

## Research Article

# Design Flow Instantiation for Run-Time Reconfigurable Systems: A Case Study

Yang Qu,<sup>1</sup> Kari Tiensyrjä,<sup>1</sup> Juha-Pekka Soininen,<sup>1</sup> and Jari Nurmi<sup>2</sup>

<sup>1</sup> Communication Platforms, Technical Research Centre of Finland (VTT), Kaitoväylä 1, 90571 Oulu, Finland

<sup>2</sup> Institute of Digital and Computer Systems, Tampere University of Technology, Korkeakoulunkatu 1, 33720 Tampere, Finland

Correspondence should be addressed to Yang Qu, yang.qu@vtt.fi

Received 25 May 2007; Revised 28 September 2007; Accepted 12 November 2007

Recommended by Donatella Sciuto

Reconfigurable system is a promising alternative to deliver both flexibility and performance at the same time. New reconfigurable technologies and technology-dependent tools have been developed, but a complete overview of the whole design flow for run-time reconfigurable systems is missing. In this work, we present a design flow instantiation for such systems using a real-life application. The design flow is roughly divided into two parts: system level and implementation. At system level, our supports for hardware resource estimation and performance evaluation are applied. At implementation level, technology-dependent tools are used to realize the run-time reconfiguration. The design case is part of a WCDMA decoder on a commercially available reconfigurable platform. The results show that using run-time reconfiguration can save over 40% area when compared to a functionally equivalent fixed system and achieve 30 times speedup in processing time when compared to a functionally equivalent pure software design.

Copyright © 2008 Yang Qu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

## 1. INTRODUCTION

Reconfigurability is an important issue in the design of system-on-chip (SoC) because of the increasing demands on silicon reuse, product upgrade after shipment, and bug-fixing ability. The reconfigurability is usually achieved by embedding reconfigurable hardware into the system. The result is a heterogeneous SoC that has the advantages of both reconfigurable hardware and traditional types of computing elements such as general-purpose processors (GPP) and application-specific integrated circuit (ASIC). Such combination allows parts of the system to be reconfigured at run time while the rest is still running. This feature is referred to as run-time reconfiguration (RTR), which can significantly increase silicon reusability.

As today's applications become more and more complex, the implementation needs more hardware resources. It means that either larger chips or more chips are needed, which might not be suitable for many products such as portable devices that require to have a limited dimension. With RTR, tasks that are nonoverlapping either in time domain or space domain can be mapped onto the same reconfigurable logic. Tasks that are required initially can be con-

figured in the beginning. When another task is required, the configuration to load it can then be triggered. For example, in a typical smartphone environment, different wireless technologies, such as GSM, WCDMA, WLAN, and WiMax in the future, have to be supported. However, it is not likely that these wireless technologies will be used at the same time. Therefore, it is possible to put them into reconfigurable logic and dynamically load the one that is needed.

A number of reconfigurable platforms are commercially available. Xilinx [1] and Altera [2] provide fine-grain FPGA platforms. They contain embedded processor cores, which make it possible to design rather complex systems in such FPGA platforms. PACT XPP [3] and QuickSilver [4] provide coarse-grain reconfigurable computing platforms, which are suitable for DSP-type tasks. The Triscend A7S [5] and the Motorola MRC6011 [6] are configurable SoCs, which bring both high flexibility and high performance.

The drawbacks of the RTR are configuration latency and power consumption related to the configuration process, which can largely degrade the system performance. How to address these problems and evaluate the effects of reconfiguration at an early phase of the design are not supported in existing system-level design methodologies and

tools. In addition, at system level, how to support and make system partitioning for not only software and hardware, but also reconfigurable logic, needs to be studied.

The ultimate goal of our work is to develop a complete design methodology and highly automatic tools for design of reconfigurable SoC (RSoC). In this paper, we present a design flow instantiation for implementing part of a WCDMA decoder in an RTR system. At system level, our supports for system partitioning and performance evaluation are applied [7, 8]. At implementation level, commercial and technology-dependent tools are applied. The structure of the paper is as follows. Related work is presented in Section 2. Brief explanations of the case study and the target platform are given in Section 3. The system-level design flow instantiation is presented in Section 4, and low-level implementation work is presented in Section 5. Finally, the conclusions are given in Section 6.

## 2. RELATED WORK

System-level design covers various issues, such as partitioning, task scheduling, and synthesis. In [9], an SW/HW partitioning and online task scheduling approach are presented. In [10], a survey of various SW/HW partitioning algorithms is presented, and a new approach to map loops into reconfigurable hardware is proposed. In [11], a codesign environment for DSPs/FPGAs-based reconfigurable platforms is presented. Both applications and architectures are modeled as graphs, and an academic system-level CAD tool is used. In [12], a macro-based compilation framework to replace logic synthesis and technology mapping is presented. In [13], a synthesis approach based on list-scheduling is presented. The target system is a single application that can be divided into a number of dependent tasks. The approach considers HW/SW partitioning, temporal partitioning, as well as context scheduling. In [14, 15], an HW/SW cosynthesis framework for real-time embedded reconfigurable system is presented. Each application consists of a number of dependent tasks and has its own period. A task can be mapped either onto a host processor or dynamically reconfigurable hardware (DRHW). Synthesis is performed by statically scheduling the applications over a hyperperiod, which is the least common multiple of the different application periods. In [16], a SystemC simulation environment for RTR systems is presented. The idea is to remove the unloaded module from the activation list of the SystemC kernel. In [17], various system-level approaches to reduce the effect of configuration latency are studied.

