The Happy Marriage of Architecture and Application in Next-Generation Reconfigurable Systems

Ingrid Verbauwhede
University of California Los Angeles,
& K.U.Leuven
ingrid@ee.ucla.edu

Patrick Schaumont
Electrical Engineering Department
University of California at Los Angeles
schaum@ee.ucla.edu

ABSTRACT

New applications and standards are first conceived only for functional correctness and without concerns for the target architecture. The next challenge is to map them onto an architecture. Embedding such applications in a portable, low-energy context is the art of molding it onto an energy-efficient target architecture combined with an energy efficient execution. With a reconfigurable architecture, this task becomes a two-way process where the architecture adapts to the application and vice-versa. This leads to the idea of a marriage between architecture and application.

These next generation reconfigurable systems consist of a heterogeneous collection of domain-specific processing units. Communication between processors occurs over a reconfigurable interconnect scheme. Global control is provided by one or more embedded micro-controllers, which operate at a low frequency since they don't run compute intensive functions. Because of the domain specific features, this architecture is low power, yet at the same time reconfigurable.

In this paper, we will describe the RINGS (Reconfigurable interconnect for next generation systems) architecture and the associated design environment, GEZEL. We will describe how applications are mapped onto RINGS architectures and how they can be modeled and simulated in the GEZEL environment.

Categories and Subject Descriptors

C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: Real-time and Embedded Systems.

General Terms

Design, Verification.

CF 04, April 14-16, 2004, Ischia, Italy.

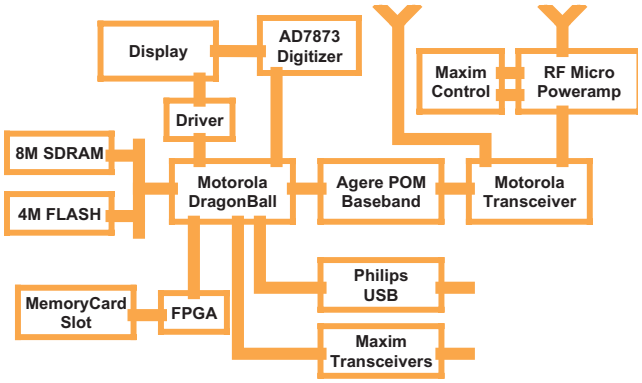

Figure 1: Typical example of an energy-optimized embedded system: PalmPilot i705

Keywords

Embedded, real-time systems,

1. INTRODUCTION

Post-PC embedded systems use an architecture platform very different from the traditional homogeneous general purpose computing platforms. For instance, they might include RF and baseband components for wireless communication, DSP engines for real-time signal processing applications, network engines for network protocol processing and co-processors or accelerator units for video and image processing. The main advantage of this type of heterogeneous platform is its energy efficiency. Indeed there is a fundamental flexibility versus energy-efficiency trade-off, referred to as the intrinsic computational efficiency by T. Claassen [5]. For example, in the battery-operated, wireless PDA in Figure 1, every component is tuned towards its application domain [3]. However, reconfiguration (and thus reuse) of this type of platform is very difficult, if not impossible.

We propose the RINGS architecture, that inherits the same computational efficiency design principles and takes the properties of the application into account before fixing the architecture. In contrast with those traditional architectures however, RINGS applies a systematic design method, rather then ad-hoc.

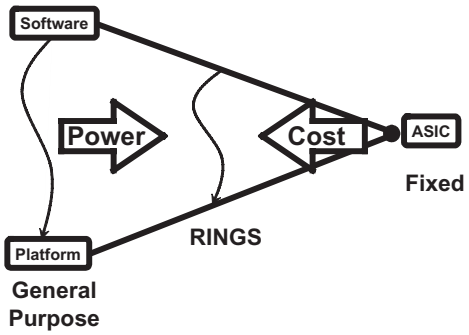

Figure 2: RINGS specializes both architecture and programming model to the application, and balances power and cost.

As illustrated in Figure 2, RINGS systems take up the middle ground between general-purpose approaches and fully-dedicated, fixed systems. General-purpose approaches provide cost-effectiveness but lack energy-efficiency. Fully dedicated approaches are best for energy-efficiency, but also very expensive to design. Moreover, dedicated architectures lack programmability, as they completely unify application and platform. RINGS seeks a balance between power and cost, by selectively reducing general-purpose programmability. The reduced programmability is achieved by targeting the architecture specialization to the application. This includes the introduction of specialized yet flexible processing elements, interconnect- and storage structures.

In Section 2, we will describe an application-centric framework to model RINGS systems. This framework allows us to define the architecture design task for RINGS more precisely. Following that in Section 3, we will take a closer look at the design methods for RINGS design, including modeling, exploration and refinement. We will then in Section 4 present the GEZEL codesign environment, which supports the design of RINGS architectures. We will discuss the architecture of a typical multiprocessor co-simulator created using GEZEL. In Section 5 we take a closer look at the architectural options available for RINGS components. We will illustrate the idea of specialized and reduced programmability by means of an example design. Section 6 provides details on the RINGS interconnect architecture, a cornerstone that enables the use of multiple, heterogeneous processors. Finally, in Section 7, we will also take a look at the various forms of coupling application and architecture by enumerating various coprocessor software interface mechanisms.

In this paper, we will use the single term programming and software as a generic term for all activity that is associated with reconfiguration, instruction-set programming, and micro-programming. Software design for RINGS comprises writing C for embedded RISC cores, writing dedicated microprograms for domain-specific processors, and developing communication protocols.

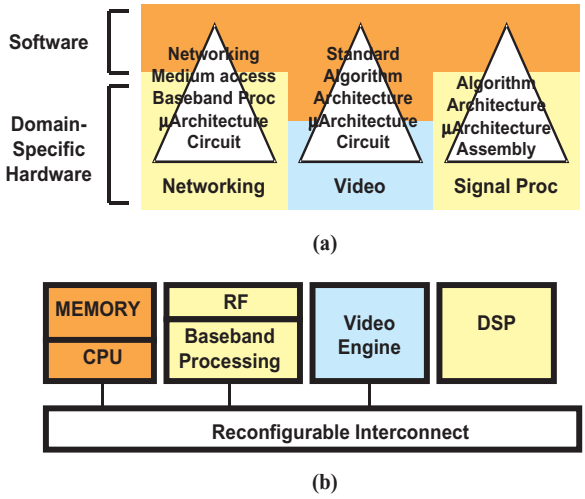

Figure 3: (a) The RINGS application model maps onto (b) the RINGS architecture.

2. THE RINGS VISION

RINGS provides an IP-based design strategy to combine flexibility and specialization, thus allowing reuse and reconfiguration without loosing the efficiency. The RINGS architecture is a platform that contains multiple heterogeneous domain-specific processors (e.g. a DSP, a video-coprocessor, a network engine, etc.) that are connected together with a reconfigurable interconnect.

