Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores

Holger Lange and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 13, D-38106 Braunschweig, Germany
lange, koch@eis.cs.tu-bs.de

Abstract. *Complex SoC and platform-based designs require integration of configurable IP cores from multiple sources. Even automatic compilation flows from a high-level description to HW/SW systems can benefit from having access to reusable sophisticated hand-optimized IP blocks. This work proposes the Parametric C Interface For IP Cores (PaCIFIC) to allow the automatic embedding of complex IP cores in a high-level language such as C. PaCIFIC provides for formal description of IP behavior and interface characteristics as well as an idiomatic programming style natural for software developers. Additionally, the techniques for integrating PaCIFIC into a compile flow and interfacing IP cores with automatically generated data paths will be discussed.*

1 Introduction

In many of current design approaches such as systems-on-chip (SoCs), embedded systems and platform-based techniques involving hardware-software codesign, a gap appears in the design flow at the interface between hardware and software. The individual hardware and software sub-flows (from RTL to layout and software to binary code) themselves are quite mature with regard to tool support, but the interface between both requires significant manual effort to establish [1][10]. This applies even more strongly if the hardware contains IP cores, as these often feature complex functionality and interfaces. The challenges the designer has to cope with include large system-specific parameter sets that span a huge space of possible combinations, not all of them legal. While configuration management is already well-explored [7][8][9], this paper concentrates on HW/SW interface design.

Two aspects play key roles in interface design. First, the interface functionality itself has to be partitioned between HW and SW realizations. Second, concrete interface mechanisms and protocols must be determined (e.g., physical connections, address ranges, transfer modes, device drivers, etc.). Both of these issues require the designer to explore a large design space, a time consuming and sometimes tedious task despite initial efforts at tool support [3].

This work focuses on the latter aspect in the context of using an ANSI C language description to embed, compose and interact with IP cores. ANSI C has been chosen as this work is part of larger project dealing with automatic HW-SW partitioning and C-to-HW compilation [4][5], enabling pure SW developers inexperienced in HW design to benefit from hardware acceleration in the context of adaptive computers. Different from building SoC by connecting IP cores and processors via on-chip

buses, a reconfigurable computer offers core integration directly into the custom datapath. As a solution, we propose the Parametric C Interface For IP Cores (PaCIFIC). It establishes an automatic design flow presenting convenient, simple C interfaces (function prototypes) to a software programmer. Our approach hides the formal descriptions of IP- or platform behaviors and interface characteristics by encapsulating them together with other IP configuration data in a dedicated repository [9].

2 Related Work

Tomiyama et. al. [2] compare several Architecture Description Languages (ADL) and determine the characterizing properties to be behavior- and structure description. They demand an explicit behavior description of processors for better compiler generation. However, they consider synthesis-based ADLs or Hardware Description Languages (HDL) neither sufficiently easy-to-use nor flexible enough for this task. Balboa [3] is a HW/SW codesign framework for system models. It abstracts IP interfaces in a two-fold intermediate layer consisting of a Component Integration Language (CIL) and the Balboa Interface Description Language (BIDL) providing automatic data type matching and interface generation. The IP behavior is implemented as C++ models. The CoWare N2C suite [10] contains a set of interface behavior descriptions expressed as prototypes or templates specialized in many detailed descendants. Despite their great number, the behavior descriptions are not universal and cannot replace a behavior description language. Handel-C [11] is an extension to the C language with explicit parallelism, hardware data types and inter-thread communication channels based on the model of Communicating Sequential Processes (CSP) [13]. SystemC and Synopsys Behavioral Compiler both provide abstractive interface modeling, but the hardware extensions of SystemC are not accepted by the software community, even less so the HDL input required by Behavioral Compiler. Carloni et. al. [16] construct an interface mechanism based on latency insensitive protocols. Thronicke [7] and Zeller [8] present configuration management methods from hard- and software domains. At present, there seem to be no attempts to combine configuration management and ADLs, although this would appear advantageous when building systems of complex IP cores and software.

3 Problem Description

Consider a scenario with two IP cores which should be arranged forming a pipeline. Assume that each core has one input and one output interface.

