Hardware/Software Partitioning and Pipelined Scheduling on Runtime Reconfigurable FPGAs

MINGXUAN YUAN, ZONGHUA GU, and XIUQIANG HE
Zhejiang University and Hong Kong University of Science and Technology
XUE LIU
McGill University
and
LEI JIANG
University of Pittsburgh

13

FPGAs are widely used in today's embedded systems design due to their low cost, high performance, and reconfigurability. Partially RunTime-Reconfigurable (PRTR) FPGAs, such as Virtex-2 Pro and Virtex-4 from Xilinx, allow part of the FPGA area to be reconfigured while the remainder continues to operate without interruption, so that HW tasks can be placed and removed dynamically at runtime. We address two problems related to HW task scheduling on PRTR FPGAs: (1) HW/SW partitioning. Given an application in the form of a task graph with known execution times on the HW (FPGA) and SW (CPU), and known area sizes on the FPGA, find an valid allocation of tasks to either HW or SW and a static schedule with the optimization objective of minimizing the total schedule length (makespan). (2) Pipelined scheduling. Given an input task graph, construct a pipelined schedule on a PRTR FPGA with the goal of maximizing system throughput while meeting a given end-to-end deadline. Both problems are NP-hard. Satisfiability Modulo Theories (SMT) is an extension to SAT by adding the ability to handle arithmetic and other decidable theories. We use the SMT solver Yices with Linear Integer Arithmetic (LIA) theory as the optimization engine for solving the two scheduling problems. In addition, we present an efficient heuristic algorithm based on kernel recognition for the pipelined scheduling problem, a technique borrowed from SW pipelining, to overcome the scalability problem of the SMT-based optimal solution technique.

This work was partially supported by Hong Kong RGC GRF Grants No. 613506 and No. 613108, National Key Technology R&D Program of China No. 2006BAH02A01, National Important Science & Technology Specific Projects Grant No. 2009ZX01038-001 and 2009ZX01038-002, Canada NSERC Strategic Grant STPGP 364910-08, and FQRNT 2010-NC-131844.
Authors' addresses: M. Yuan, Z. Gu (corresponding author), X. He, Zhejiang University, 388 Yuhangtang Road, Hangzhou, Zhejiang Province, 310058 P. R. China; email: zonghua@gmail.com; X. Liu, School of Computer Science, McGill University, Montreal, Quebec, Canada; L. Jiang, University of Pittsburgh, Pittsburgh, PA 15260.

DOI 10.1145/1698759.1698763 http://doi.acm.org/10.1145/1698759.1698763

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: HW/SW partitioning, scheduling, runtime reconfigurable FPGA

ACM Reference Format:

Yuan, M., Gu, Z., He, X., Liu, X., and Jiang, L. 2010. Hardware/Software partitioning and pipelined scheduling on runtime reconfigurable FPGAs. ACM Trans. Des. Autom. Electron. Syst. 15, 2, Article 13 (February 2010), 41 pages.

DOI = 10.1145/1698759.1698763 http://doi.acm.org/10.1145/1698759.1698763

1. INTRODUCTION

Reconfigurable HW devices, such as FPGAs, are very popular in today's embedded systems design due to their low cost, high performance, and reconfigurability. FPGAs are inherently parallel, that is, two or more tasks can execute on a FPGA device concurrently as long as they can both fit on it. *Partially RunTime-Reconfigurable* (PRTR) FPGAs allow part of the FPGA area to be reconfigured while the remainder continues to operate without interruption, so that HW tasks can be placed and removed dynamically at runtime. Early versions of Xilinx FPGA devices, such as Virtex-II and Spartan, only support 1D reconfiguration, where each task occupies a contiguous set of columns. In 2006, Xilinx introduced the Early-Access Partial Reconfiguration flow (EAPR) to permit 2D reconfiguration. Virtex-4 and Virtex-V devices support independent reconfiguration of a minimum of 16 *Configurable Logic Blocks* (CLBs) in the same column, making it possible to have 2D dynamic reconfiguration [Xilinx 2005], where each task occupies a rectangular area with a width and height in terms of number of *Configurable Logic Blocks* (CLBs), also referred to as *cells*, on the two dimensions. Even though 2D reconfiguration is feasible with today's FPGA technology, there are still many technical obstacles to its wide adoption in industry, hence we limit our attention to 1D reconfigurable FPGAs in this article. As an example, Figure 1 shows a 1D reconfigurable FPGA with 5 columns C_1 to C_5, and two tasks on it: T_1 occupies 2 columns and T_2 occupies 1 column.

Task scheduling for 1D reconfigurable FPGAs bears some similarity to scheduling on identical multiprocessors [Carpenter et al. 2004], where all processors in the system have identical processing speed. The difference is that each HW task may occupy multiple contiguous columns on the FPGA while a SW task always occupies a single CPU at any given time. For example, consider the problem of scheduling a task graph on a FPGA with 3 columns. If each task occupies 3 columns, then it is identical to a single-processor scheduling problem, as the entire FPGA can be viewed as a single CPU. If each task occupies 1 column, then it is similar to a multiprocessor scheduling problem (except for the constraints imposed by a single configuration controller, as discussed later), as each FPGA column can be viewed as a CPU. But if each task can occupy multiple columns, then it becomes a more general and difficult problem than multiprocessor scheduling.

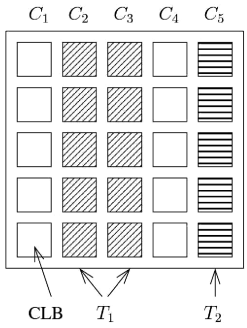


Fig. 1. A 1D reconfigurable FPGA with two HW tasks running.

HW task reconfiguration on a FPGA has some unique characteristics compared to SW task context-switch on the CPU: large reconfiguration delay and unique reconfiguration controller as a global shared resource.

—Unlike CPU scheduling, where task context-switch overhead is often small enough to be ignored, FPGA reconfiguration carries a significant overhead in the range of milliseconds which is proportional to the size of area being reconfigured. Several techniques have been proposed to reduce the impact of reconfiguration delay. Notably, *configuration prefetch* [Li and Hauck 2002] is an effective technique for hiding the large reconfiguration delay by allowing a task's configuration to be loaded on the FPGA sometime before the start of its actual computation. As a result, a task's reconfiguration and execution stages may be separated by a time gap. Each task invocation consists of two distinct stages: *reconfiguration* and *execution*. This can be useful for reducing or eliminating impact of reconfiguration delays by overlapping one task's reconfiguration stage with some other task's execution stage.

—The typical commercial FPGA has a single reconfiguration controller, so the reconfiguration stages of different tasks must be serialized on the timeline, while their execution stages can be concurrent as long as they occupy different areas on the FPGA. This is a major source of complexity of real-time scheduling on FPGAs. (Although there are research prototypes that support multiple reconfiguration controllers [Noguera and Badia 2006], we do not consider them in this article.)

In this article, we consider applications described as task graphs. A *task graph* is a directed acyclic graph where each vertex represents a task, and each edge represents precedence relationship between two tasks. $task_i$ has a tuple of attributes (E_i, R_i, W_i), where E_i is its computation time, R_i is its reconfiguration delay, and W_i is its width in terms of the number of contiguous FPGA columns it occupies. A task's reconfiguration delay R_i is proportional to its width W_i. Each task invocation consists of a reconfiguration stage with length R_i followed by an execution stage with length E_i. A task's execution stage can start when all its predecessors have finished their execution stages, that is, precedence relationships in the task graph constrain task execution stages, not reconfiguration stages, which can occur in arbitrary order.