A conceptual application-model and architecture-model for RINGS is shown in Figure 3. A system design is a co-design of application domains. In the example, a wireless videophone application is represented as a combination of three different domains: networking, video and signal processing. This system view is different from a traditional HW/SW co-design view, which uses a co-design of implementations. Each application domain is represented by a hierarchy of abstraction levels, represented by a pyramid. As an example: at the top level of the video pyramid will be the standard (JPEG, JPEG2000, JVT). It describes the end to end transactions between the communicating parties. One level down are the video algorithms, which are building blocks to the protocol. Video processing might rely on specialized building blocks and operations, such as motion compensation, filtering, smoothing, and so on. The next level down are the actual building blocks. For video applications, especially memory traffic and buffer design are critical issues.

As a system is expressed as a collection of application domains, ultimately all these domains will cooperate towards achieving a single goal. Thus, the collection of pyramids needs to be integrated, both from the architecture viewpoint as well as the software viewpoint. The integration

software is running on the system controller, and maintains loose interaction with the domain-specific accelerators. The RINGS interconnect architecture maintains a flexible set of communication links between the different (co-)processors in the system. In between this software layer and the architecture interconnect layer, there is a very interesting optimization boundary that cuts all of the domain pyramids in half. The top half is mapped onto software, while the bottom half is mapped onto the domain-specific processors. This hardware-software boundary can be thought of as a configuration- or programming interface. When it occurs high-up in the domain pyramid, it means that software will maintain only very loose interaction with that domain, sending only very coarse instructions to the domain-specific processors. When it occurs at low abstraction level, on the other hand, it means that there is a close interaction between the software and the domain-specific processors. In the RINGS architecture, we contend that ideal abstraction level for this hardware-software boundary *varies with the application domain*. Energy-efficiency-considerations suggest to move more activity onto a dedicated processor and thus raise the boundary. Flexibility - and cost-considerations on the other hand will call for more software and thus push the boundary down. Optimizing a RINGS system means striking a balance between these two opposing pressures. As is indicated on Figure 3, this transition from a flexible to a fixed implementation (or from software to hardware) is decided independently for each pyramid. Also, it must be observed that general-purpose systems, in contrast to the flexibility of RINGS, make an up front hard-coded decision about the boundary between software and architecture.

3. RINGS DESIGN METHODS

3.1 Design entry point

Embedded systems are complex applications. They consist of multiple components that need to be mapped on a heterogeneous architecture. The original application is seldom well defined and might be a mixture of high level specs in a natural language, block diagrams of sub components, and a C or Matlab description of other sub components, such as a JPEG or another video standard. The original specs might also include architectural features, such as the request to reuse an existing IP block or an existing assembly code library.

The task of the designer is to map the initial high level specs into a detailed spec. This task consists of two major components. One is a gradual vertical refinement of the initial spec. The second is to perform a horizontal design space exploration between design options and to optimize the energy flexibility trade-off. This is illustrated by the reversed umbrella of Figure 4.

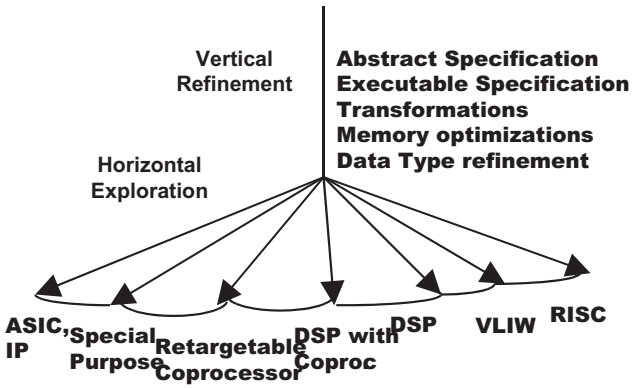

Figure 4: The reverse umbrella: vertical refinement and horizontal design space exploration.

3.2 Application specification

The initial abstract specification is a mixture of natural language (English) specifications, block diagrams and possibly executables for sub components. For instance for the wireless video phone mentioned earlier, there might a C reference code available for the coding standard. This might be combined with a written spec on the networking and wireless communication and a library of IP blocks available from previous designs.

For individual pyramids, well-suited, domain-specific languages and specification environments are available. One example is the domain of signal processing. For this domain, both high level domain-specific languages such as Silage [18] or specification environments such as Ptolemy have been developed. Data flow descriptions are another example used for real-time (multi-dimensional) signal processing examples [11][13]. For instance the DFG and MDFG [20] have been developed.

For the control-flow dominated applications, synchronous specifications [2] have been developed to deal with the intricacies of events in a formal way. Another classic modeling abstraction for control-dominated applications are Statecharts [7], currently part of the UML standard that targets to model complex software systems.

The specification of the complete system will therefore never be in one language . This language, (if it would exist) will either be too general purpose, since it has to cover a wide range of application domains. Or if it is efficient for one application class, it will fail in capturing the details of another domain.

In practice, C or C++ descriptions are widely used as specifications. The reason is that original specs are developed on general purpose (sequential) machines, onto which C or C++ run very conveniently. As a consequence, a lot of research is devoted to extract the parallelism out of a

sequential description, while the original specification might have shown the parallelism in a natural way.

This is an engineering fact of life, which we try not to address in an automatic way. In our design environment, we allow the designer to explore with different, parallel processing units, but it is the designer, who at the high abstraction level, will partition the design over the different domains. For many embedded applications, this is a quite natural task. The overall control-flow that connects the different applications together sits in the main embedded controller.

3.3 The gradual refinement

While the image of multiple domains that capture an application is intuitive and easily understood, the format in which an application is delivered will seldom take the shape of such a pyramid. Elements of both the application and the architecture can be delivered in a variety of ways that must be combined in RINGS.

System design - Abstract specification: An application can be provided as high level abstract specifications (e.g. an algorithm, a mathematical formula, a Z-domain specification), or as reference implementation (e.g. a C or Matlab program). Whatever the format, it is unlikely to have the properties required for mapping onto the final RINGS architecture. Similarly, architecture elements in RINGS come in different formats: as intellectual-property modules with detailed architecture specs (e.g. an ARM core), or as high-level black boxes (e.g. a yet-to-be-designed motion estimation module).

System to embedded system design - Executable specification: The initial abstract specifications are turned into an executable specification. For instance for video applications, this means that a C or C++ executable is available, (hopefully) including test benches.

Transformations: This original executable is not optimized for embedded implementation. A typical transformation is the explicit introduction of parallelism. This is parallelism at the task level not the individual instruction or operation level. Another example of transformation is the introduction of explicit code that describes the communication between tasks or processes.

Memory Optimizations: One of the main optimizations possible is one of memory management. The original code is written for functional correctness, without taking the memory constraints, both in size and access, of the embedded device into account. To reduce the memory size and the number of memory accesses, both code transformations and memory architecture optimizations can be performed [4]. Some of these code transformations are architecture independent, others architecture dependent. In some architectures, the memory is fixed and given, for others, the memory architectures and associated address generation can still be molded towards the application domain.

