Runtime Adaptive Extensible Embedded Processors — A Survey

Huynh Phung Huynh and Tulika Mitra

School of Computing
National University of Singapore
{huynhph1,tulika}@comp.nus.edu.sg

Abstract. Current generation embedded applications demand the computation engine to offer high performance similar to custom hardware circuits while preserving the flexibility of software solutions. Customizable and extensible embedded processors, where the processor core can be enhanced with application-specific instructions, provide a potential solution to this conflicting requirements of performance and flexibility. However, due to the limited area available for implementation of custom instructions in the datapath of the processor core, we may not be able to exploit all custom instruction enhancements of an application. Moreover, a static extensible processor is fundamentally at odds with highly dynamic applications where the custom instructions requirements vary substantially at runtime. In this context, a runtime adaptive extensible processor that can quickly morph its custom instructions and the corresponding custom functional units at runtime depending on workload characteristics is a promising solution. In this article, we provide a detailed survey of the contemporary architectures that offer such dynamic instruction-set support and discuss compiler and/or runtime techniques to exploit such architectures.

1 Introduction

The ever increasing demand of high-performance at low-power in the embedded domain is fueling the trend towards customized embedded processors [13]. A customized processor is designed specifically for an application domain (e.g., network, multimedia etc.) enabling it to offer significantly higher performance compared to its general-purpose counterparts, while consuming much lower energy. This dual improvement in power-performance is achieved by eliminating certain structures (e.g., floating-point unit) that are redundant for the particular application-domain, while choosing appropriate dimensions for other structures (e.g., cache, TLB, register file). The elimination of redundant structures cuts down energy/area wastage and tailor-made dimensioning of required structures improves performance at reduced power budget.

A further step towards customization is instruction-set extensible processors or **extensible processors** for short. An extensible processor opens up the opportunity to customize the Instruction-Set Architecture (ISA) through application-specific extension instructions or custom instructions. Each custom instruction encapsulates a frequency occurring complex pattern in the data-flow graph of the application(s). Custom instructions are implemented as Custom Functional Units (CFU) in the data-path of the processor core. As multiple instructions from the base ISA are folded into a single custom

K. Bertels et al. (Eds.): SAMOS 2009, LNCS 5657, pp. 214–224, 2009.

instruction, we save fetching/decoding costs and improve code size. More importantly, the CFU can typically achieve significantly lower latency through parallelization and chaining of basic operations (the latency is determined by the critical path in the data-flow graph of the corresponding custom instruction) compared to executing one operation per cycle sequentially in the original processor. On the other hand, as custom instructions are exposed to the programmer, extensible processors offer great flexibility just like any software-programmable general-purpose processors. The large number of commercial extensible processors available in today's market (e.g., Xtensa [9], Lx [8], ARC configurable cores [2], OptimoDE [7], MIPS CorExtend [18]) is a testament to their wide-spread popularity.

There are, however, some drawbacks of traditional extensible processors. First, we need to design and fabricate different customized processor for each application domain. A processor customized for one application domain may fail to provide any tangible performance benefit for a different domain. Soft core processors with extensibility features that are synthesized in FPGAs (e.g., Altera Nios [1], Xilinx MicroBlaze [21]) somewhat mitigate this problem as the customization can be performed post-fabrication. Still, customizable soft cores suffer from lower frequency and higher energy consumption issues because the entire processor (and not just the CFUs) is implemented in FPGAs. Apart from cross-domain performance problems, extensible processors are also limited by the amount of silicon available for implementation of the CFUs. As embedded systems progress towards highly complex and dynamic applications (e.g., MPEG-4 video encoder/decoder, software-defined radio), the silicon area constraint becomes a primary concern. Moreover, for highly dynamic applications that can switch between different modes (e.g., runtime selection of encryption standard) with unique custom instructions requirements, a customized processor catering to all scenarios will clearly be a sub-optimal design.

Runtime adaptive extensible embedded processors offer a potential solution to all these problems. An adaptive extensible processor can be configured at runtime to change its custom instructions and the corresponding CFUs. Clearly, to achieve runtime adaptivity, the CFUs have to be implemented in some form of reconfigurable logic. But the base processor is implemented in ASIC to provide high clock frequency and better energy efficiency. As CFUs are implemented in reconfigurable logic, these extensible processors offer full flexibility to adapt (post-fabrication) the custom instructions according to the requirement of the application running on the system and even midway through the execution of the application. Such adaptive extensible processors can be broadly classified into two categories:

- **Explicit Reconfigurability:** This class of processors need full compiler or programmer support to identify the custom instructions, synthesize them, and finally cluster then into one (or more) configurations that can be switched at runtime. In other words, custom instructions are generated off-line and the application is recompiled to use these custom instructions.
- **Transparent Reconfigurability:** This class of processors do not expose the extensibility feature to the compiler or the programmer. In other words, the extensibility is completely transparent to the user. Instead, the runtime system identifies the custom instructions and synthesizes them while the application is running on the

system. These systems are more complex, but may provide better performance as the decisions are taken at runtime.

