Embedded Software Integration for Coarse-grain Reconfigurable Systems

Patrick Schaumont, Kazuo Sakiyama, Alireza Hodjat, Ingrid Verbauwhede
Electrical Engineering Department
University of California at Los Angeles

Abstract

Coarse-grain reconfigurable systems offer high performance and energy-efficiency, provided an efficient run-time reconfiguration mechanism is available. Using an embedded software vantage point, we define three levels of reconfigurability for such systems, each with a different degree of coupling between embedded software and reconfigurable hardware. We classify reconfigurable systems starting with tightly-coupled coprocessors and evolving to processor networks. This results in a gradual increase of energy-efficiency when compared to software-only systems, at the cost of increasing programming complexity.

Using several sample applications including signal-, crypto-, and network-processing acceleration units, we demonstrate energy-efficiency improvements of 12 times over software for tightly-coupled systems up to 84 times for network-on-chip systems.

1. Introduction

The next generation of embedded information processing systems will require a considerable amount of computation power. An example of such a system is a portable, personal multimedia assistant. Silicon technology will be able to offer all the processing power and heterogeneity required for such a personal multimedia assistant. But we are faced with two conflicting design goals.

On one hand, we need high energy-efficiency because the system must be portable and battery-operated. Distributed, dedicated architectures are more energy-efficient than centralized, general-purpose ones. Therefore, we will use a distributed architecture that uses besides general-purpose cores also coarse-grain reconfigurable blocks with a limited instruction set [6].

On the other hand, we must also resolve how we will write programs for such a distributed and heterogeneous architecture. The problem is that there is no generally accepted programming method that covers all the elements in the system. Individual cores can be programmed in C or another general-purpose programming language, but coarse-grain reconfigurable blocks usually have very specific and architecture-dependent programming mechanisms. In addition, a system programming model should expose and promote the parallelism offered by the target architecture.

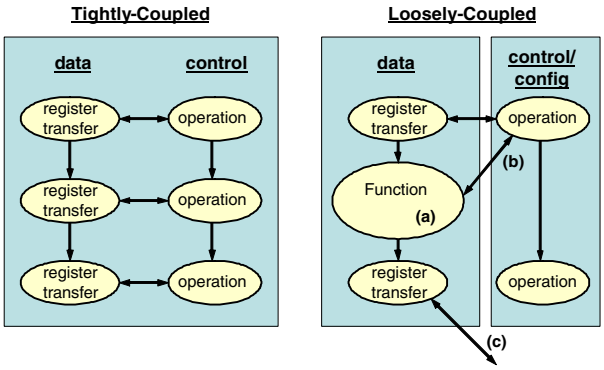

Figure 1: A coarse-grain reconfigurable system can be seen as one in which the ties between data-flow and control-flow are loosened.

In this paper we advocate that traditional embedded software design and coarse-grain reconfigurability are closely related, and that the key of a successful energy-efficient design is to identify the appropriate level of coupling of each coarse-grain reconfigurable block to the embedded software. Loosely-coupled blocks offer more opportunities for specialization, and therefore offer potentially better energy efficiency. On the other hand, loosely-coupled blocks are more difficult to program and control.

Figure 1 presents a conceptual view on this relationship. In classic embedded software, data-flow and control-flow are tightly coupled, and execute in lock-step. In a coarse-grain reconfigurable system, the links between data-flow and control-flow are loosened. Several possibilities are shown: (a) the granularity of processing can be increased, (b) a single control/configure operation can apply over an extended period, (c) control- and data-flow can be implemented with independent processors.

In this paper, we investigate the effect of coarse-grain reconfigurability on embedded software design. We will compare several architectures for loosely-coupled systems, quantifying the energy-efficiency improvement of each case over software, and discussing the impact on embedded software design. First we define three different levels of configurability from the viewpoint of embedded software. We then discuss three concrete examples to exemplify each of those three reconfiguration scenarios, and discuss design results for these examples.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

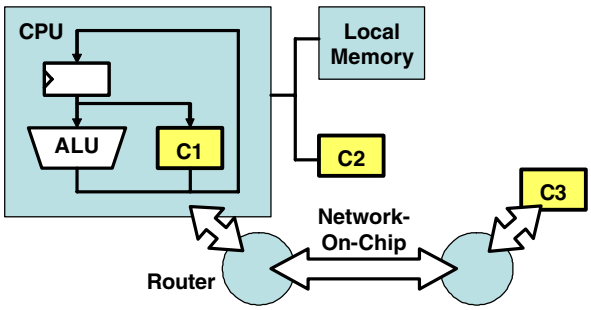

Figure 2: Coarse-grain Reconfigurable Blocks are Register-Mapped (C1), Memory-Mapped (C2) or Network-Mapped (C3)

2. Overview of Related work

Defining a programming paradigm for coarse grain reconfigurable architectures is a difficult problem. We formulate an approach based on application-specific specialization of embedded software.

