

Modeling and Synthesis of Hardware-Software Morphing

(Invited Paper)

Dirk Koch, Christian Haubelt, Thilo Streichert, Jürgen Teich
 University of Erlangen-Nuremberg; Am Weichselgarten 3; 91058 Erlangen (Germany)
 Email: {dirk.koch, haubelt, streichert, teich}@cs.fau.de

Abstract—In state of the art hardware-software-co-design flows for FPGA based systems, the hardware-software partitioning problem is solved offline, thus, omitting the great flexibility provided through partial runtime reconfiguration. The decision which functions are best suitable to be implemented in hardware or software, is typically taken with respect to the expected worst case computational demands and certain objectives like power consumption, throughput or cost. However, if these parameters change at runtime, e.g., due to environmental changes, traditional designed systems lack to adapt to the new conditions, because the hardware-software partitioning is static.

In this paper we will systematically present a new methodology that allows to change the implementation style of tasks at runtime by *hardware-software morphing*. Based on a formal model, we will demonstrate, how morphing can be performed without losing internal states. Moreover, we will present results from applying our methodology to a 16-tap FIR filter.

I. INTRODUCTION

The hardware-software partitioning has massive impact to the overall performance of a system. As a consequence, research has been done to allow for taking the final decision what functionality should be implemented in hardware or in software as late as possible [1]. Another approach [2] proposes to profile CPU operations, in order to build hardware accelerators on the CPU instruction level at runtime. In contrast, our work aims to perform the online partitioning on the task level. All these techniques require generic hardware-software interfaces [3] between these two domains. The hardware-software partitioning is typically guided through objectives like throughput or power consumption. However, the estimated parameters and thus, the objective values may change significantly at runtime. For instance, computational demands may vary due to environmental changes.

In former publications [4], we have shown how to perform hardware-software partitioning online in distributed embedded systems. For this purpose, new approaches for task migration have been proposed [5], [6]. However, to make full use of online hardware-software partitioning, we must be able to change the implementation style of a task at runtime.

In this paper, we propose a new concept called *hardware-software morphing* that permits to change the implementation style of a task between hardware or software execution at runtime without losing internal states. Starting from a unified task specification, our approach generates hardware modules and software binaries plus the required interfaces that allow a morphing of the specific task.

In the following sections, we will present the assumed task model (Section 2) and we will further discuss, how states can be transferred between the hardware and the software domain (Section 3). In Section 4, a case study of a 16-tap FIR filter demonstrates the applicability of our proposed methodology.

II. TASK MODEL FOR MORPHABLE TASKS

We will base our discussion on hardware-software morphing on the SysteMoC library [7]. SysteMoC is a System C library that allows for specifying and simulating so called *models of computation* (MoC) [8]. Before discussing hardware-software morphing, we will briefly review a derived model of SysteMoC. Our SysteMoC model is represented by a task graph $g = (A, E)$, where A denotes a set of actors and E specifies a set of edges connecting actor outputs with actor inputs. An actor $a \in A$ represents a task and is defined by a 3-tuple $T = (\mathcal{P}, \mathcal{F}, \mathcal{R})$, where $\mathcal{P} = I_p \cup O_p$ is a set of input ports I_p and output ports O_p , \mathcal{F} is a set of functions $f \in \mathcal{F}$ (also called the task functionality), and \mathcal{R} is the so called firing FSM. The *communication behavior* of an actor is encoded in its firing FSM. The firing FSM is described by a quintuple $(I, O_{fire}, S_R, \delta, s_0)$, with a set of inputs $i \in I$, a set of signals O_{fire} controlling the activation of the task functionality \mathcal{F} , a set of present states S_R , a state transition function δ , and an initial state s_0 . The inputs $i \in I$ are Boolean values obtained by *activation functions* that indicate if a function can be started and run to completion, i.e., if the input ports contain all operands required for a complete run of a function and if the available space on output ports can accept all produced results.

As an example consider Figure 1. Task1 receives some input values and when the port can access all input data required to run a specific function to completion and when all results can be written to the output port of Task1 a control FSM in the task will activate the function. The results of Task1 are transferred to Task2 that works analogue.

The actors $a \in A$ are self-scheduled, i.e., each time an input $i \in I$ evaluates to fire a function, the associated functions are executed and afterwards all the required data is consumed from input ports and the result data is produced on the output ports. In a hardware-software implementation, a set of actors is bipartitioned, i.e., $A = A^{HW} \cup A^{SW}$ with $A^{HW} \cap A^{SW} = \emptyset$. When performing hardware-software morphing, an actor $a \in A^{HW}$ is moved to A^{SW} at runtime, or vice versa. However, changing the implementation style of an actor also affects the task graph. This is due to the required communication to other actors. If two actors a_1 and a_2 are both implemented in hardware ($a_1, a_2 \in A^{HW}$) or both are implemented in software ($a_1, a_2 \in A^{SW}$) the corresponding communication has to be implemented also in hardware or software, respectively. If

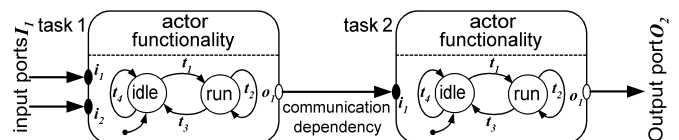


Fig. 1. Example of two communicating actor based tasks.

both actors are implemented in different domains, a hardware-software interface becomes necessary. Nevertheless, changing the implementation style of a task at runtime also affects the implementation of adjacent channels, thus, the hardware-software morphing is a twofold task: 1) morphing actor states and 2) morphing port states.

A. Morphing Actor States

An actor state is represented by the product of the possible states of the firing FSM S_R and the states of all local variables S_F used in the task functionality \mathcal{F} ($S_A = S_R \times S_F$). The ports also contain an internal state, but we will discuss the morphing of port states separately in the next paragraph. We restrict our hardware-software morphing approach only between the port update and function evaluation phase. Thus, only all *static* variables have to be translated during a morph process. In other words, all variables that can be read before they are written (after starting a function), will contain a state that has to be considered when a function is morphed.

When the task is refined to hardware or software by a synthesis or compilation process, we will get according task state sets $M_{HW} : S \rightarrow S^{HW}$ or $M_{SW} : S \rightarrow S^{SW}$. The two functions M_{HW} , M_{SW} denote a mapping of states specified by an abstract specification to the states encoded in the entire hardware or software refinements. Here, we assume that the firing FSM can be implemented in hardware as well as in software by using the same states, thus, $M_{HW} : S_R \times S_F \rightarrow S_R \times S_F^{HW}$ and $M_{SW} : S_R \times S_F \rightarrow S_R \times S_F^{SW}$. In general, S^{HW} or S^{SW} are not equal to S , because some elements $s_i \in S$ may be omitted during optimization. On the other hand, the refinements will increase the state space, thus, $|S^{HW}| > |S|$, and $|S^{SW}| > |S|$. For instance, if we utilize a library function, this may involve some internal states not specified in S . As a consequence, we will have *atomic* sequences in which it is not possible to morph into another implementation style. As an example, let us assume that we have specified the following function:

```

1: boolean example function{A, B, C, D : boolean}
2:   { tmp1 = A AND B;
3:     tmp2 = C AND D;
4:   return{tmp1 OR tmp2}; }

```

The expression could be implemented in software in three single steps (line 2-4), while it is possible to evaluate the complete function in just one look-up table, when it is mapped to an FPGA. In the latter case, the input operands may be connected directly to the table inputs. As a consequence the hardware implementation may hide the states of the signals tmp_1 and tmp_2 , thus, it is not possible to morph inside the example function.

The refined task state sets will normally differ after the high level specification has been synthesized and compiled, thus, $S^{HW} \neq S^{SW}$. As a consequence, it may in general not be possible to morph between hardware and software at any state in S^{HW} or S^{SW} .

We have identified that morphing is not possible in all states of S^{HW} and S^{SW} . In order to force equivalent states in the hardware and software refinements, we propose to define a set of *morph points* $\tau \subseteq S$.

Definition: A *morph point* represents a state $s \in S$ that has equivalent counterparts in the hardware and the software refinements and there exist inverse morph functions that allow to relocate a state $s \in S$ from the refined hardware and software states: $\exists s \in S, M_{HW}^{-1}(M_{HW}(s)) = M_{SW}^{-1}(M_{SW}(s))$.

For every morph point $s_m \in \tau$ we extend the firing

FSM by a corresponding *morph state* $s'_m \in \pi$. A firing state machine FSM that can be morphed between hardware and software is called a *morph FSM* (MFSM). Given an FSM $w = (I, O_{fire}, S_R, \delta, s_0)$ and a set of morph points $s_m \in \tau \subseteq S_R$, the corresponding MFSM $w_m = (I', O'_{fire}, S'_R, \delta', s'_0)$ can be derived as follows: $I' = I \times I_{morph}$; $I_{morph} = \{0, 1\}$ denotes an input signal indicating the intention to morph, $s' \in S' = S \cup \pi$, $O'_{fire} = O_{fire}$, $s'_0 = s_0$, and

$$\delta' = \begin{cases} \delta(s', i) & \text{if } i' = (i, 0) \wedge s' \notin \pi & \text{normal operation} \\ \delta(s', i) & \text{if } i' = (i, 1) \wedge s' \notin \tau & \text{run until next morph point} \\ s'_m & \text{if } i' = (-, 1) \wedge s' \in \tau & \text{start morphing} \\ s_m & s' \in \pi & \text{go back to morph point after morphing} \end{cases}$$

An example of a firing FSM \mathcal{R} with the states $S_R = \{\text{idle}, \text{run}\}$ and one morph point $\tau = \{\text{idle}\}$ is presented in Figure 2a. The corresponding derived MFSM (Figure 2b) will contain the additional morph state $\pi = \{\text{idle}'\}$. When the system decides to morph the task, this is indicated by the added input signal m . And if this signal is active and if the state machine is at the morph point *idle* the morphing takes place in the following morph state *idle'* by translating internal state encodings to the state representation of the entire other domain (as presented in the next section). This includes the state of the control FSM \mathcal{R} as well as the state of the functional part \mathcal{F} of the task. After finishing this process, the state machine continues its operation in the other domain beginning (transparently) from the morph point.

B. Morphing Port States

One main difference between morphing port states and morphing actor states is the rate at which they are triggered. Actor morphing is only necessary when the corresponding task changes its implementation style. In contrast, port morphing is required when the task itself or one of its adjacent tasks is morphed. The communication between different tasks may follow different semantics [8]. However, since tasks can be executed in hardware or software, because the implementation style can change due to hardware-software morphing, the semantic should be adequate for both possible implementations. In this paper we restrict the communication between tasks to i) *FIFO-channels* and ii) *buffers*. A buffer is a consecutive bounded memory block allocated in the shared global memory. A FIFO-channel has also a limited size and in contrast to buffers, a FIFO-channel allows to read out data items called *tokens* exactly in the same order as they were written to the FIFO-channel. The fill level of the FIFO-channel is provided to the port interface of both tasks, the writing task and the reading task of the FIFO-channel. As a consequence, the internal control FSMs (\mathcal{R}) of the two tasks can evaluate the number of tokens stored inside a FIFO-channel. In principle, the FIFO-channel could be located inside an output port, inside an input port, somehow external on the link between these ports, or spread in any combination over these. However, in order to reduce the design complexity, we restrict FIFOs to be located arbitrarily inside input or output ports.

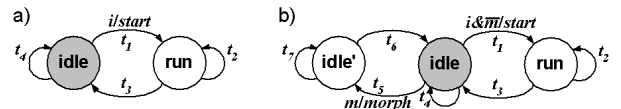


Fig. 2. Example of a task control FSM \mathcal{R} (a) and the derived MFSM (b). A morph process starts from the *morph point* *idle* (gray shaded) upon a morph request ($m = 1$). Then the state is translated into the entire other domain (HW or SW) inside the *morph state* *idle'*.

We demand that both, FIFO-channels and buffers, provide a *non-conflicting* communication between the tasks, in order to guarantee upper bounds for the execution time of the functions $f \in \mathcal{F}$. In the case of FIFOs, this implies that it is connected exactly to one input port $i_j \in I_p$ and exact one corresponding output port $o_i \in O_p$. In the case of a communication via buffers, we demand that input buffers are read only. Therefore, multiple tasks may read the same buffer without interfering each other with respect to the content, but not with respect to the time required to access data from the buffer. In the case of output buffers, we permit only one task to access the buffer (read/write) exclusively during an execution of a function $f \in \mathcal{F}$. Different buffer ports represent a pointer to always a different memory block. In the case that multiple ports of a hardware module communicate through buffers, these ports will be combined to one *master port* connected to the system bus. The access to this centralized memory port is controlled using a round robin arbitration scheme inside the task port.

Each individual port $i_j \in I_p$ and $o_i \in O_p$ is aware to the characteristic (hardware or software) of the entire adjacent task. When the characteristic of two tasks differs, it may be required to translate data transferred over a specific port between the hardware and software domain (as presented in the next section). While the hardware-software interface between different tasks is trivial when the communication utilizes buffers, we have to take care, in the case of a communication through FIFO-channels. Whenever a task has the capability of hardware-software morphing, we demand that all FIFO interface ports of the adjacent tasks provide an interface to a hardware task as well as to a software task. If these adjacent tasks are also morphable, the two interface refinements must be provided in both possible domains (hardware or software). When a morphable task utilizes a FIFO-channel communication, we demand that there is a FIFO memory located at the input port, as well as at the output ports of the morphable task and its adjacent neighbors. The FIFO memories must be large enough, such that an internal function can run to completion regardless to the state of a FIFO located outside the task that is hosting this particular function. When a FIFO port is required to be morphed, this is performed in an idle phase of the interface (e.g., after a function and the corresponding communication has terminated). In the following, the FIFO interface will be blocked, and all tokens will be copied between the two domains.

III. STATE TRANSLATION

Our hardware-software morphing approach provides to change the task execution between hardware and software without a loss of internal states. Consequently, we firstly have to read a consistent state from one domain. Next, we will translate states between the two domains (hardware to software or vice versa). Finally we have to initialize the respective other domain with this transformed state.

In the following, we will look closer on how states are represented in these two domains and more important how states can be translated between these two domains. Each implementation utilizes a set of data types to specify states, variables or some constants. However, there exist corresponding data types in the hardware and the software domain and some of them can be directly exchanged. For instance, integer types exist in hardware as well as in software. While in the software case integer types are coded in fixed sizes (e.g., as a byte or as a 32bit signed/unsigned), the hardware counterparts are more flexible as they utilize *bitvectors* optimized individually for specific number ranges. We map

bitvectors to the next largest native integer type offered in the software domain. For instance, let us assume the bit value representations of the two integer values $(I_m^{SW}, \dots, I_0^{SW})$ and $(I_n^{HW}, \dots, I_0^{HW})$, then they are converted as follows:

$$\text{conv}_{SW}^{Int} = \begin{cases} I_i^{SW} = I_i^{HW} & \forall i \leq n-1 \\ I_i^{SW} = I_n^{HW} & \forall i \geq n \end{cases}$$

$$\text{conv}_{HW}^{Int} = \begin{cases} I_i^{HW} = I_i^{SW} & \forall i \leq m-1 \\ I_i^{HW} = I_m^{SW} & \forall i \geq m \end{cases}$$

Some bits in a bit vector may not influence a state representation in the case they never change at runtime. It is further possible that two or more bits will contain always the same logic value. Such constant values or redundancy will be typically omitted by some optimizations during the hardware synthesis process and they will be handled separately according to the report files generated by the synthesis tools. If the hardware contains a single bit value it will be converted to an unsigned in the software counterpart.

Beside integer types, it is possible to declare user defined enumerated types that are in particular used to specify states in state machines. While in the software world enumerates will be mapped per default to consecutive integer values this may be inapplicable in hardware. One well known format for state encoding in hardware is the *one-hot encoding*. In this format, an enumerated type with N elements will be mapped to an N -bit wide bitvector and each element is encoded by an individual 1-out-of- N code. With this encoding, it is possible to determine a specific state in the final hardware by simply evaluating the signal of a single wire.

We can summarize that enumerated types are typically not mapped in such a way that they can be directly exchanged between the hardware and software domain. A translation of states may be performed using the following techniques: 1) force the encoding of enumerates in both domains to pre-defined values (e.g. one hot), 2) force the hardware domain to use the same state encoding as determined during software compilation, 3) force the software domain to use the same state encoding as determined during hardware synthesis, 4) generate a translate function that performs a mapping between states in the hardware or software domain (e.g., by a look-up table). Note that in VHDL the only specified synthesizable attribute is an attribute that forces a user defined encoding of enumerated values. As the state encoding has massive impact to the hardware module with respect to the performance (achievable clock frequency) and cost (logic resources) we favor the last two techniques.

With the capability of translating integer types and enumerated types we are able for applying morphing to most tasks suitable to be implemented in hardware as well as in software. There exist further datatypes like floating-point numbers, but these types are unlikely to be implemented in hardware on FPGAs.

A. Further Discussion on State Translation

Hardware and software tasks may further differ massively in the way how *hierarchies* are represented and how the hierarchy level is controlled or encoded. In the software world, it is common to use a stack to build hierarchies and functions can be reentered, thus, allowing recursion. Inside a certain hierarchy the program counter and the CPU registers mainly represent the present state of the control flow. In contrast to this, a task implemented in hardware may typically express hierarchies by communicating state machines. Furthermore, hierarchies found in the hardware domain can lead to multiple instances of the same functionality that may run in parallel and

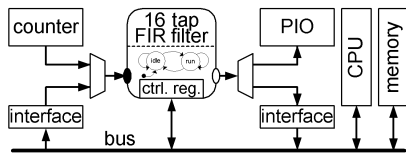


Fig. 3. Test system allowing hardware-software morphing of a task with all possible input and output interface permutations (HW and SW).

recursion is not appropriate to be implemented in hardware. Therefore, we restrict our model to be flat in the sense that the control FSM \mathcal{R} can start functions, but a function will run atomically to completion, thus, we do not allow morphing inside the execution phase of a function, and a function has to terminate in an *idle* state. Therefore, functions may contain hierarchies, but this will be transparent to the morphing capability and we do not require to interpret stacks or other constructs that represent hierarchies.

The state translation can either be done in the hardware or in the software domain of the system, depending on allowed latencies and on the rate how often a state translation process is required.

In order to access the state of a software module, we demand that all state variables are globally accessible from inside the software task. We further demand that after compilation and synthesis the respectively mapped states can be identified to allow a translation as described above in this section. Note that there may exist temporary states in each domain not required to be concerned for morphing (see also Section 2). It may be required to guide the hardware synthesis steps, in order to allow an identification of the states. In the software domain, this is not required because we access states at the source code level and variables are globally accessible within a task. For the hardware design flow, we are restricted to the capabilities offered by the synthesis tools, in order to get efficient morphable hardware tasks. However, we propose to extend a hardware module after the synthesis process on the netlist level, in order to access the state of only the memory elements that are involved in the morphing process. This allows us to take most benefit from the optimizations offered by the synthesis tools. An implementation of this procedure is presented in the case study section.

IV. CASE-STUDY

In this section, we present our experimental results obtained by a 16-tap FIR filter case study (Figure 3), in order to demonstrate our hardware-software morphing methodology. Based on a C++ software specification of the software filter task, we synthesized an RTL model in VHDL by an behavioral compiler. In this generated code, we only set to all signals accessed during hardware-software morphing an attribute preventing register retiming and logic fusion. In the following, we synthesized the modified VHDL specification and the resulting netlist was automatically extended by our tool STATEACCESS [6] with some logic to access the state of the tap registers and the control FSM. In addition, we put interface templates around the generated netlist containing also the communication ports. If a task is implemented as a hardware module, FIFO-channels are mapped to on-chip memory blocks. In the special case, where a FIFO-channel has a depth of 1 (as in this case study) or even 0, the channel will be replaced by a register or simply bypassed.

The unconstrained RTL model utilizes 563 flip-flops and 1223 look-up tables (LUTs) to implement the filter. After applying the mentioned constraints and after modifying the

netlist performed by STATEACCESS, the hardware task requires 571 flip-flops (+4%) and 1890 LUTs (+54%). The achieved clock frequency is in both cases about the same (20 MHz). The pure CPU requires 1559 flip-flops (plus additional memory for the register file) and 2718 LUTs. The execution time of the filter function as a hardware task is $t_{HW} = 2,7\mu s$. When the task is morphed to software the execution time increases to $t_{SW} = 60\mu s$ (at a system clock of 50 MHz). The interface latency is about the same for all interface permutations, thus, we can save about $60\mu s$ CPU-time per task activation when the filter is morphed to hardware. Note that we utilized two clocks in order to permit maximum speed for both, the hardware task and the software task.

Furthermore, we measured the time required to copy the state between the tap registers of the hardware task and the software task: $t_{morph} = 104\mu s$. In this example, we will have a benefit with respect to the CPU time after the second task activation. Note that we specified all variables as 16-bit integer values, thus, no translation of data items between the hardware and software domain is required. If we omit additional latencies caused by the operating system, the morphing is performed within $L = t_{morph} + \max\{t_{HW}, t_{SW}\} = 164\mu s$.

The speedup value $\lambda = \frac{t_{SW}}{t_{HW}}$ is 22 in this case-study. Such high speedup values fit ideal to systems with a high load variation. It must be mentioned that the morphing methodology may further be applied for morphing between different hardware task refinements in order to meet different requirements (speed, power, area) even more flexible at runtime.

V. CONCLUSION

In this paper, we demonstrated, based on a formal model, how hardware-software morphing could be implemented in order to decide the hardware-software partitioning problem online. Morphing permits to utilize the high flexibility offered by partial reconfigurable FPGAs, in order to adapt the system to changing load and demand scenarios at runtime. We discussed the challenges of this new methodology and presented solutions applicable to real world examples. Future work will focus on advances of automation in the design flow.

ACKNOWLEDGMENT

This work was partly supported by DFG (Deutsche Forschungsgemeinschaft) under grant Te163/10-2.

REFERENCES

- [1] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens, "Hardware/Software Partitioning of embedded system in OCAPI-x1," in *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*. New York, NY, USA: ACM Press, 2001.
- [2] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 250–255.
- [3] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, no. 2, pp. 63–69, 2005.
- [4] T. Streichert, C. Strengert, C. Haubelt, and J. Teich, "Dynamic Task Binding for Hardware/Software Reconfigurable Networks," in *Proceedings of SBCCI 2006*, Ouro Preto, Brasil, Aug. 2006.
- [5] T. Streichert, D. Koch, C. Haubelt, and J. Teich, "Modeling and Design of Fault-Tolerant and Self-Adaptive Reconfigurable Networked Embedded Systems," *EURASIP Journal on Embedded Systems*, 2006.
- [6] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing – Concepts, Overhead Analysis, and Implementation," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2007)*. Monterey, California, USA: ACM, Feb. 2007.
- [7] J. Falk, C. Haubelt, and J. Teich, "Efficient Representation and Simulation of Model-Based Designs in SystemC," ser. Proc. FDL '06, Forum on Design Languages 2006, Darmstadt, Germany, Sept. 2006, pp. 129–134.
- [8] E. A. Lee, "Embedded software," in *Advances in Computers*, M. Zelkowitz, Ed. London: Academic Press London, Sept. 2002, vol. 56.