented by an extension instruction. The copy nodes (shaded dark-grey) should be able to be eliminated during register allocation, so won't actually result in any move instructions being used. Figure 6 shows a more realistic example where four instructions are matched to a larger basic block.

5 Summary and Future Work

We have presented a compiler-centric end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. Targeting an extensible baseline processor our tools analyse an application, identify and implement optimised instruction set extensions and extend the entire software development tool chain including simulator and compiler. Using a graph isomorphism based instruction selector this extended GCC compiler is able to automatically exploit and reuse complex instruction patterns and, hence, provides a much more scalable and powerful approach to ISE exploitation than any of the existing methodologies.

Future work will focus on integrating our tools with GCC-ICI to enable joint iterative search over compiler transformations and ISE. Furthermore, we will further improve the data movement between the base and extension registers as our current approach may occasionally introduce redundant register-to-register copy operations.

References

1. Leupers, R., Karuri, K., Kraemer, S., Pandey, M.: A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In: Design Automation & Test in Europe (DATE), Munich, Germany (2006)
2. Biswas, P., Banerjee, S., Dutt, N.D., Pozzi, L., Ienne, P.: ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. IEEE Transactions on VLSI **14**(7) (2006)

3. Leupers, R., Bashford, S.: Graph-based code selection techniques for embedded processors. ACM Trans. Des. Autom. Electron. Syst. **5**(4) (2000) 794–814
4. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. Field-Programmable Logic and Applications LNCS 2778 (2003) 61–70
5. Pokam, G., Bihan, S., Simonnet, J., Bodin, F.: SWARP: a retargetable preprocessor for multimedia instructions. Concurr. Comput. : Pract. Exper. **16**(2-3) (2004) 303–318
6. Peymandoust, A., Pozzi, L., Ienne, P., Micheli, G.D.: Automatic instruction set extension and utilisation for embedded processors. In Proceedings of the 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands. (2003)
7. Atasu, K., Dundar, G., Ozturan, C.: An integer linear programming approach for identifying instruction-set extensions (2005)
8. Pozzi, L., Atasu, K., Ienne, P.: Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(7) (2006) 1209–1229
9. Goodwin, D., Petkov, D.: Automatic generation of application specific processors. In: CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems. (2003) 137–147
10. Brisk, P., Kaplan, A., Sarrafzadeh, M.: Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In: DAC '04: Proceedings of the 41st annual conference on Design automation, New York, NY, USA, ACM Press (2004) 395–400
11. Zuluaga, M., Topham, N.: Resource sharing in custom instruction set extensions. In: Proceedings of the 6th IEEE Symposium on Application Specific Processors. (Jun. 2008)
12. Ienne, P., Verma, A.K.: Arithmetic transformations to maximise the use of compressor trees. In: Proceedings of the IEEE International Workshop on Electronic Design, Test and Applications, Perth, Australia. (2004)
13. Bonzini, P., Pozzi, L.: Code transformation strategies for extensible embedded processors. In: CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, New York, NY, USA, ACM Press (2006) 242–252
14. Bennett, R., Murray, A., Franke, B., Topham, N.: Combining source-to-source transformations and processor instruction set extension for the automated design-space exploration of embedded systems. In: LCTES 2007. (2007) 83–92
15. Inc, T.: XTensa LX Product Brief - http://www.tensilica.com/pdf/xtensa_LX.pdf
16. Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H.: Compiler-in-loop architecture exploration for efficient application specific embedded processor design (2004)
17. Smit, G.J.M., Andr B. J. Kokkele and, P.T.W., Hlzenspie, P.K.F., van de Burgwal, M.D., , Heysters, P.M.: The Chameleon architecture for streaming DSP applications. EURASIP Journal on Embedded Systems (2007)
18. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for SIMD. SIGPLAN Not. **41**(6) (2006) 132–143
19. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactons on Pattern Analysis and Machine Intelligence **26**(10) (Oct. 2004) 1367–1372