In this article, we will first provide a quick survey of the architecture of explicit runtime adaptive extensible processors followed by the compiler support required for such processors. Next, we will discuss transparent reconfigurable processors and their runtime systems. Finally, we will conclude this survey by outlining the challenges and opportunities in this domain.

2 Explicit Runtime Adaptive Extensible Processors

In this section, we will focus on extensible processors that require extensive compiler or programmer intervention to achieve runtime reconfigurability.

2.1 Architecture

Temporal Reconfiguration. We start with architectures that enable temporal reconfiguration, but only one custom instruction can exist at any point of time. That is, there is no spatial sharing of the reconfigurable logic among custom instructions.

PRISC (PRogrammable Instruction Set Processor) [17] is one of the very first architectures to include temporal reconfigurability of the custom functional units. Temporal reconfiguration virtually enlarges the limited reconfigurable hardware, which is tightly attached to the datapath of core processor. PRISC supports a set of configurations, each of which contains a computation kernel or a custom instruction. At any point of time, there is only one active configuration for reconfigurable hardware. However, each of the configurations can become active at some point of time through time-multiplexing. Therefore, temporal reconfiguration can extend the computational ability of the reconfigurable hardware at the cost of reconfiguration overhead.

Figure 1 shows the Programmable Functional Unit (PFU) in parallel with the other traditional functional units in the datapath of the PRISC processor. PFU data communication is similar to the other functional units. However, PFU can support only two input operands and one output operand. With the limitation on the number of input and output operands, PRISC cannot implement large custom instructions that can potentially provide more performance benefit though instruction-level parallelism as well as higher latency reduction. Moreover, as each configuration can include only one instruction, PRISC effectively restricts the number of custom instructions per loop body to one;

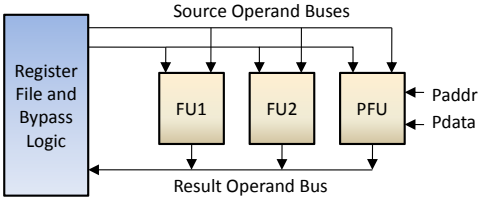

Fig. 1. PRISC Architecture [17]

otherwise, the temporal reconfiguration cost within loop body will typically outweigh any benefit of custom instructions.

OneChip [14] reduces reconfiguration overhead by allowing multiple configurations to be stored in the PFU, but only one configuration is active at any point of time. Moreover, OneChip comprises of a superscalar pipeline with PFU to achieve higher performance for streaming applications. However, OneChip lacks the details of how programmers specify or design the hardware that is mapped onto the reconfigurable logic.

Spatial and Temporal Reconfiguration. Both PRISC and OneChip allow only one custom instruction per configuration that can result in high reconfiguration cost specially if two custom instructions in the same code segment are executed frequently, for example, inside a loop body. Our next set of architectures enable spatial reconfiguration, that is, the reconfigurable hardware can be shared among multiple custom instructions. The combination of spatial and temporal reconfiguration is a powerful feature that partitions the custom instructions into multiple configurations, each of which contains one or more custom instructions. This clustering of multiple custom instructions into a single configuration can significantly reduce the reconfiguration overhead.

Chimaera [22], which is inspired by PRISC, is one of the original works considering temporal plus spatial configuration of the custom functional units. Chimaera tightly couples Reconfigurable Functional Unit (RFU) with a superscalar pipeline. The main innovation of the Chimaera RFU is that it uses nine input registers to produce the result in one destination register. Simple compiler support is provided to automatically map group of normal instructions into custom instructions. However, Chimaera compiler lacks support for spatial and temporal reconfiguration of custom instructions so as to make runtime reconfiguration more efficient.