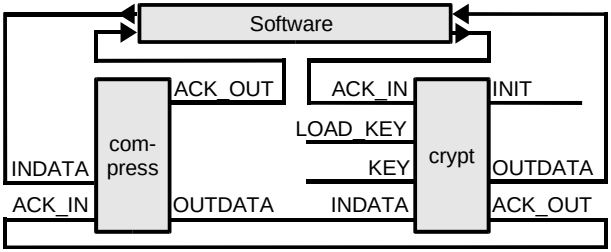

Fig. 1. Hardware pipeline used by software

As shown in Fig. 1, the data path comprising both cores is supposed to be used from a software description that also sources and sinks the data. A natural approach for plain software would consider the two IP cores to be C functions, leading to the following code:

```
int *indata, *outdata, *intermediate;
for (n = 0; n < 64; n++) {
  compress (indata++, intermediate);
  crypt (intermediate++, outdata++);
}
```

From such a code description, the HW pipeline shown in Fig. 1 should be automatically inferred. This requires additional information about the hardware "functions" **`compress`** and **`crypt`**. The software developer should not have to be aware of the actual mechanisms involved in realizing the structure.

To this end, several issues must be addressed when dealing with hardware embedded in a software description:

- Recognition of IP cores
  Since C cannot distinguish between hardware and software, function calls aiming at IP core instances have to be detected somehow.

- Low-level interface control
  In contrast to HDLs, plain C has no notion of timing- or cycle-accurate execution schedules. Thus, for each IP core, interface parameters like signal timing, handshaking and bus arbitration must be provided in an external representation.

- Data transfer
  There are several ways to exchange data between software and hardware. IP cores are often programmed via register files. Thus, a Programmed I/O (PIO) mode is mandatory in this case. On the other hand, this is highly inefficient for the large data sets which are commonly processed by complex IP cores (video, networking). In these cases, Streaming I/O (SIO) mechanisms are generally employed, often assisted by rate matching and buffering using FIFOs. We will refer to such a setup as a *stream engine*. For each use of an IP core, the appropriate transfer method used has to be determined based upon data-traffic statistics and interface descriptions delivered by the IP provider.

- Hardware events
  Some transactions are initiated not by the software, but by the IP core, e.g., the

acceptance to process the next data block. Asynchronous events such as interrupts or error notifications are beyond the semantics of a C function. The functional synchronization, such as the indication of the current state of a hardware function, must be realized, for example, to determine the end of a C function call (=IP core execution) and proceed with the rest of the program.

4 Proposed Solution

PaCIFIC consists of rules for an idiomatic programming style which must be used when embedding IP cores in a C source program, and interface control semantics which describe the interface behavior of an IP core (see Fig. 2). To this end, PaCIFIC includes a data model and human-readable description language for the characteristics of individual IP blocks as well as entire platforms. In this paper, we will only concentrate on the scope of IP blocks. All components are tied together in a number of dedicated compiler passes that perform the necessary analysis and synthesis steps for both hard- and software. These extra steps access the PaCIFIC descriptions to find idiomatic HW function calls in the C source program. For the first practical realization, the Compiler for Adaptive Systems (COMRADE) [4][5] will act as the host compiler. PaCIFIC enables COMRADE to access and integrate IP cores which are too complex to be generated automatically from a software description, especially if they are hand-optimized. For brevity, the details of PaCIFIC's integration into COMRADE have to be omitted.