Other researchers have proposed systematic programming models. SCORE [3] uses the data-flow model of computation to partition an application into pieces that can be configured individually on reconfigurable hardware. PipeRench [5] defines a virtual pipeline model, that can be mapped onto reconfigurable pipeline elements. For our applications, we do not use such a homogeneous programming model. Instead, we assume the presence of a general-purpose core running software that can be specialized by the designer, according to the needs of the application.

There are many examples available of systems that combine embedded software design with hardware acceleration. XiRISC [2] uses an embedded software environment to program the combination of a general-purpose processor with a reconfigurable function unit. Another recent success in applying a software methodology to loosely coupled reconfigurable architectures is OS4RS [4]. This approach uses a task-oriented formulation of the application. The resource management and communication mechanisms of an operating system are used to manage the combination of embedded software as well as reconfigurable hardware. Our key contribution to this existing work is that we provide an integrated approach to the combination of embedded software with different forms of reconfigurable hardware.

3. Embedded software view on coarse-grain reconfiguration

3.1. The RINGS system architecture

The discussion on coarse-grain reconfigurability is given in the context of the RINGS [7] system architecture. A RINGS architecture is a collection of heterogeneous and application-domain-specialized blocks embedded into a reconfigurable network-on-chip. There is at least one CPU that is responsible to maintain overall system control. The program running on this CPU is the *embedded software* that we will use in our discussion.

Figure 2 demonstrates that there are three levels of coupling between embedded software and reconfigurable hardware. These levels correspond to the level of integration between the CPU and the reconfigurable hardware. We distinguish register-mapped, memory-mapped and network-mapped reconfigurable blocks.

3.2. Register-mapped reconfigurable blocks

Register-mapped reconfigurable blocks give the tightest integration with embedded software. They can be created by modifying the micro-architecture of an embedded core, for example by integrating a custom reconfigurable datapath next to the ALU. In this case, the presence of such a block is directly visible in the instruction-set of the embedded core.

This type of reconfigurable blocks is popular because their design can be tightly integrated into the existing tool- and architecture infrastructure for this core. However, we should also realize that this solution requires a tight coupling of control-flow and data-flow. The parallelism that can be obtained with these solutions is primarily data-parallellism. We cannot easily modify the data-flow and control-flow of an algorithm outside the model provided by the CPU. Another issue is that control- and data-flow dependencies need to be resolved instruction-by-instruction. For example, pipeline conflicts in the CPU will also affect the processing performance of the reconfigurable block.

3.3. Memory-mapped reconfigurable blocks

By providing reconfigurable blocks with a memory interface, they can be integrated into the memory-map of a processor. This method results in looser coupling between software and the reconfigurable block. A set of shared memory locations between the software and the configurable block is defined. These shared memory locations can convey control- as well as data-flow oriented information, depending on the requirements of the design. As a result, coupling between data-flow and control-flow is less tight. A typical example of loose control-data coupling is the use of so-called continuous instructions in streaming-media processors. A continuous instruction is one that is assumed to be applicable to a stream of data elements. For this purpose, the processor can be programmed into a predefined mode of operation using a continuous instruction. The same type of instruction can also be created for a reconfigurable block: one memory location of the interface is used to configure the operation of that block, and after that another location accepts a stream of data values to be processed.

