# Scheduling and Communication-Aware Mapping of HW/SW Modules for Dynamic Partial Reconfigurable SoC Architectures

Sándor P. Fekete, Department of Mathematical Optimization, Braunschweig University of Technology, 38106 Braunschweig, Germany
Jan C. van der Veen, Department of Mathematical Optimization, Braunschweig University of Technology, 38106 Braunschweig, Germany
Josef Angermeier, Department of Computer Science 12, University of Erlangen-Nuremberg, Erlangen, Germany
Diana Göhringer, Department of Computer Science 12, University of Erlangen-Nuremberg, Erlangen, Germany
Mateusz Majer, Department of Computer Science 12, University of Erlangen-Nuremberg, Erlangen, Germany
Jürgen Teich, Department of Computer Science 12, University of Erlangen-Nuremberg, Erlangen, Germany

## Abstract

In this paper, we present an approach for simultaneous scheduling and placement of communicating modules for SoC architectures including devices with partial reconfiguration support and at least one CPU. This approach includes **(a)** a detailed modeling of the communication of modules and an optimization model for finding the best temporal and spatial placement of modules on either CPU or on the reconfigurable device including communication and reconfiguration time overheads, **(b)** a real SoC platform for slot-based module relocation and on-chip inter-module communication called ESM, and **(c)** real experimental data based on experiments on this machine. Existing approaches either neglect inter-module communication, are not able to solve the related problem, or do not provide real applications implemented on real platforms.

## 1 Introduction

### 1.1 Classification of partitioning approaches

Hardware-Software Partitioning is known as one of the major problems and challenges of co-design throughout the last decade. It has lead to numerous approaches starting with simple bi-partitioning algorithms, more complex partitioning methods [20, 7], as well as exact solutions based on ILP formulations, e.g., [13, 18]. Current approaches may be categorized and compared only loosely with respect to the criteria *application modeling*, where applications are described in languages such as C, C++, SystemC [16], or modeled directly by one or more abstract communicating tasks (or task graphs); *architecture modeling*, where many approaches are able to model even multiple and heterogenous processor and hardware modules communicating via either buses or dedicated links and memories (MPSoC); *partitioning algorithm* as described above; and finally, *objectives*. Whereas the first approaches in literature were simple single-objective approaches optimizing for either execution time or cost, some of the newest approaches are able to approximate Pareto-optimal solutions (e.g., [4, 9]), thereby considering many objectives simultaneously such as speed, power, cost, reliability, and many more, and thus allowing a designer to shift decision-making after partitioning.

### 1.2 Motivation

In platform-based design, many systems incorporate hardware components such as FPGAs [11, 10], which are at least partially reconfigurable, so the system architecture is not fixed but can vary at run-time. Except of a few results on how to model reconfigurable architectures by hierarchical architecture graphs [19], only very few approaches have extended the HW/SW partitioning problem to such architectures [11, 10]. In this paper, we provide one of the first approaches for hardware-software partitioning that is able to a) model accurately reconfigurable hardware, i.e., FPGAs included as part of a MPSoC, including b) reconfiguration times, and space management of free and occupied resources. Our subsequently presented optimization model is used in an exact approach employing an ILP solver to statically compute optimal hardware-software partitions with respect to execution time. c) Our approach also allows blocks mapped to hardware to communicate with each other using a concept of a multiple reconfigurable bus (RMB) [5]. Thus, we provide an accurate incorporation of inter-module communication between hardware and hardware as well as between hardware and software modules.

### 1.3 Related Work

In [15] it is approved that our choice to target dynamically reconfigurable architectures is well-founded, because our desired applications mainly depend on performance.
HW/SW scheduling algorithms for System-on-Chip platforms with dynamically reconfigurable logic architectures are exhaustivly studied in [14]. System performance for tasks with data-dependent execution times is improved by using dynamic schedulers instead of a static, compile time scheduling techniques. Hereby no worst-case execution

times have to be assumed, which may heavily deviate from the average case. This may prevent a highly under-utilized hardware/software system at run-time. Our static scheduling approach instead, prevents ready tasks ahead of time and under-utilization by a fine-tailored resource and cost model (e.g. placement dependent task communication costs, single reconfiguration interface, FPGA heterogenities) and detailed cost values measured on the corresponding, existing dynamically partial reconfigurable platform in dependency of the data input.

The reconfiguration latency of dynamically reconfigurable devices represents a major drawback, that must not be neglected. New techniques to alleviate this problem are developed in [17] and integrated into an existing scheduling environment. A run-time task scheduler decides, which task implementation to execute by using a pareto curve reflecting the energy/performance trade-off. Hereby a initial schedule, which neglects the reconfiguration overhead is generated. Based on this a reuse, a prefetch and a replacement unit modify the schedule and considerably reduce the latency even for highly dynamic tasks. Our approach differs therein, that those reconfiguration aspects are already taken into account for the decision, which task implementation to choose. Thus, a possibly more suitable implementation is selected.

Very closely related to our approach is the work presented in [2][3]. That approach, however, differs in two major aspects: a) We are able to provide exact solutions to the hardware-software partitioning problem including reconfigurable devices, and b) contrary to [2], we are able to model and realize also on-chip inter-module communication based on a real architecture called ESM, see also Fig. 1. This architecture, first presented in [6], is able to relocate HW modules freely in areas called *slots* on a main FPGA. Also, the architecture allows for online module communication, which is established automatically at run-time by issuing requests between source and target HW module.

The reconfigurable computing HW/SW multitasking platform, presented in [12], shows some similarities to our platform. Tasks can be executed on the reconfigurable logic or run in software and can communicate with each other. The infrastucture consists of a uniformed HW/SW communication scheme and a common HW/SW behavior. The uniform communication is realized by a message passing system in the operating system and a packet switched interconnection network on the reconfigurable hardware. On the ESM a circuit switched interconnection network can be used by multiple HW modules independent of their positions. In contrast, the ESM platform allows a more flexible HW task placement. Partial configuration bitstreams for a task can be freely relocated, on the fly at runtime and placed to each position. The needed I/O pins to the periphery, to access i.e. video input or output devices, get connected to the actual position of the HW modul. Furthermore, the ESM offers different communication methods with varying transmission rates and bandswidths. Depending on the placement and requirements of the communicating modules one choice might me more favorable than another.

### 1.4 Organization of the Paper

In Section 2, we present the target architecture called ESM. The optimization model for hardware-software partitioning is presented in Section 3. Detailed experimental results are reported in Section 4. We conclude with ideas on future work.

## 2 The Erlangen Slot Machine

The main idea of the Erlangen Slot Machine (ESM) [5, 6] architecture is to accelerate application development as well as for research in the area of partially reconfigurable hardware. The ESM platform is centered around an FPGA that serves the main reconfigurable engine, and an FPGA realizing a crossbar switch, see Figure 1. The advantage of the ESM platform is its unique slot-based architecture that allows 1D slot modules to be reconfigured independently by delivering peripheral data through the separate crossbar switch. The ESM architecture is based on the flexible decoupling of I/O-pins from a direct connection to an interface chip. This flexibility allows the independent placement of application modules at run-time in any available slot. Thereby, run-time placement is not constraint by physical I/O-pin locations, as the I/O-pin routing is performed automatically in the crossbar.

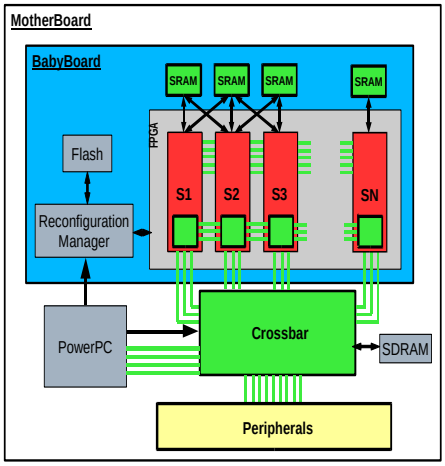


**Figure 1** ESM architecture.

The two FPGAs are placed onto two physical boards called BabyBoard and MotherBoard, using a Xilinx Virtex-II 6000 and a Xilinx Spartan-II 600 FPGA. Figure 1 shows the slot-based architecture of the ESM, consisting of the Virtex-II FPGA, local SRAM memories, configuration memory, and

a reconfiguration manager. The top pins in the north of the FPGA connect to local SRAM banks. These SRAM banks solve the problem of restricted intra-module memory, e.g., for video applications. The bottom pins in the south connect to the crossbar switch.

## 2.1 Inter-module Communication

One of the central limiting factors for the wide use of partial dynamic reconfiguration is the problem of inter-module communication. Each module that is placed in one or more slots on the device must be able to communicate with other modules. For the ESM, we provide four main paradigms for communication among different modules: a) direct communication using bus-macros between adjacently placed modules; b) secondly, shared memory communication using SRAMs or BlockRAMs is possible. However, only adjacent modules can use these two communication modes. For modules placed in non-adjacent slots, we provide c) a dynamic signal switching communication architecture called Reconfigurable Multiple Bus (RMB) [8, 21, 1]. Finally, the communication between two different modules can also be realized through d) the external crossbar.

### 2.1.1 Communication between adjacent modules

On the ESM, bus-macros are used to realize direct communication between adjacently placed modules, providing fixed communication channels that help to keep the signal integrity upon reconfiguration.

### 2.1.2 Communication via SRAM

Communication between two neighboring modules can be done by using external SRAM. This is particularly useful in applications in which each module must process a large amount of data, which is then sent to the next module, as in the case of video streaming applications. On the ESM, each SRAM bank can be accessed by the module placed below as well as the direct neighbors placed right and left to it. A controller is used to manage the SRAM access. Depending on the application, the user may set the priority of accessing the SRAM for these three modules.

### 2.1.3 Communication via RMB

In its basic definition, the Reconfigurable Multiple Bus architecture consists of a set of processing elements or modules, each having access to a set of switched bus connections to other processing elements. The switches are controlled by connection requests between individual modules. The RMB is a one-dimensional arrangement of switches between N slots. In our FPGA implementation, the horizontal arrangement of parallel switched bus line segments allows for the communication among modules placed in the individual slots. The request for a new connection is done in a wormhole fashion, where the sender (a module in slot $S_k$) sends a request for communication to its neighbor (slot $S_{k+1}$) in the direction of the receiver. Slot $S_{k+1}$ sends the request to slot $S_{k+2}$, etc., until the receiver receives and acknowledges the request. The acknowledgment is then sent back on the same way to the sender. Each module that receives an acknowledgment sets its switch to connect two line segments. Upon receiving the acknowledgment, the sender can start the communication (circuit routing). The wired and latency-free connection is then active until an explicit release signal is issued by the sender module.

## 2.2 Hardware-Software Communication

Hardware tasks on the FPGA can communicate with Software tasks on the PowerPC and vice versa via a single-ported SDRAM Memory, which is connected to the crossbar. The access to this shared memory is supervised by a dedicated Hardware-Controller on the crossbar. The communication bandwith between a Hardware and a Software task is therefore limited to 84,37 MBit/s. The setup time for a transfer is 20 clock cycles.

## 2.3 Physical Properties

In order to model the physical platform, all physical properties and constraints of the platform itself as well as of the provided infrastructure must be taken into account. Ideally the FPGA device is a homogeneous two-dimensional CLB array on which rectangular application modules are placed. However, communication requirements and physical FPGA constraints do not provide this kind of environment.

### 2.3.1 Slot Arrangement

The main FPGA of the ESM is divided into 22 micro slots with 12 I/O-pins each. Because the left and right slots of the FPGA are connected to dedicated I/Os, one micro slot on both the left and the right side of the FPGA is excluded. As the middle CLB columns are connected to external clock lines, two micro slots in the middle of the device are also excluded. Three micro slots can be grouped logically into one so-called *macro slot* in order to allow access to the RMB and to the SRAM banks. The resulting slot refined architecture is shown in Fig. 2.
Due to the incorporation of BlockRAM and multipliers, the Virtex-II FPGA architecture from Xilinx is divided into columns. Each BlockRAM block occupies a whole column in the device; In the following we assume our XC2V6000 is divided into six macro slots that are spread over the device. Thereby, only macro slots 2 and 5 contain one BlockRAM column.

### 2.3.2 Communication Costs

The ESM platform supports four different communication schemes. Each approach has its own properties, such as maximum bandwidth, signal delay and setup latency. The RMB is the only scheme that has a varying setup latency that is the product of the number of RMB elements to destination and the setup time of four clock cycles. Using bus macros for communication is the preferred choice, but it

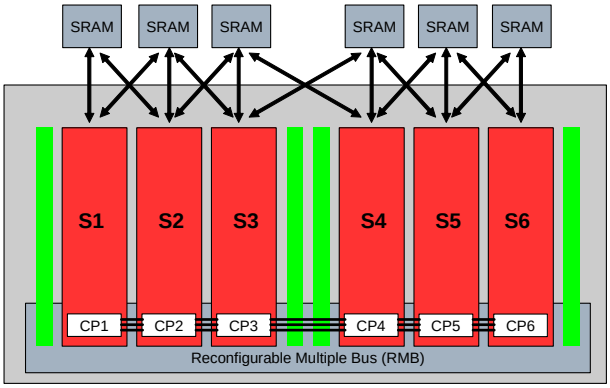


**Figure 2** ESM slot architecture with six macro slots. In order to allow access to the RMB crosspoints (CP) and SRAM banks, one macro slot consists of three micro slots.

| Scheme | Data Bandwidth | Delay | Setup |
|---|---|---|---|
| Bus Macro | 19.2 Gbits/s | 2 ns | none |
| RMB | 6.4 Gbits/s | 3 ns * CP | 4 cycles * CP |
| Crossbar | 1.8 Gbits/s | 15 ns | 18 cycles |
| SRAM | 0.4 Gbits/s | 20 ns | 2 cycles |

**Table 1** Communication bandwiths and signal delays.

only works for adjacent modules. The maximum bandwidth in all communication schemes is a factor of clock speed and data bandwidth. In our experiments we assume for the ESM a global clock speed of 50 MHz. All properties are listed in Table 1.

#### 2.3.3 Reconfiguration Times

A Virtex-II device is basically configured in entities called frames, which cover the whole height of the FPGA and are one bit wide [22]. Depending on the resources the number of frames are different. The resources used in the different macro slots are: CLBs, BRAMs and BRAM-Interconnect. The BRAM and the BRAM-Interconnect columns are always next to each other. The BRAM-Interconnect columns are used for wiring purposes and can therefore also be used by a module, that does not require the actual BRAM column. To configure one CLB column or a BRAM-Interconnect column, 22 frames must be written. For a BRAM column this are 64 frames. The frame length in terms of 32-bit words depends on the height of the FPGA. For the Virtex-II 6000 FPGA, which consists of 96 rows by 88 columns of CLBs, the frame length consists of 246 32-bit words. As the reconfiguration manager of the ESM uses the SelectMAP interface for programming the main FPGA, a bandwidth of eight signals and a maximum frequency of 50 MHz have to be taken into account. The reconfiguration time for one CLB column as well as for one BRAM-Interconnect column therefore requires 246 words $*$ 22 frames $*$ 4 clock cycles $*$ 20 ns = 433 $\mu$s. The reconfiguration time for a BRAM column is 246 words $*$ 64 frames $*$ 4 clock cycles $*$ 20 ns = 1259,52 $\mu$s. Out of this we get the following reconfiguration times for the different types of macro slots:

- one macro slot with 12 CLB-Columns: 12 $*$ 433 $\mu$s = 5196 $\mu$s.
- one macro slot with 12 CLB-Columns + 1 BRAM-Column + 1 BRAM-Interconnect Column: 13 $*$ 433 $\mu$s + 1259,52 $\mu$s = 6888,52 $\mu$s. If a module, that does not use the BRAM, is reconfigured into a macro slot, which has a BRAM-Column, the above described reconfiguration time is reduced to: 13 $*$ 433 $\mu$s = 5629 $\mu$s.

# 3 Model for HW/SW Partitioning

In the following, we provide a detailed model of communication, placement and scheduling on reconfigurable SoC architectures based on the ESM as a representative example.

## 3.1 Mathematical Model

### 3.1.1 ESM Geometry and Task Graphs

The FPGA is split into 6 slots; Slots 2 and 5 provide access to BlockRAM and multipliers. It takes $c_r$ time units to reconfigure one slot. We assume that loading a module on the processor takes $c_l$ time units ($c_l \ll c_r$). Each slot can access the SRAM blocks directly to the left, on top and directly to the right of the slot. The RMB has $h_{\text{RMB}}$ parallel bus segments. The maximal length of any segment can be restricted by setting $w_{\text{RMB}}$ to a value smaller than six. Each slot has access to 36 pins located on the bottom of the FPGA.

Given a task graph $G = (T, A)$ with $T = \{t_1, t_2, \ldots, t_n\}$, the set of tasks to be executed on the platform and $A = \{(t_i, t_j) \in T \times T\}$ the set of predecessor successor relations. If an edge has a non-zero weight, this weight specifies the amount of data $a_{(t_i,t_j)}$ that is to be sent from a task to its successor. For each task there can be up to three different implementations: *Implementations using special resources* like BlockRAM or multipliers (SP). These implementations have to be placed in slots 2 or 5. *Hardware implementations* (HW) can be placed in any of the slots 1, 2, ..., 6. *Software implementations* running on the processor (SW); for ease of notation we identify the processor with slot 0, even if on the ESM the processor(PowerPC) is located externally on the Motherboard.

### 3.1.2 Communication

To be able to correctly model inter-module communication each task $t_i \in T$ is split up into subtasks:

- **Load ($L_{t_i}$)**: the subtask where the task is loaded.
- **Acquire Data from Predecessors ($A_{t_i}$)**: a dummy task for modelling the earliest time when the task can start to acquire data from its predecessors.
- **Acquire Data from Predecessor ($A_{(t_j,t_i)}$)**: There are as many subtasks $A_{(t_j,t_i)}$ as there are predeces-

sors to $t_i$. In this task data is acquired from all predecessors $t_h$ of $t_i$.

- **Run ($R_{t_i}$)**: the task $t_i$ runs for its implementation-dependent fixed execution time $r_{t_i}^{\text{SP}}$, $r_{t_i}^{\text{HW}}$, $r_{t_i}^{\text{SW}}$.
- **Forward Data to Successors ($F_{t_i}$)**: the dummy task for modelling the earliest time when the task can start to forward data to its successors.
- **Forward Data to Successors ($F_{(t_i,t_j)}$)**: There are as many subtasks $F_{(t_i,t_j)}$ as there are successors to $t_i$. This task forwards data to each successor $t_j$ of $t_i$. Unless this is done asynchronously $F_{(t_i,t_j)}$ equals $A_{(t_i,t_j)}$.
- **End ($E_{t_i}$)**: a dummy subtask just for scheduling purposes that will be explained later on.

We obtain a more detailed task graph $G_D = \{T_D, A_D\}$. For every $t_i \in T$ the detailed set of tasks $T_D$ contains all of the above mentioned tasks $L_{t_i}$, $A_{(t_h,t_i)}$, $R_{t_i}$, $F_{t_i}$, $F_{(t_i,t_j)}$ and $U_{t_i}$. $A_D$ contains an edge for all subtask dependencies ($\{(L_{t_i}, A_{(t_h,t_i)})$ for all $(t_h,t_i) \in A$, $(A_{(t_h,t_i)}, R_{t_i})$ for all $(t_h,t_i) \in A$, $(R_{t_i}, F_{t_i})$, $(F_{t_i}, F_{(t_i,t_j)})$ for all $(t_i,t_j) \in A$, $(F_{(t_i,t_j)}, E_{t_i})$ for all $(t_i,t_j) \in A$).

Hardware modules can acquire and communicate data in five different ways. In the following we consider two tasks $t_i$ and $t_j$. Without loss of generality we assume that $t_i$ precedes $t_j$. Then we get the following modes of communication.

**SRAM.** Tasks $t_i$ and $t_j$ can transfer data via the SRAM interface if they both support the SRAM interface and are placed in two consecutive slots. Without loss of generality assume that $t_i$ is located directly to the left of $t_j$. Data to be transfered has to be stored in the SRAM blocks directly above $t_i$ or $t_j$. To be able to access the data $t_j$ has to ask $t_i$ where to find it. After $t_j$ has acquired all data $t_i$'s slot can be reconfigured with another module. Communicating via the SRAM interface has constant setup and deletion cost $s^{\text{SR}}$ and $d^{\text{SR}}$. The data bandwidth is denoted by $b^{\text{SR}}$.

**Bus macro.** Two tasks can communicate by utilizing bus macros if they both support bus macros and are located in consecutive slots. Upon request $t_i$ forwards all data to $t_j$. After doing so its slot can be reconfigured again. Communicating via the bus macro interface has neither setup nor deletion cost ($s^{\text{BM}} = d^{\text{BM}} = 0$). The data bandwidth is given by $b^{\text{BM}}$.

**RMB.** To communicate using the RMB both tasks have to support it. The distance of $t_i$ and $t_j$ may not exceed $w^{\text{RB}}$. Not more than $h^{\text{RB}}$ pairs of modules may simultaneously use the RMB. To acquire data $t_j$ has to request a connection to $t_i$. After the connection has been established $t_i$ forwards the data to $t_j$. Upon termination $t_i$ frees the RMB segments used by that connection. Both setting up and deleting a RMB connection causes distance dependent costs $s^{\text{RB}}(m_i, m_j)$ and $d^{\text{RB}}(m_i, m_j)$. The data bandwidth is given by $b^{\text{RB}}$.

**Crossbar.** The crossbar interface is the most versatile as it imposes no placement restriction at all. It is the only way to directly forward data to or directly acquire data from a software module is via the crossbar interface. Setup and deletion of crossbar connections take constant time $s^{\text{CB}}$ and $d^{\text{CB}}$. The data bandwidth is given by $b^{\text{CB}}$.

**Common memory.** Additionally there is a block of SDRAM accessible by both the processor and the FPGA through the crossbar interface. This way two hard- or software modules can communicate asynchronously. Setup and deletion of connections takes constant time dependent on the setup and deletion time of the crossbar connection $s^{\text{CM}}$ and $d^{\text{CM}}$. The data bandwidth is given by $b^{\text{CM}}$.

## 3.2 ILP Formulation

The formulation resembles a resource-constrained open shop scheduling with single server, setup cost, transportation setup cost, and transportation cost.

The objective is $\min C_{\max}$, i.e., minimize the makespan.

### 3.2.1 Variables

We use the following variables:

- $\chi_t^L \geq 0$: start of reconfiguration for task $t$.
- $\chi_{(t_h,t_i)}^A \geq 0$: start of data acquisition from task $t_h$ by $t_i$.
- $\chi_t^R \geq 0$: start of task $t$'s execution.
- $\chi_t^F \geq 0$: start of data forwarding of task $t$.
- $\chi_{(t_i,t_j)}^F \geq 0$: start of data forwarding from $t_i$ to $t_j$.
- $\chi_t^E \geq 0$: end of task $t$.
- $\iota_t^{\text{SP}}$, $\iota_t^{\text{HW}}$, $\iota_t^{\text{SW}} \in \{0,1\}$: indicator whether task $t$ is implemented using special resources (SP), hardware (HW) or software (SW).
- $\pi_{t_it_j}^L \in \{0,1\}$: indicates if task $t_i$ is reconfigured before task $t_j$.
- $\pi_{t_it_jt_kt_l}^{\text{CM}} \in \{0,1\}$: indicate the order in which two pairs $(t_i,t_j)$, $(t_k,t_l) \in A$ of tasks may access the common memory. As the forwarding operation of $t_i$ (respecively $t_k$) have to precede the the data acquisition operation of $t_j$ (resp. $t_l$) there are six orders $(t_i,t_j,t_k,t_l)$, $(t_i,t_k,t_j,t_l)$, $(t_i,t_k,t_l,t_j)$, $(t_k,t_i,t_j,t_l)$, $(t_k,t_i,t_l,t_j)$, and $(t_k,t_l,t_i,t_j)$. We will denote these six orders by $\mathcal{O}$.
- $\delta_{t_it_j}$: distance of $t_i$ and $t_j$ for $(t_i,t_j) \in A$.
- $\lambda_{ts} \in \{0,1\}$: indicates if task $t$ is placed in slot $s$. For ease of notation, $s = 0$ is the processor.
- $\lambda_{t_it_js} \in \{0,1\}$: auxiliary variable indicating that $t_i$ and $t_j$ are placed in slot $s$.
- $\gamma_{(t_i,t_j)}^{\text{SR}}$, $\gamma_{(t_i,t_j)}^{\text{BM}}$, $\gamma_{(t_i,t_j)}^{\text{RB}}$, $\gamma_{(t_i,t_j)}^{\text{CB}}$, $\gamma_{(t_i,t_j)}^{\text{CM}} \in \{0,1\}$: indicator what communication paradigm is used to transfer data from task $t_i$ to task $t_j$.

### 3.2.2 Constraints

Next we list and explain the set of constraints. We denote by $\mathcal{P} = \{\text{SP}, \text{HW}, \text{SW}\}$ the possible implementations of a task and by $\mathcal{C} = \{\text{SR}, \text{BM}, \text{RB}, \text{CB}, \text{CM}\}$ all availabe means of communication.

For all $t_i, t_j \in T$, either $t_i$ is reconfigured before $t_j$ or vice versa:

$$\pi_{t_it_j}^L + \pi_{t_jt_i}^L = 1 \tag{1}$$

If $t_i$ is to be reconfigured before $t_j$, this should be well reflected in the starting times for reconfiguration:

$$\chi^L_{t_i} + \sum_{p\in\mathcal{P}} c_p \iota^p_{t_i} \leq \chi^L_{t_j} + (1 - \pi^L_{t_i t_j})M \quad (2)$$

A task $t_i$ can start to aqcuire data from its predecessors $t_h$ only after it has been loaded on either the FPGA or the processor:

$$\chi^L_{t_i} + \sum_{p\in\mathcal{P}} c_p \iota^p_{t_i} \leq \chi^A_{t_i} \quad (3)$$

$$\chi^A_{t_i} \leq \chi^A_{(t_h,t_i)} \quad (4)$$

Except for CM all means of inter-module communication are synchronous or at least require the transmission of some additional address information (SR). So in general the starting times for acquisition and forwarding of data between two tasks $t_i$ and $t_j$ are equal:

$$\chi^F_{(t_i,t_j)} \leq \chi^A_{(t_i,t_j)} \quad (5)$$

$$\chi^A_{(t_i,t_j)} \leq \chi^F_{(t_i,t_j)} + M\gamma^{\text{CM}}_{(t_i,t_j)} \quad (6)$$

For ease of notation let $f^c_{(t_i,t_j)}$ denote the time needed to transfer data from task $t_i$ to to $t_j$ using communication medium $c \in \mathcal{C}$:

$$f^c_{(t_i,t_j)} = \begin{cases} s^c + \frac{a_{(t_i,t_j)}}{b^c} + d^c & c \neq RB \\ (s^c + d^c)\delta_{t_i t_j} + \frac{a_{(t_i,t_j)}}{b^c} & \text{else} \end{cases} \quad (7)$$

A task $t_i$ with no predecessors can start to run directly after it is ready to acquire data (8). All other tasks have to wait until all data is transmitted. For $c \in \mathcal{C}$ this is taken care of in (9). Of course (9) needs linearization for the case $c = \text{CB}$:

$$\chi^A_{t_i} \leq \chi^R_{t_i} \quad (8)$$

$$\chi^A_{(t_h,t_i)} + f^c_{(t_h,t_i)}\gamma^c_{(t_h,t_i)} \leq \chi^R_{t_i} \quad (9)$$

After task $t_i$ ran for its implementation dependent time data can be forwarded to $t_i$'s successor:

$$\chi^R_{t_i} + \sum_{p\in\mathcal{P}} \iota^p_{t_i} r^p_{t_i} \leq \chi^F_{t_i} \quad (10)$$

Task $t_i$ can start to forward data only after it is ready (11) and must not end before all its data has been forwarded to all successors:

$$\chi^F_{t_i} \leq \chi^F_{(t_i,t_j)} \quad (11)$$

$$\chi^F_{(t_i,t_j)} + f^c_{(t_i,t_j)}\gamma^c_{(t_i,t_j)} \leq \chi^E_{t_i} \quad (12)$$

Our objective is to minimize the makespan $C_{max}$. The makespan is the same as the termination time of the last task $t_i$:

$$\chi^E_{t_i} \leq C_{\max} \quad (13)$$

For each task $t_i$ exactly one implementation should be selected:

$$\sum_{p\in\mathcal{P}} \iota^p_{t_i} = 1 \quad (14)$$

Each task $t_i$ should be executed either on the processor (slot 0) or in any of the six FPGA slots

$$\sum_{s=0}^{6} \lambda_{t_i s} = 1 \quad (15)$$

If $t_i$ is to be executed in software it must be loaded on the processor:

$$\lambda_{t_i 0} = \iota^{\text{SW}}_{t_i} \quad (16)$$

If the tasks implementation uses special resourcers it can only be located in columns 2 and 5, so for all $t_i \in T$ and $s \in \{0, 1, 3, 4, 6\}$:

$$\lambda_{t_i s} \leq 1 - \iota^{\text{SP}}_{t_i} \quad (17)$$

Task $t_i$ is located in slot $\lambda_{t_i}$:

$$\sum_{s=0}^{6} s\lambda_{t_i s} = \lambda_{t_i} \quad (18)$$

The distance between the location of $t_i$ and $t_j$ is $\delta_{t_i t_j}$. This distance is relevant for communication via SR and BM only:

$$\lambda_{t_i} - \lambda_{t_j} \leq \delta_{t_i t_j} \quad (19)$$

$$\lambda_{t_j} - \lambda_{t_i} \leq \delta_{t_i t_j} \quad (20)$$

If $t_i$ and $t_j$ are to communicate via the SRAM or the bus macro interface their distance has to be one, so for all $(t_i, t_j) \in A$ and $c \in \{\text{SR}, \text{BM}\}$:

$$\delta_{t_i t_j} \leq 1 + 6(1 - \gamma^c_{(t_i,t_j)}) \quad (21)$$

If $t_i$ and $t_j$ are located in slot $s$, then set $\lambda_{t_i t_j s}$ to 1; for this purpose, implement a logical **and** by the following linearizition of $\lambda_{t_i s}\lambda_{t_j s}$ for all $t_i < t_j \in T$ and $s \in \{0, 1, \ldots 6\}$:

$$\lambda_{t_i s} + \lambda_{t_j s} - 1 \leq \lambda_{t_i t_j s} \quad (22)$$

$$\lambda_{t_i t_j s} \leq \lambda_{t_i s} \quad (23)$$

$$\lambda_{t_i t_j s} \leq \lambda_{t_j s} \quad (24)$$

$$\pi^L_{t_i t_j} + \pi^L_{t_j t_i} \geq \lambda_{t_i t_j s} \quad (25)$$

$$\chi^E_{t_i} \leq \chi^L_{t_j} + (1 - \pi^L_{t_i t_j})M + (1 - \lambda_{\min\{t_i,t_j\}\max\{t_i,t_j\}s})M \quad (26)$$

(25) makes sure that if $t_i$ and $t_j$ use the same slot, $t_i$ has to precede $t_j$ or vice versa; moreover, if $t_i$ and $t_j$ are located in the same slot and $t_i$ precedes $t_j$ then $t_j$ can be loaded only after $t_i$ unblocks all resources used (26).

For all $(t_i, t_j) \in A$ select exactly one type of communication medium:

$$\sum_{c \in \mathcal{C}} \gamma^{c}_{(t_i,t_j)} = 1 \quad (27)$$

If task $t_i$ is run on the processor, it can only use the crossbar or the common memory attached to the crossbar for acquiring and forwarding data:

$$\iota^{\text{SW}}_{t_i} \leq \gamma^{\text{CB}}_{(t_h,t_i)} + \gamma^{\text{CM}}_{(t_h,t_i)} \quad (28)$$
$$\iota^{\text{SW}}_{t_i} \leq \gamma^{\text{CB}}_{(t_i,t_j))} + \gamma^{\text{CM}}_{(t_i,t_j)} \quad (29)$$

If any other communication paradigm is used for forwarding, the task has to be run in hardware, so for all $(t_i, t_j) \in A$ and $c \in \{\text{SR}, \text{BM}, \text{RB}\}$:

$$\iota^{\text{SP}}_{t_i} + \iota^{\text{HW}}_{t_i} \geq \gamma^{c}_{(t_i,t_j)} \quad (30)$$
$$\iota^{\text{SP}}_{t_j} + \iota^{\text{HW}}_{t_j} \geq \gamma^{c}_{(t_i,t_j)} \quad (31)$$

Finally there are some constraints that deal with the access to the memory attached to the crossbar. Only one task may access this memory at any given point in time. As mentioned above there are six feasible data forwarding/acquisition orders denoted by $\mathcal{O}$ for every pair $(t_i, t_j)$, $(t_k, t_l) \in A$. If both of these edges are common memory edges exactly one of the access orders has to be selected (32, 33, 34):

$$\gamma^{\text{CM}}_{(t_i,t_j)} + \gamma^{\text{CM}}_{(t_k,t_l)} - 1 \leq \sum_{o \in \mathcal{O}} \pi^{\text{CM}}_{o} \quad (32)$$
$$\sum_{o \in \mathcal{O}} \pi^{\text{CM}}_{o} \leq \gamma^{\text{CM}}_{(t_i,t_j)} \quad (33)$$
$$\sum_{o \in \mathcal{O}} \pi^{\text{CM}}_{o} \leq \gamma^{\text{CM}}_{(t_k,t_l)} \quad (34)$$

Selecting one order $o \in \mathcal{O}$ fixes an access pattern to the common memory. For each order there is a group of three equations. We only list one group for the order $(t_i, t_k, t_j, t_l)$:

$$\chi^{F}_{(t_i,t_j)} + f^{\text{CM}}_{(t_i,t_j)} \pi^{\text{CM}}_{t_i t_k t_j t_l} \leq \chi^{F}_{(t_k,t_l)} + (1 - \pi^{\text{CM}}_{t_i t_k t_j t_l}) M \quad (35)$$
$$\chi^{F}_{(t_k,t_l)} + f^{\text{CM}}_{(t_k,t_l)} \pi^{\text{CM}}_{t_i t_k t_j t_l} \leq \chi^{A}_{(t_i,t_j)} + (1 - \pi^{\text{CM}}_{t_i t_k t_j t_l}) M \quad (36)$$
$$\chi^{A}_{(t_i,t_j)} + f^{\text{CM}}_{(t_i,t_j)} \pi^{\text{CM}}_{t_i t_k t_j t_l} \leq \chi^{A}_{(t_k,t_l)} + (1 - \pi^{\text{CM}}_{t_i t_k t_j t_l}) M \quad (37)$$

# 4 Experimental Results

We have successfully tested our ILP on a wide range of randomly generated task graphs. Based on this computational experience, it is safe to say that we can solve any small instance up to 20 tasks in less than an hour on a Pentium IV clocked at 3 GHz. As a solver for the ILP we employed ILOG CPLEX 10.0. Instead of listing results on randomly generated instances we will discuss the results obtained for a small JPEG example.

| Task | HW8 | SW8 | HW256 | SW256 |
|---|---|---|---|---|
| RGB2YCbCr | 1.38 | 5.52 | 1310.82 | 5243.28 |
| 2D-DCT | 9.06 | 36.24 | 9175.14 | 36700.56 |
| Quantize | 1.54 | 6.16 | 1310.98 | 5243.92 |
| Huffman | 1.56 | 62.40 | 1310.00 | 5244.00 |

**Table 2** Hardware execution times in micro seconds for the JPEG case study for 8x8 and 256x256 8 bits data blocks.

This example is based on the JPEG encoder depicted in Fig. 3; it is similar to the case study made in [2]. The data for tasks like color conversion, DCT, quantize and Huffman were obtained after place and route of the synthesized modules under placement and routing constraints. Multiple instances of the JPEG encoder tasks are used to process the problem in parallel, as the nature of problem allows independent data processing.
For SW implementations we assume for each task a execution time that is four times slower than the HW implementation.

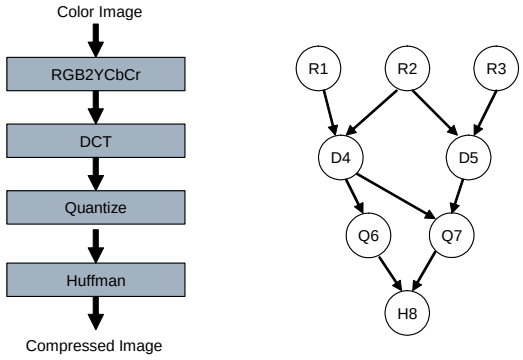


**Figure 3** Task graph for the JPEG encoder.

Implementation details of each task are presented in Table 2. These eight tasks are to be partitioned between six available macro slots in HW and one processor. All execution times are for a 8x8 and 256x256 blocks of 8 bits data.
Solving the ILP for the 8x8 case took 10.19s with the above mentioned equipment. The solution implies that the 8x8 instance can be executed in roughly 0.94976ms. This is less than it would take to configure all eight tasks on the FPGA. Consequently all tasks are mapped to the processor. As software tasks can only communicate using the common memory all communication between tasks is done this way.
When increasing the amount of data to be processed, the picture changes. Solving the ILP for the 256x256 case took 1.29s. Without reconfiguration overhead executing all tasks on the FPGA would take roughly 13ms. Now some of the tasks are executed in software. Others are placed on the FPGA. Communication is mostly done using the reconfigurable multiple bus and the common memory.
When further increasing the amount of data all tasks are executed in hardware.

# 5 Conclusion

In this paper, we presented an exact approach to hardware-software partitioning for SoC architectures including at least one processor and multiple, concurrently running hardware slots. Novel with respect to existing approaches is the joint consideration of schedulding and module placement during partitioning, while considering not only reconfiguration cost, but also communication cost. To the best of our knowledge, our approach constitutes the first *integrated* model for a real-world reconfigurable architecture; we show that it is not only possible to formulate this complex task as an integer linear program, but also to solve it for actual instances.

Although there may be other ways of communication on SoC architectures, we believe that those five considered here are quite representative for FPGA-based reconfigurable devices; this makes our model interesting beyond the particular ESM architecture that was used to verify the real experiments presented in Section 4.

Obviously, solving integer linear problems is an NP-complete problem, so it is not too surprising that the computational difficulties increase with the problem size, in particular with the number of tasks. However, some part of the practical difficulties may also be due to the relatively large number of binary variables to model logical dependencies. We are optimistic that constraint programming in combination with integer linear programming will allow us to solve larger instances.

# 6 References


[1] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. Fekete, and J. van der Veen. A practical approach for circuit routing on dynamically reconfigurable devices. In *Proc. of the 16th IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 84–90, Montreal, Canada, 2005.

[2] S. Banerjee, E. Bozorgzadeh, and N. Dutt. HW-SW partitioning for architectures with partial dynamic reconfiguration. *Technical Report CECS-TR-05-02, UC Irvine*, 2005.

[3] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 335–340, New York, NY, USA, 2005. ACM Press.

[4] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. In R. Gupta, editor, *Design Automation for Embedded Systems*, 3, pages 23–62. Kluwer Academic Publishers, Boston, Jan. 1998.

[5] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. Increasing the flexibility in fpga-based reconfigurable platforms: The erlangen slot machine. In *IEEE Conference on Field-Programmable Technology (FPT)*, pages 37–42, Singapore, Dec. 2005.

[6] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, S. P. Fekete, and J. van der Veen. The Erlangen Slot Machine: A highly flexible FPGA-based reconfigurable platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320, 2005.

[7] K. S. Chatha and R. Vemuri. An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling. *Design Automation for Embedded Systems*, 5(3–4):281–293, Aug. 2000.

[8] H. A. ElGindy, A. K. Somani, H. Schröder, H. Schmeck, and A. Spray. RMB - a reconfigurable multiple bus network. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 108–117, San Jose, California, Feb. 1996.

[9] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 25–30, San Jose, California, Nov. 2001.

[10] B. Jeong, S. Yoo, S. Lee, and K. Choi. Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 169–174, New York, NY, USA, 2000. ACM Press.

[11] B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of ProRISC 2000*, pages 405–411, Veldhoven, Netherlands, Nov. 2000.

[12] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10986, Washington, DC, USA, 2003. IEEE Computer Society.

[13] R. Niemann and P. Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):165–193, Mar. 1997.

[14] J. Noguera and R. M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 205–210, New York, NY, USA, 2002. ACM Press.

[15] J. Noguera and R. M. Badia. Power-performance trade-offs for reconfigurable computing. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hard-*

*ware/software codesign and system synthesis*, pages 116–121, New York, NY, USA, 2004. ACM Press.

[16] OSCI. *SystemC Version 2.0 Users Guide*. Open SystemC Initiative, 2002. www.systemc.org.

[17] J. Resano and D. Mozos. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 119–124, New York, NY, USA, 2004. ACM Press.

[18] M. Schwiegershausen and P. Pirsch. A system level design methodology for the optimization of heterogeneous multiprocessors. In *Proceedings of 8th International Symposium on System Synthesis*, pages 162–167, Cannes, France, Jan. 1995.

[19] T. Streichert, C. Haubelt, and J. Teich. Distributed HW/SW-partitioning for embedded reconfigurable systems. In *Proceedings of Design, Automation and Test in Europe*, pages 894–895, Munich, Germany, Mar. 2005.

[20] F. Vahid and T. D. Le. Extending the Kernighan/Lin heuristic for hardware and software functional partitioning. *Design Automation of Embedded Systems*, 2(2):237–261, Mar. 1997.

[21] R. Vaidyanathan and J. L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms*. IEEE Computer Society, 2003.

[22] Xilinx Inc. *Xilinx Virtex-II Platform FPGA User Guide.*, 2005. http://www.xilinx.com.