In this article, we address the following two problems.

—*HW/SW Partitioning and Scheduling on a Hybrid FPGA/CPU Device*. Given an input task graph, where each task has two equivalent implementations: a HW implementation that can run on the FPGA, and a SW implementation that can run on the CPU, partition and schedule it on a platform consisting of a PRTR FPGA and a CPU, with the objective of minimizing the schedule length (makespan). The CPU may be a *softcore CPU* carved out of the configurable logic in the FPGA, for example, MicroBlaze for Xilinx FPGAs, or a separate CPU connected to the FPGA via a bus. We present a Satisfiability Modulo Theories (SMT) encoding for finding the optimal solution, and compare its performance with a heuristic algorithm in the literature [Banerjee et al. 2005, 2006].

—*Pipelined Scheduling on a PRTR FPGA*. Given an input task graph, construct a pipelined schedule on a PRTR FPGA with the objective of maximizing system throughput while meeting a given task graph deadline.[1] The deadline constraint is imposed since certain applications may be latency sensitive as well as throughput oriented, hence maximizing throughput without regard to latency may negatively impact application QoS. We present a SMT encoding for finding the optimal solution, as well as a heuristic algorithm based on Kernel Recognition [Aiken et al. 1995].

These two problems are found in different application contexts: the first problem is useful when different task graph iterations do not overlap with each other, perhaps the task graph is executed only once, or periodically driven by a periodic timer; the second problem is useful when the task graph is iterated many times. The objective is to maximize system throughput, hence finish the "batch job" of all task graph iterations as soon as possible by overlapping execution of different task graph iterations. This article is an extended version of our previous conference paper [Yuan et al. 2008], which addressed the problem of HW/SW partitioning and scheduling.

We make the following assumptions.

—The FPGA is 1D reconfigurable with a single reconfiguration controller, as supported by commercial FPGA technology.

—Each HW or SW task has a known worst-case execution time. Each HW task has a known size in terms of the number of contiguous columns it occupies on the FPGA.

—Tasks are not preemptable on either the FPGA or the CPU. This is a common assumption adopted in the literature on HW/SW partitioning.

—The communication delay between a HW task on the FPGA and a SW task on the CPU is proportional to the data size transmitted. (We make the simplifying assumption that each edge in the task graph represents the same

[1]It is also possible to formulate the pipelined scheduling problem for a hybrid CPU/FPGA device, where the CPU is used to execute one or more pipeline stages. However, this approach has the drawback that the CPU is typically very slow, and the pipeline throughput is limited by the slowest pipeline stage, so we do not consider it in this article.

communication data size, hence the communication delay is the same for all task graph edges that cross the FPGA/CPU boundary. Our framework can be easily extended to handle the more general case.) Intertask communication between two HW tasks on the FPGA or between two SW tasks on the CPU takes place via shared memory and takes no time.

—The entire FPGA area is uniformly reconfigurable, and each dynamic task can be flexibly placed anywhere on the FPGA as long as there is enough empty space. In practice, it is common to preconfigure some FPGA columns for dedicated purposes as static components where dynamic tasks cannot be placed. This situation can be handled by denoting these columns as always occupied. In particular, it is common to allocate an area of shared memory that spans the entire width of the FPGA and acts as a global shared communication medium, which reduces the height of the FPGA columns that are available to dynamic tasks but does not affect our task model otherwise.

This article is structured as follows: we present the SMT model for HW/SW partitioning on a hybrid CPU/FPGA device in Section 2; the SMT model and a heuristic algorithm based on kernel recognition for HW task pipelined scheduling in Section 3; related work in Section 4; performance evaluation results in Section 5; conclusions in Section 6.

2. HW/SW PARTITIONING

As motivation for HW/SW partitioning using FPGA as the coprocessor, FPGAs may be used in place of ASICs as HW coprocessors, also called HW accelerators, for computation-intensive kernels in the application. As HW coprocessors, FPGAs have a number of advantages over ASICs.

—FPGAs are more cost effective than ASICs in terms of ease of design and reduced time-to-market.

—FPGAs are more flexible than ASICs. Whereas ASIC coprocessors accelerate specific functions, coprocessors based on FPGAs can be applied to the speedup of arbitrary SW programs with some distinctive characteristics, for example, programs with parallelizable bit-level operations.

—Dynamically reconfigurable FPGAs may be used to achieve further cost effectiveness and flexibility by reconfiguring the FPGA to run different acceleration tasks at different times, thus avoiding the need for multiple ASIC coprocessors.

There are some commercial products that support using of FPGA-based coprocessors as HW accelerators, for example, Altera developed HW and design tools for a flexible coprocessor architecture. Figure 2 shows one possible configuration using Altera's FPGA coprocessor with Texas Instruments' digital signal processor, where the coprocessor is Direct Memory Access (DMA)-driven via the TI External Memory InterFace (EMIF), and the data is buffered using First-In First-Out (FIFO) buffers. It is also possible to build a System on a Programmable Chip (SoPC) with the NIOS embedded processor core instead

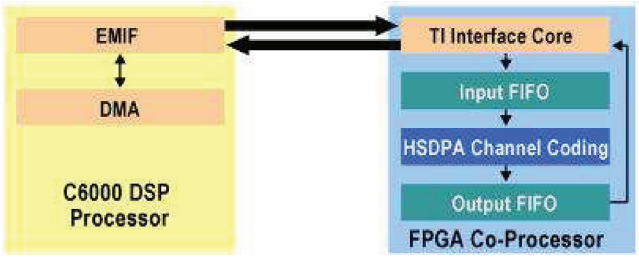
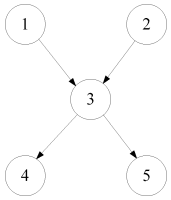


Fig. 2. One example configuration of Altera FPGA coprocessor, taken from Altera's Web site (www.altera.com).

Table I. Task Graph Example for HW Task Scheduling on FPGA



ID	W_i	R_i	E_i
1	1	1	3
2	2	2	1
3	1	1	2
4	1	1	1
5	2	2	2

of an external processor. Combined with PRTR feature of modern FPGAs, this provides a flexible and high-performance approach to HW acceleration.

2.1 Motivating Examples

We first present an example of HW task scheduling on FPGA, then present another example of HW/SW partitioning and scheduling on a hybrid CPU/FPGA device.

Consider the problem of scheduling the task graph in Table I on a FPGA with 3 columns. Figure 3 shows some possible schedules. Dark areas denote the reconfiguration stage; white areas denote the execution stage; striped areas denote the gap between a task's reconfiguration stage and execution stage, within which no other task can execute. After a $task_i$'s columns are reconfigured and before $task_i$ finishes execution, they are reserved for $task_i$ and should not be allocated to other tasks, as indicated by the shaded areas. However, $task_i$ may not be able to start its execution stage immediately since it has to wait for all its predecessors to finish. For example, in schedule (d), reconfiguration of column C_3 for $task_3$ has finished at time 2, but $task_3$'s execution stage E_3 cannot start until time 3 when $task_1$'s execution stage E_1 finishes due to task graph precedence constraints.

Even though only one task graph iteration is shown, the task graph can be executed repeatedly and periodically in an actual system. Figure 3 shows that the task graph is executed with a period of 13. The source tasks $task_1$ and $task_2$ that start at time 0 are preconfigured at the end of the previous period and before the start of the current period, so their reconfiguration delays do not contribute to the schedule length. This implies that there is enough slack time at the end of each period to preconfigure certain source tasks for the next period.

