Perfecto: A SystemC-Based Design-Space Exploration Framework for Dynamically Reconfigurable Architectures

PAO-ANN HSIUNG, CHAO-SHENG LIN, and CHIH-FENG LIAO
National Chung Cheng University

To cope with increasing demands for higher computational power and greater system flexibility, dynamically and partially reconfigurable logic has started to play an important role in embedded systems and systems-on-chip (SoC). However, when using traditional design methods and tools, it is difficult to estimate or analyze the performance impact of including such reconfigurable logic devices into a system design. In this work, we present a system-level framework, called Perfecto, which is able to perform rapid exploration of different reconfigurable design alternatives and to detect system performance bottlenecks. This framework is based on the popular IEEE standard system-level design language SystemC, which is supported by most EDA and ESL tools. Given an architecture model and an application model, Perfecto uses SystemC *transaction-level models* (TLMs) to simulate the system design alternatives automatically. Different hardware-software copartitioning, coscheduling, and placement algorithms can be embedded into the framework for analysis; thus, Perfecto can also be used to design the algorithms to be used in an operating system for reconfigurable systems. Applications to a simple illustration example and a network security system have shown how Perfecto helps a designer make intelligent partition decisions, optimize system performance, and evaluate task placements.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.8 [**Operating Systems**]: Performance—*Simulation*

General Terms: Design, Experimentation, Performance, Verification

Additional Key Words and Phrases: Reconfigurable systems, partitioning, scheduling, placement, performance evaluation, design-space exploration

ACM Reference Format:

Hsiung, P.-A., Lin, C.-S., and Liao, C.-F. 2008. Perfecto: A SystemC-based design-space exploration framework for dynamically reconfigurable architectures. ACM Trans. Reconfig.

This work was supported by the National Science Council, Taiwan, under project grant NSC96-2221-E-194-065-MY2.

Authors' address: P.-A. Hsiung (corresponding author), C.-S. Lin, C.-F. Liao, National Chung Cheng University, 168 University Road, Min-Hsiung, Chiayi, Taiwan-62102, ROC; email: hpa@computer.org.

Technol. Syst. 1, 3, Article 17 (September 2008), 30 pages, DOI = 10.1145/1391732.1391737
http://doi.acm.org/10.1145/1391732.1391737

1. INTRODUCTION

With rapidly increasing consumer requirements for product features and the entrance of the electronics industry into the low-profit era, there is a pressing financial reason for adding more flexibility to system designs without sacrificing the performance and possible parallelism of ASIC. Reconfigurable technologies [Compton and Hauck 2002] are viewed as the solution for getting this flexibility; however, the introduction of dynamic reconfiguration adds several new dimensions to the system design-space, since the same logic area can be configured to execute different functions at different times. These new dimensions have created problems in evaluating the performance of system designs, since now this depends not only on fixed hardware and software scheduling, but also on how the reconfigurable space is efficiently used to accelerate system tasks.

Figure 1 compares the traditional system architecture with a dynamically reconfigurable one. Figure 1(a) shows a traditional SoC with a processor, memory, and some hardware accelerators. Figure 1(b) illustrates a *dynamically reconfigurable system-on-chip* (DRSoC) architecture [Pelkonen et al. 2003], which is an SoC architecture enhanced with an embedded *dynamically reconfigurable logic* (DRCL), such as FPGA, and a ROM for storing the reconfigurable functions.

Currently, there are many research groups studying reconfigurable technologies [Baleani et al. 2002; Chang and Marek-Sadowska 1998; Desmet et al. 2002; Loo and Wells 2006; Mei et al. 2000; Noguera and Badia 2003; Rakhmatov and Vrudhula 2002; Trimberger 1998]. Most of the research is focused on hardware-software partitioning, task scheduling, and placement algorithms. However, there is no framework that integrates all these emerging design technologies for DRSoC because existing frameworks, such as the ADRIATIC design flow [Pelkonen et al. 2003] and SyCERS [Santambrogio 2008], focus mainly on the design-space exploration of reconfigurable architectures and not on the design algorithms. In fact, the current practice is to synthesize the hardware circuits into a vendor-specific FPGA such as Xilinx or Altera and then do some performance evaluations using CAD tools such as CoWare's ConvergenSC or Synopsys' System Studio. However, these evaluations are all for some fixed design configuration, as these commercial tools do not support dynamic or partial reconfigurations of the system under design. In this work, a design-space exploration framework called Perfecto is proposed and developed to fill this integration gap. Perfecto can be used not only to explore the design space of dynamically and partially reconfigurable systems, but also to explore the construction of the design algorithms that must be implemented into an *operating system for reconfigurable systems* (OS4RS).

In creating such a framework, there are several issues to be resolved, described as follows. Solutions or pointers to solutions are also presented.

—*Architecture Model*. How must one define a parameterized architecture model such that users can configure it by simply setting the parameters?

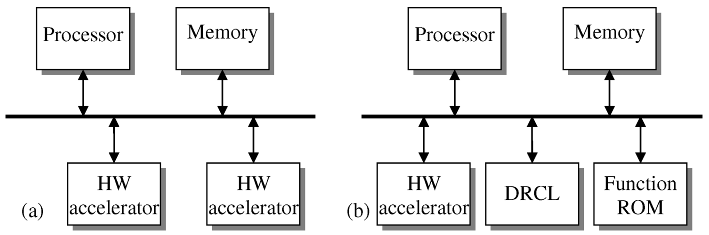

Fig. 1. SoC architectures: (a) traditional; (b) reconfigurable.

Section 3.1 will give a formal definition of the reconfigurable architecture model used in Perfecto.

—*Application Model*. How must one define an application model such that a generic application can be executed on the architecture model? Section 3.2 will give a formal definition of the application model used in Perfecto.

—*Design Algorithms*. What kind of design algorithms must be supported by the framework such that the execution of the application model on the architecture model can be thoroughly analyzed? Partitioning, scheduling, and placement algorithms are considered in Perfecto and described in Sections 3.3, 3.4, and 3.5, respectively.

—*System Evaluation*. What kinds of system features must be evaluated such that they are of use to system designers? Section 3.6 will give the four task features and five partition features that are evaluated in Perfecto.

—*Guidelines*. In what ways can Perfecto guide designers in choosing the right design alternatives that meet user requirements? Perfecto helps designers in three ways: (a) making intelligent partition decisions, (b) optimizing performance, and (c) evaluating task placements. Examples are used in Section 4 to illustrate these guidelines.

Perfecto is based on SystemC [OSCI 2008], an IEEE 1666 standard system-level modeling language supporting both software and hardware specifications. Executable specification with simulation is an added benefit of SystemC, which is rapidly becoming the language of choice for system-level design. This is partly due to the fact that all large EDA vendors support or plan to support SystemC in their tools. Perfecto takes full advantage of unique features of SystemC, such as built-in simulation, transaction-level modeling, software-hardware modeling, communication modeling, and performance evaluation.

The rest of the article is organized as follows. Related previous work is described in Section 2. Section 3 describes the Perfecto framework in detail, including the SystemC-based reconfigurable architecture model, the application model, and the sample partitioning, scheduling, and placement algorithms implemented. Section 4 illustrates the benefits of Perfecto by applying it to two examples. Section 5 gives some conclusions with consideration for future work.

2. PREVIOUS WORK

Being a simulation-driven system description language, the IEEE 1666-2005 SystemC standard language has been used for design-space exploration at system level in several application domains, such as embedded software, SoC, multicore systems, and reconfigurable architectures. The *transaction-level modeling* (TLM) interfaces supported by SystemC not only make simulation possible at different abstraction levels, but also accelerate simulation, thus enabling rapid design-space exploration. Further, due to the C++-based design, SystemC can be used to model both hardware and software. All these features make SystemC a very suitable language for exploring and evaluating reconfigurable system architectures and applications. Nevertheless, SystemC is still restricted in its capability to model dynamic reconfiguration because runtime instantiation of `sc_module` and dynamic binding of `sc_method` and `sc_thread` are not allowed in the current IEEE 1666 version of SystemC. Nevertheless, it was shown by Rissa et al. [2005] that SystemC can achieve a much faster simulation speedup compared to traditional HDLs.

SystemC has been used for modeling and exploring reconfigurable system architectures in some previous work such as the ADRIATIC project [Pelkonen et al. 2003; Qu et al. 2004; Tiensyrjä et al. 2004] and the SyCERS framework [Santambrogio 2008]. The ADRIATIC project used SystemC to model dynamically reconfigurable systems by introducing a *dynamically reconfigurable fabric* (DRCF) [Pelkonen et al. 2003]. Reconfigurable components are all mapped to a DRCF that is generated from a template which contains a context scheduler, an instrumentation process, and a multiplexer that routes data transfers to correct instances. Interfaces and ports of reconfigurable components are all added to the DRCF. Bus cycle-accurate performance evaluations such as reconfiguration delay, context size, and computation delays can be obtained. The limitations are as follows. All components mapped to DRCF must be at the same level of hierarchy. Partial configuration was not supported because DRCF uses context switching to change between the different reconfigurable functions. DRCF was later extended into *dynamically reconfigurable coprocessors* (DRC) [Qu et al. 2004], which support partial reconfiguration using a configuration scheduler and an input splitter. However, only reconfiguration latencies were evaluated. Energy consumptions were not modeled in DRCF and DRC and the support for design-space exploration of system architecture alternatives was limited.

SyCERS [Santambrogio 2008] is also a SystemC-based framework for design-space exploration of dynamically reconfigurable systems. Partial reconfiguration is supported. Instead of static binding of multiple functions through a multiplexer, as in DRCF, or through an input splitter, as in DRC, SyCERS uses function pointers that are changed at runtime to model dynamic reconfiguration. SyCERS allows users to model the application through a fixed set of interfaces and to model the system architecture using several TLM black boxes constituting the YaRA architecture. SyCERS uses `sc_thread` to model a reconfigurable component and uses `sc_mutex` to synchronize configurations and executions. Elaborating on details of the configuration process and configuration controller is the focus of SyCERS, which can help a designer to decide on a more optimal architecture for a particular application in terms of the number of black boxes. SyCERS does not focus on how hardware-software partitioning, scheduling, and placement are performed for an architecture-application combination.

The newly proposed Perfecto framework is most similar to SyCERS because we are also trying to find a match between an application model and a reconfigurable architecture TLM model. Its difference from DRCF, DRC, and SyCERS is that instead of considering fixed algorithms, Perfecto evaluates partitioning, scheduling, and placement algorithms along with an architecture and an application. Further, in Perfecto, design-space exploration is automatically performed by providing an interface to random task graph generation, evaluating multiple partitionings of the system, detecting performance bottlenecks, and evaluating the placement of all reconfigurable tasks in each partitioning. A designer can choose the best architecture by referring to the partition evaluations in Perfecto. Section 3.6 will present more details on how Perfecto performs design-space exploration.

3. PERFECTO FRAMEWORK

Design-space exploration and performance evaluation are extremely important, but difficult for reconfigurable system designers to achieve, due to the complex dynamic nature of such systems and due to the multitude of combination possibilities in hardware-software partitioning, reconfigurable hardware scheduling, and reconfigurable hardware placement. Moreover, scheduling and placement must be concurrently considered for a feasible design solution. Perfecto is a framework proposed for integrating the design algorithms and for the design-space exploration of dynamically reconfigurable systems through performance evaluation.

As shown in Figure 2, the design flow for dynamically reconfigurable systems is divided into two phases, namely, a front-end design and a back-end design. The front-end design phase takes an architecture model and an application model, which might be derived from user-given system specifications, and then generates three kinds of tasks, namely, hardware task, reconfigurable hardware task, and software task. The back-end design phase synthesizes the three kinds of tasks, performs more detailed hardware-software cosimulation, and then implements the full system using back-end tools such as a hardware synthesis tool, software compiler, gate-level simulator, and power estimation tool. Perfecto nicely fits into the front-end design phase as the main tool for design-space exploration and performance evaluation.

As shown in Figure 3, Perfecto takes two inputs, namely an architecture model and an application model, which will be defined later in Sections 3.1.4 and 3.2, respectively. Hardware-software partitions are then generated by Perfecto. For each partition, the scheduler in Perfecto schedules the reconfigurable hardware and the software tasks (it is assumed here that the fixed hardware accelerators are part of the DRSoC architecture and thus not scheduled by Perfecto). Then, Perfecto simulates the execution of the software tasks on a processor, places the hardware tasks in the reconfigurable logic, and simulates their execution on the DRCL. Finally, after all task executions in all partitions

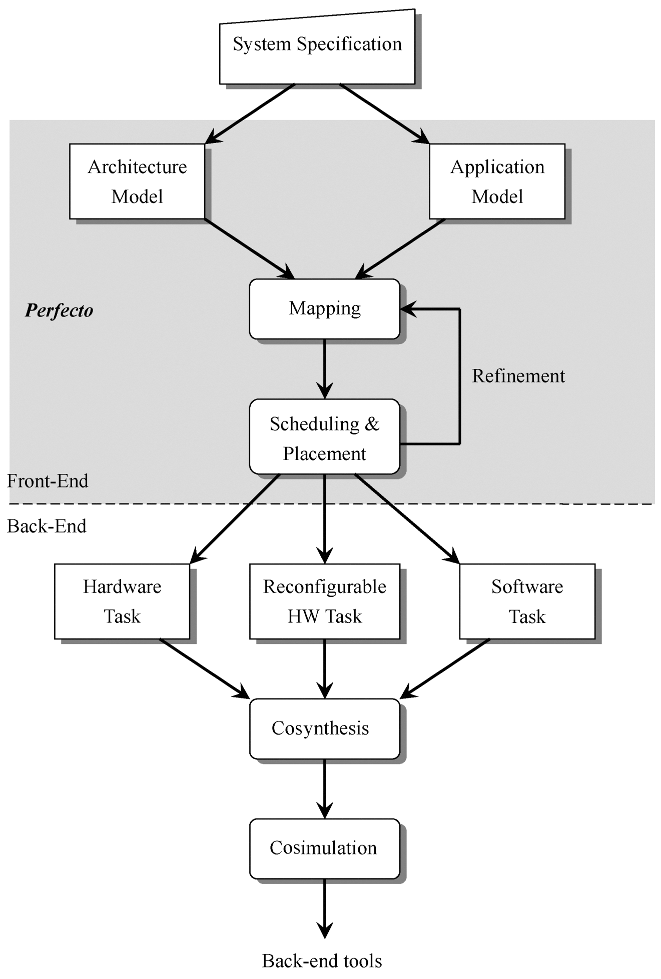

Fig. 2. Design flow of DRSoC.

have been simulated, Perfecto generates several reports for the system designer, including partition evaluations, bus-access conflicts, and real-time placement information, which are described in Section 3.6.

In the rest of this section, we will first describe the two inputs of Perfecto, namely, the reconfigurable architecture model and parameters in Section 3.1 and the application model and task parameters in Section 3.2. Later, in Sections 3.3, 3.4, and 3.5, respectively, we will also describe the partitioning, scheduling, and placement algorithms that were implemented into Perfecto for illustration purposes. Designers can always change these algorithms or use the real-time information provided by Perfecto to construct a suitable algorithm. It must be noted here that the algorithms themselves are not the focus of design, but rather the evaluation framework design and how it can benefit designers. The algorithms will be the focus of design when we design an operating system for OS4RS.

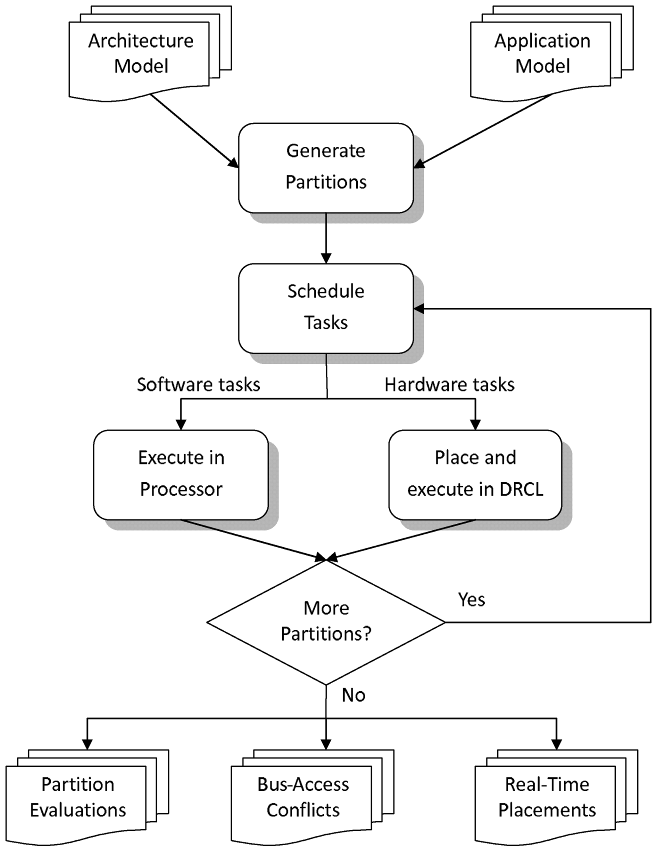

Fig. 3. Perfecto simulation flow.

3.1 Reconfigurable Architecture Model

For performance evaluation, we need a basic architecture model of the target dynamically reconfigurable system. As shown in Figure 4, the basic reconfigurable system architecture model in Perfecto consists of a processor model, a memory model, a function ROM model, a bus model, an arbiter model, and a dynamically reconfigurable logic (DRCL) model. We use the SystemC design language [OSCI 2008] to develop this architecture model because we are doing design-space exploration at the system level and because the design contains both hardware and software functions. In this architecture, a simple bus model is used as communication infrastructure for the hardware tasks. Here, "simple" means that there is no pipeline and no split transaction. The bus model eliminates the need to do global routing after tasks are placed into the DRCL. Function ROM is a memory storage to save configurations (e.g., bitstreams) that will be loaded into and executed in DRCL. The other system models are described in the rest of this subsection.

3.1.1 *Processor Model.* A processor is required to execute the software in a DRSoC. Besides controlling peripheral devices, it has mainly two behaviors.

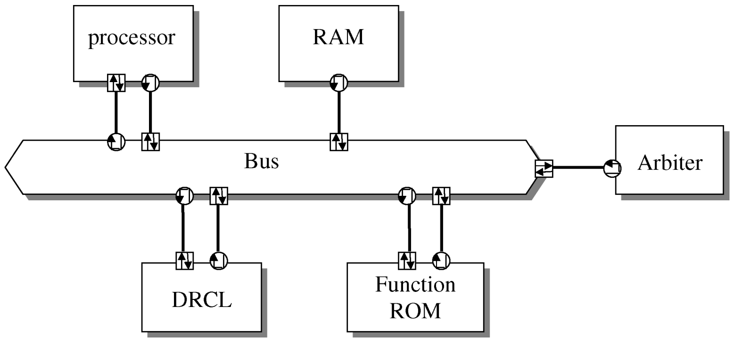

Fig. 4. DRSoC SystemC architecture model.

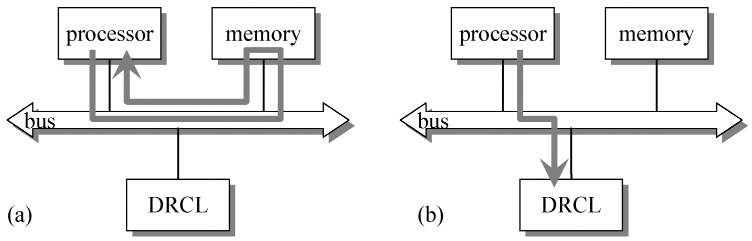

Fig. 5. Behavior of processor.

One is using the bus to access data from memory for executing software instructions, as shown in Figure 5(a). The other is to issue commands to DRCL, such as reconfigure or execute, as shown in Figure 5(b). These two behaviors of the processor are simulated at transaction level.

3.1.2 *Bus Arbiter Model.* An arbiter model is required, as there is more than one master device on the bus in a DRSoC, including the processor and the DRCL. The main behavior of an arbiter is to arbitrate when more than one request is made for the bus. The arbiter selects the most suitable request to grant bus access according to the following policy, while the other requests are kept in the waiting queue.

(1) If the current request is a locked burst request, then it is always selected.
(2) If the last request had its lock flag set and the corresponding master is "requesting" again, this request is selected from the waiting queue and returned.
(3) The request with highest priority (smallest number) is returned from the waiting queue.

3.1.3 *DRCL Model.* There are three main behaviors in a DRCL model. First is memory access. As shown in Figure 6(a), when a DRCL receives an execute command from the processor, the DRCL will access memory according to the address parameters. Next, DRCL may request a bitstream from the function ROM, as shown in Figure 6(b). As shown in Figure 6(c), DRCL may issue a response command to the processor when it completes a task. Note that the

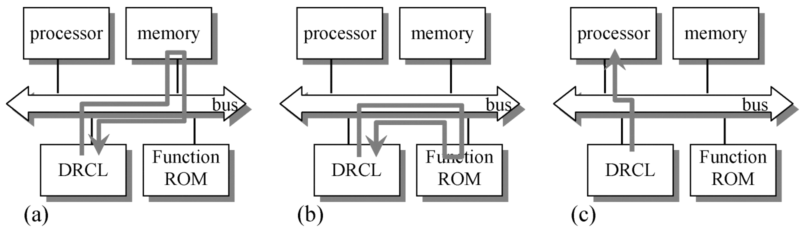

Fig. 6. DRCL behaviors.

configuration controller is embedded within the DRCL model, which is similar to the embedded ICAP controller in Xilinx Virtex FPGAs.

Reconfigurable logics can be divided into two configuration styles, namely, full configure (static) and partial configure (dynamic). The DRCL model in Perfecto can simulate both types of configuration. However, we must note here that SystemC does not allow dynamic binding of modules with their behaviors and also does not allow a module to have multiple behaviors that can be configured at runtime. We can thus say that SystemC does not support reconfiguration of any kind. There are several workarounds for simulating reconfiguration in SystemC. A straightforward method is the static binding of a SystemC module to multiple behaviors and then selecting one of the behaviors for dynamic execution. This method is the simplest, but quite inflexible because for every new function, we need to modify the DRCL model. Similar to DRCF [Pelkonen et al. 2003] and DRC [Qu et al. 2004], Perfecto adopts this method because it is simple and fast to simulate. For design-space exploration, speed of simulation is of utmost importance. Other methods include the use of C function pointers [Santambrogio 2008] and C++ templates, all of which might cause an overhead in SystemC simulation performance and are thus not very suitable for design-space exploration. Further, multiple `sc_threads` are used in the DRCL for modeling partial reconfiguration. To avoid the use of function pointers, Perfecto uses a function table and a task table, with interfaces for automatic insertion of new functions and new tasks.

3.1.4 *Architecture Model and Parameters.*

In Perfecto, the basic architecture model as illustrated in Figure 4 is simulated using the aforesaid models of the processor, memories, the arbiter, and the DRCL. A software task executes in the processor model by accessing the memory. A reconfigurable hardware task executes in the DRCL model by accessing the memory for input and output data. Communications between a software task and a hardware task are accomplished by the processor and the DRCL models. A hardware function reconfiguration is accomplished by the DRCL model by accessing the function ROM. The arbiter grants access to the bus for memory accesses by the processor and the DRCL.

If Perfecto simulates only a fixed basic architecture model, then it will be of little use to a designer who wants to experiment with different system design alternatives. Thus, Perfecto allows a system designer to tune the basic architecture model through several architecture parameters, as described in the following definition.

Definition 3.1. An architecture model is defined as a tuple $S = \langle W_{bus}, N_{mem}, T_{mem}, N_{slice}, A_{part}, A_{sched}, A_{place}\rangle$, where the parameters are as follows.

—W_{bus} is the bus width. The basic unit is a word of 4 bytes. A designer may specify the bus width in units of word. This parameter affects the memory-access counts of an application running in the target system.

—N_{mem} is the memory size. This is a multiple of 4 because the smallest memory unit is 4 bytes. The parameter affects the application-response time.

—T_{mem} is the memory-access time. This parameter is the time for one memory access. This parameter also affects the application-response time.

—N_{slice} is the DRCL size. This is the total number of slices in a DRCL.

—A_{part} is the partitioning algorithm. The default partitioning algorithm is function-based partitioning. Users can implement their own partitioner for task mapping or choose an optional method, including the common-hardware-first or the random partitioning as described in Section 3.3.

—A_{sched} is the scheduling algorithm. The default scheduling algorithm is simply FIFO. Users can implement their own scheduler to optimize an application. Another scheduler also currently implemented in Perfecto is an energy-efficient hardware-software co-scheduler [Liu 2006] for reconfigurable systems.

—A_{place} is the placement algorithm. The default placement algorithm is a rule-based one, which will be described in Section 3.5. Users can implement their own placer to place the hardware tasks into DRCL.

As an illustrative example, the architecture model could be as follows. The bus width is 2 words, the memory size is 100MB, the memory-access time is 10ns, the maximum number of DRCL slices available is 5, and the partitioning, scheduling, and placement algorithms are all the default ones.

3.2 Application Model

Besides using parameters to model a user-desired dynamically reconfigurable system architecture for simulation, Perfecto further allows designers to specify the application model that represents an application to be executed on the reconfigurable architecture model. An application is defined as set of concurrent tasks with possible precedence relations among these tasks. Thus, an application can be formalized as follows.

Definition 3.2. An application is represented by a directed acyclic graph $G(V, E)$, where these aspects are as follows.

—V is a set of nodes representing the application tasks. Each task T_i invokes a function F_j, which is represented by the tuple $F_j = (f_{name}, t_{sw}, t_{cfg}, t_{hw}, n_{slice}, f_{code})$, where f_{name} is a unique function name, such as DES, AES, DCT, etc.; t_{sw} is the computation time of the software implementation of the function, without considering memory-access time; t_{cfg} is the configuration time of the hardware implementation of the function; t_{hw} is the computation time of the hardware implementation of the function; n_{slice} is the area

```
bool func_rom::func_1(unsigned int priority, int task_no, int CT,
int ET,
    unsigned int src1, unsigned int src2, unsigned int src3, unsigned int des1,
    unsigned int des2, int slice_no, int sw_hw) {
  double time_1 = 0;
  int time_2 = 0;
  int data1[src3], data2[src3];

  time_1 = sc_simulation_time(); // record start time
  if(CT != 0){ time_2 = CT; wait(CT, SC_NS); } // CT: configuration time
  func_rom_bus_port->burst_read(priority, task_no, data1, src1, src3);
  func_rom_bus_port->burst_read(priority, task_no, data2, src2, src3);
  for(unsigned int i=0; i<src3; i++)   data1[i] = data1[i] + data2[i] + 1;
  wait(ET, SC_NS); // ET: execution time, without communication
  func_rom_bus_port->burst_write(priority, task_no, data1, des1, des2);
  time_1 = sc_simulation_time() - time_1; // calculate total time
  func_rom_bus_port->direct_response(sw_hw, task_no, (int)time_1-ET-CT,
                                   time_2, slice_no);
  return true;
}
```

Fig. 7. SystemC function behavior code for simple example.

of DRCL required by the function in terms of the number of slices, where a slice is basic unit of configuration such as a frame, column, or tile in Xilinx Virtex FPGAs; and f_{code} is the *function behavior code* implemented as SystemC transaction-level code and is used to model the function behavior (see Figure 7 for an example).

Note that the same function can be invoked by different tasks, but without any data sharing between the different invocations.

—E is a set of edges representing the task precedence relations. An edge $(u, v) \in E$ means that task v must wait for task u to complete before starting execution.

An application is specified by a designer through several task parameters extracted from Definition 3.2, including the set of tasks and functions, the mapping between tasks and functions, the six function attributes, and the precedence relations among the tasks. Note that modeling a new application into an appropriate set of tasks could be a complicated job, which is out of the scope of the current work.

To illustrate the aforesaid task parameters, we will use a simple application that has six tasks invoking four functions as given in Table I(a), where the mappings between tasks and functions are given and where it is also specified that task T3 starts execution only after task T5 is done. The function attributes specified by the user are shown in Table I(b). For example, function F3, when implemented in software, requires 1300ns execution time without considering memory accesses, and when implemented in hardware requires 150ns configuration time, 600ns execution time, and uses 2 slices. A generic example of function behavior code is shown in Figure 7. Thus, Table I and Figure 7 depict the task parameters.

Table I. Tasks in Illustration Example

$V = \{T1, T2, T3, T4, T5, T6\}, E = \{(T5, T3)\}$						
Tasks (V)	T1	T2	T3	T4	T5	T6
Function	F1	F3	F2	F2	F4	F3

(a) Task Graph $G(V, E)$

$fname$	$t_{sw}(ns)$	$t_{cfg}(ns)$	$t_{hw}(ns)$	n_{slice}
F1	200	0	0	0
F2	1000	100	500	1
F3	1300	150	600	2
F4	2000	200	1000	1

(b) Function Parameters

The architecture parameters for this example are similar to what were specified previously, at the end of Section 3.1.4. The results of performance evaluation for this simple example using Perfecto will be presented in Section 4.1.

Sometimes a designer might want to evaluate a specific reconfigurable system architecture along with some specific combinations of partitioning, scheduling, and placement algorithms, but might not want to target it for some specific application. To perform such application-independent evaluations, we designed a randomized application model interface of Perfecto based on the *task graphs for free* (TGFF) tool [Dick et al. 1998], as described in Section 3.2.1.

3.2.1 *TGFF Interface.* *Task graphs for free* (TGFF) [Dick et al. 1998] is a tool for generating random task graphs based on some user-specified requirements on the graphs. TGFF has been widely used by many academic and industrial tools for computer-aided design. Since Perfecto is used for design-space exploration, an interface to TGFF would eliminate the need for users specifying the exact application for evaluation. The TGFF interface allows users to thoroughly evaluate a reconfigurable system along with its partitioning, scheduling, and placement algorithms because the task graphs are randomly generated.

TGFF generates the task set information from a template by varying the seed for the random number generator per template. The template parameters to be defined include the number of task graphs, the average number of functions in a task graph, and the function attributes. The textual representation of task graphs generated by TGFF is then automatically parsed by Perfecto into intermediate task data structures that can then be used for partitioning, scheduling, and placement, thus automating design-space exploration and performance evaluation.

3.3 Partitioning

A partitioning algorithm maps each task in an application model to either software or hardware, based on some estimation criteria such that the task is executed either on the microprocessor or in the DRCL, respectively. Formally, it is defined as follows.

Definition 3.3. Given an architecture model S and an application model $G(V, E)$, the partitioning algorithm is defined as $A_{part}(G(V, E), S) = \mu(V, \{0, 1\})$, where $\mu(T_i) = 0$ represents that T_i is mapped to software, and $\mu(T_i) = 1$ represents that T_i is mapped to hardware.

By selecting the mapping criteria, we can have different partitioning algorithms. A mapping result of a partitioning algorithm is called a *system partition*, or simply *partition*. A partitioning algorithm thus generates a set of system partitions. For experiment purposes, Perfecto implemented three hardware-software partitioning algorithms, namely function-based, common-hardware-first, and random partitioning, as described next.

The *function-based partitioning* algorithm maps each function into hardware and/or software, according to the function attributes. For a set F of functions, at most $2^{|F|}$ partitions are generated, irrespective of the number of tasks. Hence, in a partition, even if the same function is invoked by multiple tasks, all of them are mapped to the same implementation (either hardware or software, depending on the partition). Though nonexhaustive, this mapping greatly reduces the number of partitions generated since the number of functions is usually smaller than the number of tasks.

The *common-hardware-first partitioning* algorithm first counts the number of times each function is invoked, denoted by $c(f)$ for a function $f \in F$. If $c(f) > 1$, then f is called a *common* function. The common functions are then sorted in descending order according to $c(f)$. Given a parameter $k > 0$ representing the number of common hardware functions desired, we map the first k common functions from the ordered list into hardware and map the rest of the functions into software. This partitioning algorithm is useful because several scheduling and placement algorithms often employ heuristics based on common hardware functions, for example, energy-efficient hardware-software scheduling [Hsiung and Liu 2007] and configuration-reuse scheduling and placement methods [Noguera and Badia 2003; Resano and Mozos 2004].

A third random partitioning method generates random partitions according to user requirements. We did not employ any complex partitioning algorithm in Perfecto because our purpose was not to propose a new partitioning algorithm; our purpose was merely to check whether the framework can be used to efficiently evaluate dynamically reconfigurable system designs.

Though several partitioning algorithms were implemented in Perfecto, users have to compare the results of the different partitioning algorithms manually after Perfecto has generated the partitions, by applying the algorithms one-by-one. The partitioning results of different algorithms can be compared by considering either the total number of partitions generated by an algorithm or the quality of the partitions generated. After Perfecto applies the partitioning, scheduling, and placement algorithms, the quality of the partitions can be gauged. Details of the characteristics of partitions can be found in Section 3.6. An ideal partitioning algorithm is one that can generate the optimal partition within a minimal number of partition results. Optimality in quality and minimality in quantity are conflicting goals, and thus a real partitioning algorithm can only generate a near-optimal partition in a manageable number of partitions. Users can thus select a partitioning algorithm based on the desired trade-off between quality and quantity.

Users can also invent a new partitioning algorithm and implement it into Perfecto to check whether the evaluated performance improves. Perfecto was modularly designed such that independent data structures were used for the set of all tasks, the set of software tasks, and the set of hardware tasks. Thus, a user has to merely rewrite the `Generate_Partition()` function to implement a new partitioning algorithm. Well-defined interfaces between this function and the consequent scheduling algorithm have thus helped users to implement new algorithms into Perfecto.

3.4 Scheduling

A scheduling algorithm associates an ordering to the set of software tasks that are ready so that they will be executed in this order on the microprocessor. It also associates an ordering to the set of hardware tasks that are ready so that they will be placed and executed in this order in the DRCL. The algorithm is defined formally in Definition 3.4.

Definition 3.4. Given a set of tasks V, a system partition μ, and a time instant t, a scheduling algorithm is defined as $A_{sched}(V, \mu, t) = \langle \gamma_{sw}(V, t), \gamma_{hw}(V, t) \rangle$, where γ_{sw} associates a strict total order to the set of software tasks that are ready at time t, namely, $\{T_j \mid \mu(T_j) = 0, (T_i, T_j) \in E \rightarrow T_i$ has terminated$\}$ and γ_{hw} associates a strict total order to the set of hardware tasks that are ready at t, namely, $\{T_j \mid \mu(T_j) = 1, (T_i, T_j) \in E \rightarrow T_i$ has terminated$\}$.

By selecting the ordering criteria, different scheduling algorithms can be designed. A specific order of tasks generated by a scheduling algorithm is called a *schedule* and the time required to complete the tasks in this order is called the *schedule length*. Two scheduling methods were implemented in Perfecto: a default FIFO method and an energy-efficient hardware-software coscheduling algorithm [Hsiung and Liu 2007].

Given the same system partition μ, different scheduling algorithms result in different schedule lengths. Smaller schedule lengths are desired so that the set of tasks complete execution sooner. The overall system-schedule length is the maximum of the hardware-schedule length and the software-schedule length. However, due to task dependencies, the hardware schedule and software schedule usually affect each other; thus, in general it might not be possible to optimize only one of the two schedules without considering the other one. Perfecto allows the application of different scheduling algorithms on the same set of partitions. Users can select the scheduling algorithm that results in the smallest system-schedule length. However, it must be noted here that the hardware-schedule length is directly affected by the placement algorithm.

Here, also, we were not intent on proposing a new scheduling algorithm. Users can always implement an existing algorithm such as the Horizon or Stuffing algorithms [Steiger et al. 2004], classified stuffing [Hsiung et al. 2008], and real-time relocatable task scheduling [Chiang 2007], or can invent a new one. Since the algorithms themselves are out of the scope of the present work, interested readers may refer to Hsiung and Liu [2007] for further details on the energy-efficient hardware-software coscheduling algorithm that was implemented in Perfecto. Integrating new scheduling algorithms into Perfecto also allows another dimension of exploration, whereby we can tune a scheduling algorithm or select the best for a specific application. Due to modular design and the well-defined interfaces in Perfecto, users have merely to modify or rewrite the `Scheduler()` function. This function takes two sets of tasks Q_1 and Q_2 and sorts them according to some criteria, where Q_1 is a set of ready hardware tasks and Q_2 is a set of ready software tasks.

3.5 Placement

A placement algorithm tries to find a feasible location in a fixed-size DRCL for a given hardware task of some fixed size. The placement algorithm depends on the underlying configuration model, namely, paged one-dimensional, segmented one-dimensional, or two-dimensional, where the basic units of configuration are a fixed-size *slot*, a *column*, or a *tile*, respectively. In Perfecto, an abstract model is used where the basic unit of configuration is simply called a *slice*. Thus, in Perfecto the underlying configuration model could be any of the three.

Definition 3.5. Given a DRCL of N_{slice} slices, a list L_{used} of spaces allocated to tasks, a list L_{free} of free spaces, and a task T_j of size $n_{slice}(T_j)$ slices to be placed, a placement algorithm is defined as $A_{place}(T_j, L_{used}, L_{free}) = \langle loc, L'_{used}, L'_{free}\rangle$, where loc is either NULL or a pointer to a feasible location for the task T_j in the DRCL such that $n_{slice}(loc) \geq n_{slice}(T_j)$. Suppose that after placing T_j in loc, the used part of loc is denoted as loc' and the remaining free part, if any, is denoted by loc'', that is, $n_{slice}(loc) = n_{slice}(loc') + n_{slice}(loc'')$, where $n_{slice}(loc') = n_{slice}(T_j)$. Then, the lists are updated as follows: $L'_{used} = L_{used} \cup \{loc'\}$. $L'_{free} = \text{Merge\_Adj}(L_{free} \backslash \{loc\} \cup \{loc''\})$, where the function Merge_Adj() tries to merge adjacent free spaces into contiguous blocks of free space.

By changing the selection criteria for the feasible location to place a task, different placement algorithms can be invented. Currently in Perfecto, a placement policy whereby the DRCL selects a block for configuration according to the following rules.

(1) If there exists a block which is already configured with the same circuit but which is not executing currently, that is, it is in the *done* status, then reuse the block by selecting it for the current task.
(2) If there exists a configured block with the same slice count, but with a different function and at the done status, then configure the new task in this block.
(3) If there exists an unconfigured block, that is, in the *idle* status, with enough slices, then configure in this block.
(4) If there is no free block with enough slices, the blocks at the done status will be released into the idle status, then check rule 1 to rule 3 again.

An example is shown in Figure 8(a), where a DRCL is divided into several slices, S0 to S5. Suppose there are three circuits already configured in the DRCL, but that C1 and C3 are at the done status, while C2 is in the *execute* status. If a request for circuit C2 is issued to the DRCL, then, according to aforesaid placement rules, this task will be configured and executed in S0. If a

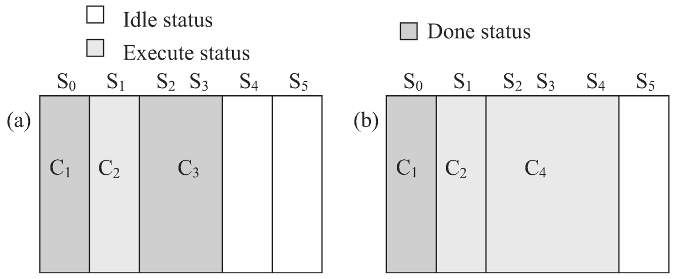

Fig. 8. Hardware task placement in perfecto.

request is received for circuit C4 by the DRCL and if circuit C4 needs 3 slices, then circuit C3 will be released first, and then circuit C4 will be configured into S2 to S4, as shown in Figure 8(b).

The aforementioned is only a sample one-dimensional placement strategy implemented in Perfecto. The algorithms themselves are not the focus of this article. Users can always implement two-dimensional and other placement strategies such as multi-objective placement [Liao 2007] and integrate them into Perfecto for further evaluation of both algorithms and systems. The function `Placer()` takes a task, a list of used blocks, and a list of free blocks. It then computes the ideal location for the task and updates the two lists as described in this section. Users simply have to modify or design a new function to replace the default Perfecto `Placer()`.

As a final note to this subsection, we remark that in a real system, a bitstream must be *relocated* to the target location before it can be used for configuration, where relocation simply means a change of the major address (MJA) to the target column or tile. This relocation can be achieved using software such as Parbit, or hardware filters such as REPLICA. Since the bitstream relocation does not affect the performance of the system significantly, it is abstracted in Perfecto.

3.6 Performance Evaluation Results

Applying the partitioning, scheduling, and placement algorithms described in Sections 3.3, 3.4, and 3.5, respectively, to a user-parameterized reconfigurable architecture model and a user-specified application model, the built-in simulation capabilities of SystemC are used to simulate the system. During simulation, as shown in Figure 3, several performance readings are collected and three results generated, including partition evaluations, bus-access conflicts, and real-time placements.

Before describing how the results are evaluated by Perfecto, some basic terminologies and definitions are required. Given a task t that invokes a function $F = (f, t_{sw}, t_{cfg}, t_{hw}, n_{slice}, f_{code})$ in a partition P, we use $\lambda(t, P) = (f, u)$ to denote the implementation of the task t (i.e., function f) as a software function if $u = 0$ and as a hardware function if $u = 1$. The computation time $ET(t, P)$, configuration time $CT(t, P)$, and reconfigurable resource requirements $RR(t, P)$ are defined as follows. For a software task, $ET(t, P) = t_{sw}$, $CT(t, P) = 0$, and

$RR(t, P) = 0$, while for a hardware task $ET(t, P) = t_{hw}$, $CT(t, P) = t_{cfg}$, and $RR(t, P) = n_{slice}$.

For each task t in each system partition P, Perfecto accurately evaluates the total *task execution time* ($TET(t, P)$), which is the sum of the computation time ($ET(t, P)$), configuration time ($CT(t, P)$), memory-access time ($MAT(t, P)$), and bus-wait time ($BWT(t, P)$). The first two are as defined earlier, and the last two are obtained through simulation.

For each partition P, Perfecto evaluates five attributes of the partition, called *partition evaluations*, which include the total *partition execution time* (PET) in nanoseconds, the *average DRCL utilization* (ADU %), the *maximum number of DRCL slices used* (MS), the percentage of *average configuration time* (ACT %), and the percentage of *average bus-waiting time* (AWT %). Out of these five attributes, the values of PET and MS depend on the scheduler and the placer chosen in Perfecto, respectively. The other three attributes are defined as follows.

$$ADU(P) = \frac{\sum_t (TET(t, P) \times RR(t, P))}{PET \times MS}, \qquad ACT(P) = \frac{\sum_t CT(t, P)}{\sum_t TET(t, P)},$$

$$AWT(P) = \frac{\sum_t BWT(t, P)}{\sum_t TET(t, P)}$$

The bus-access conflicts show the real-time information of the number of tasks competing for bus access and also the tasks that are actually making requests. From this information, a designer can detect whether there is a bottleneck in system performance. The real-time placement information for each task in each partition can be used for further tuning and optimization.

After simulation, by analyzing the aforementioned results generated by Perfecto, a designer can then decide to select one or more partitions that best fit his/her needs. The criterion could be the least total execution time, least average DRCL utilization, or least average bus-waiting time. All of these results would be more apparent and intuitive through application examples, as described in Section 4.

4. APPLICATION EXAMPLES

We implemented the proposed performance evaluation framework Perfecto in IEEE 1666-2005 SystemC on a Linux Fedora Core-3 workstation with Intel Pentium 4 2.4 GHz CPU and 1GB RAM. Perfecto was applied to several designs. We use a simple example to illustrate the framework and then show its application to a more complex real-world network security example. Note that the partitioning, scheduling, and placement algorithms applied to the examples in this section are all the default ones in Perfecto, so that we can focus on the framework itself.

4.1 Simple Illustration Example

The simple illustration example was introduced in Section 3.2, which has six tasks invoking four functions, with a precedence relation (T5, T3). Note that from the function attributes in Table I, we can conclude that F1 has a software

Table II. Partitions for Simple Illustration Example

	Function Name (f_{name})				Num Tasks	
Partition#	F1 {T1}	F2 {T3, T4}	F3 {T2, T6}	F4 {T5}	SW	HW
P0	0	1	1	1	1	5
P1	0	1	1	0	2	4
P2	0	1	0	1	3	3
P3	0	1	0	0	4	2
P4	0	0	1	1	3	3
P5	0	0	1	0	4	2
P6	0	0	0	1	5	1
P7	0	0	0	0	6	0

0: implement in software; 1: implement in hardware; SW: number of software tasks; HW: number of hardware tasks.

implementation only. Perfecto uses function-based partitioning to generate the partitions for the example as shown in Table II. Partition P0 has the most hardware tasks and P7 is the all-software partition. In the following, we show how Perfecto helps a system designer to make intelligent partition decisions, optimize system performance, and evaluate task placement strategies.

4.1.1 *Making Intelligent Partition Decisions.* All eight hardware-software partitions for this example were evaluated by Perfecto as shown in Table III, which gives the composition of time and the number of slices required by each task. Take partition P0 as an example, whose total execution time is 2033ns, average DRCL utilization 54.63%, the maximum usage of DRCL is 3 slices, average configuration time is 11.96%, and average bus-waiting time is 2.28%. Further, the execution time of task T4 in partition P0 is 752ns, which includes pure execution time (500ns), configuration time (100ns), memory-access time (90ns), and bus waiting time (62ns). From Table III, some useful conclusions can be drawn as follows; these will help a reconfigurable system designer to make intelligent design choices.

(1) The most-hardware partition, P0, requires the least execution time, while all-software partition P7 requires the most. This observation, though intuitively expected, might not be true if the reconfiguration time is very large, as exemplified by the network security system example in Section 4.2.
(2) The execution time for partitions P1 and P2 are quite close; however, P2 uses fewer DRCL slices than P1. If P2 is selected for system design, we can use a smaller DRCL (2 slices instead of 5) for this application. Architecture exploration can thus be performed.
(3) Partition P4 gives a very good trade-off between hardware and software because its total execution time (2623ns) is quite close to that of the most-hardware partition P0 (2033ns). Also, its average bus-waiting time (0.2%) is negligible just like that of the all-software partition P7, which instead has a much higher execution time (7351ns).

Table III. Evaluation by Perfecto of Hardware-Software Partitions for a Simple Example

	PET	*ADU*		*ACT*	*AWT*	$\frac{CT \mid MAT \mid BWT}{\text{Task Execution Time}}$ (ns), where Task Execution Time = $ET + CT + MAT + BWT$																	
P#	(ns)	(%)	*MS*	(%)	(%)	T1			T2			T3			T4			T5			T6		
P0	2033	54.63	3	11.96	2.28	0	240	0	150	20	3	100	90	0	100	90	62	200	60	40	0	20	0
						440			773			690			752			1325			620		
P1	3119	29.54	5	7.27	2.80	0	240	0	150	20	4	0	90	0	100	90	61	0	60	0	150	20	89
						440			774			590			751			2085			859		
P2	3083	17.16	2	4.62	1.23	0	240	0	0	20	0	0	90	0	100	90	51	200	60	29	0	20	0
						440			1320			590			741			2085			1320		
P3	5169	5.19	1	1.54	0.94	0	240	0	0	20	0	0	90	9	100	90	52	0	60	0	0	20	0
						440			1320			599			742			2085			1320		
P4	2623	33.45	5	9.16	0.20	0	240	0	150	20	2	0	90	0	0	90	0	200	60	0	150	20	9
						440			772			1090			1090			1285			779		
P5	4709	13.20	4	4.79	0.22	0	240	0	150	20	3	0	90	0	0	90	0	0	60	0	150	20	11
						440			773			1090			1090			2085			781		
P6	5265	4.88	1	3.06	0.00	0	240	0	0	20	0	0	90	0	0	90	0	200	60	0	0	20	0
						440			1320			1090			1090			1285			1320		
P7	7351	0.00	0	0.00	0.00	0	240	0	0	20	0	0	90	0	0	90	0	0	60	0	0	20	0
						440			1320			1090			1090			2085			1320		

PET: partition execution time; *ADU*: average DRCL utilization;
MS: maximum usage of DRCL slices (out of a totally of 5 slices); *ACT*: average configuration time;
AWT: average bus-waiting time; *ET*: pure execution time $\in \{t_{sw}, t_{hw}\}$; *CT*: configuration time $\in \{0, t_{cfg}\}$;
MAT: memory-access time; *BWT*: bus-wait time.

4.1.2 *Optimizing Performance.* Perfecto not only helps designers in making intelligent hardware-software partitioning choices, but also makes debugging performance bottlenecks easier by providing designers with detailed real-time information on how and when the tasks in each partition compete for bus access. From Table III, we can observe that task T6 in partition P1 has the largest bus-waiting time of 89ns among all tasks in all partitions. Further, this also directly reflects on the average bus-waiting time of partition P1 (2.8ns), which is largest among all the eight partitions. To debug this performance bottleneck, we can analyze the bus-conflict accesses reported by Perfecto. Figure 9 shows the aggregate and individual task bus accesses for partition P1 along the time axis. Analyzing the aggregate diagram, we can immediately identify a performance bottleneck in partition P1, where the maximum number of concurrent bus accesses is 3. Suppose the system designer would like to solve this bottleneck. Through the individual task accesses provided by Perfecto, we can identify the tasks that are competing in this bottleneck of 3 concurrent bus accesses, namely, tasks T1, T3, and T6. We can also exactly pinpoint the time (200ns) during which the three tasks are competing for the bus. These information details help designers to resolve bottlenecks and optimize their designs.

4.1.3 *Evaluating Task Placements.* Besides guiding system designers in making intelligent partitioning decisions and in resolving performance bottlenecks, Perfecto also helps designers in tuning hardware task placement algorithms by providing designers with detailed real-time placement information for each task in the DRCL area. For the simple illustrative example, from Table III, we can observe that partition P1 uses a maximum of 5 DRCL slices, but has quite a low average DRCL utilization of 29.54% only. Thus, we would like to investigate the task placement in P1, which is shown in Figure 10, for a DRCL of 5 slices. The numbers in the top-left corner of each block represent the time when the blocks were placed. For example, task T2 was placed and configured at 6ns, T4 at 8ns, T6 at 10ns, and T3 at 2528ns. The status of a slice indicates whether it is idle, executing, or done. A slice in an idle or done state can be reused for placing newly arrived tasks. As we can observe, there are large periods of time where several slices are unused. Intuitively, we can see that if we delay the execution of T6 to 780ns, we would place it in slices S0 and S1, following T2. Further, we could also delay T4 and place it in S1, following T6. Finally, we would place T3 also in S1, following T4. Thus, only 2 slices are allowable without affecting the total execution time of P1. Note that T3 can start only at 2528ns due to its precedence constraint with T5, a software task in P1. This is an illustration of how placement information can help designers to change the placement algorithm or tune it manually.

4.2 Network Security System Example

Nowadays, a given company has headquarters and several branch offices located at different places. These communicate and transfer data through the Internet. This is dangerous when they transfer data that is not encrypted because the Internet is an open network; therefore, they encrypt data before transfer

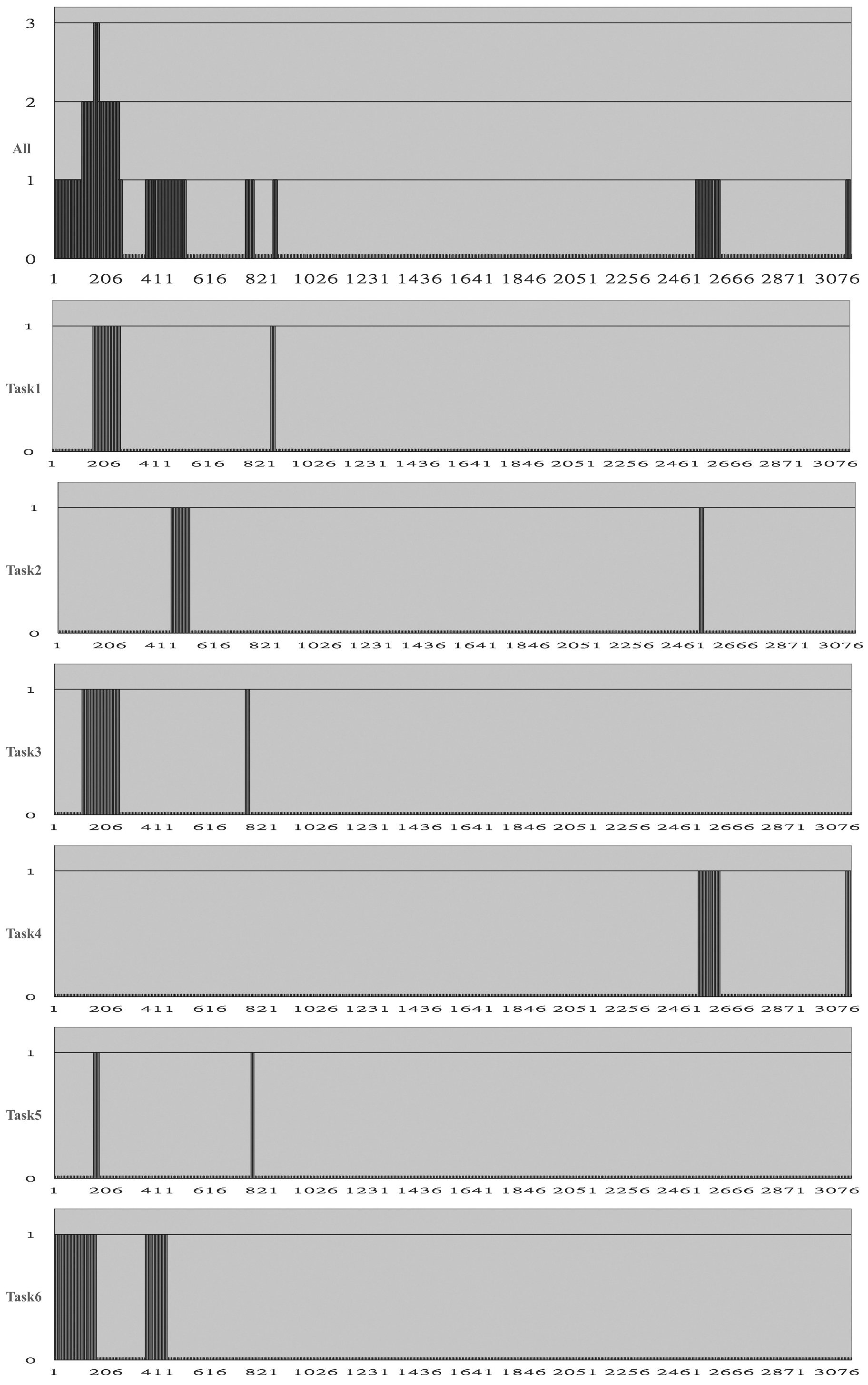

Fig. 9. Bus-access conflicts for partition P1 of the simple example.

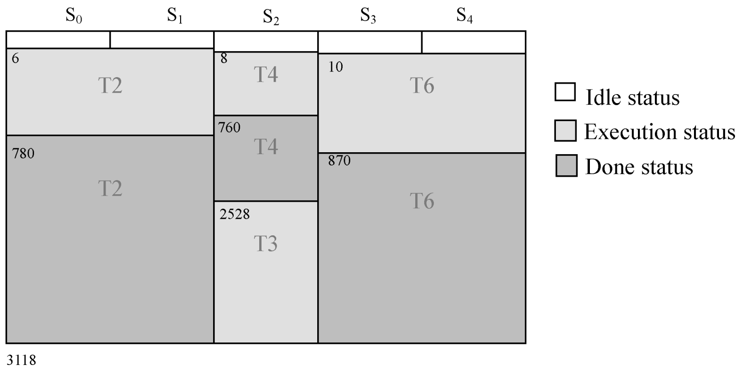

Fig. 10. Hardware task placement for partition P1 of the simple example.

and decrypt after receiving. The network; security system is used when headquarters need to send a ciphertext to some branch office through the Internet. It is also used for decrypting it back into plain text at the branch offices.

Cryptography algorithms can be implemented either in software or in hardware. Table IV compares software and hardware implementations. In the case of software, there is flexibility enough to update the cryptography algorithm, but it may not provide sufficient throughput. Fixed hardware provides good performance, but has low flexibility because devices may need to be changed when the cryptography algorithm is updated. However, if reconfigurable logic is designed into such a security system, it will not only provide high throughput as in the case of hardware, but will also allow good flexibility as in the case of software.

The network security system is represented by a set of 23 concurrent tasks that invoke 5 different encryption functions, namely, MD5, SHA-1, DES, 3-DES, and AES. The number of invocations for each of these functions is given in Table V and the function attributes are specified as shown in Table VI. For example, the DES cryptography algorithm when implemented in software requires 990ns execution time without memory access, and when implemented in hardware requires 63.6μs configuration time, 60ns execution time, and uses 255 slices. Note that the input data-block sizes for the 5 encryption functions are 16, 16, 2, 2, and 4 words, respectively, where a word consists of 4 bytes. The output data sizes for the 5 functions are 4, 5, 2, 2, and 4, respectively.

The architecture parameters for the network security system are as follows. As far as bus width is concerned, we evaluated two versions: 1 word and 2 words. Memory size is unlimited and the memory-access time is 10ns. Since our target FPGA is a Xilinx Virtex-II Pro (XC2VP2-7), the maximum available number of DRCL slices is 1420.

To cope with simulation of large applications, Perfecto employs the following techniques. Since the total number of placed and configured tasks cannot exceed the number of DRCL slices N_{slice}, the size of the task table is fixed at N_{slice}, which allows Perfecto to handle most applications. Further, because the architecture models in Perfecto are transaction-level models, the differences in hardware and software task execution times do affect the efficiency of Perfecto.

Table IV. Comparisons of Security System Implementations

Implementation	Cryptography Algorithm Update	Throughput	Flexibility
Software	Update Software	Low	High
Hardware	Change Device	High	Low
Reconfigurable Logic	Reconfigure with Bitstream	**High**	**High**

Table V. Task Set of Network Security System Example

f_{name}	MD5	SHA-1	DES	3-DES	AES
Number of Invocations	2	2	7	6	6

Table VI. Function Parameters for Network Security System

Function	Algorithm (f_{name})	SW Exec Time $t_{sw}(ns)$	Config Time $t_{cfg}(\mu s)$	HW Exec Time $t_{hw}(ns)$	Slice Count (n_{slice})
F1	MD5	200	119.2	190	478
F2	SHA-1	670	141.7	190	568
F3	DES	990	63.6	60	255
F4	3-DES	1960	190.3	70	763
F5	AES	910	33.2	220	133

It is also assumed that the DRCL size is fixed, irrespective of a change in bus width.

4.2.1 *Making Intelligent Partition Decisions.* Perfecto generates all 32 possible partitions for the network security system as shown in Table VII. For instance, in partition 25, DES and 3-DES are implemented in hardware, while MD5, SHA-1, and AES are implemented in software, which gives a total of 13 hardware and 10 software tasks.

For legibility, in Table VIII, we show the detailed Perfecto simulation results for 8 representative partitions out of the total 32 partitions in the network security system, which include P0, P1, P2, P4, P21, P25, P26, and P31. All functions are implemented in hardware in partition P0. There are four functions that are implemented in hardware and one in software for partitions P1, P2, and P4. There are two functions implemented in hardware and three in software for partitions P21, P25, and P26. All functions are implemented in software in partition P31.

From Table VIII, the following conclusions can be drawn.

(1) Conventionally, for nonreconfigurable system designs a full-hardware system implementation usually has the shortest execution time compared to a hardware-software system. However, the all-hardware partition P0 of the network security system does not have the shortest execution time. Compared with the total execution time of 944.6μs for P0 with 1-word bus width, there are at least two other partitions, P1 and P25, that have shorter execution times, that is, 637.9μs and 504.9μs, respectively. The reason is that

Table VII. Hardware-Software Partitions of Network Security System

P #	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
MD5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
SHA-1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
DES	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
3-DES	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
AES	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
P #	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
MD5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SHA-1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
DES	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
3-DES	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
AES	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

0: Implement in Software, 1: Implement in Hardware

the reconfiguration time overhead becomes a bottleneck in the all-hardware partition.

(2) The average DRCL utilizations for tasks in partitions P2 and P21, that is, 10.55% and 12.29%, respectively, are quite close, but their total execution times of 2992μs and 1718μs differ significantly. The reason is that partition P2 requires a much larger average configuration time compared to partition P21. The difference is at least 20% for both bus-width architectures.

(3) All three of the partitions P1, P2, and P4 have 4 functions in hardware and 1 in software. However, their total execution times differ significantly. For example, they are 637.9μs, 2992.1μs, and 1351.4μs, respectively, for a 1-word bus width. The reason is that the partitions have different average DRCL utilizations of 64.95%, 10.55%, and 28.59%, respectively, depending on the different sizes of cryptography function when implemented in hardware. We observe that a larger average DRCL utilization, such as 64.95% by partition P1, gives a shorter execution time, specifically 637.9μs.

(4) Comparing the two different architectures with 1- and 2-word bus widths, we can observe that the execution time for each partition of the architecture with 1-word bus width is larger than the corresponding partition of the architecture with 2-word bus width. The reason is that a smaller bus width results in an increase in memory-access time, which is included in the total execution time.

If total execution time is the sole criterion for selecting the best hardware-software partition, then the partition P25 gives the smallest execution time (504.9μs) and will thus be selected as the best choice for further implementation of the network security system. However, from Table VIII, we can see that

Table VIII. Summary Results of Applications with Different Bus Widths in Network Security System Example

	Bus Width = 1 word					Bus Width = 2 words				
P#	Partition Execution Time (μs)	Avg. DRCL Utilization (%)	Max Slice Usage (*Max_S*)	Avg. Config. Time (%)	Avg. Bus Waiting Time (%)	Partition Execution Time (μs)	Avg. DRCL Utilization (%)	Max Slice Usage (*Max_S*)	Avg. Config. Time (%)	Avg. Bus Waiting Time (%)
0	944.6	57.58	1406	88.44	0.71	905.2	56.97	1344	93.91	0.23
1	637.9	64.95	1331	82.17	2.32	575.6	66.65	1331	90.32	0.80
2	2992.1	10.55	1344	84.23	1.36	2962.1	7.37	1344	87.49	1.60
4	1351.4	28.59	1331	81.40	1.95	1325.2	26.91	1331	89.81	0.50
21	1718.2	12.29	1331	64.18	1.60	1672.6	11.32	1331	77.17	0.68
25	504.9	45.94	1273	72.21	6.14	459.9	43.68	1273	84.38	2.38
26	3016.0	3.06	1408	71.40	3.95	2975.0	2.77	1408	83.86	0.42
31	4723.8	0.00	0	0.00	0.00	4642.1	0.00	0	0.00	0.00

Max_S: Maximum Usage of DRCL Slices (out of totally 1420 slices).

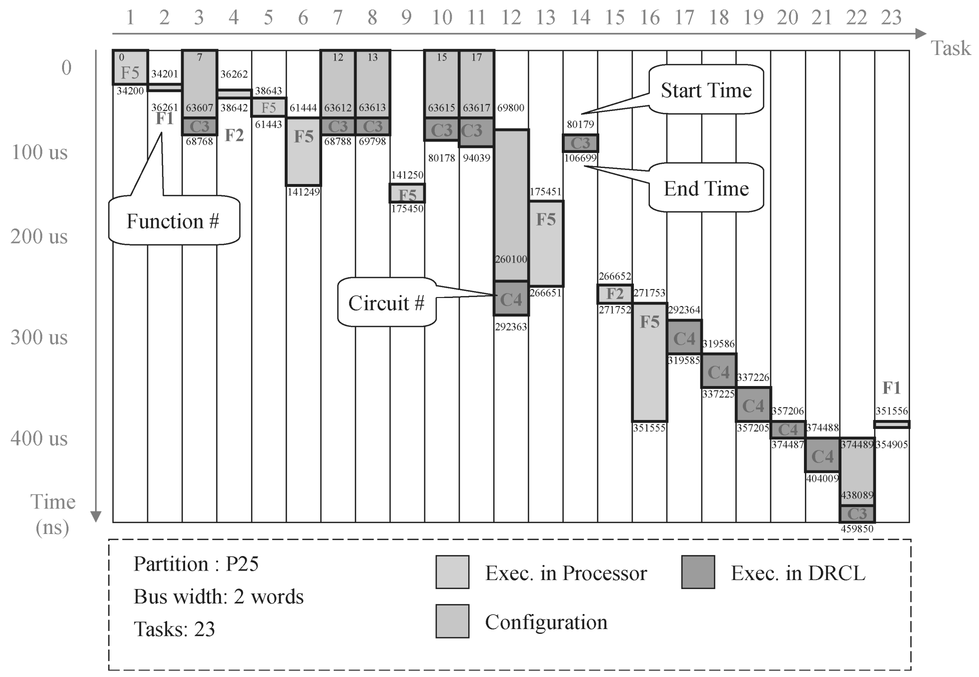

Fig. 11. Execution status for partition P25 of the example network security system.

the average bus-waiting time for tasks in P25 is 6.14% of its execution time, which is quite high compared to that of the other partitions. If the task-response time has a higher priority compared to other design criteria, then P25 might not be the best choice. Nevertheless, Perfecto can still be used to analyze the exact execution status of P25, which is as shown in Figure 11 for an architecture with bus width of 2 words. In this figure, the x-axis represents the task number and the y-axis represents time progress. The function label F? associated with a task, such as F5 in the region of task T1, means that task T1 invokes function F5, namely, the AES algorithm implemented in software. Further, the blocks in the region of task T12 indicate that the task is implemented in hardware with circuit C4, namely, the 3-DES algorithm, whose hardware configuration takes place in the time period from 69800ns to 260100ns, and whose execution is in the time period from 260100ns to 293263ns. Using Figure 11, as generated by Perfecto, a system designer can understand the comparative execution status of each task to decide whether partition P25 is the best choice for further implementation.

4.2.2 *Optimizing Performance.* A portion of the aggregate bus accesses by the 23 tasks in partition P25 of the network security system is illustrated in Figure 12. Referring to this report, we can easily locate at least two system bottlenecks, at 63677ns and 65114ns, where the number of concurrent bus requests reaches a maximum of 5. Similar to the simple illustration example, the designer can then pinpoint the exact tasks that are contributing to these

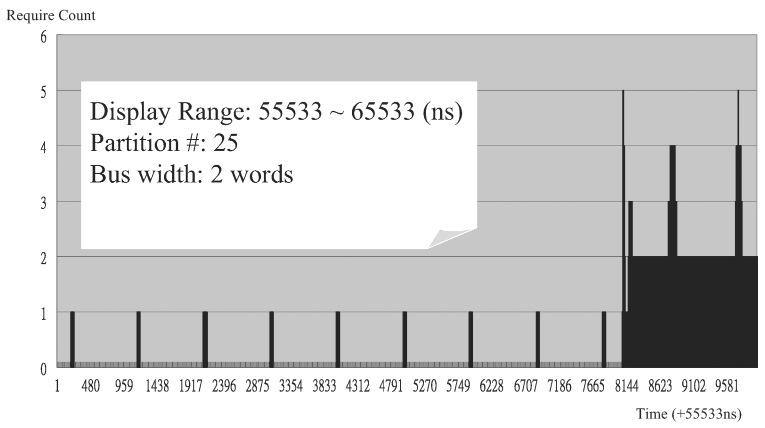

Fig. 12. Bus-access conflicts for partition P25 of the network security system.

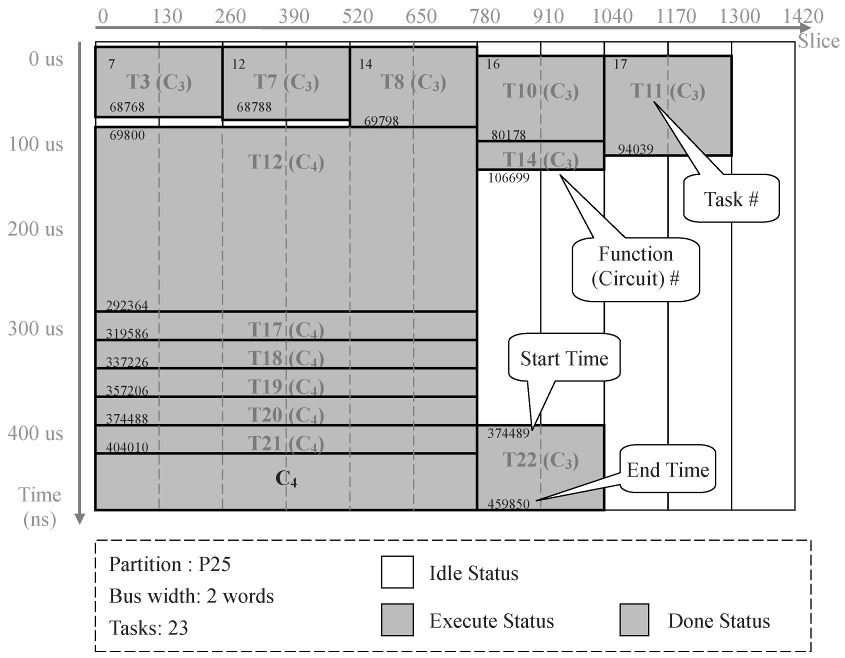

Fig. 13. Placement diagram of DRCL in the network security system example.

bottlenecks and try to eliminate the latter by modifying the tasks themselves, or by changing the schedules or partitions.

4.2.3 *Evaluating Task Placements.* To explore how the tasks are placed in a partition, Perfecto allows designers to study the detailed real-time placement of each task in a reconfigurable system. For partition P25 of the network security system, the task placements in a DRCL (Xilinx Virtex-II Pro XC2VP2-7) of 1420 slices are illustrated in Figure 13, where the x-axis represents the slices and the y-axis represents the time progress. This diagram can help a designer to study the task placements in DRCL and also to manually alter some, if desired. For example, we can easily observe that for partition P25, the maximum slice count is 1273; however, the resource utilization is not high for the slices numbered from 780 to 1273. If a smaller DRCL is required, we can manually move and reschedule task T11 that invokes function F3 (DES implemented as hardware circuit C3) to a later slot, after 106699ns and before 374489ns. The size of the DRCL can thus be reduced to $1273 - 255 = 1018$ slices. Further, since tasks T10, T14, and T22 all invoke the same function F3 (DES implemented as hardware circuit C3), if T11 is inserted between T14 and T22, we can also reuse the hardware configuration in slices 780 to 1035 and thus save hardware reconfiguration time and power. In summary, we can say that the placement diagram allows a designer to fine-tune his/her design schedules and placements such that a more optimal system can be constructed in terms of smaller DRCL sizes, shorter total execution time, higher resource utilization, and lesser power consumption through fewer hardware reconfigurations.

5. CONCLUSIONS

A SystemC-based performance evaluation framework called Perfecto for dynamically, partially reconfigurable systems was proposed and developed. Perfecto can be used for design-space exploration, as shown in the application examples, and also for designing or evaluating hardware task scheduling and placement algorithms that have to be integrated in an operating system for reconfigurable systems. Perfecto not only takes advantage of SystemC simulation capabilities, but also generates very detailed reports on the performance of different partitions. The partitioning, scheduling, and placement algorithms in Perfecto can be replaced by a user to suit his/her needs. This is a step towards filling in the gap that keeps widening between rapidly advancing reconfigurable technologies and slowly improving reconfigurable design technologies, including methods, tools, and environments.

In the future, we plan to improve on the interfaces of Perfecto for hooking in user-designed partitioning, scheduling, and placement algorithms. Further, we also plan to extend Perfecto by integrating it into the UML/SystemC-based design flow for dynamically reconfigurable systems [Tseng and Hsiung 2005]. Perfecto will act as a design-space exploration and performance evaluation tool in the design flow. Finally, we plan to apply Perfecto to other application domains.

REFERENCES

Baleani, M., Gennari, F., Jiang, Y., Patel, Y., Brayton, R., and Sangiovanni-Vincentelli, A. 2002. Hardware-Software partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES)*. ACM Press, New York, 151–156.

Chang, D. and Marek-Sadowska, M. 1998. Partitioning sequential circuits on dynamically reconfigurable FPGAs. In *Proceedings of the 6th International Symposium on FPGAs*. ACM Press, New York, 161–167.

CHIANG, C.-C. 2007. Hardware/Software real-time relocatable task scheduling and placement in dynamically partial reconfigurable systems. M.S. thesis, National Chung Cheng University, Chiayi, Taiwan.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv. 34,* 2 (Jun.), 171–210.

DESMET, D., AVASARE, P., COENE, P., DECNEUT, S., HENDRICKX, F., MARESCAUX, T., MIGNOLET, J.-Y., PASKO, R., SCHAUMONT, P., AND VERKEST, D. 2002. Design of Cam-E-leon, a run-time reconfigurable Web camera. In *Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS).* Lecture Notes in Computer Science, vol. 2268. Springer, 274–290.

DICK, R. P., RHODES, D. L., AND WOLF, W. 1998. TGFF: Task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (CODES)*. IEEE Press, 97–101.

HSIUNG, P.-A., HUANG, C.-H., AND CHEN, Y.-H. 2008. Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC. *J. Embed. Comput.* (to appear).

HSIUNG, P.-A. AND LIU, C.-W. 2007. Exploiting hardware and software low power techniques for energy efficient co-scheduling in dynamically reconfigurable systems. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE Computer Society Press, 165–170.

LIAO, H.-W. 2007. Multi-Objective placement of reconfigurable hardware tasks in real-time systems. M.S. thesis, National Chung Cheng University, Chiayi, Taiwan.

LIU, C.-W. 2006. Energy efficient hardware/software co-scheduling in reconfigurable systems. M.S. thesis, National Chung Cheng University, Chiayi, Taiwan.

LOO, S. AND WELLS, B. 2006. Task scheduling in a finite resource reconfigurable hardware/software co-design environment. *INFORMS J. Comput. 18,* 12 (Spring), 151–172.

MEI, B., SCHAUMONT, P., AND VERNALDE, S. 2000. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of the 11th ProRISC Workshop on Circuits, Systems and Signal Processing Veldhoven*.

NOGUERA, J. AND BADIA, R. 2003. System-Level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*. ACM Press, New York, 73–83.

(OSCI), O. S. I. 2008. *SystemC User's Guide*. http://www.systemc.org/.

PELKONEN, A., MASSELOS, K., AND CUPÁK, M. 2003. System-Level modeling of dynamically reconfigurable hardware with SystemC. In *Proceedings of the 10th Reconfigurable Architectures Workshop, International Parallel and Distributed Processing Symposium*, 174–181.

QU, Y., TIENSYRJÄ, K., AND MASSELOS, K. 2004. System-Level modeling of dynamically reconfigurable co-processors. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application (FPL).* Lecture Notes in Computer Science, vol. 3203. Springer, 881–885.

RAKHMATOV, D. AND VRUDHULA, S. 2002. Hardware-Software bipartitioning for dynamically reconfigurable system. In *Proceedings of the 10th International Workshop on Hardware-Software Codesign (CODES)*, 145–150.

RESANO, J. AND MOZOS, D. 2004. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In *Proceedings of the 41th Annual Design Automation Conference (DAC)*. ACM Press, New York, 119–124.

RISSA, T., DONLIN, A., AND LUK, W. 2005. Evaluation of SystemC modelling of reconfigurable embedded systems. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, vol. 3, 253–258.

SANTAMBROGIO, M. 2008. Hardware-Software codesign methodologies for dynamically reconfigurable systems. Ph.D. thesis, Politecnico Di Milano, Italy.

STEIGER, C., WALDER, H., AND PLATZNER, M. 2004. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput. 53,* 11 (Nov.), 1393–1407.

TIENSYRJÄ, K., QU, Y., ZHANG, Y., CUPAK, M., RYNDERS, L., VANMEERBEECK, G., MASSELOS, K., POTAMIANOS, K., AND PETTISSALO, M. 2004. SystemC and OCAPI-xl based system-level design for reconfigurable systems-on-chip. In *Proceedings of the International Forum on Specification and Design Languages (FDL)*, 428–429.

TRIMBERGER, S. 1998. Scheduling designs into a time-multiplexed FPGA. In *Proceedings of the 6th International Symposium on FPGAs*. ACM Press, New York, 153–160.

TSENG, C.-C. AND HSIUNG, P.-A. 2005. UML-Based design flow and partitioning methodology for dynamically reconfigurable computing systems. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*. Lecture Notes in Computer Science, vol. 3824. Springer, 479–488.

Received March 2008; revised May 2008; accepted July 2008