A drawback of this type of integration is that a reconfigurable block must share the memory address space with other memories and peripherals. Also, both the control and

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Table 1: Coarse Grain Reconfiguration Mechanisms

Mapping	Architecture Strategy	Reconfiguration Mechanism	Data-flow/ Control-flow Coupling	Energy Efficiency Improvement	Simulation Technology	Integration Technology
Register-Mapped	Custom Datapath	Custom Instructions	Tightly Synchronized	Low	Custom ISS	Custom Compiler
Memory-Mapped	Coprocessor	Memory-mapped Instructions	Loosely Synchronized	Medium	ISS/Coprocessor Cosimulation	Software Library (function call)
Network-Mapped	Peer Processor	Configuration Packets	Uncoupled	High	ISS/Coproc/NoC Cosimulation	Communication primitive

data-flow are eventually routed through the CPU and the embedded software. Direct-memory access techniques can help to break this bottleneck but do not eliminate the fundamental problem of a shared memory address space. The CPU remains a bottleneck in the overall system.

3.4. Network-mapped reconfigurable blocks

Reconfigurable blocks can also be attached as independent entities in a Network-on-Chip [1]. In this case, integration of embedded software and reconfigurable blocks can be done using communication primitives. These can be integrated into an operating system such as in [4].

Network-mapping allows to treat the integration of data- and control-flow independently. In a network-on-chip, network packets can contain control- as well as data-flow information. Therefore, data-flow and control-flow might literally have a different route in the system. For example, it is possible to create a system where a CPU sends configuration and control packets to reconfigurable blocks that at the same time have high-throughput data-streams between them. In this case the embedded software on the CPU maintains overall system synchronization, rather than being a data pipe. This programming model is the most complicated, because it deviates the most from a classic sequential programming model.

3.5. Impact on embedded software design

Each of the three schemes discussed has specific requirements towards system- and embedded software design. In Table 1, we give an overview of the issues that are relevant to select a particular strategy, as well as the impact of each strategy on design support.

- *Architecture Strategy* relates to the reconfigurable block. Self-contained architectures such as peer processors are harder to design because their integration interface is more complicated.
- *Reconfiguration Mechanism* indicates how instructions are provided to the reconfigurable block.
- *Data-flow/Control-Flow Coupling* indicates how close the design of data-flow is linked to the design of control-flow. Uncoupled offers higher performance, potential better energy improvement, but is also the hardest to program.
- *Energy Efficiency Improvement* is a relative appreciation how energy-efficient a coarse grain reconfigurable system will perform when compared to a software-only, single-CPU system with the same functionality.
- *Simulation Technology* indicates the required simulation technology to design software for this reconfigurable system effectively. Each of the three approaches requires instruction-set simulation (ISS), but the complexity of the cosimulation setup shows large variations.
- *Integration Technology* indicates the requirements towards embedded software development. A tightly coupled, register-mapped system requires a compiler that can create custom instructions. Memory-mapped systems can be supported using software libraries. Network-mapped systems need communication primitives, and can require the introduction of specialized operating system software.

In our experience, each of these three models for coarse-grain reconfigurability has virtues and deficiencies, and none of them can be pointed at as a universal solution. In the following section, we discuss three concrete design cases.

4. Example design cases

The three examples will illustrate the characteristics of the three classes of coarse-grain reconfigurable blocks. The first two, a DFT acceleration unit and an AES coprocessor, are part of the biometrics system discussed earlier. A third one is a TCP/IP checksum coprocessor.

We use the DFT unit as an example of a tightly-coupled (register-mapped) block, the AES unit as an example of a loosely-coupled (memory-mapped) block and the checksum processor as an example of an uncoupled (network-mapped) block. We will discuss the impact of each of the coprocessors on embedded software.

4.1. DFT signal processor

This DFT processor implements a one-dimensional 24-point DFT on four different discrete sample frequencies, namely for k = 1, 2, 3 and 4 (corresponding to 1, 2, 3, and 4 periods per 24 samples). The micro-architecture of this processor is shown in Figure 3. The processor is organized as a memory-mapped coprocessor. Two memory locations are