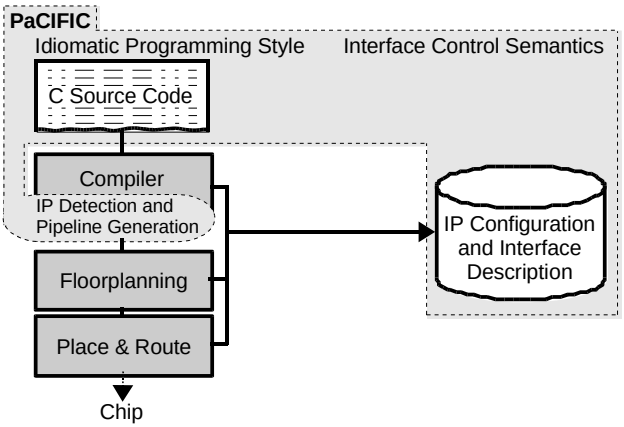

Fig. 2. Design flow with PaCIFIC

The data models and representations are based on the study of more than thirty commercial IP blocks, that were classified using the attributes of the PaCIFIC interface template [12]. The aim was the capability to describe all of the IP cores' interface semantics with the existing attribute catalog. The majority of the evaluated cores belong to the domains of multimedia and networking. The first cores generally presented a data path oriented interface, with the video or audio stream processing being the main task. In contrast, the networking IP cores employed a processor-based

register interface. More complex IP blocks even use multiple different interfaces of both kinds.

5 Hardware Interface Description

The PaCIFIC interface description [12] is used to define the static properties for all IP interfaces as well as the dynamic flow of the interface protocols based on synchronous logic operating without a central flow control authority. As usual, properties are expressed as attributes and values. Some of the many defined attributes are:

- Identification (class, type, version, name).
- Auxiliary information (author, comments).
- Port definitions (transaction type, direction, width, associated clock, abstract data type, associated address, handshaking protocol, data traffic statistics, bus arbitration).
- External resources required by the IP core and their allocation modes (shared, exclusive, persistent). This might include external memories or special I/O requirements (e.g., access to multi-Gbps transceivers).

Port transaction types and handshaking protocols will be examined in more detail in Section 5.1. A fragment of an interface template for the **`crypt`** IP core is shown below:

Example: Part of PaCIFIC Interface for the crypt IP

```
interface crypt
  type:    custom
  version: 1
  port INDATA
    transaction: data
    direction:   in
    width:       32
    sequence repeat: inf
      bigendian: 32 bit signed
    end sequence
    enableout:   name ACK_OUT offset 0 latency 0
  end port
  port ...
...
end interface
```

The abstract data type defined by the **`sequence`** block of the example (here just a single scalar integer), plays a central role in the data exchange between software and hardware. It arranges the nature, order and count (**`repeat`**) of the data items that are transferred over the port or bus. Every sequence block corresponds to formal parameter of the fictitious C function representing the IP core.

Interface *templates* can be used to group and reuse the same or similar interfaces in a fashion analogous to the classes and inheritance of object-oriented programming.

5.1 Interface Protocol Description

The fundamental interface flow control mechanism in PaCIFIC is a handshaking scheme which consists of an incoming and an outgoing signal per port. For an outgoing signal, the asserted state (selectable as high or low) means that the IP block is ready to consume data (on an input port) or that data is waiting to be fetched (on an output port). The incoming signal is the outgoing signal from the connected port at the other end of the communication. It is not necessary to specify both signals, a one-way handshake is possible as well as no handshake. A transaction is considered complete when all specified handshake signals are active at a clock edge. If both signals are specified, it is illegal to reset the first active signal before the second signal has been activated. For all handshakes, a time offset or initial latency with regard to another handshake may be specified. Additionally, an interrupt semantic can be selected for the handshaking signals. This is useful if no actual data transfer is required during the transaction, e.g., to indicate that data must be fetched from a mailbox register.

When such static interface properties no longer suffice to describe the characteristics of an IP core's interface, an enhanced version of the FLAME UCODE notation is employed [6] to describe dynamic behavior. A UCODE block is a list of statements most of which are executed sequentially. It represents the state machine of an interface controller. An excerpt of the UCODE statements is shown below:

- The **`level`** statement asynchronously sets ports to the values given as arguments of the form **`port=value`**.
- The **`posedge`** statement is similar to **`level`**, but operates synchronously with a rising clock edge.
- The **`continue`** statement takes three kinds of parameters: an optional **`timeout: n`**, optional **`error: port=value`** expressions and normal **`port=value`** expressions which are interpreted as conditions. The first two branch to the **`exception`** block either if all error conditions are true or the timeout in clock cycles has expired. If no timeout or error occurs, the control flow is halted until all normal continue conditions are valid. As stated in [6], multiple conditions in the same **`continue`** statement are logically ANDed, multiple successive **`continue`** statements are ORed. The asynchronous **`continue`** statement can be synchronized by a following **`posedge`**.
- The **`exception`** block, if present, is located at the end of the UCODE block. It marks the branch target for all **`error`** and **`timeout`** clauses and puts the interface or IP block into a well defined error state. The normal control flow terminates, if the **`exception`** block or the end of the UCODE block is reached.
- The mandatory **`transfer`** ***`n name`*** block represents the transfer of ***`n`*** sequences to port ***`name`***, with the nature of the sequence being defined in the PaCIFIC interface description. Without a sequence description on a port, **`transfer`** indicates ***`n`*** scalar transfers using the full port width. It acts as a loop in the UCODE control flow. Each iteration is triggered by the handshaking protocol defined for the port.

6 Software Interface Description

The last sections dealt with the hardware realization of the interface to the IP cores. In this section, the corresponding software mechanisms will be examined.

From the study of multimedia IP cores it is obvious, that a powerful data streaming service is needed to source and sink the data path interfaces of the IP. The stream engine fetches and stores data from respectively to shared memory, which is accessible to the SW running on the CPU. The start address of the memory range to be streamed can be expressed as a pointer to C structures reflecting the composition of the sequences defined in a PaCIFIC interface.

In all cases, the IP cores also require programming (e.g., for initialization) using a register interface. This can be realized by simply mapping the registers into a SW accessible memory region (but not necessarily the main memory space).

To recognize the actual IP core embedding and establish both communication methods, an idiomatic C programming style is required: Only two modes of instantiating IP cores from C are supported by PaCIFIC, but they are sufficient to cover all interface types under discussion.

First, there is the fully automatic interface generation, which results in the creation of read and write *primitives* for access to the ports of the IP core in both direct (register) and streaming fashions. This method works from the PaCIFIC interface definition, the IP designer (or more precisely, the author of the PaCIFIC description) does not have to provide any additional data. However, the SW has to explicitly call the primitives in the required order to actually get the IP core to perform the desired function.

Second, there are functions which atomically perform complex operations without requiring incremental prodding by a SW program. For the realization of these *monoliths*, the IP designer has to supply an algorithmic description of the control and data patterns that must be applied to the interfaces of an IP core for the required function. The monoliths are then generated automatically and their call resembles conventional C library functions (all individual control steps have been hidden and implemented automatically).

Note that POSIX threads still work with PaCIFIC enabling parallel execution of HW and SW. This can be beneficial when calling data-intensive IP cores, e.g., while the HW is still running, the SW prefetches the next data block into memory and writes processed data to disk.

6.1 Primitives

Consider an input port without an associated address that is 32 bits wide (cf. example in Section 5). For this case, the C function **`write_indata`** is generated. It writes 32-bit integers (sequence **`bigendian: 32 bit signed`**) and terminates data dependently (**`repeat: inf`**):

```
void write_indata (int *data);
```

Unrelated to the previous example, an output port with an associated address that delivers a sequence of composite data items (here mapped to the **`struct comp`**) produces the following function:

```
void read (int address, struct comp *data);
```

If the **`repeat`** value in a sequence definition equals one, SW wrapper functions may be used to eliminate the unwieldy pointer in favor of just passing scalar data. Primitives are most suitable for use with simple register- or memory-style interfaces.

6.2 Monoliths

Due to the strictly sequential semantics of C, it is not possible to directly describe pipelined accesses using primitives. However, this is achievable using monoliths. The example below reconsiders the compress-crypt scenario from Section 3 and describes the underlying control protocol for the behavior “encrypt” in PaCIFIC-extended UCODE [6] (see also Section 5.1).

```
behavior encrypt
proc crypt(plaintext, ciphertext)

; load key
posedge LOAD_KEY=1
        KEY=10027821 ; fixed key
posedge LOAD_KEY=0

; process single data item
transfer 1 INDATA
  level    INDATA=plaintext
           ACK_IN=1
  continue timeout: 16
           error: INIT=0 ACK_OUT=1
  posedge  ciphertext=OUTDATA
  level    ACK_IN=0
endtransfer INDATA

; wait for end of pipeline flush
exception
continue INIT=1
; execution terminates here
end behavior
```

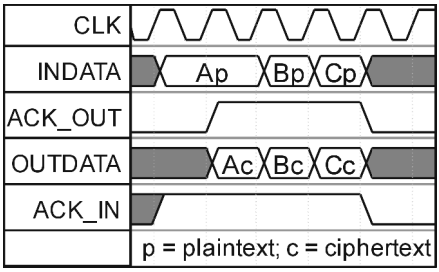

Fig.3. Signal timing for the crypt IP

The function prototype in the **`proc`** statement corresponds to the C function, with variables being passed by reference. Fig. 3 displays the signal timing described by the **`transfer`** block above with **`INIT`** := 1.

The following sequence is defined in the PaCIFIC specification for the port **`INDATA`**:

```
; data
sequence
  bigendian 32 bit signed
endsequence
```

From this description, the **`crypt`** function in the example of Section 3 can be generated. The function terminates after encrypting one data word from the memory pointed to by **`plaintext`** and delivering it to **`*ciphertext`**:

```
crypt(int *plaintext, int *ciphertext)
```

7 Experimental Results

To evaluate the feasibility and efficiency of the PaCIFIC approach, the Xilinx High-Performance 16-Point Complex FFT/IFFT [14] from the Core Generator was coupled to an ANSI C program applying the PaCIFIC algorithms manually, since an automatic tool flow is not yet available. The FFT expects data to be continuously streamed to its input buses as well as from its outputs. For simplicity, the 16 bit real and imaginary buses are combined to 32 bit buses carrying complex numbers. The output data is available after an initial latency of 82 cycles. To efficiently source and sink data, two stream engines are employed with a FIFO capacity of 256x32 bit each.

The test platform was an ACE-V card [15]. The relevant platform hardware used in this scenario was a microSPARC IIep processor at 100 MHz with 64 MB of DRAM and a Virtex 1000 -4 FPGA. The microSPARC accesses the FPGA via PCI and a PLX PCI9080 local bus bridge. For comparison, the FFT was exercised on a second test platform, an ADM-XRC card attached via PCI to a standard PC (AMD Duron 800 MHz, 256 MB SDRAM). The ADM-XRC is a subset of the ACE-V providing the same Virtex FPGA and PLX local bus bridge.

The C program executed by the processor reads the source data from a file into the DRAM, calls the FFT hardware implemented on the FPGA and finally writes the result back to disk:

```
int main(int argc, char* argv[]) {
  FILE* infile, * outfile;
  int* dram_in, * dram_out;

  infile   = fopen("time.dat", "r");
  outfile  = fopen("freqspec.dat", "w");
  dram_in  = calloc(16384, sizeof(int));
  dram_out = calloc(16384, sizeof(int));
  fread (dram_in, sizeof(int), 16384, infile);
  vfft16(dram_in,  dram_out);
  fwrite(dram_out, sizeof(int), 16384, outfile);
...
}
```

The FFT logic is sourced and sinked by two stream engines co-located on the FPGA which access the DRAM in bus master mode. The naive approach without PaCIFIC would require to set up the stream engines and the control signals for the FFT manually, which PaCIFIC combines in one C function call.

After application of the PaCIFIC algorithms, the software part was compiled using gcc, while the resulting RTL description for the stream engines and interface control logic was synthesized with Synplify 7.3.3. It was subsequently mapped with ISE 6.2.01i, integrating the FFT core netlist underway. The achievable clock speed without optimized floorplanning for the mapping results in Table 1a is 27 MHz.