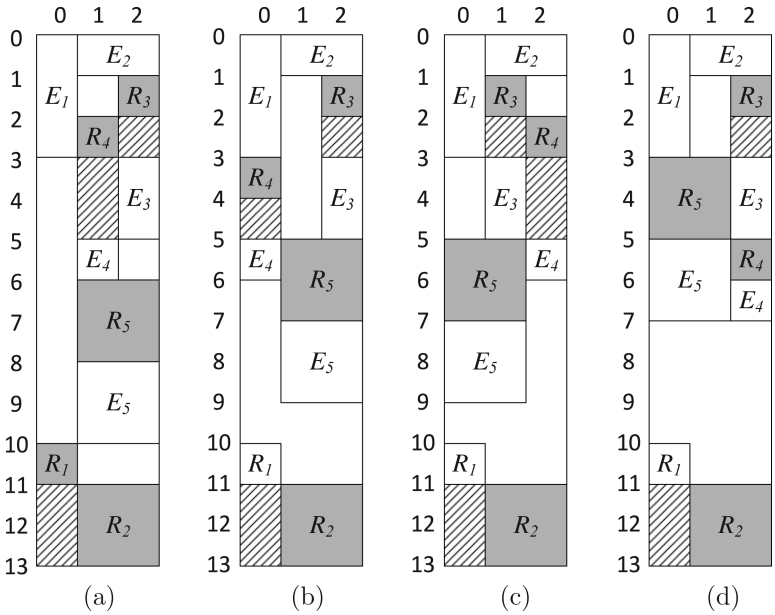


Fig. 3. Some possible schedules of the task graph in Table I on a FPGA with 3 columns. Vertical axis denotes time; horizontal axis denotes the column position on the FPGA.

For example, for the schedule in Figure 3(a), the task graph period should be larger than or equal to 13 ($SL+R_1+R_2 = 10+3$ in order to leave enough slack at the end of each period for reconfiguration of tasks T_1 and T_2 for the next period, while for the schedule in Figure 3(d), the task graph period must be larger than or equal to 10 ($SL+R_1+R_2 = 7+3$). This approach results in reduced task graph latency compared to the approach where all task configurations are performed at the beginning. This should be beneficial to the application QoS, similar to a common approach to implementation of control loops, where time-consuming state updates are performed after computing controller output [Klein et al. 1993].

Figure 4 shows one possible HW/SW partition and schedule for the task graph in Table II, assuming task graph period is 11. Each task has an additional attribute SE_i to denote its execution time on the CPU for its SW implementation. $task_6$ is assigned to the CPU and the other tasks are assigned to the FPGA. The box labeled "com" represents the communication delay between the CPU and FPGA when SW $task_6$ on the CPU sends a message to HW $task_5$ on the FPGA.

2.2 SMT Model for HW/SW Partitioning

Scheduling problems are typically NP-hard, and efficient heuristic algorithms are often devised to obtain near-optimal solutions. Researchers have also used various techniques to obtain exact solutions to these NP-hard problems, including Integer Linear Programming (ILP) solvers, Constraint Programming (CP) solvers, Binary Decision Diagram (BDD) packages, Satisfiability (SAT) solvers,

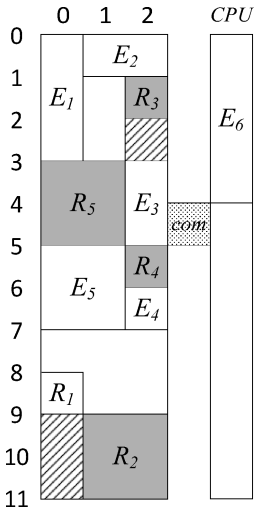
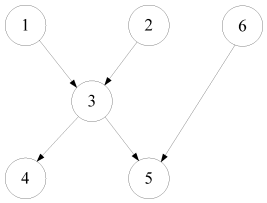


Fig. 4. One possible schedule of the task graph in Table II on a hybrid FPGA/CPU device.

Table II. Task Graph Example for HW/SW Partitioning on a Hybrid CPU/FPGA Device



ID	W_i	R_i	E_i	SE_i
1	1	1	3	12
2	2	2	1	4
3	1	1	2	8
4	1	1	1	4
5	2	2	2	8
6	2	2	1	4

model-checkers, etc. In particular, SAT is a well-known NP-complete problem of assigning values to a set of boolean variables to make a propositional logic formula true. The formula is typically written in Conjunctive Normal Form (CNF) consisting of a conjunction of boolean disjunctions. SAT solvers have become amazingly fast in recent years, and a good SAT solver can routinely handle up to 10^{300} states. SAT can encode bounded integers with bit vectors, but it cannot encode unbounded types such as real variables, or infinite structures such as queues or linked lists. Even for bounded variables, the number of variables can be very large, and SAT solving can be very slow if there are a large number of variables. Since the number of boolean variables needed to encode integer variables grows large quickly for large integer values, SAT is not very suitable for optimization problems involving large integer values.

Satisfiability Modulo Theories (SMT) is an extension to SAT by adding the ability to handle arithmetic and other decidable theories, such as equality with uninterpreted function symbols, linear integer arithmetic, linear real arithmetic, integer difference logic, and real difference logic. Early attempts at solving SMT problem instances involved translating them to boolean SAT instances

(e.g., a 32-bit integer variable would be encoded by 32 boolean variables, and word-level operations such as addition would be replaced by lower-level boolean operations) and passing this formula to a boolean SAT solver. This allows us to use existing SAT solvers and leverage their performance and capacity improvements over time. On the other hand, the loss of high-level semantics means that the SAT solver has to work a lot harder than necessary to discover obvious facts such as $x + y = y + x$ for integer addition. This observation led to the development of a number of SMT solvers that tightly integrate the boolean reasoning of a Davis, Putnam, Logemann, and Loveland (DPLL)-style search [Davis et al. 1962] with theory solvers that handle conjunctions of predicates from a given theory. This architecture, called *DPLL(T)*, gives the responsibility of boolean reasoning to the DPLL-based SAT solver which, in turn, interacts with a solver for theory T through a well-defined interface. The theory solver checks the feasibility of conjunctions of theory predicates passed onto it from the SAT solver as it explores the boolean search space of the formula. Different SMT solvers may use different theory solvers and different techniques of integrating them within the DPLL(T) framework. Integration of SAT with a theory solver results in dramatic performance improvements for problems that can be expressed directly in the theory solver compared to developing a boolean encoding and using a SAT solver only. In this article, we use the SMT solver Yices [Dutertre and de Moura 2006] from Stanford Research Institute (SRI), as an alternative to more conventional optimization techniques such as ILP. We can view all time-related variables as either integer or real variables. The SMT solver Yices provides the *Linear Integer Arithmetic* (LIA) theory or the *Linear Real Arithmetic* (LRA) theory, so either option is acceptable. The solutions generated with either the LIA or LRA theory are the same, but their runtime efficiency may be different. Our experience shows that SMT with LIA theory exhibits better performance for most examples, so we declare all variables to be integers in this article.