Data type refinement: Transformations for memory optimization cross loop boundaries, affect function calls and in general have a more global impact. Once these are stable, more local optimizations are performed to better match the application code onto the embedded architecture. One such example is the floating-point to fixed point conversion. Indeed, most low power embedded processors do not have a floating point execution unit. Moreover, the fixed point units are often very limited in size and options. Some of these data refinements are executed independently of the target architecture. Some of them are very processor specific and are executed only when the target architecture is known. For instance, if one selects a 16-bit fixed point DSP processor, then the data type refinement will consist of mapping the arithmetic operations into 16 bit fixed point operations with the specific accumulator options provided by the processor architecture. For instance the DSP processor might have 2 or 4 accumulators of 32 bits with 8 bits overflow and specific overflow and saturation logic. Other processors, provide a SIMD type of data paths with associated instruction sets. For instance, 64 bit processors come with 8 parallel 8 bits wide SIMD instructions. These are typical examples of target dependent optimizations.

3.4 The horizontal options

For each of the domains an architecture platform needs to be chosen. Between the two extremes of a fixed ASIC and a general purpose programmable CPU, there is almost a continuum of options available to implement a domain. The two extremes, ASIC and CPU correspond to the two extremes shown in Figure 2.

ASIC s are still used because the know-how is available in the form of library IP modules: e.g. a video DCT or IDCT module, a crypto DES unit, a Viterbi or Turbo acceleration unit for wireless communications. To increase the reuse possibilities, the designers of IP blocks add features, parameters and other options to the IP blocks to turn them into special-purpose acceleration units. E.g. the DES unit can be programmed to execute encryption or decryption. It can also be programmed to execute DES or triple DES. Similarly, the Viterbi unit can be programmed for the number of states or samples, and so on.

As one moves more to the right on the horizontal axis of Figure 4, the processor architecture becomes more programmable. A processor has fundamentally four basic components: the data path or execution units, the control part, the memory part and the interconnect. Each of these components can be fixed or can be made programmable. At the same time, the granularity of programming needs to be defined. If a data path is still reprogrammed at the bit level or CLB level, it is called reconfiguration (typically on a FPGA or an FPGA block). If it is reprogrammed at the instruction level, it is called instruction set reconfiguration.

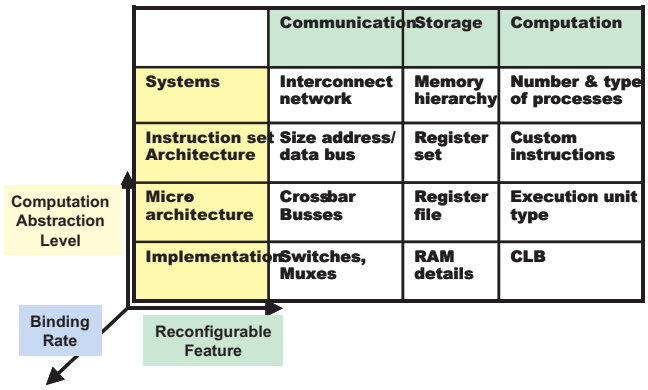

Figure 5: The reconfiguration hierarchy.

Reconfigurability is a concept with many different faces. We have defined the reconfiguration hierarchy [17] shown in Figure 5 as a representation of the design space of reconfigurable systems. The three axes of the reconfiguration space express three orthogonal reconfigurability characteristics: the configuration binding time, the reconfiguration abstraction level, and the nature of the reconfiguration. Reconfigurable systems can occupy lines, planes or volumes in this space. A general purpose programmable processor that has all components of the processor fixed and that can only be reprogrammed, occupies a line in this space: reconfiguration of this device proceeds at a fixed rate (the instruction stream) and at a fixed abstraction level (micro-architecture), and reconfiguration is done though ALU operations, register file addresses and multiplexer control inputs. Such a general purpose processor can now be unfolded to a more flexible reconfigurable system that occupies a plane or even a volume.

Typical architectural modifications will result in a reconfiguration plane: A first type of re-configuration is added when the data paths can be changed. This is e.g. the case in certain VLIW type of architectures, such as the ART design environment [12]. Other design environments allow the addition of specialized data paths and associated instruction sets. This is e.g. the case in the design environment of Lisatek or Tensilica [16].

3.5 An example refinement: AES Co-processing

We will give an example of the impact of the vertical refinement/horizontal exploration process. A dedicated AES128 coprocessor can execute the advanced encryption standard (AES) algorithm in 11 clock cycles. Figure 6a illustrates how it is attached as a memory-mapped coprocessor to a 32-bit embedded Sparc microprocessor. The raw performance of this dedicated coprocessor is however only useful in the context of a larger application.

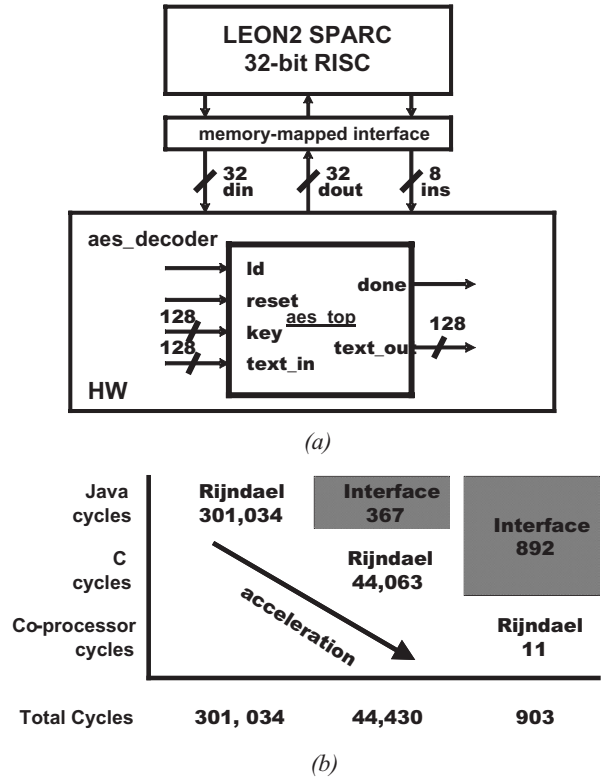

Figure 6: (a) A memory-mapped AES128 Coprocessor shows (b) a 300-times performance increase over a pure software implementation in

We implemented an embedded Java virtual machine (KVM) onto the Sparc processor, similar to the one that can be found on mobile phone applications. Then we integrated the AES128 coprocessor so that it could provide security services to the Java KVM. As illustrated in Figure 6b, this is a two-step process. The first is to create a *native interface* onto the Java Virtual machine that runs on top of the Sparc processor. Such a native interface is written in C. The second step consists of further adapting the code so that it directly interfaces to the memory-mapped coprocessors.

As illustrated in Figure 6b, the performance improvement is quite dramatic. If we run the AES encryption algorithm in Java software (and thus bypassing the coprocessor altogether), then we need 300K cycles for one encryption iteration. By building the native interface to C-code, and using an optimized version of the AES algorithm in the C language, the total amount of clock cycles drops to 44K. The best performance is obtained when we make use of the coprocessor, in which case the total amount of cycles drops to below 1Kcycles. Because the total amount of cycles for AES encryption has shrunk by a factor of 300, we obtain significant power savings. Even taking the additional overhead of a hardware coprocessor into account, we demonstrated a 25-fold improvement in power consumption assuming an FPGA technology.

Figure 6b also makes clear that the cost of additional processing levels (i.e. Java/C/Hardware) is high. Indeed, for the coprocessor, the execution overhead of interfacing dedicated hardware to software is 80 times! This problem is a generic one and not limited to AES encryption. Modern technology makes dedicated hardware and gates virtually free, however making effective use of them can be a major hurdle. The interconnect strategy of RINGS provides a specific approach to this interface overhead problem, by promoting a maximum amount of flexibility in on-chip information-flow.

4. RINGS DESIGN TOOLS

As was pointed out before, RINGS approaches system design as a codesign of domains. It should come as no surprise that the RINGS design environment is, in essence, a codesign environment. We need an efficient hardware modeling and verification environment to create domain-specific processors and their on-chip interconnection network. But, we also need co-simulation with software running on the central controller. In this section, we will present an environment that allows us to achieve both of these goals at the same time.

We will first present our codesign model. Next, we present the design tool architecture.

4.1 The GEZEL codesign model

Our codesign model is based on combining cycle-accurate FSMD (finite-state machine + data path) models for hardware with instruction-set simulation for software. We will illustrate the codesign model in the most simple form, as a single ISS combined with a memory-mapped interface to the hardware.

Consider again the architecture in Figure 6a. The AES encryption coprocessor is controlled out of C code running on the Sparc core. The encryption coprocessor includes an AES IP core (aes_top) with 128-bit input/output busses. This core is instantiated inside of a decoder module aes_decoder that multiplexes the 128-bit I/O data busses of aes_top on the 32-bit data connections to aes_decoder.

A number of memory addresses on the Sparc have been reserved for communication with the AES coprocessor. In this case, we have reserved two 32-bit data channels (din, dout), and an 8-bit instruction bus ins. By accessing an absolute memory address in C, we will be able to exchange data with the AES hardware. Figure 7a shows a sample C program that provides a new key value as 12 subsequent memory writes.

```
typedef volatile char* vcp;
typedef volatile int*  vip;
vcp ins  = (vcp) 0x80000000;
vip din  = (vip) 0x80000008;
vip dout = (vip) 0x80000004;

enum {ins_idle, ins_load, ins_key};

void load_key(int w0, w1, w2, w3) {
  *din = w0;   *ins = ins_load;   *ins = ins_idle;
  *din = w1;   *ins = ins_load;   *ins = ins_idle;
  *din = w2;   *ins = ins_load;   *ins = ins_idle;
  *din = w3;   *ins = ins_key;    *ins = ins_idle;
}
```

(a)

```
dp aes_decoder(in   ins : ns(8);                    Module
               in   din : ns(32);
               out dout : ns(32)) {
  reg key                       : ns(128);
  reg wrkreg0, wrkreg1, wrkreg2 : ns(32);
  reg ir                        : ns(8);
  reg dinreg                    : ns(32);

  use aes_top(rst, ld, sigdone, key, txtin, txtout);

  sfg decode   { insreg = ins;
                 dinreg = din; }
  sfg putword  { wrkreg0 = dinreg;
                 wrkreg1 = wrkreg0;
                 wrkreg2 = wrkreg1; }
  // The '#' operator bit-concatenates
  sfg setkey   { key     = wrkreg2 # wrkreg1 #
                           wrkreg0 # dinreg; }

}

fsm faes_decoder(aes_decoder) {          Module contro
  initial s0;
  state s1, s2;
  @s0 (decode)                                      -> s1;
  @s1 if (ir == 1)       then (decode, putword)     -> s2;
      else if (ir == 2)  then (decode, setkey)      -> s2;
      else (decode)                                 -> s1;
  @s2 if (ir == 0) then (decode)                    -> s1;
                   else (decode)                    -> s2;
}

                                         HW/SW Interface
ipblock b_ins(out data : ns(8)) {
  iptype "armsource"; ipparm "address=0x80000000";
}

ipblock b_datain(out data : ns(32)) {
  iptype "armsource"; ipparm "address=0x80000008";
}

ipblock b_dataout(in data : ns(32)) {
  iptype "armsink"; ipparm "address=0x80000004";
}

system S {
  aes_decoder(ins, din, dout);
  b_ins(ins);
  b_datain(din);
  b_dataout(dout);
}
```

(b)

Figure 7: (a) C driver program for (b) GEZEL description of aes_decoder

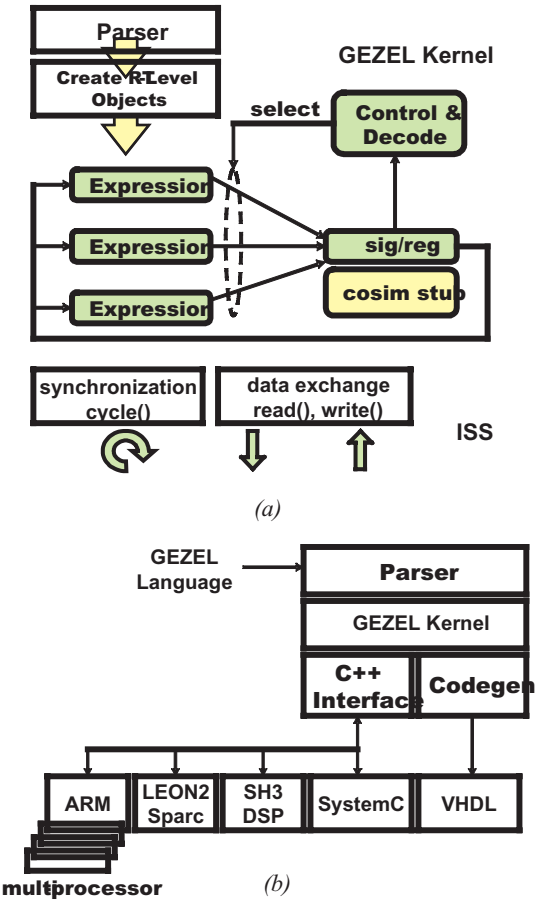

Figure 8: (a) GEZEL is implemented as a C++ library that links to an instruction-set simulator. (b) Several cosimulation interfaces are available, including VHDL

The AES hardware is expressed in a dedicated language called GEZEL that uses FSMD semantics. Figure 7b illustrates (part of) the GEZEL description of the top-level aes_decoder. The code shows how a 128-bit key is assembled out of four subsequent 32-bit data input values. The module aes_decoder contains registers, in addition to signal flow graphs (`sfg`) that contain operations on these registers. Each sfg represents one clock cycle of processing. A module can hierarchically include another one by means of the use statement. A finite state machine (`fsm`) expresses control as a state transition graph that indicates which sfg will execute each clock cycle. One transition takes one clock cycle. Conditional state transitions can be expressed using boolean conditions on data path registers. In the example, the fsm decodes one of three instructions received through ins from the C software. Two of them (`ir==1` and `ir==2`) assemble the key using 32-bit chunks of din. A third one is an idle instruction that is used to synchronize the operation of aes_decoder to the C software on the ISS. As shown in Figure 7a, a single-sided handshake is created by providing an idle instruction after each active instruction.

The aes_decoder module is interfaced to a driver C program by describing the characteristics of each memory-mapped interface. In GEZEL, an ipblock is used for such a memory-mapped interface. An ipblock can have a custom implementation, both for the purpose of simulation as well as for mapping. Finally, the memory-mapped interfaces are connected to the aes_decoder in a system block, which is the top-level GEZEL module.

4.2 GEZEL Architecture

The architecture of the co-simulator for this single-processor system is illustrated in Figure 8a. GEZEL is organized as a C++ library with a built-in parser. The library is linked against the ISS to create a co-simulation environment. A GEZEL description is parsed and converted into simulation objects. These objects are sets of expressions, extracted out of sfg descriptions. The expressions define the values of signals or registers. The control description of each module determines for each clock cycle which expression is valid.

The co-simulation interface consists of two elements: a synchronization interface and a data exchange interface. The synchronization interface keeps the GEZEL description running in coordination with the ISS. The data exchange interface allows to exchange data from the C software to the GEZEL program. To implement a memory-mapped interface, we intercept memory read/write in the ISS and forward those with a matching address to the GEZEL simulation, where they are available as port values on an ipblock.

As shown in Figure 8b, we have created GEZEL co-simulators with several different cores, including ARM, LEON2-Sparc, and SH3-DSP. We also build a co-simulation interface to SystemC to provide seamless co-existence of GEZEL code with SystemC. An important element of the GEZEL environment is the ability to generate synthesizable VHDL code, which closes the path to implementation for RINGS hardware processors.

4.3 ARMZILLA: A multiprocessor simulator

We have built the ARMZILLA environment to evaluate one class of RINGS architectures, namely those that can be built with one or more ARM cores, a network-on-chip, and dedicated hardware processors. Figure 9 illustrates the ARMZILLA setup. There are three components: a hardware simulation kernel (GEZEL), one or more instruction-set simulators (ISS), and a configuration unit. The GEZEL kernel captures hardware models with the FSMD (Finite-State-Machine with Data path) model-of-computation. For the ARM ISS we use the cycle-true SimIT-ARM environment [14]. The ARM ISS uses memory-mapped channels to connect to the GEZEL hardware models. Finally, the configuration unit specifies a symbolic name for each ARM ISS, and

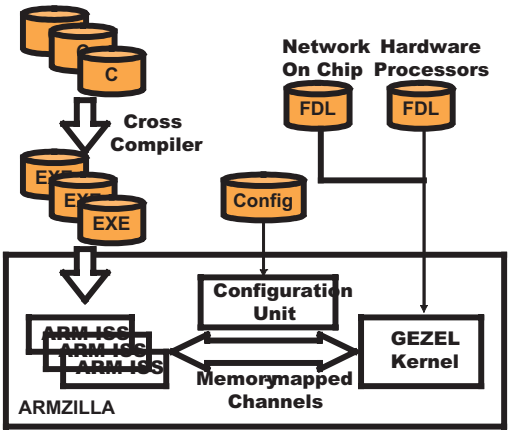

Figure 9: ARMZILLA is a combination of GEZEL and one or more ARM instruction-set simulators.

associates each ISS with an executable. This way the memory-mapped communication channels can be set up, and the hardware GEZEL models can address each ARM memory space uniquely.

5. RINGS ARCHITECTURE ELEMENTS

The RINGS architecture accommodates a heterogeneous collection of architecture elements. These comprise a number of the following components: General-purpose RISC and VLIW processors, DSP and other special-purpose processors, and ASIC. Choosing one or the other is a matter of how well the application is understood and can be molded, in a cost-effective manner, into a specialized architecture.

Indeed, if there are no constraints on performance and power consumption, then a high level simulation model of a system is as good an implementation as the embedded version of it. This implies that a general-purpose processor is the most likely architecture in the absence of performance and power constraints. When performance/power is taken into account, architectures are required which offer a more efficient utilization of the intrinsic computing efficiency of silicon, as discussed in the introduction section. From the system level viewpoint, this means that the programming/configuration mechanism has to be specialized towards the application.

General-purpose FPGA are playing part in this picture insofar they offer a general-purpose, spatial programming platform. They are at the same level of general-purpose programmability as a general-purpose processor. General purpose LUT-based FPGA are a factor of 10 less energy-efficient then dedicated architectures [15]. New generations of FPGA introduce dedicated hard-macros, such as processors, RAM-blocks, multipliers, I/O ports and so on to improve better power-figures.

As we discussed before, the RINGS architecture is a partner in a marriage between application and architecture. Also the application affects the selection of architecture elements. We will illustrate this by comparing the implementation of an embedded web server on different target platforms.

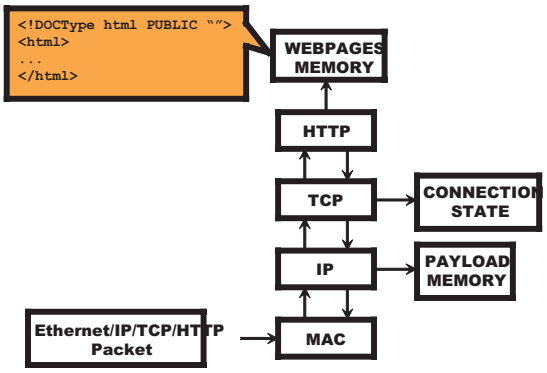

Figure 10: Embedded Webserver Application uses a

5.1 Example: Embedded Web server

Figure 10 shows the hierarchy of protocol stacks used by an embedded web server. Ethernet packets are received by a medium access layer and stripped down and processed subsequently by Internet Protocol (IP), Transmission Control Protocol (TCP) and Hypertext Transfer Protocol (HTTP) stacks. There is a stream of control connecting all these protocol stacks as packets are being decomposed and processed.

We started from a reference implementation in C for such a web server, optimized for execution on an 8-bit controller. Then, we considered what architecture options would be available to improve the protocol stack performance. Up front we ruled out an all-hardware implementation because of the excessive design cost (both in time and in resources). Instead, we considered how existing programmable solutions can be applied optimally to select the correct target architecture.

Two different architectural styles were considered, each with two variations.

One approach was to apply a DSP processor. While a protocol stack has no compute operations that would benefit from the signal processing operators inside a DSP, we must recognize that such a processor is proficient at working with indexed data. We used a TI C54 and a BlackFin processor, both of which have a Harvard-style architecture, and have dedicated index address generation hardware. Moreover, both of these processors have a C compiler, which simplifies the porting of the reference implementation to the new platform.

A second approach was to consider a combination of a classic RISC processor with a coprocessor that can do in-band processing on the packet stream fed into the RISC. We considered two RISC processors, both of them 32 bit. The first is the LEON-2 Sparc processor, the second is the Strong-ARM processor.

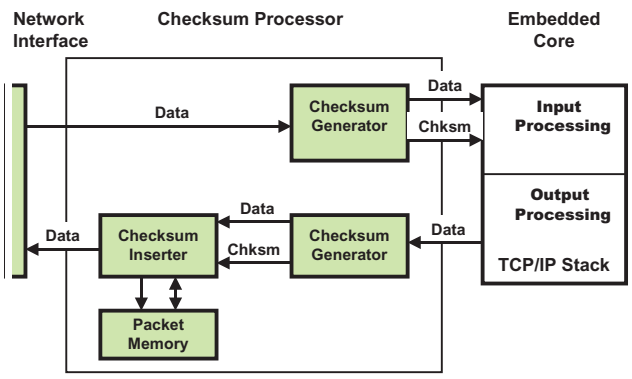

Figure 11: TCP/IP Checksum Generation/Verification

One obvious optimization that helps in all platforms is to convert the 8-bit code to the native word length of the platform, 16 bit for the DSPs and 32 bit for the RISCs. This transformation is not trivial due to various alignment and endianess issues, but still can be expressed in C code.

From an initial profiling of the C code, it was obvious that a major amount of cycles was devoted to TCP/IP checksum verification and insertion. In the case of the DSP processor, the evaluation of this function can be expressed as a multiply-accumulate operation for which native support is available. In addition, the address generation hardware for DSP runs in parallel with the accumulation process. In the case of BlackFin, a dual-MAC unit is available, so that we can perform two accumulates in parallel.

In the case of the RISC-with-coprocessor platforms, the checksum functions are the preferred candidate for off loading to the coprocessor. This is illustrated in Figure 11. 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.

Many other optimizations are possible. We enumerate a few of them to illustrate how the capabilities of a programmable platform are used to improve the performance of an application.

- DSP processors provide circular addressing capabilities for working with FIFO structures, such as needed with digital filters. For checksum operations, those circular addressing modes can be used for efficient network buffer managements.
- Both DSP and RISC processors have pipelines that are sensitive to branches in the code. Loop merging, function inlining and in some cases rewriting conditional statements help to minimize the number of branches during processing.

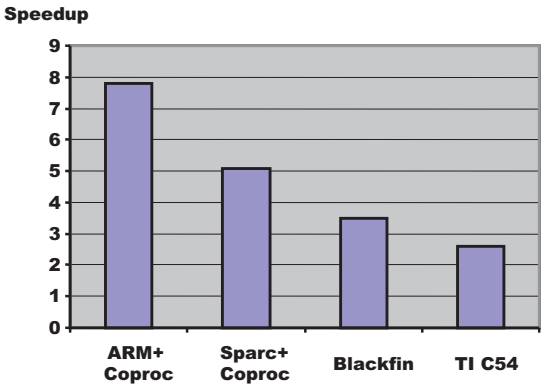

Figure 12: Relative Effect of Optimizing the Webserver for each the target platforms.

- The checksum operation is a 16-bit 1-complement sum. This requires overflow checking and possible sum adjustment after each iteration. However, the DSP have internal dual 40-bit accumulators, and this overflow checking can be hoisted out of the accumulation loop by increasing the accumulator precision. With hand-coding of double precision operations in assembly, it is possible to process 8 bytes for each checksum iteration loop, a dramatic improvement over the 1-byte per iteration in the reference code.
- Modern compilers are proficient in optimizing code towards the architecture. This is an optimization that can improve but however not completely replace application-specific optimization techniques as discussed above.

The result of these optimizations, shown in Figure 12, illustrate the impact of the interaction between application and target architecture for various target platforms. The speedup factor in the figure compares the unoptimized application mapping versus an architecture-optimized mapping on the same platform. This demonstrates that it is vital to know the detailed characteristics of a programmable component in order to use it optimally. At the same time without a detailed insight into the applications, we are clueless on how to use the architecture.

6. RINGS INTERCONNECT

Flexible interconnect is a central theme in RINGS. As was already illustrated in Figure 3, integration is needed at different abstraction levels. In this section, we will describe these interconnection schemes starting at the lowest abstraction level and ramping up in abstraction level up to the software running on the central controller.

6.1 Physical layer reconfigurable interconnect.

Interconnect is used to transport data over a transmission medium between the different components of a system,

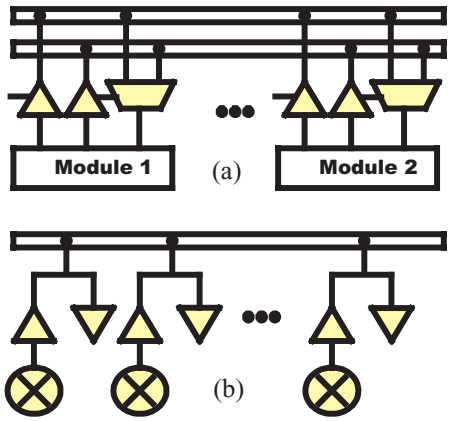

Figure 13: Reconfigurable Interconnect using (a) TDMA and (b) source-synchronous CDMA bus interface

called senders and receivers. Current interconnect schemes are all based on space or time division. Similarly, reconfiguration is confined to a space and a time axis [6]. New interconnect schemes are proposed that introduce frequency and code division or a combination of all above [19].

6.1.1 SDMA - Space division multiple access

If every sender, receiver pair has its own physical transmission medium, e.g. its own metal wire, there will be no access conflicts. But the amount of wires will grow exponential with the required number of sender, receiver pairs. Hence, to keep the space, i.e. number of wires, under control, space and time division multiple access schemes are introduced. This has been the reason to add ever more layers of metal.

6.1.2 TDMA - Time division multiple access

Given a set of metal layers and a set of senders and receivers, the current approach to solve the interconnect demand is to introduce time division. It is illustrated conceptually in Figure 13(a). This has been done in multiple approaches and multiple levels of hierarchy. Busses are the most well-known example of time multiplexing. Multiple protocols to arbitrate the bus have been designed.

Examples of bus architectures are the Amba bus [1]. Since resources are limited there is always a latency, throughput, bandwidth flexibility trade-off.

Providing general multiplexer based reconfigurable interconnect architectures can be very expensive in terms of area and power. It is described in [10], that for a Xilinx XC4003A FPGA, 65% of the power is attributed to interconnect, 21% to clock power, 9% to I/O power and only 5% to the actual calculations (CLB) power. Although this is an older FPGA device, new devices still focus on providing high speed switching matrices.

6.2 CDMA-based Interconnect

To save area and power the physical real-estate needs to be used more efficiently. Yet at the same time, reconfiguration is required. One option to combine both is the CDMA-based reconfigurable inter-connect. Figure 13b shows a conceptual picture of a source synchronous CDMA implementation. Each sender and receiver gets a unique spreading code. By changing the Walsh code, a different configuration is obtained. Traditional buses, which are a TDMA channel, require hardware switches for reconfiguration such as illustrated in Figure 13a. CDMA interconnect has the advantage that reconfiguration can occur on-the-fly.

For example for multi-memory bus systems, source-synchronous CDMA has been proven to improve the communication bottle neck between the CPU and the multiple DRAMs in a processor system [8][9]. The memory wall is a major problem in increasing the bandwidth and latency of DRAM based memory systems, without having to increase the number of I/O pins. By choosing orthogonal codes, two memory requests can be handled by one channel at the same time. As a result, it reduces the latency dramatically. A reconfigurable DRAM uses half the number of high-speed buses of the conventional D-RDRAM and hence reduces the channel power dissipation up to 50%.