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

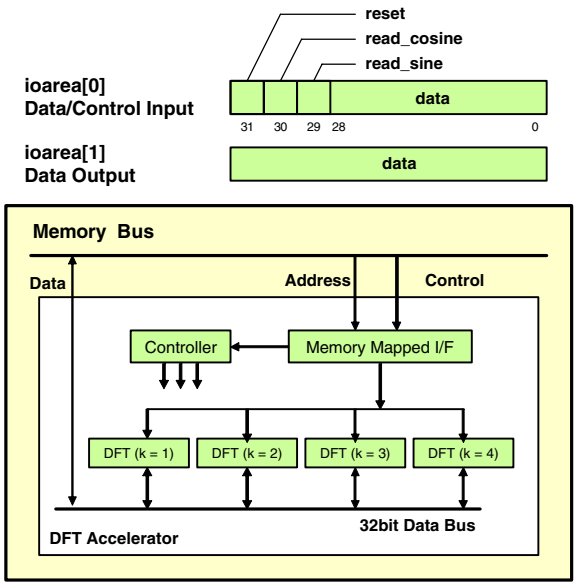

Figure 3: DFT Processor Micro-Architecture

```
for (k=0; k<3; k++) { /* DFT at 4 frequencies */
  /* Initialize accumulators */
  cospart[k] = 0;
  sinpart[k] = 0;

  /* Accumulate cos and sin components */
  for (i = 0; i < 16; i++) {
    cospart[k] += (rowsums[i] * wave[k]->cos[i]);
    sinpart[k] += (rowsums[i] * wave[k]->sin[i]);
  }
}
```

(a) DFT accumulation loop in C

```
volatile int *ioarea = (int *) 0x20000000;

for (i = 0; i < 16; i++) {
   ioarea[0] = 0x80000000; /* cmd = send next */
   ioarea[0] = rowsums[i];
}
for (k=0; k<3; i++) {
  ioarea[0]  = 0x40000000; /* cmd = read cos */
  cospart[k] = ioarea[1];
  ioarea[0]  = 0x20000000; /* cmd = read sin */
  sinpart[k] = ioarea[1];
}
```

(b) DFT accumulation loop in C using a memory-mapped coprocessor mapped at 0x20000000 (GEZEL code not shown)

Figure 4: Embedded Software for (a) software-only DFT and (b) DFT-coprocessor

used between the CPU and the coprocessor: one for data input and control, and one for data output. After a 'reset' command, the processor will accept a sequence of 24 samples during the 24 consecutive writes to the data-input. This is an example of a continuous, implicit instruction.

The 24 samples are used to accumulate sine and cosine components of the DFT at all four frequencies in parallel. After 24 samples have been provided, a 'read_cosine' or

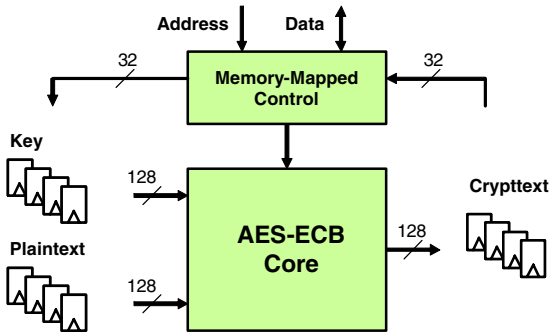

Figure 5: Memory-mapping of AES Processor Micro-Architecture

```
volatile int *ioarea = (int *) 0x20000000;

void aes(int data[4], int result[4]) {
  for (int i=0; i<4; i++) {
     ioarea[0] = data[i];      // data input
     ioarea[1] = LOAD_DATA;    // instruction
  }
  ioarea[1] = ENCRYPT;
  for (int i=0; i<4; i++) {
     ioarea[1] = READ_OUTPUT; // data ouput
     result[i] = ioarea[2];   // instruction
  }
}
```

Figure 6: Embedded Software for AES Processor

'read_sine' command will retrieve the sine and cosine accumulators for each of the four frequencies in a sequence of consecutive reads. The embedded software that controls this memory-mapped processor is shown in Figure 4. The top part (a) shows the original, software-only implementation of the 4-point DFT. The bottom part (b) shows the C-code that drives the coprocessor. While this code uses memory-mapped accesses, it exhibits the characteristics of a register-mapped design: the control-flow of the DFT algorithm is not carried over to the DFT processor, but remains in C on the embedded processor.