Since SMT does not support optimization directly, but only provides a yes/no answer to the feasibility of a given constraint set, we use a binary search algorithm at the top level to search for the shortest schedule length *SL*, as shown in Algorithm 1. The SMT solver is invoked as a subroutine to check the feasibility of each possible schedule length *SL*. *LB* and *UB* denote the minimum and maximum possible values of the schedule length, respectively. To ensure that *LB* is a safe lower bound, we set *LB* to be one less than the length of the *critical path* in the task graph, that is, the longest delay path from the source task's task to the sink tasks assuming zero reconfiguration delays. To ensure that *UB* is a safe upper bound, we set *UB* to be sum of the execution times of all tasks and their reconfiguration delays. "GenSMTModel" refers to the code generator that takes as input the schedule length *SL* to be checked for feasibility, and generates the SMT model as input to Yices.

Next, we present the SMT model for HW/SW partitioning on a hybrid FPGA/CPU device for a given upper bound on schedule length *SL* in Conditions 1 to 10. Table III summarizes the notations used in this section. We follow the convention of using lower-case letters to denote variables, and upper-case letters to denote constants.

Table III. Notations Used in This Section

M	size of the reconfigurable area on the FPGA in terms of number of contiguous columns
N	number of tasks in the task graph
R_i	$task_i$'s reconfiguration delay on the FPGA
E_i	$task_i$'s execution time on the FPGA
W_i	$task_i$'s size in terms of the number of contiguous FPGA columns it occupies
SE_i	$task_i$'s execution time on the CPU for its SW implementation
SL	upper bound of the schedule length
$Period$	period of the task graph, assuming the task graph is executed periodically
r_i	actual reconfiguration delay of $task_i$
pl_i	position of task i's leftmost column on the FPGA
pr_i	position of task i's rightmost column on the FPGA
tr_i	start time of task i's reconfiguration stage
fr_i	finish time of $task_i$'s reconfiguration stage
te_i	start time of task i's execution stage
fe_i	finish time of $task_i$'s execution stage
hs_i	boolean variable denoting if $task_i$ is a HW task on the FPGA ($hs_i = 0$) or a SW task on the CPU ($hs_i = 1$)

Algorithm 1. Top-Level Binary Search Algorithm for Finding the Minimum Schedule Length

Input: Lower bound LB and upper bound UB of the schedule length
Output: Minimum schedule length and the corresponding schedule
$lb = LB, ub = UB$;
while $lb < ub$ **do**
    $SL = \lfloor (lb + ub)/2 \rfloor$;
    $hasSolution$:=InvokeYices(GenSMTModel(SL));
    if $hasSolution$ **then**
        $ub := SL$, and record the schedule.;
    else
        $lb := SL + 1$;
return ub as the minimum schedule length, along with the corresponding schedule.

As shown in Figure 5, each HW task occupies a 2D rectangle in the 2D time-position chart, which consists of a reconfiguration stage and an execution stage plus a possible gap between them due to configuration prefetching. The thick border indicates the "forbidden" task area not available for use by other tasks. HW task scheduling can be formulated as a problem of nonoverlapping placement of rectangles.

—A HW task must fit on the FPGA.

$$\forall i, hs_i = 0 \implies pr_i = pl_i + W_i - 1 \wedge pl_i \geq 1 \wedge pr_i \leq M \tag{1}$$

—Preemption is not allowed for either the execution stage or the reconfiguration stage of a HW task, or for a SW task.

$$\begin{aligned} \forall i, hs_i = 0 &\implies fr_i = tr_i + R_i \wedge fe_i = te_i + E_i \\ \forall i, hs_i = 1 &\implies fe_i = te_i + SE_i \end{aligned} \tag{2}$$

—If a HW task is a source task in the task graph without any predecessors and it starts at time 0, then it has been preconfigured in the previous period

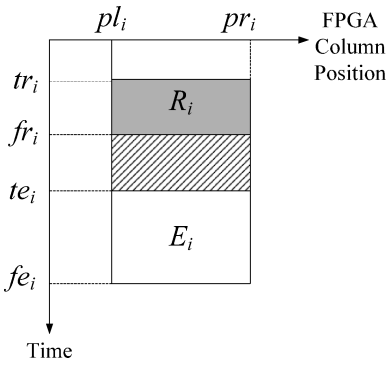


Fig. 5. $task_i$ on the 2D time-FPGA position chart.

and does not experience any reconfiguration delay. (Note that the FPGA may not be large enough to permit preconfiguration of all source tasks. The SMT model encodes all possible choices of the subset of source tasks to be preconfigured, so that the SMT solver will find the optimal choice as part of the solution to the optimization problem.)

$$\begin{aligned} \forall i, (task_i \text{ is not a source node} \wedge hs_i = 0) &\implies r_i = R_i \\ \forall i, (task_i \text{ is a source node} \wedge hs_i = 0) &\implies \\ &((r_i = 0 \vee r_i = R_i) \wedge \\ &(r_i = 0 \Leftrightarrow te_i = 0)) \end{aligned} \tag{3}$$

—Related to Condition 4, the task graph period (if it is periodically executed) should be larger than or equal to sum of reconfiguration delays of all source tasks whose reconfiguration stages have been pushed to the end of the period, that is, their reconfiguration delays do not contribute to the schedule length.

$$SL + \sum_i R_i \leq Period, \text{ where } r_i = 0. \tag{4}$$

In this article, we assume the task graph period to be large enough so that this condition becomes a null constraint.

—A task's reconfiguration stage (which may have length 0) must precede its execution stage.

$$\forall i, hs_i = 0 \implies (fr_i \leq te_i) \tag{5}$$

—Two HW task rectangles in the 2D time-position chart should not overlap with each other.

$$\begin{aligned} \forall i, j, i \neq j, hs_i = 0 \wedge hs_j = 0 &\implies \\ &(fe_i \leq tr_j \\ &\vee \ fe_j \leq tr_i \\ &\vee \ pr_i < pl_j \\ &\vee \ pr_j < pl_i) \end{aligned} \tag{6}$$

This constraint ensures that at least one of the 4 listed conditions must be true, that is, if two tasks do not overlap on the time dimension, then they can overlap on the space dimension, and vice versa. This guarantees that the two rectangles representing T_i and T_j do not overlap in the 2D time-position chart.

—Reconfiguration stages of different tasks must be serialized since the reconfiguration controller is a shared resource.

$$\begin{aligned} \forall i, j, i \neq j, hs_i = 0 \wedge hs_j = 0 \implies & \\ & fr_i \leq tr_j \\ \vee\ & fr_j \leq tr_i \end{aligned} \tag{7}$$

—If there is an edge from $task_i$ to $task_j$ in the task graph, then $task_j$ can only begin its execution after $task_i$ has finished its execution, taking into account any possible communication delay between the FPGA and CPU. Note that there is no precedence constraint on the two tasks' reconfiguration stages, which enables configuration prefetch.

$$\begin{aligned} &\forall i, j, i \neq j, Edge_{ij} = true \wedge hs_i = hs_j \implies fe_i \leq te_j \\ &\forall i, j, i \neq j, Edge_{ij} = true \wedge hs_i \neq hs_j \implies fe_i + ComDelay \leq te_j \end{aligned} \tag{8}$$

—Two SW tasks cannot overlap on the time axis due to the shared CPU resource.

$$\begin{aligned} \forall i, j, i \neq j, hs_i = 1 \wedge hs_j = 1 \implies & \\ & (fe_i \leq te_j \\ \vee\ & fe_j \leq te_i) \end{aligned} \tag{9}$$

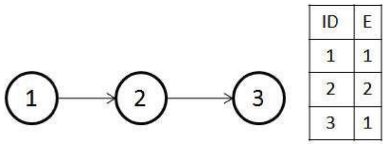
—Sink tasks must finish before the specified schedule length SL (this is the deadline constraint).

$$\forall i, (task_i \text{ is a sink node})\ fe_i \leq SL \tag{10}$$

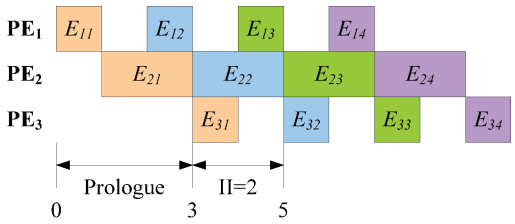
Conditions 1 to 10 form a constraint set that can be input to a SMT solver to check satisfiability.