Table 1. FFT mapping results (a) and performance results (b) with PaCIFIC

a)	Area Slices	Total V1000	BSR*	Total V1000
FFT	1386	11%	0	0%
S/I*	1385	11%	4	13%
Sum	2771	22%	4	13%

*S/I: Stream engines and interface control; BSR: BlockSelectRam

b)	Clk cycles ACE-V	Clk cycles ADM-XRC/PC
S/I* read startup latency	8	8
PCI read startup latency	39	29
FFT processing	4178	4178
Memory transfer overh.	4096	4096
PCI processing overh.	8036	6333
PCI write flush overh.	199	58
Sum	16556	14702

Table 1b shows the performance results for the FFT processing 4096 words on both ACE-V and ADM-XRC/PC at 27 MHz FPGA clock. The time spent in software processing is not considered here since it depends mostly on the host's file I/O capabilities rather than PaCIFIC interface design assuming that a naive approach would also access memory in master mode.

8 Conclusions and Future Work

We presented PaCIFIC, a strategy for using complex IP cores from within ANSI C programs as seamlessly as pure C software functions. The HW-specifics unfamiliar to a SW developer are encapsulated in the PaCIFIC framework. Instead, the IP provider supplies the details required for core integration as a machine-readable description. The HW/SW interfaces are then generated automatically without user intervention, thus raising design productivity by closing the gap between the vertical hardware and software design flows.

This approach applies not only to COMRADE or the specific domain of adaptive computing systems, but generally to all HW/SW co-design environments. The unified notation for IP configuration and interface protocol description enables (semi-) automatic design composition. Reusable interface descriptions allow the separation of interfaces and implementation details. Although most ideas we presented are known separately, their combination catalyzes a new and easy-to-use HW/SW codesign flow. Hence, PaCIFIC is applicable to the entire spectrum of SoC, platform-based and derivative designs.

Our future work will concentrate on the actual implementation of tool support for PaCIFIC within COMRADE.

References

1. **P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau.** EXPRESSION: An ADL for System Level Design Exploration. Technical Report, University of California, Irvine, USA, 1998
2. **H. Tomiyama, P. Grun, A. Halambi, N. Dutt, A. Nicolau.** Architecture Description Languages for Systems-on-Chip Design. 6th Asia Pacific Conference on Chip Design Language, Fukuoka, Japan, 1999
3. **F. Doucet, M. Otsuka, S. Shukla, R. Gupta.** An Environment for Dynamic Component Composition for Efficient Co-Design. Proc. Design Automation and Test in Europe, 2002
4. **N. Kasprzyk, A. Koch, U. Golze, M. Rock.** An Improved Intermediate Representation

for Datapath Generation. International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, 2003
5. **N. Kasprzyk, A. Koch.** Advances in Compiler Construction for Adaptive Computers. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 2001
6. **A. Koch.** FLAME: A Flexible API for Module-based Environments - User's Guide and Manual. TU Braunschweig (E.I.S.), Braunschweig, Germany, 2003
7. **W. Thronicke.** Konzept und Realisierung einer allgemeinen Parametrisierungsstrategie von Systemmodellen unter besonderer Berücksichtigung der Wiederverwendbarkeit, PhD thesis, University Paderborn, Germany, 2000
8. **A. Zeller.** Configuration Management with Version Sets, PhD thesis, TU Braunschweig, Germany, 1997 (http://www.infosun.fmi.uni-passau.de/st/papers/zeller-phd/)
9. **H. Lange, M. Radetzki.** IP Configuration Management with Abstract Parameterizations. Proc. International Workshop on IP Based SoC Design, Grenoble, France, 2002
10. **CoWare Inc.** N2C Scenario Library, 2001
11. **Celoxica Ltd.** Handel-C Language Reference Manual, 2001
12. **H. Lange.** PaCIFIC. Technical Report, TU Braunschweig (E.I.S.), Germany, 2003
13. **T. Hoare.** Communicating Sequential Processes. Prentice Hall International Series in Computer Science, 1985
14. **Xilinx Inc.** High-Performance 16-Point Complex FFT/IFFT V1.0 Product Specification, http://www.xilinx.com
15. **A. Koch.** A Comprehensive Prototyping Platform for Hardware-Software Codesign. Workshop on Rapid Systems Prototyping, Paris, France, 2000
16. **L. Carloni, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli.** A methodology for correct-by-construction latency insensitive design. International Conference on Computer-Aided Design, San Jose, USA, 1999