4.2. AES encryption processor

The AES coprocessor is a stand-alone, memory-mapped block that evaluates the AES encryption algorithm. The integration onto a 32-bit memory bus is illustrated in Figure 5. The encryption is done on a 128-bit key and a 128-bit plaintext, and produces 128-bit of crypttext. The processor uses three memory locations for interfacing to the embedded software - one for instructions, one for data-input and one for data output. The instructions allow to assemble the 128-bit key or plaintext out of 32-bit writes, and execute the encryption. The embedded software that controls this AES processor is shown in Figure 6. This fragment assumes that the key was already programmed, and that the AES processor encrypts subsequent blocks of plaintext with this key.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

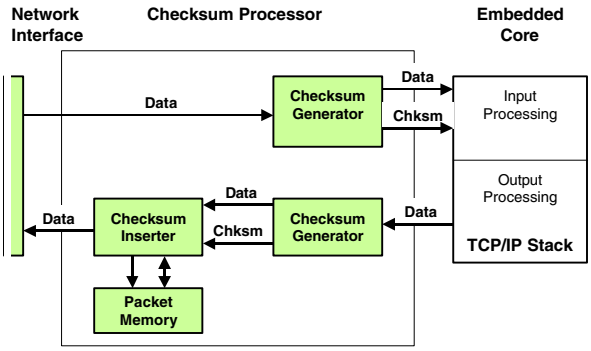

Figure 7: TCP/IP Checksum Generation/Verification

```
volatile int *ioarea = (int *) 0x20000000;

int input_available() {
  return (ioarea[0]);
}

int input_complete() {
  return (ioarea[1]);
}

void input_read(u8_t *dbuff) {
  *(u32_t *) dbuff = ioarea[2];

  ioarea[3] = 0x1; // acknowledge
  ioarea[3] = 0x0;
}

void input_get_checksum () {
  ipchecksum_val  = ioarea[4];

}
```

Figure 8: Embedded Software Communication Primitives for TCP/IP Input Channel

While the software fragment looks similar to the one shown for DFT coding, the granularity of the two functions is considerably different. For the case of the DFT, the control-flow of the original algorithm in software still shows up in the resulting accelerated algorithm. For the AES, the software corresponding to a complete encryption has been substituted by a write to the instruction memory location. Thus, if we consider the AES processor as a migration of software to hardware, both control- and data-flow of the AES algorithm C have been moved onto hardware, and the functionality in C is reduced to a few memory writes and reads. We can see however that data leaves from software - as plaintext - and also returns to it - as crypttext. The embedded software remains responsible for the system-level data-flow.

4.3. TCP/IP checksum evaluation

A third example shows system-level migration of data- and control-flow to a coprocessor. A TCP/IP protocol stack runs on an embedded core. Each packet processed by the protocol stack has checksum fields embedded into it. The checksums are generated using the header bytes only (for the IP-layer), or else using bytes from both the header and the payload (for the TCP-layer). Checksum verification and generation is a computation-intensive process because it affects the complete packet. It is therefore migrated to independent coprocessor units as illustrated in Figure 7. A checksum generator generates TCP and IP checksums. For the input channel, they are used for checksum verification. For the output channel, they are used by a checksum inserter to place them in the correct position in a packet. An extra packet memory is required because TCP checksums are non-causal - they are evaluated on bytes that succeed the TCP checksum bytes.

The software that accesses the input channel is illustrated in Figure 8. The checksum processor operates independently from the embedded core and interfaces with it using a set of communication primitives. The embedded core has four functions for access.

- input_available indicates if the checksum processor has a word available. The embedded core polls the checksum processor (rather then using interrupts) to improve throughput when reading blocks of packet data.
- input_complete indicates if a complete packet was provided.
- input_read accepts the next word in a packet. Bytes are grouped by 4 in a word to improve throughput.
- input_get_checksum reads the checksum values obtained by the checksum processor.

This example corresponds to the network-mapped strategy discussed earlier - even if no network on chip is used here. When we compare the embedded software to a complete, software-only TCP/IP stack, we see that checksum evaluation has completely migrated to another (peer) processor. The checksum evaluation functions have been substituted with a set of communication primitives.

5. Results

We have implemented the three design cases starting from a LEON-2 embedded SPARC v8 core, and extending it with coarse-grain accelerator processors. For each case we obtained the implementation cost and energy cost for the software-only implementations as well as for the hardware-accelerated systems. We used a LEON2 32-bit, 50 MHz Sparc core as embedded software host. We used Xilinx Virtex-2 FPGA technology as the mapping target and obtained power estimates with Xilinx xpower. The energy figures still show the overhead of a fine-grain platform and have primarily a relative value. Our results are presented in Table 2.