The reconfiguration feature brings the DRAM close to the CPU in a dynamic fashion. Similarly, a reconfigurable interconnect on chip can bring co-processors close to the main processor or the memory units in a reconfigurable way. The example of Section 3.5 illustrates this principle. Depending on the application, a different co-processors needs to be close to the CPU, without losing the throughput. In case of cryptographic applications, a small set of specialized co-processors is provided on chip, including e.g. a triple DES, an AES and a hash function. The reason these units are distinct is that the arithmetic and operation is very different. Hence the units can not be merged efficiently. Then a different CDMA code setting, can in a dynamic fashion rearrange the processors depending the application.

6.3 Network-on-chip

The physical layer reconfigurable interconnect busses can be used as components in the network-on-chip interconnect architecture. The reconfiguration settings have to travel down from the application to the CDMA controllers. One systematic way of implementing this, is by using a network-on-chip paradigm. The flexibility is contained within the network topology. Designers can instantiate an arbitrary network of 1D and 2D router modules. Furthermore, they can reconfigure internal buffer size of each router, and in this way, trade area for speed. These two features allow cre-

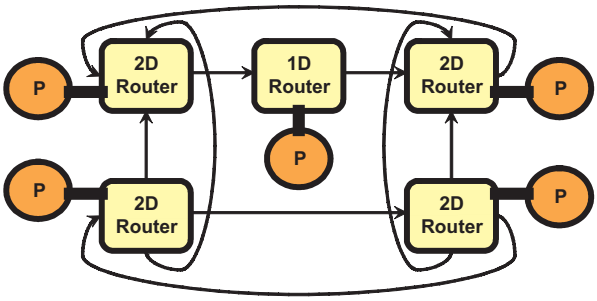

Figure 14: A network-on-chip has a configurable topology made up out of 1D and 2D routers.

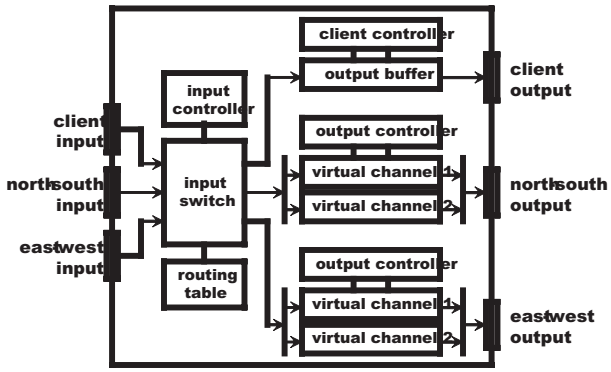

Figure 15: Architecture of a 2D-router.

ation of a network topology that is matched to the traffic patterns of a special purpose SoC.

As illustrated in Figure 15, a 2D router contains three concurrent controllers: an input controller, a router output controller and a processor output controller. The input controller handles simultaneous input requests from neighboring routers and the processors. Priority is given to router inputs because the processor interfaces are driven by software, which is typically slower. A round-robin scheme is employed to arbitrate requests of equal priority. The router output controller and two virtual channels handle communication to neighboring routers. The two virtual channels can avoid deadlocks in a two dimensional torus network topology. Finally, the processor output controller interfaces with the processor core to receive packets from the network. Because the communication between network and processor is handled in a blocking-send and receive manner, an additional output buffer is added between the routing channel and the processor output to relieve possible congestion caused by the blocking. A 1-D router has a similar structure but with a reduced interface and reduced number of virtual channels. A routing table is used to determine the subsequent routing path of each packet.

6.4 Software interfaces

The network-on-chip in RINGS accommodates a number of processors and dedicated hardware processors as network

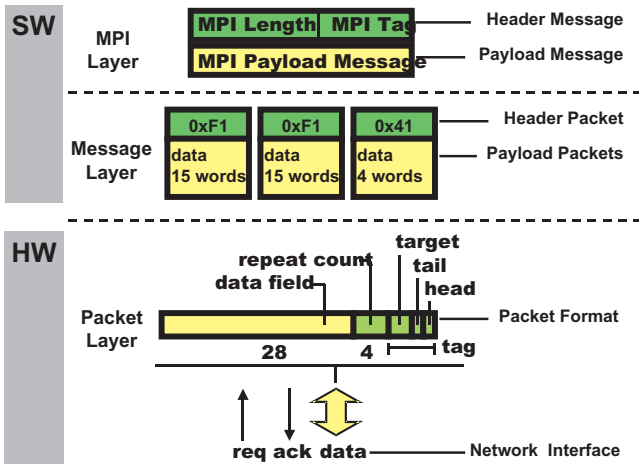

Figure 16: Network Protocol Hierarchy

clients. Using the router modules, these processors exchange information packets. Each packet carries an address stamp that allows to route the packet to the interface of the component with the corresponding address.

We now discuss how a hardware or software processor is interfaced to this on-chip network. The interface between router and processor uses request/acknowledge signaling to synchronize communication between a router and a component. For the hardware processors, these request/acknowledge signals are hardwired. For embedded software on the cores, we use memory-mapped interfaces that map the request- and acknowledge signals into well-defined bits of shared memory locations. In C, we can control the value of the handshake signal by bit-manipulation on the shared memory addresses. The advantage of such a memory-mapped scheme is that it is independent of the processor architecture, so that it can be applied on a broad range of cores. On the other hand the memory bandwidth of a processor is a scarce resource, and therefore we must minimize the number of memory accesses required in executing the synchronization cycle. We paid special attention to this in the development of the communication protocol stacks. At the level of request/acknowledge signaling, we transfer data on both phases of the handshake.

On top of this architecture, we define a three-layer network protocol. While such a protocol must accommodate both hardware and software blocks, we have optimized the protocol stack towards 32-bit embedded processors. The three layers in our protocol are the packet layer, the message layer, and the Message Passing Interface (MPI) layer. Figure 16 shows the data format used by each abstraction level.

The packet layer is at the lowest abstraction level and takes care of the transport of individual packets in the network-on-chip. The lower-order bits of a network-on-chip packet are used for control signalling, and include a header bit, a tail bit and a target network address. This field is

called the packet tag. A header bit signals the first packet of a longer stream, while a tail bit indicates the final packet. Beyond the 4-bit tag, a data word of 32 bits is available. The lowest nibble of this word has the additional special meaning of repeat count. When the repeat count field has a non-zero value N, then the router hardware will generate the packet tags automatically for the next N words. During this repeat mode, the embedded software can use all 32-bits of the memory bus for data transfer.

The next two layers are focused on embedded software. The message layer transports arbitrary length messages in chunks of 16 packets, using the repeat counter mechanism described earlier. The first packet is formatted as a header packet with a nonzero repeat counter field.

The MPI layer defines an embedded interface compliant with the MPI standard. The basic MPI semantics are blocking send/ blocking receive communications of arbitrary length messages. Each message also has a symbolic tag that can qualify a logical channel or a particular message data type. In our implementation, we map each MPI message into a header message and a payload message. The header message specifies the length of the payload as well as the MPI-standard tag field. The payload message can contain a variable number of words.

We validated this architecture using the ARMZILLA design environment. ARMZILLA supports cycle-true co-simulation of one or more ARM instruction-set simulators (ISS) embedded in a user-specified hardware model. The hardware model captures the network-on-chip as well as any additional hardware processor.

The performance metrics of each abstraction layer in our protocol, as measured with ARMZILLA, are illustrated in Table 1. The figures show the cycle cost of each abstraction layer for a ping-pong message - a message that is send from A to B and back. The table points out that higher abstraction layers for design induce considerable overhead. Clearly the comfort and flexibility of MPI-modeling comes at a price.

Table 1: Protocol Overhead for ping-pong messaging in cycles per word (cpw).

Abstraction Level	**Payload = 8 words**	**Payload = 64 words**
MPI-Layer (SW)	135 cpw	62 cpw
Message Layer (SW)	72 cpw	42 cpw
Packet Layer (HW)	8 cpw	8 cpw

7. RECONFIGURABILITY AND EMBEDDED SOFTWARE

As illustrated in Figure 3, the integration of domain-specific processors proceeds at two level. Specialized interconnect architectures as discussed in Section 6 integrate the architecture. Software interfaces, which we will discuss next, integrate the application.

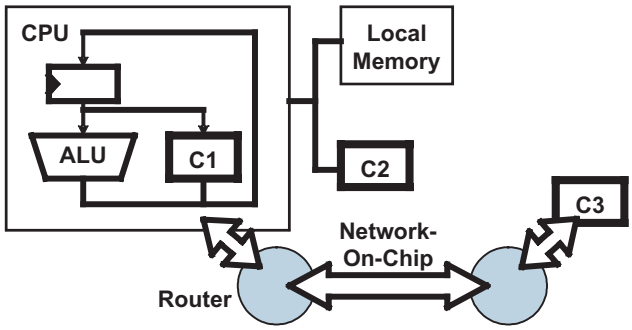

Figure 17: Coarse-grain Reconfigurable Blocks are Register-Mapped (C1), Memory-Mapped (C2) or Network-Mapped (C3)

Figure 17 demonstrates that there are three options of coupling the embedded software and the domain-specific co-processors. These levels correspond to the level of integration between the CPU and these acceleration hardware units. We distinguish register-mapped, memory-mapped and network-mapped reconfigurable blocks.

7.1 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 data-path 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.

7.2 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 pro-

Table 2: 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	Tight	Low	Custom ISS	Custom Compiler
Memory-Mapped	Coprocessor	Memory-mapped Instructions	Loose	Medium	ISS/Coprocessor Cosimulation	Software Library (function call)
Network-Mapped	Peer Processor	Configuration Packets	Uncoupled	High	ISS/Coproc/NoC Cosiumulation	Communication primitive

cessors. 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 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.

7.3 Network-Mapped Reconfigurable Blocks

Reconfigurable blocks can also be attached as independent entities in a Network-on-Chip. In this case, integration of embedded software and reconfigurable blocks can be done using communication primitives.

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.

7.4 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, but the complexity of the co-simulation 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.

8. CONCLUSIONS

In this paper, we pointed out the matching making process of architecture and application in the context of reconfigurable systems. We believe the ability to efficiently connect the world of programs and the world of architectures will become a central topic in reconfigurable design. This is because of two reasons: on the one hand, because

future system design is becoming prohibitively expensive and it makes second guesses out of the question, and on the other hand, because general-purpose programmable systems alone cannot reach the energy efficiencies required for a pervasively connected world.

In the paper, we outlined the tools that help us answer this question. It requires to subdivide a system in its composing domains, and approach each of these domains individually for optimal energy-efficiency. Next, the domain implementations are integrated back into a system, at the architecture level using a flexible interconnect scheme, and at the algorithm level using a layer of integration software. We proposed the RINGS architecture as the architecture template for such systems and the GEZEL design environment to support the design of RINGS applications.

9. ACKNOWLEDGEMENTS

The authors would like to thank the support of NSF grant CCR-0310527, SRC grant 2003-HJ-1116, and Atmel, Panasonic and Xilinx through UC-Micro 02-079.

10. REFERENCES

[1] ARM, Amba Specification, available from www.arm.com

[2] Benveniste, et al, *The synchronous languages 12 years later*, Proceedings of the IEEE, Vol 91(1), Jan 2003

[3] Carey, *Palms latest wireless wonder*, EETimes, Under-the-hood, April 2002.

[4] Catthoor, et al., *Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer Academic Publishers, Boston, 1998.

[5] Claasen, *High speed: not the only way to exploit the intrinsic computational power of silicon*, Proc. Solid-State Circuits Conference 1999 (ISSCC 1999), pp. 22-25.

[6] A. Dehon, J. Wawrzynek, Reconfigurable computing: what, why, and implications for design automation, Proc. DAC 1999, pg. 610-615.

[7] Harel, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, 8(3):231--274, June 1987.

[8] J. Kim, Z. Xu, M.F. Chang, Reconfigurable Memory Bus Systems using Multi-Gbs/pin CDMA I/O Transceivers, Proc. ISCAS, pg. II-33- to II-36, 2003.

[9] J. Kim, Z. Xu, M.F. Chang, A 2Gb/s/pin Source Synchronous CDMA Bus Interface with simultaneous multi-chip access and Reconfigurable I/O capability, Proc. IEEE CICC, Sept. 2003.

[10] E. Kusse, J. Rabaey, Low-energy embedded FPGA structures, Proc. 1998 International Symposium on Low Power Electronics and Design, ISLPED 1998, pg. 155 -160

[11] E.A. Lee, D. Messerschmitt, "Synchronous data flow," Proceedings of the IEEE, Vol. 75, no.9, Sept. 1987.

[12] P. Mosch ,G. van Oerle, S. Menzl, N. Rougnon-Glasson, K. Van Nieuwenhove, M. Wezelenburg, A 660- W 50-Mops 1-V DSP for a hearing aid chip set, IEEE Journal of Solid-State Circuits, vol. 35, pp. 1705 - 1712, November 2000.

[13] P. Murthy, E.A. Lee, "Multi-dimensional Synchronous Data flow graphs" IEEE Transactions on Signal Processing, Vol. 50, No. 7, July 2002.

[14] W. Qin et al., *Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation*, Proc. DATE 03, March 2003, Munchen.

[15] J. Rabaey, Platforms for the next generation wireless systems - What role does Reconfigurable Hardware plays? Proc. FPL 2000, LNCS 1896, pp. 277-285, 2000.

[16] C. Rowen, *Reducing SoC Simulation and Development Time*, IEEE Computer, 35(12), pp. 29-34.

[17] P. Schaumont, I. Verbauwhede, K. Keutzer, M. Sarrafzadeh, A Quick Safari through the Reconfiguration Jungle, Proceedings Design Automation Conference, DAC-2001, Las Vegas, June 2001, pg. 172-177.

[18] I. Verbauwhede, C. Scheers, J. Rabaey Analysis of Multi-dimensional DSP Specifications IEEE Transactions on Signal Processing, Vol. 44, No. 12, December 1996, pp. 3169-3174.

[19] I. Verbauwhede, M.F. Chang, Reconfigurable Interconnect for next generation systems, Proc. ACM/Sigda 2002 International workshop on System Level Interconnect Prediction (SLIP02), Del Mar, CA.

[20] W. Verhaegh, et al. A Two-Stage approach to Multidimensional Periodic scheduling, IEEE Transactions on CAD, Vol. 20, No. 10, October 2001.