Stretch S6000 [10] commercial processor follows this research trend. Figure 2 shows the Stretch S6000 engine that incorporates Tensilica Xtensa LX dual-issue VLIW processor [9] and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. It consists of a plane of Arithmetic/logic Units (AU) and a plane of Multiplier Units (MU) embedded and interlinked in a programmable, hierarchical routing fabric. This configurable fabric acts as a functional unit to the processor. It is built into the processor's datapath, and resides alongside other traditional functional units. The programmer defined application specific instructions (Extension Instructions) are implemented in this fabric. When an extension instruction is issued, the processor checks to make sure the corresponding configuration (containing the extension instruction) is loaded into the ISEF. If the required configuration is not present in the ISEF, it is automatically loaded prior to the execution of the user-defined instruction. ISEF provides high data bandwidth to the core processor through 128-bit wide registers. In addition, 64KB embedded RAM is included inside ISEF to store temporary results of computation. With all these features, a single custom instruction can potentially implement a complete inner loop of the application. The Stretch compiler fully unrolls any loop with constant iteration counts.

Partial Reconfiguration. Partial reconfiguration provides the ability to reconfigure only part of the reconfigurable fabric. With partial reconfiguration, idle custom instructions

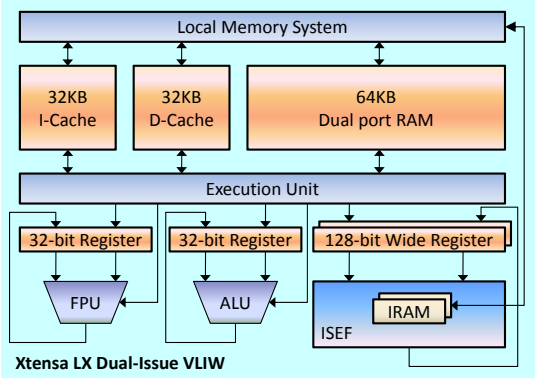

Fig. 2. Stretch S6000 datapath [10]

can be removed to make space for the new instructions. Moreover, as only a part of the fabric is reconfigured, it saves reconfiguration cost.

DISC (Dynamic Instruction Set Computer) [20] is one of the earliest attempts for an extensible processor to provide partial reconfiguration feature. DISC implements each instruction of the instruction set as an independent circuit module. It can page-in and page-out individual instruction modules onto reconfigurable fabric in a demand-driven manner. DISC supports relocatable circuit modules such that an existing instruction module can be moved inside the fabric to generate enough contiguous space for the incoming instruction module. The drawback of DISC system is that both standard and custom instructions are implemented in reconfigurable logic, causing significant performance overhead. On the other hand, the host processor is under-utilized as it only performs resource allocation and reconfiguration.

Extended Instruction Set RISC (XiRisc) [15] follows this line of development to couple a VLIW datapath with a pipelined run-time reconfigurable hardware. XiRisc has a five-stage pipeline with two symmetrical execution flows called Data Channels. Reconfigurable datapath supports up to four source operands and two destination operands for each custom instruction. Moreover, reconfigurable hardware can hold internal states for several computations so as to reduce the register pressure. However, configuration caching is missing in XiRisc leading to high reconfiguration overhead. Moreover, there is lack of compiler support for designer to automatically generate custom instructions.

Molen [19] polymorphic processor incorporates an arbitrary number of reconfigurable functional units. Molen resolves the issue of opcode space explosion for custom functions as well as data bandwidth limitation of the reconfigurable hardware. Moreover, Molen architecture allows two or more independent functions to be executed in parallel in the reconfigurable logic. To achieve these features, Molen requires a new programming paradigm that enables general-purpose instructions and hardware descriptions of custom instructions to coexist in a program. An one-time instruction set extension

of eight instructions is added to support the functionality of reconfigurable hardware. Molen compiler automatically generates optimized binary code for C applications with pragma annotation for custom instructions. The compiler can also generate appropriate custom instructions for each implementation of reconfigurable logic. The reconfiguration cost is hidden by scheduling the instructions appropriately such that the configuration corresponding to a custom instruction can be prefetched before that custom instruction is scheduled to execute.

2.2 Compiler Support

Most of the runtime adaptive extensible processors lack appropriate compiler support to automate the design flow. However, given the tight time-to-market constraint of embedded systems, compiler support is instrumental in developing greater acceptability of these architectures. Currently, the burden is entirely on the programmer to select appropriate custom instructions and cluster them into one or more configurations. Choosing an appropriate set of custom instructions for an application itself is a difficult problem. Significant research effort has been invested in developing automated selection techniques for custom instructions [13]. Runtime reconfiguration has the additional complication of both *temporal and spatial partitioning* of the set of custom instructions in the reconfigurable fabric.

We have recently developed an efficient framework [12] that starts with an application specified in ANSI-C and automatically selects appropriate custom instructions as well as clubs them into one or more configurations (see Figure 3). We first extract a set of compute-intensive candidate loop kernels from the application through profiling. For each candidate loop, one or more Custom Instruction Set (CIS) versions are generated differing in performance gain and area tradeoffs. The control flows among the hot loops are captured in the form of a loop trace (execution sequence of the loops) obtained through profiling. The hot loops with multiple CIS versions and the loop trace are fed to the partitioning algorithm that decides the appropriate CIS version and configuration for each loop. The key component of the framework is an iterative

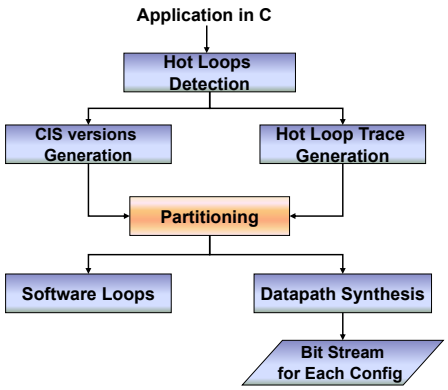

Fig. 3. Compiler framework for runtime adaptive extensible processors [12]

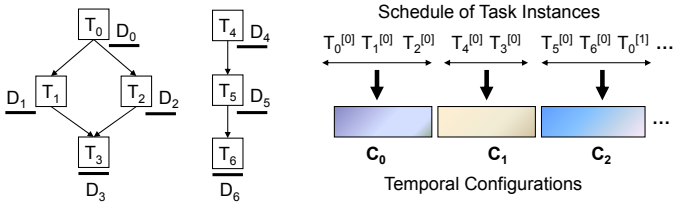

Fig. 4. A set of periodic task graphs and the corresponding schedule [11]

partitioning algorithm. We model the temporal partitioning of the custom instructions into different configurations as a *k-way graph partitioning* problem. A dynamic programming based pseudo-polynomial time algorithm determines the spatial partitioning of the custom instructions within a configuration. The selected CIS versions to be implemented in hardware pass through a datapath synthesis tool. It generates the bitstream corresponding to each configuration (based on the outcome of the temporal partitioning). These bitstreams are used to configure the fabric at runtime. The remaining loops are implemented in software on the core processor. Finally, the source code is modified to exploit the new custom instructions.

We also extend our work to include runtime reconfiguration of custom instructions for multiple tasks along with timing constraints [11]. An application is modeled as a set of periodic task graphs, each associated with a period and a deadline. Multiple CIS versions are generated for each constituent task of a task graph. Each task has many instances in the static non-preemptive schedule over the hyper-period (the least common multiple of the task graph periods) as shown in Figure 4. The objective is to minimize processor utilization by exploiting runtime reconfiguration of the custom instructions while satisfying deadline constraints. To achieve this goal, *temporal partitioning* divides the schedule into a number of configurations, where area constraint is imposed on each configuration. For example, Figure 4 illustrates an initial fragment of the schedule and its partitioning into three configurations. Note that each configuration contains a disjoint subsequence of task instances from the original schedule. Temporal partitioning allows a larger virtual area at the cost of reconfiguration overhead. The area within a configuration is *spatially partitioned* among the task instances assigned to it by choosing appropriate CIS version for each task instance. A dynamic programming based algorithm is enhanced with various constraints to efficiently solve the problem.

3 Transparent Extensible Processors

We now proceed to describe extensible processors that are reconfigured transparently by the runtime system.

Configurable Compute Accelerators (CCA): Transparent instruction-set customization supports a plug-and-play model for integrating a wide range of accelerators into a pre-designed and verified processor core. Moreover, instruction-set customization occurs at runtime. An architectural framework for transparent instruction-set

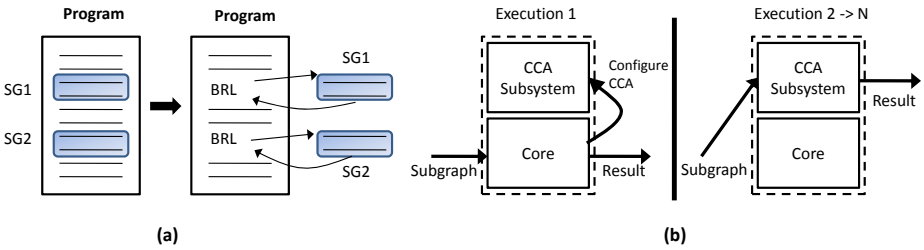

Fig. 5. Transparent Instruction Set Customization. (a) Subgraph Identification and (b) Runtime Processing [6].

customization has been proposed in [5]. The framework comprises of static identification of subgraphs for execution on CCA [6] and run-time selection of custom instructions to be synthesized to CCA as shown in Figure 5. First, the program is analyzed to identify the most frequent computation subgraphs (custom instructions) to be mapped onto CCA. Figure 5(a) shows that two subgraphs have been selected. They are considered as normal functions and will be replaced by function calls. At runtime, the first time a selected subgraph is encountered, it is executed in the core pipeline while a hardware engine determines the CCA configuration concurrently. From the second execution onwards, the subgraph is implemented in the CCA as shown in Figure 5(b).

Static subgraph extraction and replacement are achieved by adding a few steps into the conventional code generation process, which comprises of prepass scheduling, register allocation and postpass scheduling of spill code as shown in Figure 6. These steps are shaded in gray in the figure. First, given a dataflow graph, subgraph identification selects a set of potential subgraphs, which will be later implemented on CCA. Subgraph identification is a well-studied problem; interested readers can refer to [13] for a detailed exposition of the solutions. Note that subgraph identification is performed before register allocation to avoid false dependencies within data flow graph. After subgraph identification, selected subgraphs are collapsed into a single instruction. However, when collapsing subgraphs, code motion ensures the correctness if the subgraph crosses branch boundaries. Before getting into register allocation, the collapsed instruction is expanded so that register allocator can assign the registers to internal values. The advantage of this approach is that even a processor without CCA can execute the subgraphs as well (because they are treated as normal functions). More importantly, subgraph expansion ensures that register allocation remains relatively unchanged. After register

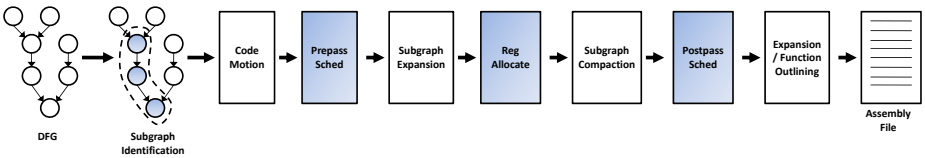

Fig. 6. Compiler Flow for CCA Architecture [6]

allocation, each subgraph is compacted to an atomic node and passed on as input to postpass scheduling. When postpass scheduling completes, each subgraph is expanded once again and a function is created for each subgraph along with a function call.

WARP: At the other end of the spectrum, we have WARP [16] that has been developed with completely transparent instruction-set customization in mind. WARP processor consists of a main processor with instruction and data caches, an on-chip profiler, WARP-oriented FPGA and an on-chip computer-aided design (CAD) module. The execution of an application starts only on the main processor. During the execution, the profiler determines the critical kernels of the application. Then, CAD module invokes the Riverside On-Chip CAD (ROCCAD) tool chain.

ROCCAD tool chain starts with *decompilation* of the application binary code of software loops into high-level representation that is more suitable for synthesis. Next, the partitioning algorithm determines the most suitable loops to be implemented in FPGA. For the selected kernels, ROCCAD uses behavioral and Register Transfer Level (RTL) synthesis to generate appropriate circuit descriptions. Then, ROCCAD configures the FPGA by using Just-In-Time (JIT) FPGA compilation tools. The JIT compiler performs logic synthesis to optimize the hardware circuit followed by technology mapping to map the hardware circuit onto reconfigurable logic fabric. Placement and route are then performed to complete the JIT compilation. Finally, ROCCAD updates the application binary code to utilize the custom accelerators inside the FPGA.

RISPP (Rotating Instruction Set Processing Platform) [4] is a recent architecture that offers a unique approach towards runtime customization. RISPP introduces the notion of *atoms* and *molecules* for custom instructions. Atom is the basic datapath, while a combination of atoms creates custom instruction molecule. Atoms can be reused across different custom instruction molecules. Compared to the contemporary reconfigurable architectures, RISPP reduces the overhead of partial reconfiguration substantially through an innovative gradual transition of the custom instructions implementation from software into hardware. At compile time, only the potential custom instructions (molecules) are identified, but these molecules are not bound to any datapath in hardware. Instead, a number of possible implementation choices are available including a purely software implementation. At runtime, the implementation of a molecule can gradually "upgrade" to hardware as and when the atoms it needs become available. If no atom is available for a custom instruction, it will be executed in core pipeline using the software implementation. RISPP requires fast design space exploration technique at runtime to combine appropriate elementary data paths and evaluate tradeoffs between performance and hardware area of the custom instructions [3]. A greedy heuristic is proposed to select the appropriate implementation for each custom instruction.

4 Conclusions

In this article, we focused on a detailed survey of extensible processors that provide runtime reconfiguration capability of the custom instruction sets. We observe that these architectures span a large spectrum starting from simplest solution that provides only temporal reconfiguration of a single custom instruction to more complex partial recon-

figuration and finally completely transparent reconfiguration solution where the custom instructions are identified and implemented at runtime. We also discuss compiler support necessary to exploit and harness this unique reconfiguration capability. Even though the architectural landscape in this domain looks quite promising, there is a serious lack of software tool support to take these solutions forward. In particular, runtime reconfiguration demands spatial and temporal partitioning of the custom instructions of an application into multiple configurations — a challenging problem for which only preliminary solutions exist today. Transparent extensible processors offer an interesting alternative to customization; however, the runtime overhead for design space exploration and synthesis is somewhat limiting the effectiveness of these proposals. We hope future research will bridge the gap between architecture and application to create an end-to-end solution for mapping applications to dynamic architectures.

Acknowledgements

This work is partially supported by NUS research project R-252-000-292-112.

References

1. Altera. Introduction to the Altera Nios II Soft Processor, ftp://ftp.altera.com/up/pub/Tutorials/DE2/Computer_Organization/tut_nios2_introduction.pdf
2. ARC. Customizing a Soft Microprocessor Core (2002), http://www.arc.com/upload/download/ARCIntl_0126_CustomizingSoftMicCore_wp.pdf
3. Bauer, L., Shafique, M., Henkel, J.: Run-time instruction set selection in a transmutable embedded processor. In: DAC (2008)
4. Bauer, L., Shafique, M., Kramer, S., Henkel, J.: RISPP: Rotating instruction set processing platform. In: DAC (2007)
5. Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., Flautner, K.: An architecture framework for transparent instruction set customization in embedded processors. In: ISCA (2005)
6. Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K.: Application-specific processing on a general-purpose core via transparent instruction set customization. In: MICRO (2004)
7. Clark, N., Zhong, H., Fan, K., Mahlke, S., Flautner, K., Van Nieuwenhove, K.: OptimoDE: Programmable Accelerator Engines through Retargetable Customization. In: Hot Chips (2004)
8. Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., Homewood, F.: Lx: A technology platform for customizable VLIW embedded processing. In: ISCA (2000)
9. Gonzalez, R.E.: Xtensa: A configurable and extensible processor. IEEE Micro. 20(2) (2000)
10. Gonzalez, R.E.: A software-configurable processor architecture. IEEE Micro. 26(5) (2006)
11. Huynh, H.P., Mitra, T.: Runtime reconfiguration of custom instructions for real-time embedded systems. In: DATE (2009)
12. Huynh, H.P., Sim, J.E., Mitra, T.: An efficient framework for dynamic reconfiguration of instruction-set customization. In: CASES (2007)
13. Ienne, P., Leupers, R. (eds.): Customizable Embedded Processors. Morgan Kauffman, San Francisco (2006)
14. Jacob, J.A., Chow, P.: Memory interfacing and instruction specification for reconfigurable processors. In: FPGA (1999)

15. Lodi, A., Toma, M., Campi, F., Cappelli, A., Canegallo, R., Guerrieri, R.: A VLIW processor with reconfigurable instruction set for embedded applications. IEEE Journal of Solid-State Circuits 38(11) (2003)
16. Lysecky, R., Stitt, G., Vahid, F.: WARP processors. ACM Transactions on Design Automation of Electronic Systems 11(3) (2006)
17. Razdan, R., Smith, M.D.: A high-performance microarchitecture with hardware-programmable functional units. In: MICRO (1994)
18. MIPS Technologies. MIPS Configurable Solutions, http://www.mips.com/everywhere/technologies/configurability
19. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The MOLEN Polymorphic Processor. IEEE Transactions on Computers 53(11) (2004)
20. Wirthlin, M.J., Hutchings, B.L.: A Dynamic Instruction Set Computer. In: FCCM (1995)
21. Xilinx. Microblaze Processor, http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm
22. Ye, Z.A., Moshovos, A., Hauck, S., Banerjee, P.: CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In: ISCA (2000)