Different from these approaches, our ultimate goal is not to develop a fully automatic system partitioning approach, which we believe will not succeed. This is because nowadays' applications and platforms are becoming so complex that it is not possible to quantitatively characterize them precisely in the early design phase so that complex mathematical formulas can be applied to fully partition the design in such a way that optimal solutions can be guaranteed. However, providing supports to designers at this phase can help to prune the design space and possibly avoid re-designs. In our work, approaches to support partitioning and modeling for fast de-

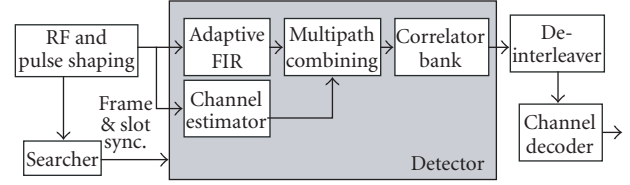


FIGURE 1: The WCDMA base-band receiver system.

sign space exploration are provided. To reduce coding effort, a tool to automatically generate reconfigurable components is developed.

## 3. APPLICATION AND TARGET PLATFORM

Reconfigurable system is a promising alternative to deliver both flexibility and performance at the same time. Technology-dependent tools and high-level abstract supporting tools have been proposed to solve the various design problems at different abstraction levels. However, a complete overview of how to integrate them into a single design flow is missing. In this work, we use a real case study to demonstrate our design flow of RTR systems. The design case is a set of wireless communication functions [18], and the target is a RTR-type implementation on VP20FF1152 development board from Memec Design group [19], which contains one Virtex2P XC2VP20 FPGA [1] that supports partial RTR.

The whole WCDMA base-band receiver system is depicted in Figure 1. The case study focuses on the detector portion (shaded area in Figure 1) of the receiver and a limited set of the full features were taken into account. It uses 384 kbits/s user data rate without handover. The functions are an adaptive filter, a channel estimator, a multipath combiner, and a correlator bank. The adaptive filter is performing the signal whitening and part of the matched filtering implemented traditionally with the RAKE receiver. The channel estimator module calculates the phase references. In the combiner part, the different multipath chip samples are phase rotated according to the reference samples and combined. Finally, the received signal is despread in the correlator bank.

## 4. SYSTEM-LEVEL DESIGN FLOW AND INSTANTIATION STEPS

Our design flow is divided into system-level design and implementation-level design. The task at system-level design is to make various partitioning decisions and evaluate system performance. At implementation level, executable code and HW netlist are generated using technology-dependent tools. A generic view of the system-level design flow is depicted in Figure 2. The following new features are identified in each phase when reconfigurability is taken into account.

- (i) *System requirements/specification capture* needs to identify requirements and goals of reconfigurability (e.g., flexibility for specification changes and performance scalability).

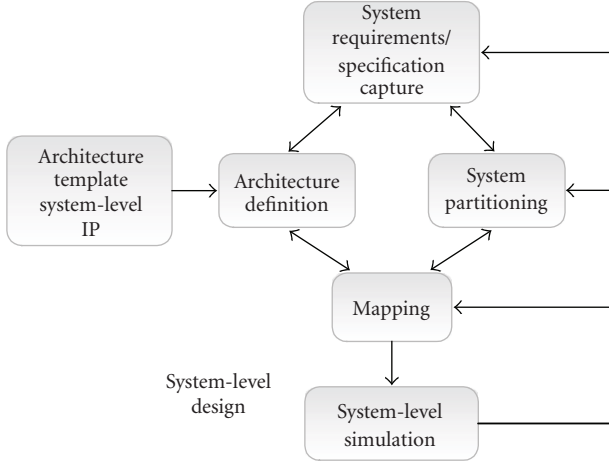


FIGURE 2: A generic system-level design flow.

- (ii) *Architecture definition* needs to model the reconfigurable technologies of different types and vendors at abstract level and include them in the architecture models.
- (iii) *System partitioning* needs to analyze and estimate the functions of the application for SW, fixed HW, and reconfigurable HW. The parts of the targeted system that will be realized on reconfigurable HW must be identified. There are some rules of thumb that can be applied. If an application has roughly several same-sized hardware accelerators that are not used at the same time, these accelerators can be implemented onto DRHW. If an application has some parts in which specification changes are foreseeable or there are foreseeable plans for new generations of the applications, it may be beneficial to implement it onto DRHW.
- (iv) *Mapping* needs to map functions allocated to *reconfigurable* hardware onto the respective architecture model. The special concerns at this step are the temporal allocation and the scheduling problem. Allocation and scheduling algorithms need to be made either online or offline.
- (v) *System-level simulation* needs to observe the performance impacts using reconfigurable resources for a particular system function. The effect of configuration overhead should be highlighted in order to support designers to perform system analysis or design space exploration.

It should be noted that reconfigurability does not appear as an isolated phenomenon, but as a tightly connected part of the overall SoC design flow. Our approach is therefore not intended to be a universal solution to support the design of any reconfigurability. Instead, we focus on a case, where the reconfigurable components are mainly used as coprocessors in SoCs. In addition, we assume that RTR system design does not start from scratch, but it is a more advanced version of an existing device. The new architecture is defined partly based on the existing architecture and partly using the system specification as input. The initial architec-

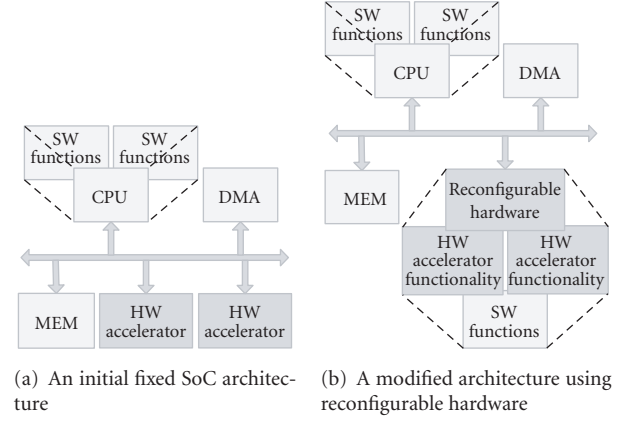


FIGURE 3: Creating RSoC from fixed platform.

ture is often dependent on many things not directly resulting from the requirements of the application. The company may have experience and tools for certain processor core or semiconductor technology, which restricts the design space and may produce an initial hardware/software (HW/SW) partition. Therefore, the initial architecture and the HW/SW partition are often given at the beginning of system-level design.

SystemC 2.0 is selected as the backbone of the approach since it provides a universal environment to model HW/SW and seamlessly cosimulate them at different abstract levels. The way that the SystemC-based approach incorporates dynamically reconfigurable parts into architecture is to replace SystemC models of some hardware accelerators, as shown in Figure 3(a), with a single SystemC model of reconfigurable block, as shown in Figure 3(b). The objective of the SystemC-based extensions is to provide a mechanism that allows designers to easily test the effects of implementing some components in DRHW. Referring to the system-level design flow, as shown in Figure 2, we provide estimation support for system partitioning and modeling support for system-level simulation.

#### 4.1. Estimation approach to support system analysis

The estimation approach [20] is used to support system analysis to identify functions that are beneficial to be implemented in DRHW. It focuses on VirtexII-like FPGA [1] DRHW, in which the main resources are lookup-tables (LUTs) and multipliers. The estimation approach starts from function blocks represented using C language, and it can produce the following estimates for each function block: execution time in terms of running the function on DRHW and resource utilization of DRHW. The framework of the estimation approach is shown in Figure 4. The designer decides the granularity of partitioning by decomposing the algorithm down to function blocks. The estimation tool produces the estimates for each of the functions.

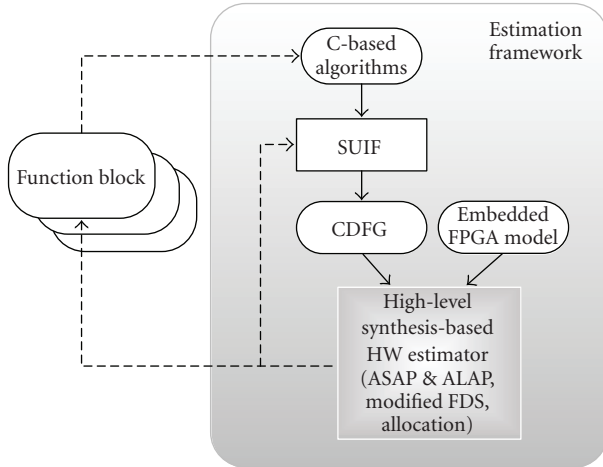


FIGURE 4: The estimation framework.

#### 4.1.1. High-level synthesis-based hardware estimation

A graphical view of the hardware estimation is shown in Figure 4. Taking control-data flow graph (CDFG) and a model of embedded FPGA as inputs, the hardware estimator carries out a high-level synthesis-based approach to produce the estimates. Main tasks performed in the hardware estimator as well as in a real high-level synthesis tool are scheduling and allocation. Scheduling is the process in which each operator is scheduled in a certain control step, which is usually a single clock cycle, or crossing several control steps if it is a multi-cycle operator. Allocation is the process in which each representative in the CDFG is mapped to a physical unit, for example, variables to registers, and the interconnection of physical units is established.

In the estimator, a function block is represented as a CDFG, which is a combined representation of data flow graph (DFG) that exposes the data dependence and parallelism of algorithms, and control flow graph (CFG) that captures the control relation of a group of DFGs. A SUIF-based [21] front-end pre-processor is used to extract CDFG from the C code. First it dismantles all high-level loops (e.g., while loop and for loop) into low-level jump statements and restrict the produced code to minimize the number of jumps. Then, basic blocks are extracted. A basic block contains only sequential statements without any jump in between. Data dependence inside each basic block is analyzed, and a DFG is generated for each basic block. After the creation of all DFGs, the control dependence between these DFGs is extracted from the jump statements to construct the CDFG. Finally, profiling results, which are derived using gcov [22], are inserted into the CDFG as attributes.

The basic construction units of the embedded FPGA are static random access memory (SRAM)-based lookup tables (LUT) and certain types of specialized function units, for example, custom-designed multiplier. Routing resources and their capacity are not taken into account. The model of the embedded FPGA is in a form of mapping-table. The index of the table is the type of the function unit, for example, adder. The value mapped to each index is hardware resources in

terms of the number of LUTs and the number of specialized units for this type of operation.

As-soon-as-possible (ASAP) scheduling and as-late-as-possible (ALAP) scheduling [23] determine the critical paths of the DFGs, which together with the control relation of the CFGs are used to produce the estimate of hardware execution time. For each operator, the ASAP and ALAP scheduling processes also set the range of clock cycles within which it could be legally scheduled without delaying the critical path. These results are required in the next scheduling process, a modified version of force-directed-scheduling (FDS) [24], which intends to reduce the number of function units, registers, and buses required by balancing the concurrency of the operations assigned to them without lengthening the total execution time. The modified FDS is used to estimate the hardware resources required for function units.

Finally, allocation is used to estimate the hardware resources required for interconnection of function units. The work of allocation is divided into 3 parts: register allocation, operation assignment, and interconnection binding. In register allocation, each variable is assigned to a certain register. In operation assignment, each operator is assigned to a certain function unit. Both are solved using the weighted-bipartite algorithm, and the common objective is that each assignment should introduce the least number of interconnection units that will be determined in the last phase, the interconnection binding. In this approach, multiplexer is the only type of interconnection unit, which eases the work of interconnection binding. The number and type of multiplexers are determined by simply counting the number of different inputs to each register and each function unit. After allocation, the clock frequency is determined by searching for the longest path between two registers. Because routing resources are not modeled, the delay takes into account only the function units and the multiplexers.

We assume that all variables have the same size, since our goal is to quickly produce estimates with pure ANSI-C code instead of generating optimal synthesizable RTL code, which often uses some kinds of subset C code and applies special meanings to variables. Our estimation framework also supports to explore parallelism for loops. This is done at the SUIF-level, where we provide a module that allows designers to perform loop unrolling (loops must have fixed number of iterations) and loop merging (loops must have the same number of iterations).

#### 4.1.2. Instantiation for the case study

For the case study, we started with C-representation of the system. It contained a main control function and the four computational tasks, which lead to a simple system partition that the control function was mapped onto SW and the rest onto RTR hardware. The estimation tool was used first to produce the resource estimates. The results are listed in Table 1, where LUT stands for lookup table and register refers to word-wide storages. The multiplexer refers to the hard-wired  $18 \times 18$  bits multipliers embedded in the target FPGA.

Based on the resource estimates, the dynamic context partitioning was done as follows. The channel estimator was



TABLE 1: Estimates of FPGA resources required by the function blocks.

Functions	LUT	Multiplier	Register
Adaptive filter	1078	8	91
Channel estimator	1387	0	84
Combiner	463	4	32
Correlator	287	0	17
Total	3215	12	224

assigned to one context (1387 LUTs), and the other three processing blocks were assigned to a second context (1078 + 463 + 287 = 1828 LUTs). This partition resulted in both balanced resource utilization and less interface complexity compared to other alternatives. In implementation phase, both contexts are mapped onto the same region of the target FPGA, and the system dynamically loads the one that is needed.

#### 4.2. Modeling of DRHW and the supporting transformation tool

The modeling of reconfiguration overhead is divided into two steps. In the first step, different technology-dependent features are mapped onto a set of parameters, which are the size of the configuration data, the clock speed of configuration process, and the extra delays apart from the loading of the configuration data. Thus, by tuning the parameters, designers can easily evaluate the tradeoffs between different technologies without going into implementation details. In the second step, a parameterized SystemC module that models the behavior of run-time reconfiguration process is created. It has the ability to automatically capture the reconfiguration request and present the reconfiguration overhead during performance simulation. Thus, designers can easily evaluate the tradeoffs between different technologies by tuning the parameters.

##### 4.2.1. DRHW modeling approach

We model DRHW in such a way that the component can automatically capture reconfiguration requests during simulation and trigger reconfigurations when needed. In addition, a tool to automate the process that replaces some existing SystemC models by a DRHW SystemC model is developed, so system designers can easily perform the test-and-try and thus the design space exploration process is easier. In order to let the DRHW component be able to capture and understand incoming messages, SystemC modules must implement predefined but interface methods, such as `read()`, `write()`, `get_low_addr()`, and `get_high_addr()`. With the forthcoming SystemC TLM 2.0 standard [25], new interface methods could be defined to comply with the TLM 2.0. Equivalently, OCP standard transaction-level interfaces [26] can be used.

A general model of RSoC is shown in Figure 5. The left-hand side depicts the architecture of the RSoC. The right-hand side shows the internal structure of the DRHW component. It is a single hierarchical SystemC module, which

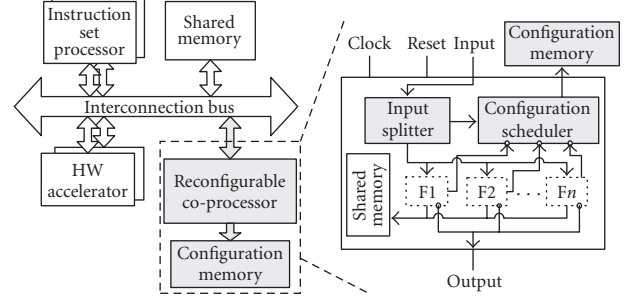
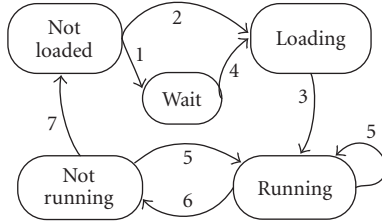


FIGURE 5: A generic model of RSoC.

implements the same bus interfaces as other HW/SW modules do. A configuration memory is modeled, which could be an on-chip or off-chip memory that holds the configuration data. Functions mapped onto DRHW ( $F1$  to  $F_n$ ) are individual SystemC modules that implement the predefined bus interfaces with separate system address space. The input splitter (IS) is an address decoder and it manages all incoming interface-method-calls (IMCs). The configuration scheduler (CS) monitors the status of the mapped function and controls reconfiguration processes.

The DRHW component works as follows. When the IS captures an IMC, it will hold the IMC and pass the control to the CS, which decides if reconfiguration is needed. If so and the CS detects the DRHW is free to use, it issues a reconfiguration that uses the technology-dependent parameters to generate the memory traffic and the associated delays to mimic the reconfiguration latency. If the CS detects the DRHW is loaded with another module, a request to reconfigure the target module will be put into a FIFO queue and the reconfiguration will be started after the DRHW has no running module. After finishing the reconfiguration, the IS will dispatch the IMC to the target module. This is a generic description of the context switching process, and designers can develop different CS models when different types of RTR hardware are used such as partial reconfiguration or multi-context device. In our approach, context switching with preemption is not supported because of its high implementation cost in DRHW.

There is a state diagram common to each of the mapped function modules. Based on the state information, the CS makes reconfiguration decisions for all incoming IMCs and DONE signals. A state diagram of partial reconfiguration is presented in Figure 6. For single context and multicontext reconfigurable resources, similar state diagrams can be used in the model. In fact, different techniques for reducing the effect of the configuration latency can be applied, for example, configuration prefetching [27]. The idea is to load a module before it is needed. In the state diagram, this can be achieved by enabling the branch 2 when the module is known to be executed soon, so the module can be loaded before an IMC to it is issued. However, prefetching conditions should be decided at design time and stored in a table, which can be accessed by the CS at run-time.



State definitions:

Not loaded: module is only in the configuration memory

Loading: module is being loaded

Wait: module is waiting in a FIFO queue to be loaded

Running: module is running

Not running: module is loaded, but not running

State transition conditions (\*) for configuration prefetching

1. IMC to the module occurs & not enough resources
2. (IMC to the module occurs & enough resources)|  
(The module is to be used soon & enough resources)\*
3. CS finishes the loading
4. Other modules finish & enough resources
5. IMC to the module occurs
6. Module finishes
7. CS flushes the module

FIGURE 6: State diagram of functions.

#### 4.2.2. An automatic code transformer for DRHW component

In order to reduce the coding effort, we have developed a tool that can automatically transform SystemC modules of the function blocks into a DRHW component. The inputs are SystemC files of a static architecture and a script file, which specifies the names of the mapped functions and the associated design parameters such as configuration latency. The tool contains a hard-coded DRHW template. It first parses the input SystemC code to locate the declarations of the candidate components (the C++ parser is based on opencxx [28]). Then the tool creates a DRHW component by filling the DRHW template with the declarations and establishing appropriate connections between the CS, the IS, and the functions. Finally, in the top-level structure, original SystemC modules of the mapped functions are replaced with the generated DRHW component. During simulation, reconfigurations will be automatically monitored and saved in a text file for analysis. A value change dump (VCD) file will also be produced to visualize the reconfiguration effects.

#### 4.2.3. Instantiation for the case study

For the case study, we first created a SystemC model of a fixed system, which had two purposes in the design. The first was to use its simulation results as reference data, so the data collected from the reconfigurable system could be evaluated. The second purpose was to automatically generate the reconfigurable system model from it via the transformation tool.

In the fixed system, each of the four WCDMA decoding functions was mapped to an individual hardware accelerator, and pipelined processing was used to increase the per-

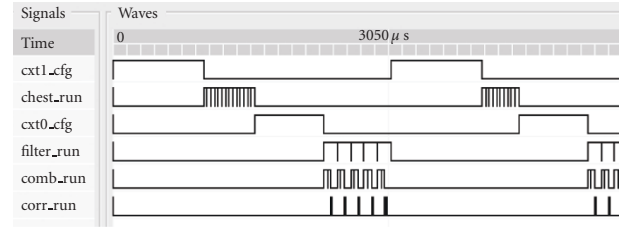


FIGURE 7: Simulation waveform shows the reconfiguration latency.

formance. A small system bus was modeled to connect all of the processing units and storage elements. The channel data used in the simulation was recorded in text files, and the processor drove a slave I/O module to read the data from the file. The SystemC models were described at transaction level, in which the workload was derived based on the estimation results but with manual adjustment. The results showed that 1.12 milliseconds were required for decoding all 2560 chips of a slot when the system was running at 100 MHz.

The transformation tool was then used to automatically generate the reconfigurable system model from the fixed model. The reconfiguration latency of the two dynamic contexts was derived based on the assumption that the size of the configuration data was proportional to the resource utilization, the number of LUTs required. The total available LUTs and size of full bitstream were taken from the Xilinx XC2VP20 datasheet. In the future, accurate approaches to derive the reconfiguration latency will be studied.

The performance simulation showed that the system required two reconfigurations per processing one slot of data. This is presented by the cxt0\_cfg and cxt1\_cfg in Figure 7. When the configuration clock was running at 33 MHz and the configuration bit width was 16, the reconfiguration latency was 2.73 milliseconds and the solution was capable of processing 3 slots of data in a frame.

## 5. LOW-LEVEL IMPLEMENTATION

The task at low-level implementation is to generate C code for SW, RTL-VHDL code for HW, and further generate executable binary code and netlist. For the HW part, there are commercial high-level synthesis tools that could be used to reduce the design time. However, considering the cost of such tools and the fact that the four WCDMA decoding functions can be implemented straightforward, we manually generated synthesizable RTL code for HW implementation. Simulation of the reconfigurable system was also performed at the RTL level by using the dynamic circuit switching (DCS)-based technique [29]. Multiplexers and selectors are inserted after the outputs of the modules and before the inputs of the modules. They are automatically switched on or off according to the configuration status. In the cycle-accurate simulation model, the reconfiguration is modeled as pure delay. For implementing the RTR, technology-dependent tools were used. Reconfigurations are triggered and managed by the main controlling SW task. The reconfiguration is implemented using the SystemACE compact flash (CF) solution

and the configuration data is stored in a CF card. A simple device driver that controls the SystemACE module is developed and the reconfiguration request is implemented as function calls to the SystemACE device driver.

### 5.1. Detailed design and implementation

In the low-level design phase, the main controlling SW task is mapped onto the embedded PowerPC core in the target FPGA, and the data memories are mapped onto the embedded block memories. Other components are mapped onto Xilinx IP cores, if corresponding matches can be found, for example, the bus model to the Xilinx processor local bus (PLB) IP core. In addition to the basic functionality, we added a few peripherals for debugging and visualization. Vendor-specific tools were used in the system refinement and implementation phases. Other than the traditional design steps for HW/SW implementation, additional steps for interface refinement, configuration design, and partially reconfigurable module (PRM) design were needed.

### 5.2. Interface refinement

The number of signals crossing the dynamic region and the static region must be fixed, since the dynamic region cannot adapt itself for changing the number of wires. In our work, the step to define the common set of boundary signals shared between the PRMs is referred to as interface refinement. In Xilinx FPGAs, the boundary signals are implemented as bus macros [30], which are prerouted hard macros used to specify the exact routing channels and will not change when modules are reconfigured. Because each bus macro is defined to hold 4 signals and there are only a limited number of bus macros, the boundary signals cannot be oversized. Therefore, it is more beneficial to minimize the number of signals crossing the dynamic region and the static region, which can also relax the constraint during placement and routing. In this case study, the number of boundary signals is reduced to 82, which correspond to the signals connected to the two 16-bit dual-port data memories and the PLB bus adapter. 21 bus macros are needed.

### 5.3. Partial reconfigurable module design

Synthesis results of the four functions are listed in Table 2. When considering the estimation, the results are overestimated at about 55% in average. The main reasons are that: (1) the estimator assumes fixed-length computation for all variables, (2) the estimator maps all multiplexers directly to LUTs but real synthesis tools usually utilize the internal multiplexers in individual logic elements. For the PRM, the Xilinx module-based partial reconfiguration design flow [30] was used. First, each of the four detector functions was implemented as a single block. Then a context wrapper that matches the boundary signals was generated to wrap the channel estimator as one module and the other three as another module. The static part was assigned to the right side of the FPGA, because most used IO pads were in the right side. The dynamic region was in the left side of the FPGA.

TABLE 2: HW synthesis results.

Functions	LUT	Multiplier	Register (bits)
Adaptive filter	553	8	1457
Channel estimator	920	0	2078
Combiner	364	4	346
Correlator	239	0	92

The size of the configuration data was 279 KB for the context 1 and 280 KB for the context 2.

Routed PRMs on the dynamic region are shown in Figure 8. The context 1 that contains the channel estimator is shown in Figure 8(a), and the context 2 that contains the other three modules is shown in Figure 8(b). In addition, a routed design after module assembly is shown in Figure 8(c), which is the integration of the context 2 and the static part. The bus macros that are used for providing reliable connections for the boundary signals are marked by the block in the middle.

### 5.4. Comparison with other approaches

In addition to the RTR implementation, a fixed hardware implementation and a pure software implementation were made as reference designs. In the fixed-hardware implementation, the processing blocks were mapped onto static accelerators and the scheduling task was mapped onto SW that ran on the PowerPC core. The resource requirements were 4632 LUTs (24% of available resources), 55 Block RAMs (62%) and 12 Block Multipliers (13%). The system was running at 100 MHz. The execution time for processing one slot of data was 1.06 ms. Compared to the fixed reference system, the dynamic approach achieved over 40% resource reduction in terms of the number of LUTs, but at the cost of 8 times longer processing time.

For the full software implementation, the design was done as a standalone approach and no operating system was involved. Everything was running in a single PPC core and data were entirely stored in internal Block RAMs. For the same clock frequency, the processing time of one slot of data was 294.6 milliseconds, which was over 30 times of the processing time in run-time reconfiguration case. This did not fulfill the real-time requirements.

## 6. CONCLUSIONS

The main advantage of RTR systems is the combined flexibility and performance. However, implementing RTR does require extra efforts in the various design stages, from the abstract system-level down to the timing-accurate circuit-level. In this work, we present a design flow instantiation for RTR systems. It combines design supports at system level for design partitioning and system modeling to evaluate the effect of reconfiguration overhead. In implementation level, commercial and technology-dependent tools are applied. A set of wireless communication functions is used in the case study. Compared to a completely fixed implementation, the reduction of LUTs is over 40%. Compared to a full software



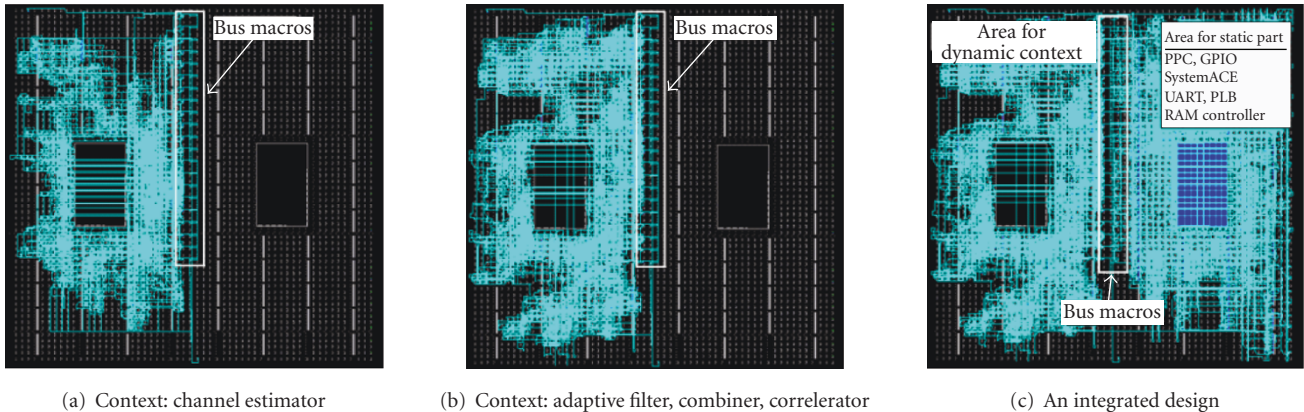


FIGURE 8: Routed design of PRM on the dynamic region.

implementation, the run-time reconfiguration approach is over 30 times faster. The commercial off-the-shelf FPGA platform caused limitations on the implementation of run-time reconfiguration. Although the selected approach used partial reconfiguration, the required configuration time affected the performance a lot in the data-flow type WCDMA design case. The ratio of computing to configuration time was about 1/8 in this design case. The results clearly show that the configuration overhead is nowadays the main bottleneck of RTR systems. In the future, techniques to reduce its effect will be studied. In addition, improvements of our system-level design supporting tools are needed, such as power analysis and more accurate HW resource estimation approach, which will be studied.

## ACKNOWLEDGMENTS

This work was previously supported by the European Commission under the Contract IST-2000-30049 ADRIATIC, and later by Tekes (National Technology Agency of Finland) and VTT under EUREKA/ITEA Contract 04006 MARTES.

## REFERENCES

- [1] Xilinx, "Virtex platform datasheet," May 2007, <http://www.xilinx.com>.
- [2] Altera, "Stratix platform datasheet," May 2007, <http://www.altera.com>.
- [3] PACT XPP technologies, "XPP performance media processor datasheet," May 2007, <http://www.pactxpp.com>.
- [4] QuickSilver Technologies, "Adapt2000 ACM system platform overview," May 2007, <http://www.qstech.com>.
- [5] Triscend, "A7 field configurable system-on-chip datasheets," 2004, <http://www.triscend.com>.
- [6] Motorola, "Press release of MRC6011 RCF device," 2003, <http://www.motorola.com>.
- [7] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-level modeling of dynamically reconfigurable co-processors," in *Proceedings of the 14th International Conference on FPL*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 881–885, Tampere, Finland, August 2004.
- [8] Y. Qu, K. Tiensyrjä, and J.-P. Soininen, "SystemC-based design methodology for reconfigurable system-on-chip," in *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD '05)*, vol. 2005, pp. 364–371, Porto, Portugal, August 2005.
- [9] J. Noguera and R. M. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.
- [10] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Partitioning methodology for dynamically reconfigurable embedded systems," *IEE Proceedings: Computers and Digital Techniques*, vol. 147, no. 6, pp. 391–396, 2000.
- [11] F. Berthelot, F. Nouvel, and D. Houzet, "Design methodology for runtime reconfigurable FPGA: from high level specification down to implementation," in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS '05)*, vol. 2005, pp. 497–502, Athens, Greece, November 2005.
- [12] M. Handa, R. Radhakrishnan, M. Mukherjee, and R. Vemuri, "A fast macro based compilation methodology for partially reconfigurable FPGA designs," in *Proceedings of the 16th International Conference on VLSI Design*, pp. 91–96, New Delhi, India, January 2003.
- [13] K. S. Chatta and R. Vemuri, "Hardware-software codesign for dynamically reconfigurable architectures," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL '99)*, pp. 175–184, Glasgow, UK, August–September 1999.
- [14] L. Shang and N. K. Jha, "Hardware-software cosynthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proceedings of the 7th Asia and South Pacific and the 15th International Conference on VLSI Design Automation Conference (ASP-DAC '02)*, pp. 345–352, Bangalore, India, January 2002.
- [15] L. Shang, R. P. Dick, and N. K. Jha, "SLOPES: Hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 508–525, 2007.
- [16] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher, "Modeling and simulation of dynamical and partially reconfigurable systems using SystemC," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 35–40, Porto Alegre, Brazil, March 2007.



- [17] Z. Li, *Configuration management techniques for reconfigurable computing*, Ph.D. thesis, Department of Electrical and Computer Engineering, Northwestern University, Evanston, Ill, USA, 2002.
- [18] M. J. Heikkilä, "A novel blind adaptive algorithm for channel equalization in WCDMA downlink," in *Proceedings of the 12th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '01)*, vol. 1, pp. A41–A45, Diego, Calif, USA, September 2001.
- [19] Memec, "VirtexIIpro demonstration board datasheet," 2003, <http://www.memec.com>.
- [20] Y. Qu and J.-P. Soininen, "Estimating the utilization of embedded FPGA co-processor," in *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD '03)*, pp. 214–221, Belek-Antalya, Turkey, September 2003.
- [21] R. P. Wilson, R. S. French, C. S. Wilson, et al., "SUIF: an infrastructure for research on parallelizing and optimizing compilers," in *Proceedings of the 7th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 37–48, San Diego, Calif, USA, December 1994.
- [22] Redhat, "gcov: the test coverage tool," May 2007, <http://www.redhat.com>.
- [23] D. D. Gajski, et al., *High-level synthesis: Introduction to chip and system design*, Kluwer Academic Publishers, Boston, Mass, USA, 1997.
- [24] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.
- [25] The Open SystemC Initiative (OSCI), "The SystemC TLM 2.0 documentation," May 2007, <http://www.systemc.org/home>.
- [26] OCP-IP, "OCP 2.2 Specification," February 2007, <http://www.ocpip.org/home>.
- [27] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *Proceedings of the 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 65–74, Monterey, Calif, USA, February 1998.
- [28] S. Chiba, "OpenC++ reference manual," May 2007 <http://opencxx.sourceforge.net>.
- [29] P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 3, pp. 381–390, 1996.
- [30] Xilinx, "Xilinx application note: XPP290 Two Flows for Partial Reconfiguration: Module-Based or Difference-Based," May 2007.