Before presenting the conclusions, it is useful to motivate the methodology of the experiment.

- First, we use three different applications, based on actual designs, rather than a single one. The designs therefore are representative to what a designer would do.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Table 2: Design results for the three design cases

Application	Target Architecture	Performance (ms)	Implementation Cost (Memory + LUT)	Estimated Power (mW/MHz)	Estimated Energy (mJ)
DFT (1000 iterations)	SW on LEON2	118.7	9.7 KByte ROM 4856 LUT	11.4	67.6
	LEON2 with Accelerator HW	9.23	8.9 KByte ROM 7700 LUT	12.5	5.76 **(12X Improvement)**
AES (175 iterations)	SW on LEON2	158.3	36.3 KBytes ROM 4856 LUT	11.4	89.2
	LEON2 with Accelerator HW	5.23	8.6 KByte ROM 8330 LUT	13.5	3.5 **(25X Improvement)**
TCP/IP Chksm (100 packets from HTTP seq)	SW on LEON2	5.54	10.0 KByte ROM 4856 LUT	11.4	17.0
	Accelerator HW (stand-alone *)	0.699	1556 LUT	5.78	0.20 **(84X Improvement)**

(*) The implementation and performance is that from a stand-alone checksum verifier/inserter

- Second, we compare each design case to itself, without a change of the target technology. The goal of our experiment is to demonstrate the potential of effective system level design regardless of the target technology. In addition, any improvement in target technology will benefit both the software-only as well as the hardware-accelerated case.
- Third, while the energy-improvement of hardware over software is well known, our techniques also demonstrate how such improvements can be obtained. We specifically point to the role of intelligent on-chip interconnection mechanisms as a way to make coarse-grain acceleration hardware easily available.

From these observations, the conclusions are as follows. Introducing accelerator hardware in the form of coarse-grain reconfigurable blocks reduces energy consumption. The DFT design can evaluate Fourier Transforms with 12 times less energy than equivalent software. By moving not only data processing, but also control-flow to the accelerator, further improvements are achieved. The AES design can evaluate Rijndael encryption with 25 times less energy as software. The biggest savings are obtained by detaching the coarse grain reconfigurable block from all bottlenecks in the embedded core (such as the memory bus). The TCP/IP Checksum design, which makes this transformation, can perform checksum insertion and verification with 84 times less energy than embedded software.

6. Conclusions

We have illustrated the impact various types of coarse-grain reconfigurable systems on embedded software. Coarse-grain reconfigurable systems are good candidates for energy-efficient designs. In our examples, we have shown energy-efficiency improvements of 12 to 84 times over software. We have presented three strategies for coarse-grain reconfigurable block integration, each with a different effect on data- and control-flow of the original software implementation. An important aspect is that a C programming model, despite the assumption of sequential execution, does not *prevent* any of the strategies explained above. Indeed, a single application could use register-mapped, memory-mapped as well as network-mapped reconfigurable blocks.

7. Acknowledgements

This work has been made possible by NSF (Grant CCR 0310527) and SRC (Grant SRC - 2003-HJ-1116).

8. References

[1] L. Benini et al., *Networks on chips: a new SoC paradigm*, IEEE Computer, volume 35, pp. 70--78, January 2002.

[2] Capelli et al., *A Reconfigurable Processor Architecture and Software Development Environment for Embedded Systems*, Proc 10th Reconfigurable Architectures Workshop (RAW03), Nice, France.

[3] Caspi et al., *Stream computations organized for reconfigurable execution (SCORE)*, 2000 Conference on Field Programmable Logic and Applications (FPL00), Villach, Austria, August 2000.

[4] Nollet et al., *Desigigning an Operating System fot a Heterogeneous Reconfigurable SoC*, Proc. 10th Reconfigurable Architectures Workshop (RAW03), Nice, France.

[5] Schmit et al., *PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology*, 2002 IEEE Custom Integrated Circuits Conference (CICC02).

[6] Rabaey, *Silicon platforms for the next generation wireless systems - What role does reconfigur able hardware play?*, 2000 Field Programmable Logic and Applications (FPL00).

[7] Verbauwhede et al., *Reconfigurable Interconnect for next generation systems,* Proc. ACM/Sigda 2002 International workshop on System Level Interconnect Prediction (SLIP02), Del Mar, CA.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE