

Chapter 2

Background and Related Work

Embedded processors are the key for rapidly growing application fields ranging from automotive to personal mobile communication, computation, entertainment, etc. The work presented in this monograph envisions an embedded processor that follows the concepts of extensible processors and uses a reconfigurable fabric to provide application-specific accelerators. Even though this work is mainly related to run-time reconfigurable processors, this chapter presents a short overview about the concepts of extensible processors and provides general background for reconfigurable computing before reviewing the most prominent work from the area of reconfigurable processors with a focus on state-of-the-art processors with a run-time reconfigurable instruction set.

2.1 Extensible Processors

In the early 1990s, the term ASIP has emerged denoting processors with an application-specific instruction set, i.e., they are specialized toward a certain application domain. They present a far better efficiency in terms of “performance per area,” “performance per power,” etc. compared to mainstream processors and eventually make today’s embedded (and often mobile) devices possible. The term ASIP comprises nowadays a far larger variety of embedded processors allowing for customization in various ways including (a) instruction set extensions, (b) parameterization, and (c) inclusion/exclusion of predefined blocks tailored to specific applications (like, e.g., an MPEG-4 decoder) [Hen03]. A generic design flow of an embedded processor can be described as follows:

1. An application is analyzed/profiled.
2. An extensible instruction set is defined.
3. The extensible instruction set is synthesized together with the core Instruction Set Architecture (cISA).
4. Retargetable tools for compilation, instruction set simulation, etc., are (often automatically) created.

5. The application characteristics (e.g., area, performance, etc.) are analyzed.
6. The process might be iterated several times until design constraints comply.

A general overview of the benefits and challenges of ASIPs is given in [Hen03, KMN02]. Tool suites and architectural IPs for embedded customizable processors with different flavors are provided by major vendors like Tensilica [Tena], CoWare/LisaTek [CoW], ASIP Solutions [ASI], ARC [ARC], and Target [Tar]. In addition, academic approaches like PEAS-III [IHT+00, KMTI03], LISA [HKN+01, ISS], and Expression [HGG+99] are available.

Using these tools suites, the designer can now implement a specific instruction set that is tailor-made for a certain set of applications. Typically, these suites come with a whole set of retargetable tools (compiler, assembler, simulator, debugger, etc.) such that the code can be generated conveniently for a specific ASIP. As the instruction set definition requires both, application and hardware architecture expertise, major research effort was spent in design-space exploration [CAK+07] and automatically detecting and generating so-called special instructions (SIs) from the application code [CZM03, HSM07]. A library of reusable functions (manually designed but therefore of high quality) is used in [CHP03], whereas in [API03, SRRJ03] the authors describe methods to generate SIs from matching profiling patterns. The authors in [BCA+04] investigate local memories in the functional units, which are then exploited by SIs. An automated, compiler-directed system for synthesizing accelerators for multiple loops (multifunction loop accelerators) is presented in [FKPM06]. The authors in [BKS04] present an approach to exploit similarities in data paths by finding the longest common subsequence of multiple data paths to increase their reusability. [CPH04] introduces an estimation model for area, overhead, latency, and power consumption under a wide range of customization parameters. In [BNS+04] an approach for an instruction set description on architecture level is proposed, which avoids inconsistencies between compiler and instruction set simulator.

However, facing the requirements of modern embedded systems, ASIPs experience a practical limitation. These approaches assume that customizations are undertaken during design time with little or no adaptation possible during run time. Therefore, they may perform poorly when deployed in scenarios that were not considered during optimizations. The proposed RISPP approach combines the paradigms of extensible processor design with the paradigm of dynamic reconfiguration in order to address the following concerns in embedded processing:

1. An application might have many computational hot spots (instead of only a few) and would require a large additional chip area in order to comprise all customizations necessary.
2. The characteristics of an application may widely vary during run time due to switching to different operation modes, change in design constraints (systems runs out of energy, for example), or highly uncorrelated input stimuli patterns.

In addition, for large applications that feature many hot spots and not just a few exposed ones, current ASIP concepts struggle. In fact, customization for many hot spots introduces a nonnegligible overhead and may bloat the initial small processor core.

Then, a designer needs to answer the question whether more powerful processor architectures would not have been the better choice. One means to address this dilemma is reconfigurable computing since resources may be utilized in a time-multiplexed manner (i.e., reconfigured over time), thus combining the performance and efficiency of dedicated hardware accelerators with a flexibility that goes beyond that of ASIPs and ASICs, respectively.

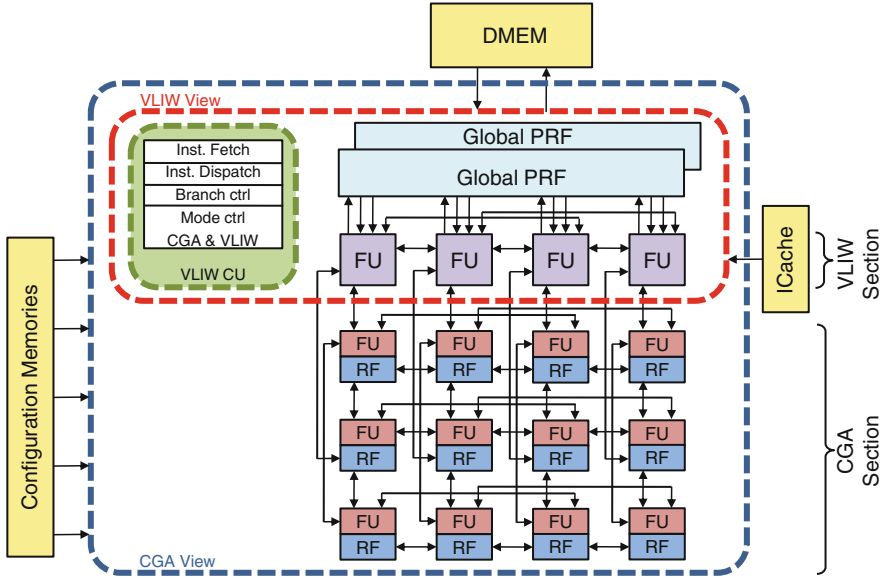
2.2 Reconfigurable Processors

The reconfigurable instruction set processing platform (RISPP) that is presented in this monograph utilizes the techniques of reconfigurable computing. Reconfigurable architectures address the challenge of supporting many hot spots by reusing the available hardware in time-multiplex, i.e., reconfiguring its functionality to support the currently executed hot spots. In this section, general techniques and concepts are presented, before reviewing state-of-the-art projects that are related to the proposed RISPP approach. General overviews and surveys for reconfigurable computing can be found in [Ama06, BL00, Bob07, CH02, Har01, HM09, TCW+05, VS07]. They also cover older projects and areas that are of less relevance for the approach that is presented in this monograph.

2.2.1 *Granularity of the Reconfigurable Fabric*

Conceptually, reconfigurable architectures can be separated into coarse and fine grained [VS07]. The coarse-grained approach maps word-level computation to a configuration of an array of arithmetic logic units (ALUs), e.g., using an automatic framework to select the appropriate configurations [HSM07]. The fine-grained approach instead reconfigures look-up tables (LUTs) on bit level (e.g., field programmable gate arrays: FPGAs). Therefore, the major difference between fine- and coarse-grained reconfigurable fabrics is the granularity of the reconfigurable elements. For coarse-grained fabrics, the operation of a word-level ALU (typically between 16 and 32 bit) can be determined with configuration bits. For fine-grained fabrics, the operation of a bit-level LUT (typically between 6:1¹ and 4:1) can be determined with configuration bits. To implement a control- and/or data-flow, multiple configurable elements (ALUs or LUTs, respectively) are connected (the connections are determined by configuration bits as well). Conceptually, word-level computation can be mapped to coarse-grained reconfigurable fabrics rather efficient and byte or sub-byte-level computation performs more efficient on fine-grained reconfigurable fabrics. For instance, fine-grained reconfigurable fabrics are

¹That is, 6-bit input and 1-bit output.



Legend:

CGA:	Coarse-Grained Array	FU:	Functional Unit
CU:	Control Unit	PRF:	Predicate Register File
DMEM:	Data Memory	RF:	Register File
DRF:	Data Register File	VLIW:	Very Long Instruction Word

Fig. 2.1 Connecting coarse-grained reconfigurable functional units

well-suited for diverse operations on image pixels (byte level), implementing state machines (sub-byte level), or calculating conditions (bit level). In addition to fine- and coarse-grained reconfigurable architectures, also combinations of both kinds were investigated [ITI, KBS+10, TKB+07].

Figure 2.1 shows a particular instance of the ADRES [BBKG07, MVV+03] coarse-grained reconfigurable fabric. The array comprises coarse-grained reconfigurable functional units (FUs). In general, an FU may correspond to an ALU; however, typically more complex compositions are used. For instance, it may comprise multiple ALUs or an ALU and Adder, etc. Furthermore, each FU may contain a local register file and also heterogeneous FUs may be used (e.g., specialized FUs for multiplication). The FUs are connected with each other. Often a two-dimensional arrangement is used and the FUs are connected to their direct neighbors (sometimes also to farther distant FUs and sometimes – when the data may through the array is predetermined – only to some neighbors). Different FU compositions, connections, heterogeneity, and memory connections (local register file and/or data memory) may be used for coarse-grained reconfigurable architectures. ADRES investigates different architecture instantiations [BBKG07, MLV+05] with a special

focus on an automatically generated compiler that can map kernels from C Code to the ADRES array [MVV+02].

A different approach for the coarse-grained reconfigurable computing is the custom compute accelerator (CCA) [CBC+05, CKP+04]. The CCA approach is based on coarse-grained reconfigurable elements and it supports a run-time placement of elementary data-flow graphs. This means, instead of preparing the configuration of the reconfigurable array at compile time (as done by ADRES), CCA identifies relevant computational kernels at compile time, but it creates configurations to accelerate them at run time. This allows that one application may be accelerated by different instantiations of the CCA reconfigurable array. To simplify the task of run-time placement, CCA is limited to a significantly narrowed hardware architecture. The reconfigurable array is meant to realize straight word-level data-flow (no external memory access, no internal state registers, and no data feedback, i.e. only acyclic data-flow graphs). As these straight data flows typically have more input data than produced output data (thus forming a triangular data-flow graph), the authors have organized their coarse-grained elements in a triangular shape as well. As shown in Fig. 2.2, the specific size, shape, and interconnections are determined application- and domain specific, depending on the occurring data-flow graphs. Placing individual nodes of a data-flow graph in a CCA can then be accomplished with one pass over the operations in the graph by placing each node

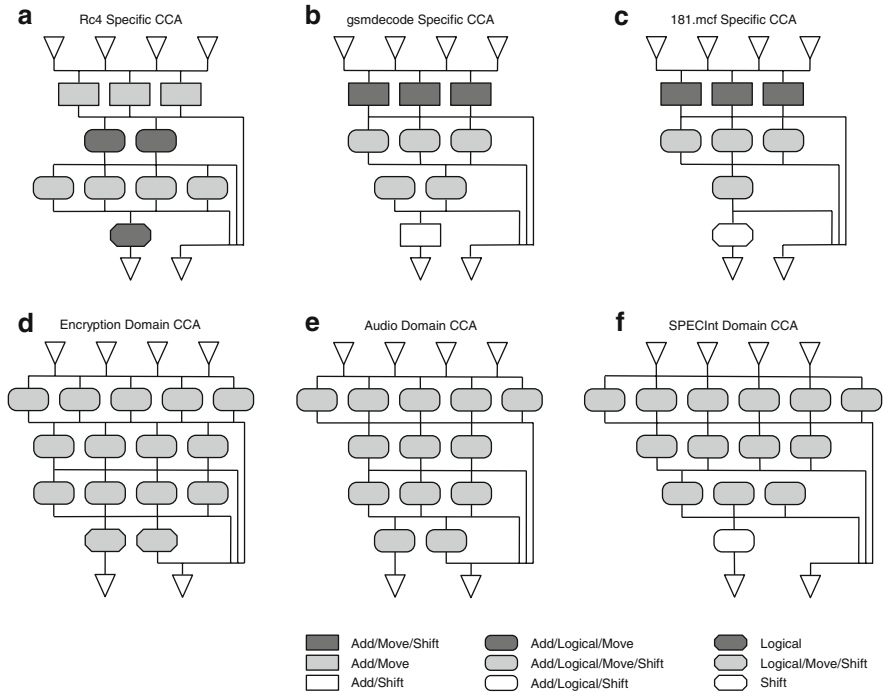


Fig. 2.2 Application-specific and domain-specific CCA design examples

in the highest row that can support the operation while respecting data dependencies. Therefore, this concept cannot be used for complex data-flow graphs (e.g., with embedded control-flow or exploiting loop-level parallelism), computations with high memory requirements (typical for multimedia applications), and multiple parallel executing applications with diverse system requirements (i.e., diverse-shaped data-flow graphs). A domain-specific optimized CCA reached an average speedup of $2.21\times$ for small application kernels [CBC+05].

In addition to these approaches, manifold coarse-grained reconfigurable approaches from academia (e.g., RAW [WTS+97], REMARC [MO99], PipeRench [GSB+00], HoneyComb [TB05]), and industry (e.g., Montium [HSM03, Rec] and PACT XPP [BEM+03, Tec06]) exist. However, this monograph presents a novel concept and architecture using a fine-grained reconfigurable fabric and thus, this chapter will focus on this type in the following. Before discussing general properties and presenting state-of-the-art reconfigurable processors, at first an overview of the underlying hardware architecture to realize fine-grained reconfigurable architectures is provided.

Figure 2.3 shows the internal structure of a Xilinx Virtex-II device [Xil07b] as an example for a fine-grained reconfigurable logic. The left side shows the so-called slice that comprises two 4:1 LUTs and two flip-flops together with multiplexers and connections. The LUTs can implement any functionality with four inputs and one

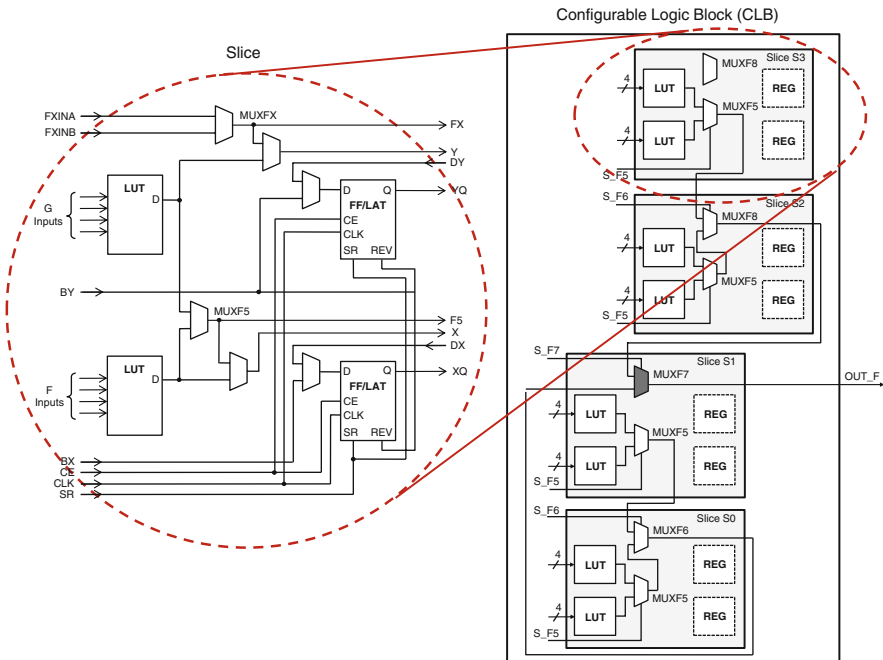


Fig. 2.3 Connecting LUTs as the basic building blocks of fine-grained reconfigurable logic to slices and configurable logic blocks

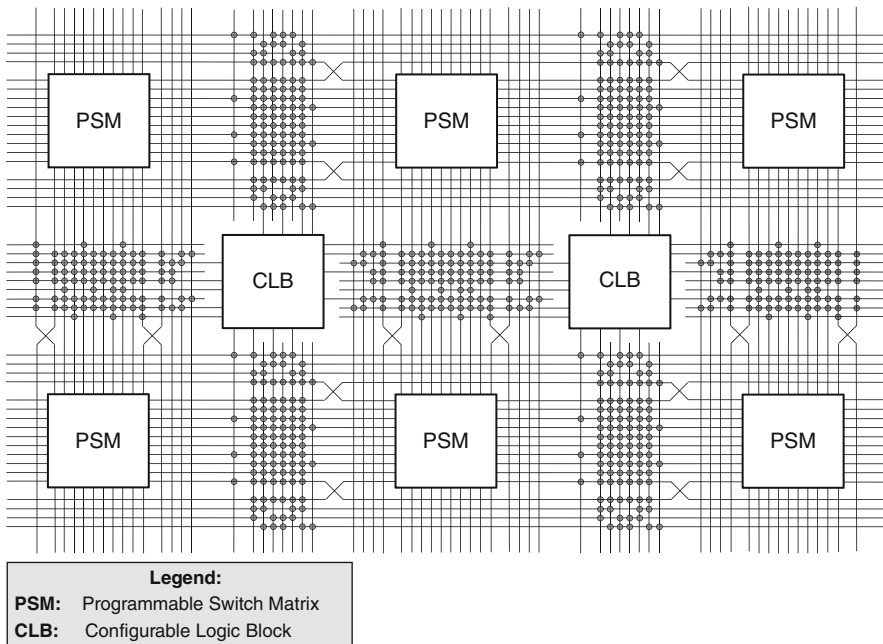


Fig. 2.4 Two-dimensional array of fine-grained reconfigurable CLBs that are connected with fine-grained reconfigurable switching matrices

output, i.e., all $2^4 = 16$ values of the corresponding truth table are configurable. To realize functions with more inputs or outputs, multiple LUTs need to be connected. The right side of the figure shows the so-called configurable logic block (CLB) that comprises four slices, i.e., eight LUTs. Here it becomes visible that some slices are directly connected with each other, which allows implementing a fast carry chain that connects multiple LUTs. Figure 2.4 shows the general structure of a fine-grained reconfigurable fabric that contains CLBs and programmable switching matrices (PSMs). The CLBs and PSMs are interconnected with dedicated wires. In addition, the PSMs are build of configurable interconnects, i.e., it is configurable which PSM input is connected to which PSM output.

These fine-grained reconfigurable fabrics provide a high flexibility, i.e., all hardware descriptions can be synthesized, placed, and routed for fine-grained reconfigurable fabrics (within their area restrictions). Bit- and byte-level computation and manipulation can be realized with rather few reconfigurable resources. In addition, small memories and finite state machines can be implemented using a fine-grained reconfigurable fabric. In addition, word-level computation can be realized; however, coarse-grained reconfigurable fabrics are typically more efficient for word-level computation as they are specifically optimized for them. Due to the high flexibility of fine-grained reconfigurable fabrics, all accelerators that might be used for an ASIP design can also be used for a fine-grained reconfigurable processor. Furthermore, the tool flow for designing accelerators for ASIPs and for a fine-grained

reconfigurable fabric is the same, i.e., a hardware description language like VHDL or Verilog can be used. Therefore, the related work for automatic detection and creation for accelerators and so-called special instructions (SIs) from ASIP research can be used for creating reconfigurable SIs as well, which improves usability because no new tool flow or programming model needs to be introduced, as it is required for coarse-grained reconfigurable arrays.

Different types of reconfiguration are distinguished and explained below. The fine-grained reconfigurable fabric allows to be reconfigured to implement different hardware designs and the major distinctions according the reconfiguration target the questions “when and how often” a reconfiguration may be performed and “which share” of the fabric is reconfigured.

Configurable. Denotes a fabric that can be configured one time, e.g., by using fuse/anti-fuse technology as used in programmable read-only memories (PROMs).

Reconfigurable. The configuration can be erased (e.g., using an UV-light source for EPROM) and a different configuration can be programmed. Often, erasing is a time-consuming operation (e.g., for EPROM) or demands different operation conditions (e.g., a higher supply voltage for EEPROM). Thus, it is performed, when the device is not active, similar to a firmware upgrade.

Run-time reconfigurable/dynamically reconfigurable. The reconfiguration is performed while the system is up and running. This demands a significantly faster reconfiguration time (microseconds to milliseconds instead of minutes).

Partially run-time reconfigurable. Often, not the entire reconfigurable fabric needs to be reconfigured. Instead, only a certain accelerator shall be reconfigured into a region of it. At the same time, the parts that are not reconfigured remain functional and active.

In addition to this classification, also the question “who triggers the reconfiguration” leads to different concepts. However, this decision is often coupled with the above-described categories. For configurable and reconfigurable systems, typically the user has to initialize and observe the reconfiguration, whereas for (partially) run-time reconfigurable designs it is common that the design itself triggers the reconfiguration (using an on-chip available access to the configuration memory, e.g., the internal configuration access port (ICAP [Xil09c]) in case of Xilinx Virtex FPGAs). In the following, the discussion will focus on the class of partially run-time reconfigurable architectures. Nowadays, basically all architectures use the Xilinx FPGAs and tools [LBM+06] to prototype partial run-time reconfiguration in practice. Especially here, the reconfiguration time may become an issue because the reconfiguration is performed as part of the normal operation of the architecture and the application may have to wait until the reconfiguration is completed, as the reconfigured accelerator is demanded to proceed. The reconfiguration time depends on (a) the amount of configuration bits that need to be transferred and (b) the reconfiguration bandwidth. For fine-grained reconfigurable logic, typically tens to hundreds of kilobyte need to be transferred (depending on the size of the design and the size of the reconfigurable fabric that shall be reconfigured). To reconfigure an entire FPGA device, even megabytes need to be transferred. The reconfiguration bandwidth can be as slow as a few MB/s (depending on the performance of the memory that stores the reconfiguration

data) or up to more than 100 MB/s [CMZS07]. The reconfiguration time is typically in the range of milliseconds. For instance, the extreme case of reconfiguring 10 KB at 100 MB/s (=100 KB/ms) demands 0.1 ms.² Vice versa, reconfiguring 100 KB at 1 MB/s demands 100 ms.³ To improve the reconfiguration performance, bitstream compression was suggested to improve the effective transfer rate from off-chip memory to the on-chip configuration port [HMW04, HUWB04, KT04, LH01]. Under certain circumstances, it may even be possible to hide the reconfiguration time by starting it before it is needed. [LH02] analyzes concepts for configuration prefetching such that the reconfigurations are completed (in the best case) before the accelerators are required. They distinguish between static prefetching (the compiler determines “when” and “what” shall be reconfigure), dynamic prefetching (a run-time systems determines the decisions), and hybrid prefetching (a compiler-guided run-time system determines the decisions). However, the reconfigurable fabric is typically highly utilized, because typically all computational blocks exploit the parallelism of the reconfigurable fabric to accelerate their kernels. Therefore, insufficient free space might be available to start prefetching. In addition, the time between computational blocks is typically small compared to the reconfiguration time (otherwise, it would rather be another computational block that benefits from accelerators). Therefore, prefetching for a computational block after the previously executed computational block finished, does not hide the reconfiguration time.

2.2.2 *Using and Partitioning the Reconfigurable Area*

Fine-grained reconfigurable systems can be partitioned into general frameworks and specific processors. Typically, they differ in the complexity of accelerators that can be reconfigured, i.e., the general frameworks often envision entire IP cores or tasks that are provided as hardware implementation, whereas specific processors typically provide reconfigurable functional units that can be reconfigured to contain an application-specific accelerator (presented in the next sections), similar to those that are created and used by nonreconfigurable ASIPs.

An example for a general framework for reconfigurable IP cores/tasks [UHGB04b] is shown in Fig. 2.5. It partitions the reconfigurable fabric into reconfigurable modules and connects them using a bus. An arbiter controls the communication of the modules with each other and with the run-time system that provides access to external periphery (e.g., a CAN interface) and manages the reconfiguration of the modules. A different approach is the Erlangen Slot Machine (ESM, [MTAB07]). As shown in Fig. 2.6, it provides different types of intermodule communication, ranging from dedicated communication to direct neighbors and shared SRAM to a reconfigurable bus system [ESS+96]. All modules have a dedicated

²Please note that 10 KB is a rather small amount of configuration data and 100 MB/s is a rather fast reconfiguration bandwidth, i.e., this indicates a lower limit.

³The other extreme indicating an upper limit.

Dedicated communication points. Run-time reconfigurable designs come with the conceptual problem that the synthesis and implementation tools (place and route) never see the entire design. This is because, in addition to the static parts of the design, some parts (the PRMs) are run-time reconfigurable and these parts are not necessarily known or fixed when the static part is synthesized and implemented. Therefore, the question arises how data can be exchanged between the static part and the PRMs. To solve this conceptual problem, dedicated communication interfaces (so-called bus macros) are placed at the borders between the static part and the PRMs. Then, the static design routes all communications to the bus macros, considering the content of the PRMs as a black box. Now, the designs for the PRMs can be synthesized and implemented independently, connecting their I/Os to the bus macros as well. The conceptual problem of establishing the communication limits the placement of the PRMs to those places, where bus macros are available and connected to the static part of the design.

Bitstream relocation. Considering that multiple PRMs are available and multiple IP cores/tasks shall be loaded into them, potentially multiple different partial bitstreams (i.e., the configuration bits to configure a PRM) need to be provided for each design that shall be loaded into any particular PRM. In the worst case, one bitstream needs to be provided for task_i that shall be loaded into PRM_i, another bitstream to load task_i into PRM₂, PRM₃, and so on. This potentially large amount of different bitstreams can be avoided when all PRMs have the same shape (i.e., outline and resource coverage). In this case, the concept of bitstream relocation can be used [BLC07, KLPR05]. Then, only one bitstream needs to be stored for each task and this bitstream can be adapted on the fly to be loaded into any particular PRM.

Writing the configuration data. To reconfigure a PRM, its configuration data needs to be changed. However, the configuration data is only accessible in so-called frames, i.e., a frame is reconfigured either entirely or not at all. For Virtex-II devices, a frame spans the whole height of the device and thus dictates that entire columns are reconfigured. That is also the reason why the PRMs in Figs. 2.5 and 2.6 correspond to horizontal slots. However, these technical constraints are specific to the particular FPGA family. For instance, in the Virtex-4 and the Virtex-5 FPGAs, the size of a frame is changed and it spans a height of 16 and 20 CLBs, respectively.⁴ Independent of the height of the frame, a PRM should always be adapted to it, i.e., it should have the same height. It is possible to reconfigure only parts of a frame by first reading the configuration data of the frame, then modifying the fraction that shall be reconfigured, and writing back the entire frame [HSKB06]. However, that increases the delay of the reconfiguration process significantly (for Virtex-II devices, the obtained flexibility may overcome the overhead). Therefore, this technical constraint also limits the shape of PRMs, namely to rectangular outlines that are aligned at frame and CLB borders.

Summarizing, the conceptual demand for communication points, the support for bitstream relocation, and the limited access to the reconfiguration data favor certain

⁴Multiple frames need to be reconfigured for a CLB column, i.e., the width of a frame does not cover a CLB.

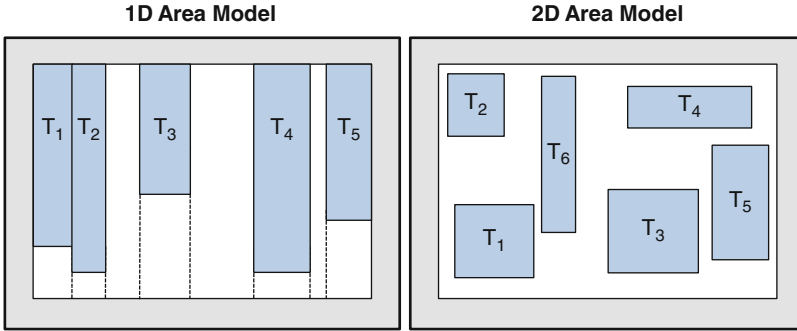


Fig. 2.7 Different area models for fine-grained reconfigurable fabric

types of designs, i.e., rectangular PRMs that are aligned to configuration frames and CLBs that have identical sizes and outlines and that provide dedicated communication points.

In addition to the full-height reconfigurable PRMs (i.e., a 1D placement of the PRMs), also a 2D placement [SWP04] is possible, when using a Virtex-4/5 FPGA or applying the read-modify-write technique from [HSKB06], as for instance demonstrated in [SKHB08]. Figure 2.7 illustrates the conceptual differences. The 2D area model is especially beneficial, if the designs that shall run on the reconfigurable fabric correspond to entire tasks (e.g., managed in [LP07]) and their hardware implementations are rather different in size. In such a scenario, it cannot be predetermined “where” a particular functionality will be placed on the FPGA. A run-time placement of tasks to an FPGA is presented in [WSP03]. A high fragmentation can lead to the undesirable situation that a task cannot be placed although there would be sufficient area available. Therefore, a service that organizes the placement of tasks (in order to avoid such situations) is investigated. The goals of a fast task placement with a high placement quality are achieved by the management of the free rectangles with a hash matrix. The hash matrix allows finding a suitable rectangle in constant time, at the cost of a subsequent updating phase of this data structure. A routing-conscious dynamic placement for reconfigurable devices is presented in [ABF+07] to reduce the communication latency. However, the actual implementation of the communication infrastructure is not addressed, i.e., how the static part can be connected to the task that is loaded into the dynamic part if this task may be placed anywhere in the dynamic part, irrespective of dedicated communication points.

One possibility to establish the communication in a 2D area model is using a network-on-chip (NoC) architecture that provides dedicated communication points at various places on the reconfigurable fabric, as proposed by DyNoC [BAM+05] shown in Fig. 2.8. When a PRM is reconfigured and it covers these communication points, then the corresponding router is disabled and an adaptive routing strategy is

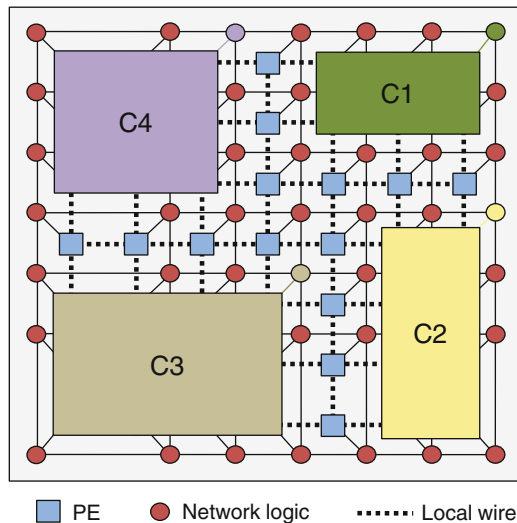


Fig. 2.8 2D area model using a dynamic network-on-chip architecture to establish communication

used to assure that the data packets still arrive at their destination.⁵ CoNoChi [PKAM09] proposes a NoC architecture that allows reconfiguring the structure of the NoC, i.e., it provides a concept and an implementation approach that allow adding or removing routers during run time. Instead of disabling routers, it allows to remove routers and change the connection topology of the network. For instance, on the right side of PRM C2 in Fig. 2.8, altogether four routers are placed. That means that each data packet that moves along that path demands four hops to traverse that distance. The concept of CoNoChi allows removing the intermediate routers and establishing a direct connection between the routers at the borders to improve the packet latency.

2.2.3 Coupling Accelerators and the Processor

As discussed in the previous section, the noticeable overhead of a flexible 2D area model is mainly beneficial if the tasks that shall be reconfigured differ in size significantly. NoCs can be used to connect the different PRMs. However, when targeting application-specific reconfigurable accelerators (similar to ASIPs), then a NoC is not necessarily the best choice to connect the accelerators with the core processor as the NoC introduces a nonlegible communication latency. Here, different alternatives exist to couple the core processor with the reconfigurable accelerator. Figure 2.9 shows conceptually different examples that vary in the amount of required processor modification and provided latency and bandwidth [CH02, TCW+05].

⁵This requires that around each PRM a ring of routers remains active.

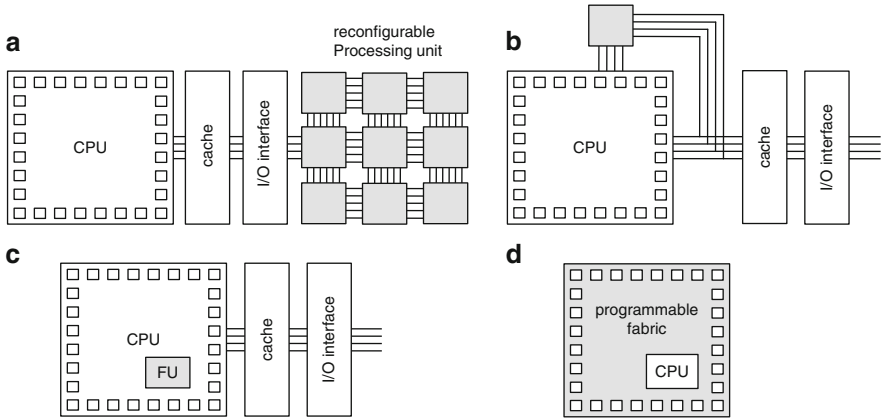


Fig. 2.9 Coupling reconfigurable accelerators with the core processor

Figure 2.9a shows a rather loose coupling, where the reconfigurable fabric is off-chip, attached via an I/O interface. The advantage is that it can be coupled to any existing chip that provides access to an I/O interface; however, the communication bandwidth and latency is limited. Therefore, a performance improvement can only be expected if a rather large amount of computation needs to be performed on a small amount of data, thus diminishing the effects of the limited communication performance. Figure 2.9b shows an on-chip connection between the reconfigurable fabric and a core processor. This connection may use a typical coprocessor interface for connection. The PRM may also be connected after the cache, depending on the provided interface of the targeted core processor. This also corresponds to the type of connection that a NoC communication infrastructure would use. The tightest coupling is shown in Fig. 2.9c. The reconfigurable fabric is embedded into the pipeline of the core processor as a reconfigurable functional unit. To establish this connection, it is required to change the processor-internal structure. The advantage is the communication bandwidth and latency, because direct connection to the register file is possible, i.e., multiple words can be read and written per cycle. This corresponds to the coupling that is used for ASIPs and typically, the amount of read/write ports of the register file is increased to extend the bandwidth even further. Figure 2.9d shows a combination that is often used for prototyping but may also be used as dedicated target architecture. An FPGA device is used as reconfigurable fabric and the core processor is implemented as nonreconfigurable part within the reconfigurable fabric (either as “hard” core or using the reconfigurable fabric).

For the architecture that is presented in this monograph, the tight coupling of Fig. 2.9c is targeted; however, for prototyping purpose the type of Fig. 2.9d is used. This tight coupling corresponds to the basic idea that reconfigurable accelerators shall be provided in a similar way like ASIPs use them. Therefore, the advantage of a rather simple area model with fixed communication points and similar-sized PRMs outpaces the rather complex 2D area model with its NoC communication paradigm. Still, a 2D placement of PRMs is used, but the communication points and PRM placements are fixed.

2.2.4 Reconfigurable Instruction Set Processors

In the scope of the work that is presented in this monograph, the focus is placed on extensions of CPUs with reconfigurable functional units (RFUs) that are tightly coupled to the core pipeline. An overview and classification for this specific area of reconfigurable computing is given in [BL00]. In this section, state-of-the-art approaches that are related to the reconfigurable processor that is presented in this monograph are presented and discussed.

Employing reconfigurable processors affects the application development in two different ways. First, the application-specific hardware accelerators need to be designed for the RFUs (typically in a hardware description language) and afterwards, these accelerators need to be operated efficiently. To obtain a benefit from the hardware accelerators, the application programmer has to insert special instructions (SIs) into the application (e.g., using inline assembly or compiler tools) as a way to access the accelerators.⁶ In a way, both steps are comparable to the development process of ASIPs, which allows that their concepts for tool flows and automatic development environments may be adapted for this class of reconfigurable processors. However, reconfigurable processors additionally need to determine which accelerators should be loaded into the RFU at which time during the application execution. This is often accomplished by so-called prefetching instructions [LH02] that trigger the upcoming reconfigurations.

The difference between ASIPs and reconfigurable processors becomes noticeable when executing SIs. An ASIP provides all SIs statically, typically using the same fabrication technology like the core pipeline. A reconfigurable processor instead uses a fine-grained reconfigurable fabric for the SIs. In addition, a reconfigurable processor may not have an SI available when it shall execute, e.g., because the process of reconfiguration has not finished yet. The partitioning of the reconfigurable fabric (e.g. how many SIs can be provided at the same time) has a high impact on whether or not an SI is available when it is demanded. Furthermore, architectural parameters (for instance the reconfiguration time) affect the ability to efficiently provide SIs on demand. All these aspects are common for the following state-of-the-art architectures (presented in chronological order); however, they address them using different techniques.

The *OneChip* project [WC96] and its successor OneChip98 [CC01, JC99] use RFUs within the core pipeline to utilize reconfigurable computing in a CPU. The RFUs are accessed by SIs as multicycle instructions. As their speedup is mainly obtained from streaming applications [CC01] they allow their RFUs to access the main memory, while the core pipeline continues executing. OneChip is based on an in-order

⁶Note that further access possibilities – e.g., memory-mapped interfaces or co-processor ports – may be used as well; however, for tightly coupled accelerators, the SIs provide a direct and low-latency interface as it is used by ASIPs.

pipeline, whereas OneChip98 uses an out-of-order superscalar architecture. Therefore, memory inconsistency problems may occur when an SI that accesses the memory executes on the RFU and – during the multicycle SI execution – load/store instructions execute in the core pipeline. For instance, the pipeline may load a value from memory, modify it, and write it back. If an SI writes the same address in between, then the core pipeline overwrites its result. In the scope of the OneChip98 project, nine different memory inconsistency problems were identified and hardware support was developed to resolve them automatically. The RFUs provide six configuration contexts and can switch between them rather fast. To load an RFU configuration into one of these contexts, the access to the main memory is used.

The *Chimaera* project [HFHK97, YMHB00] couples a dynamically scheduled superscalar processor core with a reconfigurable fabric that may contain multiple SIs. The SIs obtain up to nine inputs and create one output value directly from/to the register file (i.e., Chimaera uses a tight coupling). To avoid implementing a register file with nine read ports, these nine registers are realized as a partial physical copy of the register file that is updated when the corresponding registers in the register file are modified (so-called shadow registers). The register addresses are hard coded within the SI and the compiler/application programmer is responsible for placing the expected data in the corresponding registers. Chimaera uses a customized reconfigurable array that is based upon 4:1 LUTs that can also be used as two 3:1 LUTs or one 3:1 LUT with additional carry computation. A fast carry logic is shared among all logic blocks in a row and the routing structure of the array is optimized for arithmetic operations. When an SI demands the reconfigurable array but the required configuration is not available, then a trap is issued and the trap-handler performs the reconfiguration. Therefore, the application computation is stalled during the array reconfiguration and in the case of large working sets (i.e., many SIs within a loop), the problem of thrashing in the configuration array is reported (i.e., frequent reconfigurations within each loop iteration). Therefore, Chimaera provides a dedicated on-chip configuration cache to diminish the effects of this problem.

The *CoMPARE* processor [SGS98] provides a reconfigurable fabric that is coupled to the pipeline of the core processor similar to an ALU. The reconfigurable fabric implements an RFU that takes four inputs and returns two results. The RFU does not provide any registers or latches, i.e., it is stateless. Therefore, no state-machine or data feedback may be implemented. In addition, the RFU can only comprise the hardware implementation of a single SI at a time. If more SIs are demanded then the RFU needs to be reconfigured in between their executions.

The *Proteus* processor [Dal99, Dal03] extends a processor by a tightly coupled fine-grained reconfigurable array that is divided into multiple RFUs, where each RFU may contain one SI implementation at a time. Proteus concentrates on operating system support with respect to SI opcode management. This is especially beneficial when multiple tasks need the same SI, as they can share the same SI implementations without the need that the tasks were compiled such that they use the same opcode for the SI. However, when multiple tasks exhibit dissimilar processing characteristics, a task may not obtain a sufficient number of RFUs to execute all SIs in hardware. Therefore, some SIs will

execute in software, resulting in steep performance degradation. In addition to SI management, Proteus provides concepts for task switches that may occur during an SI execution, i.e., the context of the SI that currently executes needs to be stored, saved, and restored (when the task and thus the SI continues execution later on).

The *XiRisc* project [LTC+03, MBTC06] couples a VLIW processor with a reconfigurable gate array. The instruction set architecture of the VLIW processor is extended to provide two further instructions for reconfiguring the array and for execution SIs. The SIs have access to the register file and the main memory and may last multiple cycles. The configuration of an SI is selected out of four different contexts and reconfiguration between them can be done in a single cycle. These multiple contexts are beneficial if small applications fit into them. In [LTC+03] the fastest reported speedup (13.5 $\times$) is achieved for DES and the only context reloading happened when the application was started. However, in [MBTC06] a relevant MPEG-2 encoder is used for benchmarking. Here, run-time reconfiguration is required (as the accelerators no longer fit into the available contexts) and the achieved speedup reduced to 5 $\times$ compared to the corresponding processor without reconfigurable hardware (i.e., GPP).

The *Molen* processor [PBV06, PBV07, VWG+04] couples a reconfigurable fabric to a core processor via a dual port register file and an arbiter for shared memory. The core processor is extended to provide additional instructions to manage the reconfigurations and SI executions. The instruction set extension can be parameterized to support only one or multiple SIs at a time, i.e., they provide instructions that allow to perform partial reconfiguration, but not all Molen instances necessarily use them (i.e., they may restrict to instructions that reconfigure the entire reconfigurable fabric). The run-time reconfiguration is explicitly predetermined by these control instructions that also allow configuration prefetching. The compiler/application developer is responsible for deciding which configuration shall be loaded at which time. All SIs are executed by a generic instruction that obtains the address of the configuration bitstream as parameter to identify the demanded SI. Therefore, the amount of SIs is not limited by the number of available opcodes of any particular instruction set. To provide parameters to the SIs, data has to be explicitly moved to dedicated exchange registers that couple the reconfigurable fabric to the core pipeline.

The *Warp* processor [LSV06, LV04] automatically detects kernels (using online monitoring) while the application executes. Then, custom logic for these kernels is generated at run time through on-chip micro-CAD tools (i.e., synthesis, place, and route). These kernel implementations are then called from the application. However, as the application does not explicitly contain SIs in the binary (as the kernel was not determined at compile time yet), Warp proposes a different approach. They replace parts of the application binary (that correspond to the kernel that shall execute using the reconfigurable fabric) by a hardware initialization code. This code provides input data to the kernel implementation, starts its execution, and stalls the processor. After the kernel execution finished, the processor receives an interrupt and continues where it stopped, i.e., in the hardware initialization code. Directly following that code, a jump instruction is placed to continue the application execution with the code

that follows the kernel. Warp proposes a specialized fine-grained reconfigurable fabric that trades the performance of the reconfigurable fabric with the computational complexity for the on-chip micro-CAD tools.⁷ They focus on the memory requirements of the CAD tools, as this is a limited resource in embedded systems. Still, the online synthesis incurs a nonnegligible computation time and therefore the authors concentrate on scenarios where one application is executing for a rather long time without significant variation of the execution pattern. In these scenarios, only one online synthesis is required (i.e., when the application starts executing) and thus the initial performance degradation amortizes over time.

2.3 Summary of Related Work

Application-specific instruction set processors (ASIPs) allow accelerating applications by providing dedicated hardware accelerators. Their execution is triggered by special instructions (SIs). Research in the scope of fine-grained reconfigurable processors focused on connecting a core processor with an FPGA-like reconfigurable fabric on which SI implementations are dynamically loaded during run time. The presented state-of-the-art approaches mainly concentrated on offering and interfacing SIs in their reconfigurable fabric. Some of them only provide one SI at a time and some support multiple SIs that are available on the reconfigurable fabric at the same time. Still, all of the above-discussed approaches potentially increase the utilization of the available hardware resources by reconfiguring parts of it to match the current requirements of the application (i.e., the currently demanded SIs). However, due to the reconfiguration time the utilization of the reconfigurable fabric may often be suboptimal.

All presented approaches only consider one predetermined implementation per SI. The Proteus reconfigurable processor additionally offers the execution of an SI with the cISA, but the operating system predetermines which SI shall be executed with reconfigurable hardware and which with the cISA. Still, there is at most one hardware implementation per SI. The Warp processor could potentially change the SI hardware implementations by performing further online syntheses. However, due to the implied overhead, synthesizing SI implementations is typically only done once after the application started, which finally leads to a single SI implementation as well.

In addition, state-of-the-art fine-grained reconfigurable processors mainly focus on compile-time predefined reconfiguration decisions. Often, dedicated commands have to be inserted into the application binary by the compiler (or the application programmer) that explicitly determine “when” the reconfiguration of the hardware implementation of “which” SI shall be started. This is not suitable when computational requirements or constraints are unpredictable during compile time and change during run time.

⁷ A more regular reconfigurable fabric with limited interconnect possibilities simplifies place and route at the cost of reduced SI performance.

These and similar concerns are addressed by the paradigm of the rotating instruction set processing platform (RISPP) that is presented in this monograph and that fixes some customizations during compile time and is able to adapt dynamically during run time. Therefore, this monograph presents the novel concept of modular SIs that allows providing multiple hardware implementations per SI. A concept plus an infrastructure for modular SI is developed that provides the feature to upgrade dynamically from one implementation of an SI to another. This alleviates the problem of long reconfiguration time whenever prefetching is infeasible and additionally offers high flexibility to adapt to different requirements (e.g., by deciding which SI shall be upgraded further) that depend on the run-time situation. Furthermore, this monograph presents a novel run-time system that uses the modular SIs to enable an adaptive and efficient utilization of the available hardware without statically predetermined reconfiguration decisions. The detailed problems of state-of-the-art and the concept and basics of modular SIs are explained in Chap. 3, whereas the algorithms of the novel run-time system are presented in Chap. 4. Chapter 5 presents the hardware architecture and its implementation whereas Chap. 6 provides overall benchmarks and comparison to state-of-the-art approaches, i.e. ASIPs and reconfigurable processors.



<http://www.springer.com/978-1-4419-7411-2>

Run-time Adaptation for Reconfigurable Embedded
Processors

Bauer, L.; Henkel, J.

2011, XXI, 223 p., Hardcover

ISBN: 978-1-4419-7411-2