3. PIPELINED SCHEDULING

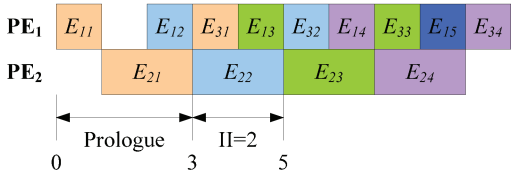
The pipelined scheduling problem is motivated by high-performance, throughput-oriented streaming applications implemented on a HW platform, which can be a PRTR FPGA, or a platform consisting of multiple FPGAs like the Galapagos prototyping platform [Noguera and Badia 2006]. The application is divided into multiple pipeline stages, and different iterations are scheduled in an overlapped manner to maximize throughput. If the HW platform is large enough to place all pipeline stages on it simultaneously, we should let each application pipeline stage occupy a fixed processing element on the HW platform. This corresponds to the traditional multiprocessor pipeline design, where each processor executes a pipeline stage. But sometimes it may be necessary and desirable to let two or more pipeline stages share the same processing element on the HW platform with time-multiplexing to increase resource utilization. There may be several possible reasons for this.



(a) Task graph example.



(b) 3 pipeline stages on 3 processing elements

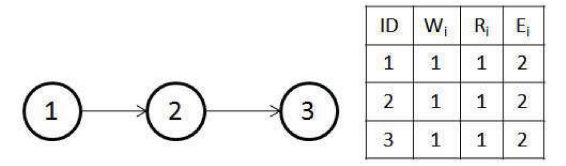


(c) 3 pipeline stages on 2 processing elements

Fig. 6. Example to show that time-multiplexing multiple pipeline stages on the same processor can increase resource utilization. The subscript ij denotes the jth iteration of task i in the task graph. This figure is best viewed in color.

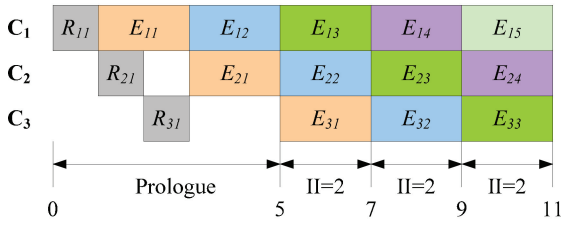
(1) The HW platform is not large enough to place all pipeline stages of an application on it.
(2) Even if the HW platform is large enough, it may be desirable to minimize limit the area occupied by one application in order to achieve isolation among multiple applications running on the same HW platform.
(3) If all pipeline stages have equal length, then the occupied area is fully utilized. But if certain pipeline stages are much shorter than the others, the HW processing element dedicated to these shorter stages will have long idle times, resulting in wasted resources. In this case, we can achieve higher resource utilization by placing multiple shorter pipeline stages on the same HW processing element.

As illustration of the third point preceding, Figure 6(b) shows a 3-stage pipeline with each stage assigned its own processing element, which can be either a CPU or a FPGA slot. We can see that processing elements PE_1 and PE_2 are underutilized. Figure 6(c) shows that letting the 1st and 3rd pipeline stage share a processing element results in increased resource utilization and reduced resource requirements. Another example of time-multiplexing of multiple pipeline stages is Piperench [Goldstein et al. 2000] (Figure 12 in Section 4).



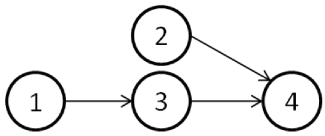
ID	W_i	R_i	E_i
1	1	1	2
2	1	1	2
3	1	1	2

(a) Task graph



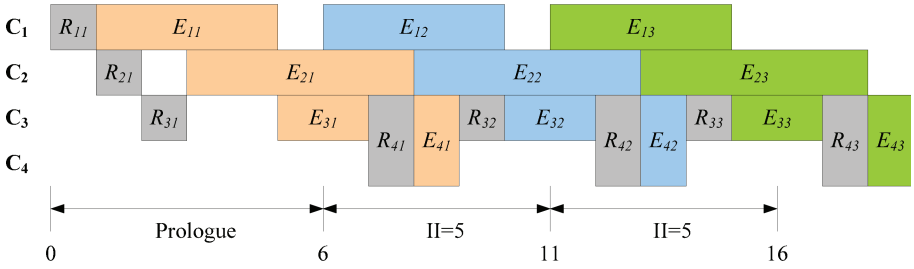
(b) Pipelined schedule.

Fig. 7. Pipelined scheduling example on a FPGA with 3 columns. $P = 2, latency = 6$. Label R denotes task reconfiguration, and E denotes task execution. This figure is best viewed in color.



ID	W_i	R_i	E_i
1	1	1	4
2	1	1	5
3	1	1	2
4	2	1	1

(a) Task graph



(b) Pipelined schedule.

Fig. 8. Pipelined scheduling example on a FPGA with 4 columns. $P = 5, latency = 8$. This figure is best viewed in color.

3.1 Motivating Examples

Figures 7 and 8 show two motivating examples of pipelined HW task scheduling on a PRTR FPGA. In Figure 7, the FPGA is large enough to fit all three tasks, so each task occupies a dedicated area on the FPGA. Reconfiguration of all three tasks only needs to be done once in the pipeline initialization stage (the prolog), but not during steady state execution. One of the FPGA columns (not

Table IV. Notations for Pipelined Scheduling, in Addition to Those in Table III

P	Initiation Interval (II) of the pipelined schedule, also called the pipeline period
r_i	actual reconfiguration delay of $task_i$. $r_i = 0$ if $task_i$'s occupied area has already been configured with its reconfiguration pattern by its instance in the previous iteration; $r_i = R_i$ otherwise.
tr'_i	relative start time of $task_i$'s reconfiguration stage within a pipeline stage
fr'_i	relative finish time of $task_i$'s reconfiguration stage within a pipeline stage
te'_i	relative start time of $task_i$'s execution stage within a pipeline stage
fe'_i	relative finish time of $task_i$'s execution stage within a pipeline stage

shown) is idle. In Figure 8, the FPGA size is not large enough to fit all tasks, so we need to time multiplex some columns between multiple tasks. In this case, tasks 3 and 4 share two columns, and their reconfiguration needs to be performed at every iteration; tasks 1 and 2 each occupy a dedicated column, and their reconfiguration only needs to be done once in the prolog.

A pipelined design is characterized by its *Initiation Interval* (II), defined as the time difference between the start of two consecutive iterations of task graph execution in the steady state. The II is equal to the inverse of system throughput. If we imagine a sliding time window with length equal to the II on the steady state Gantt chart, then any position of the moving window is a valid pipeline stage, which does not need to coincide with start or finish of any task. However, we adopt the common convention and set the start time of the first pipeline stage to coincide with the start time of the earliest source (initial) task. (If there are multiple source tasks, then the one with the earliest start time among them is chosen.) As a result, the finish time of the whole task graph is equal to the finish time of the last sink task, and must fall within the last pipeline stage. For example, the schedule in Figure 7(b) has 3 pipeline stages in the steady state; the II is 2; end-to-end latency is 6. The schedule in Figure 8(b) has 2 pipeline stages in the steady state; the II is 5; end-to-end latency is 8. There may be many possible pipelined schedules with different throughput and latency characteristics. Our objective is to find the optimal schedule with maximum throughput (minimum II) under a user-defined task graph deadline (latency) constraint.

3.2 SMT Model for Pipelined Scheduling

Table IV summarizes the notations used in this section. The nonoverlapping rectangle constraints for nonpipelined scheduling are still applicable (Figure 5), but in addition, we need to consider possible wrap-around of a task's rectangle on the time axis for pipelined scheduling when writing the constraint equations due to the overlapped execution of different task graph iterations. For example, in Figure 7, a pipeline stage contains one instance of each of the three tasks' execution stage, while in Figure 8, no matter how we choose to divide up the time axis into pipeline stages, there is always some task's reconfiguration or execution stage that is cut at the stage boundary and wraps around the pipeline stage. In addition to the time variables tr_i, fr_i, te_i, fe_i defined in Table III, we define additional primed variables $tr'_i, fr'_i, te'_i, fe'_i$, which are relative to the start of the current pipeline stage. The unprimed variables are called *absolute time*, and the primed variables are called *relative time*.

The top-level binary search algorithm is similar to Algorithm 1, except the binary search variable is the minimum II instead of the minimum schedule length. The lower bound of the II is set to $\sum_i E_i * W_i / M$, that is, when the FPGA is fully utilized without any gaps, and there are no reconfiguration stages in the pipeline steady state (e.g., the example in Figure 7); the upper bound of the II is set to the minimum schedule length SL obtained from the SMT model in Section 2 assuming there is only a FPGA without a CPU. This is because repeating the nonpipelined schedule as the pipeline steady state forms a valid pipelined schedule with II equal to SL, and we aim to find pipelined schedules with II values smaller than SL.

In addition to the same constraints as those for HW tasks in HW/SW partitioning in Section 2, we add the following additional constraints that are specific to pipelined scheduling due to the wrap-around of a task's reconfiguration or execution stages.

—Each task must fit into a pipeline stage.

$$\forall i, fe_i - tr_i \leq P \tag{11}$$

—$Task_i$'s reconfiguration delay is either R_i or 0, depending on whether it can exploit configuration reuse. If $task_i$'s reconfiguration stage has length 0, then it exploits configuration reuse between two iterations, and it must have been assigned its own dedicated HW area not shared with other tasks, for example, tasks 1 and 2 in Figure 8. The reverse is also true.

$$\forall i, (r_i = R_i \vee r_i = 0) \wedge (r_i = 0 \Leftrightarrow \forall j, (i \neq j), (pr_i < pl_j) \vee (pr_j < pl_i)) \tag{12}$$

—Relationship between absolute and relative time variables.

$$\begin{aligned} tr'_i &= tr_i \% P \\ fr'_i &= fr_i \% P \\ te'_i &= te_i \% P \\ fe'_i &= fe_i \% P \end{aligned} \tag{13}$$

—Conditions 14 to 16 are the constraints corresponding to the scenarios in Figure 9 that illustrate possible spatial and temporal relationships between two tasks in the same pipeline stage, depending on whether each task wraps around the pipeline stage. (Note that there may be other tasks running concurrently with $task_i$ and $task_j$ that are not shown.) Here is an explanation on the use of $<$ versus $\leq$ in Conditions 14 to 16: since we assume that all tasks have nonzero execution time, we have $tr'_i < fe'_i$ if $task_i$ does not wrap around the pipeline stage. In contrast, we have $fe'_i \leq tr'_i$ if $task_i$ wraps around, since it is possible to have $fe'_i = tr'_i$ if $r_i = 0$ due to configuration reuse. This is when $task_i$ executes back-to-back without any reconfiguration stage in-between different iterations.

(a) For the cases in Figure 9(a), when neither task wraps around, we have the follwoing.

$$\begin{aligned} &\forall i, j, i \neq j, ((tr'_i < fe'_i) \wedge (tr'_j < fe'_j)) \implies \\ &((fe'_j \leq tr'_i) \vee (fe'_i \leq tr'_j) \vee (pr_i < pl_j) \vee (pr_j < pl_i)) \end{aligned} \tag{14}$$

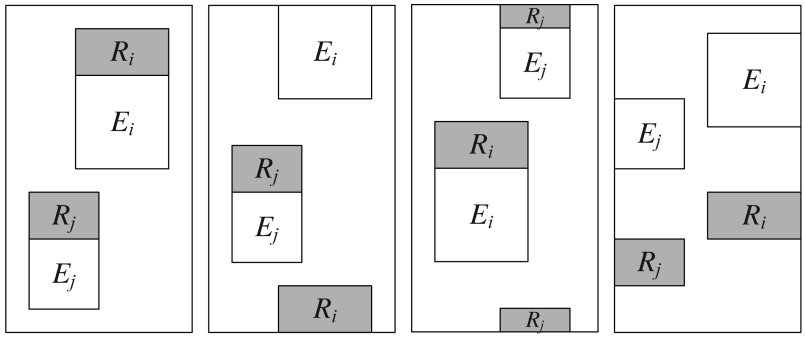


Fig. 9. Possible placement relationships between two tasks.

(b) For the cases in Figures 9(b) and 9(c), when one of the tasks wraps around, we have the following.

$$\forall i, j, i \neq j, (((tr'_i < fe'_i) \wedge (fe'_j \leq tr'_j)) \vee ((fe'_i \leq tr'_i) \wedge (tr'_j < fe'_j)) \implies (((fe'_j \leq tr'_i) \wedge (fe'_i \leq tr'_j)) \vee (pr_i < pl_j) \vee (pr_j < pl_i)) \quad (15)$$

(c) For the cases in Figure 9(d), when both tasks wrap around, we have what follows.

$$\forall i, j, i \neq j, ((fe'_i \leq tr'_i) \wedge (fe'_j \leq tr'_j)) \implies ((pr_i < pl_j) \vee (pr_j < pl_i)) \quad (16)$$

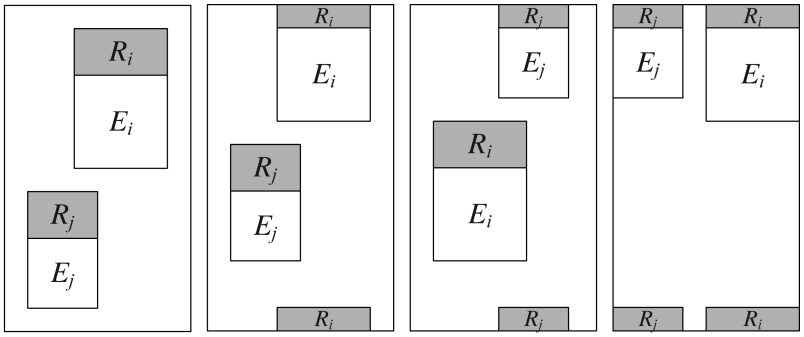
—We consider FPGAs with a single configuration controller as a global shared resource, hence reconfiguration stages of different tasks must be serialized, that is, they cannot overlap on the time axis. Conditions 17 to 19 encode the constraints on the reconfiguration stages of a task pair, depending on whether one or more of them wrap around the pipeline stage. (Note that these constraints are not redundant with Conditions 14 to 16, since they encode the serialized execution of the reconfiguration stages of different tasks.) Here is an explanation on the use of $<$ versus $\leq$ in Conditions 17 to 19: since it is possible for a $task_i$'s reconfiguration delay to be 0, we have $tr'_i \leq fr'_i$ if the reconfiguration stage does not wrap around the pipeline stage; in contrast, we have $fr'_i < tr'_i$ if the reconfiguration stage wraps around, since it is not possible to have the reconfiguration stage occupy the entire pipeline stage.

(a) For the case in Figure 10(a), when neither task's reconfiguration stage wraps around, we have the following.

$$\forall i, j, i \neq j, ((tr'_i \leq fr'_i) \wedge (tr'_j \leq fr'_j)) \implies (fr'_i \leq tr'_j) \vee (fr'_j \leq tr'_i) \quad (17)$$

(b) For the cases in Figures 10(b) and 10(c), when one of the task's reconfiguration stage wraps around, we have the following.

$$\forall i, j, i \neq j, ((fr'_i < tr'_i) \wedge (tr'_j \leq fr'_j)) \vee ((tr'_i \leq fr'_i) \wedge (fr'_j < tr'_j)) \implies (fr'_i \leq tr'_j) \wedge (fr'_j \leq tr'_i) \quad (18)$$



(a) neither task's reconfiguration stage wraps around

(b) $task_i$'s reconfiguration stage wraps around

(c) $task_j$'s reconfiguration stage wraps around

(d) illegal scenario: both tasks' reconfiguration stages wrap around

Fig. 10. Possible placement relationships between two tasks' reconfiguration stages.

(c) For the case in Figure 10(d), when both tasks' reconfiguration stages wrap around, we have what is presented next. This is an illegal scenario that needs to be prevented.

$$\forall i, j, i \neq j, \neg((fr'_i < tr'_i) \wedge (fr'_j < tr'_j)) \tag{19}$$

3.3 Kernel Recognition-Based Heuristic Algorithm

In this section, we present a heuristic scheduling algorithm to overcome the scalability issue of the SMT-based optimal solution technique. For simplicity, we assume that no deadline is associated with the task graph, although the algorithm can be modified to take into account task deadlines. The greedy nature of list scheduling prevents long schedule latencies even though no deadlines are imposed. Even with this relaxed assumption, we will see in Section 5 that the heuristic algorithm obtains inferior results compared to the optimal solution obtained by SMT with deadline constraints.

SW pipelining refers to scheduling a SW loop on multiple functional units to maximize throughput. Allan et al. [1995] presented a comprehensive survey on many SW pipelining techniques, broadly classified in two categories: *modulo scheduling* and *kernel recognition*. Modulo scheduling is not applicable to our problem, since it cannot handle HW task configuration prefetch and reuse. *Kernel recognition*, also called *perfect pipelining* [Aiken et al. 1995], is a SW pipelining technique that works as follows.

(1) Unroll the loop and note dependencies.
(2) Schedule the operations as early as dependencies allow.
(3) Look for a block of instructions that form a repeating pattern. This block represents the new loop body.

Recognition of a repeating pattern in the scheduled operations in SW pipelining is not trivial, and the repeating pattern may not even form if the scheduler and the analysis of "available operations" is not constrained in some way. Aiken

et al. [1995] presented constraints necessary to ensure the success of kernel recognition (these constraints are rather weak and can be easily satisfied, in particular, the problem of pipelined scheduling on FPGAs addressed in this article satisfies all these assumptions).

—The scheduler is a function of the available operations in the state being scheduled.
—The scheduler must schedule some operation in every state.
—The operation chosen can depend on the set of operations available and the relative distance between the iteration numbers of the available operations, but not on their actual iteration numbers.

Aiken et al. [1995] presented a SW pipelining algorithm that satisfies the previous constraints. When resource allocation constraints (e.g., due to limited number of functional units and registers) are considered, the preceding three constraints need to be strengthened to prevent resource conflicts. Aiken et al. [1995] also presented a modified SW pipelining algorithm that satisfies these constraints by using resource reservation tables. A simple greedy list scheduling heuristic is used without backtracking search. Pipelined scheduling of a task graph is a similar problem to SW pipelining, since task graphs are a special case of Homogeneous Synchronous DataFlow (HSDF) graphs considered in SW pipelining by removing all initial delay tokens on the edges. We can ensure the emergence and recognition of a repeating pattern by using a deterministic list scheduling algorithm, which satisfies the constraints presented in Aiken et al. [1995] with consideration of resource allocation. List scheduling is a well-known greedy algorithm for task graph scheduling on multiprocessors based on a priority function. At each scheduling step, the task node with the highest priority value among all the ready tasks is chosen and scheduled.

Algorithm 2 shows the overall algorithm for pipelined scheduling based on kernel recognition and list scheduling. Next, we elaborate on each of the steps.

3.3.1 *Task Graph Unrolling.* We assume there is no auto-concurrency between different iterations of the same task, that is, iteration i of a task must be finished completely before iteration $i + 1$ of the same task can begin. (This assumption may be unnecessary for some applications, since it is only necessary for tasks that need to maintain an internal state that is persistent across different iterations, but not for tasks that are stateless. It can be relaxed by selectively allowing auto-concurrency for certain tasks.) Figure 11 shows the unrolling twice of a linear task graph with 3 tasks with this assumption.

3.3.2 *Obtaining Ready List and Highest-Priority Ready Node.* The ready list can be easily obtained based on task graph dependencies. We define a simple priority function for choosing the next ready task node to schedule as a weighted sum of the task's execution time, reconfiguration delay, and its width.

$$p_i = w_1 * E_i + w_2 * R_i + w_3 * W_i \tag{20}$$

We discuss setting of the weight parameters in Section 5.2.

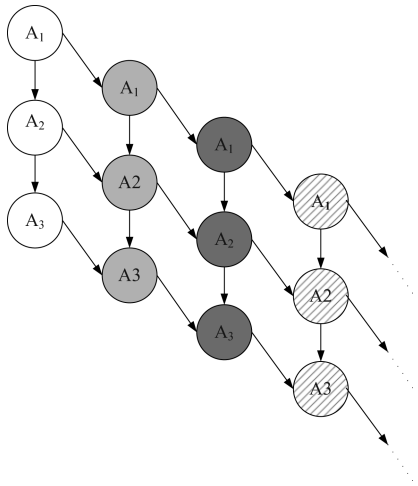


Fig. 11. Task graph unrolling.

Algorithm 2. Pseudocode for Pipelined Scheduling Based on Kernel Recognition

```
while true do
    Unroll the task graph once;
    while Some nodes are not scheduled yet do
        Obtain ready list;
        Get the highest-priority node in ready list;
        Place it on a suitable position on FPGA;
        Mark it as scheduled;
    if Repeating pattern detected then
        break;
```

3.3.3 *Finding a Suitable Position on FPGA.* In list scheduling, a task can be placed on any available processor if all processors are identical. If processors are heterogeneous, then we need to have some additional criteria to decide on which processor to place the task. For task scheduling on FPGA, we need to decide on which position (column) to place a HW task. Even though all FPGA columns are homogeneous in terms of HW task reconfiguration, it is necessary to distinguish between different columns for the purpose of task placement, since each task must occupy a contiguous set of columns. Here are the steps of choosing a suitable position to place the highest-priority task.

(1) Initially, try to spread out all task configurations to cover the entire FPGA area (or a user-specified portion of it) from left to right, in order to exploit the available parallelism to reduce the II.
(2) Try to take advantage of any opportunity of configuration reuse. If a set of contiguous columns have already been configured for one iteration of $task_i$, then we try to place the next iteration of $task_i$ on the same position to reuse the configuration pattern and eliminate the reconfiguration delay. It may be

necessary to postpone $task_i$'s start time to achieve configuration reuse, but we should not blindly postpone $task_i$'s start time indefinitely for the sake of configuration reuse. We observed in our experiments that doing so is likely to cause an increase in the II, since tasks tend to be clustered within a limited region and do not fully exploit the available parallelism. Therefore, we adopt a user-defined parameter as the length of time to look into the future for any configuration reuse opportunity, which is proportional to W_i, the size of $task_i$ in terms of the number of contiguous FPGA columns it occupies. A larger W_i implies a larger benefit brought by configuration reuse.

(3) If configuration reuse is not successful, then look for any opportunity of configuration prefetch by looking backwards from $task_i$'s ready time $Ready(task_i)$ for a time instant $t < Ready(task_i)$ when the reconfiguration controller is free. If successful, then we can start configuration prefetch at time t so that $task_i$ can start execution at time $t + R_i$. We use a first-fit policy to choose among multiple candidate positions, that is, we scan from left to right, and choose the first position that is large enough to contain $task_i$. We have tried other policies such as best-fit and worst-fit, but did not observe any significant performance differences.

(4) If configuration prefetch is not successful, then search forward from $task_i$'s ready time $Ready(task_i)$ to look for a time instant $t \geq Ready(task_i)$ and a position P to configure $task_i$. Since we assume that tasks do not have deadlines, we can always find such a time instant and a position. $task_i$'s execution starts immediately after its configuration.

3.3.4 *Detecting Repeating Patterns.* We adopt a discrete time model, and use a global clock tick to drive system simulation. Two system snapshot states s_i and s_j are defined to be identical when every task is in the same configuration or execution stage as one of its previous iterations, that is, every task and one of its previous iterations are either both in the configuration stage and have the same remaining reconfiguration delay, or are both in the execution stage and have the same remaining execution time. During system execution, whenever a task finishes execution, we record the current system snapshot state s_0 and compare it with what we have seen so far to detect a possible match. If s_0 has not been seen before, then we add it to a linked list of snapshot states from time 0. If the current snapshot state is identical to a previous snapshot state s_1, then we have detected a repeating execution pattern as the pipeline kernel: since our task scheduling algorithm is fully deterministic, this pattern must be repeated in the future until all task graph instances have been unrolled and executed. As an example, consider the execution trace in Figure 7(b). We record snapshot states at time instants 3 and 5 when certain tasks finish execution. At time instant 7, we detect an identical snapshot state at time instant 5, so a pipeline kernel has been detected between time instants 5 and 7.

3.3.5 *Complexity Analysis.* We analyze the complexity of each operation described before with regard to number of tasks N in the task graph and width of the FPGA reconfigurable area M.

—*Task Graph Unrolling*. Complexity is $O(N)$, since we need to traverse all nodes in the task graph once.
—*Obtaining Ready List and Highest-Priority Ready Node*. Complexity is $O(N)$, since we need to traverse all nodes once to obtain the ready list, then traverse the ready list once to get the highest-priority node.
—*Finding a Suitable Position on FPGA*. Complexity is $O(NM)$, where M is the width of FPGA reconfigurable area, since we need to examine $O(M)$ possible candidate positions on the FPGA, and for some of the candidate positions, we need to perform forward and backward search looking for configuration reuse and prefetch opportunities within a time window size bounded by $O(N)$.
—*Detecting a Repeating Pattern*. Complexity is $O(NM)$, since we need to perform O(N) number of comparisons of the current snapshot to $O(N)$ previous snapshots, and each comparison takes time $O(M)$.

So the complexity of the statements within the **while** loop is $O(NM)$, dominated by the last two steps of finding a suitable position and detecting a repeating pattern. The total number of times that the **while** loop is executed is the number of times the task graph is unrolled before a repeating pattern is detected, which is $O(N)$. So the overall algorithm complexity is $O(N^2M)$.

4. RELATED WORK

4.1 Work on HW/SW Partitioning

HW/SW partitioning of task graphs is a well-studied problem, but most prior work did not consider FPGA dynamic reconfiguration, that is, each HW task is typically assigned a fixed position on the FPGA and never reconfigured. Even though sometimes a HW task may be inactive, that is, waiting its next invocation trigger, the area it occupies is not reclaimed for use by other tasks, that is, there is no time-multiplexing of different tasks at the same FPGA position. This model removes the major complexities in FPGA scheduling, but may not be efficient if some HW tasks have low utilization on the time axis, for example, large periods and small execution time. Fekete et al. [2001] addressed optimal placement of a task graph on a 2D PRTR FPGA by treating each task as a 3D box in space and time, and converting the problem into an optimal box packing problem solved using an efficient search algorithm. But if task precedence constraints and configuration prefetch are considered, then there may be a gap between a task's reconfiguration and computation stages, so the problem is no longer a box packing problem, and the algorithm in Fekete et al. [2001] is no longer applicable. Shang et al. [2007] used a genetic algorithm to partition a task graph between the CPU and FPGA, and a list scheduling heuristic for task graph scheduling on the FPGA in the inner loop of the genetic algorithm. They assume the HW task reconfiguration can be processed for each individual column independently, instead of for each HW task atomically, which may not be very realistic for commercial FPGAs. Banerjee et al. [2005] presented a heuristic algorithm for partitioning and scheduling of a task graph on an execution platform consisting of a CPU and a FPGA, which we will compare with for the

performance evaluation. They also presented an ILP encoding of the problem, which works by using two boolean variables for each FPGA column at each time point to describe the state of each task and their position on the FPGA. Our SMT encoding is more compact and efficient than their ILP encoding, not only due to the expressiveness of SMT compared to ILP for expressing disjunctive constraints, but mainly because our SMT formulation turns the task graph scheduling problem into a problem of nonoverlapping placement of rectangles, where each HW task (including both its reconfiguration stage and computation stage) is modeled as a rectangle in the 2D time-position chart, and the scheduling constraint is that no two rectangles can overlap with each other (Eq. (7)). In contrast, the ILP encoding in Banerjee et al. [2005] adds a new set of boolean variables at every time step (clock tick), similar to the SAT formulation for the problem of High-Level Synthesis [Memik and Fallah 2002], which can become very inefficient, as the number of boolean variables can grow very large with large task execution times, while our encoding is independent of the size of task execution times. We implemented a tool that accepts as input the task graph parameters and generates the corresponding ILP encoding according to the method in Banerjee et al. [2005]. If the task execution times are relatively large, we encountered "code explosion", where the number of boolean variables in the ILP encoding grew so large that the ILP solver lp_solve was not even able to read in the input file.

4.2 Work on Pipelined Scheduling

Some authors have applied ILP to pipelined scheduling of task graphs on multiprocessors. Jin et al. [2005] developed an ILP formulation for task mapping on a multiprocessor system for maximizing throughput, with the assumption that each pipeline stage is mapped to one processor, so the number of processors is equal to the number of pipeline stages. Lin et al. [2007] developed an ILP formulation of mapping a hierarchical Synchronous Dataflow Graph (SDF) onto a multiprocessor system with a Direct Memory Access (DMA) engine to maximize throughput. Kudlur and Mahlke [2008] developed an integrated ILP formulation of splitting of data-parallel actors and mapping of actors to processors on the Cell Broadband Engine, followed by a greedy algorithm to assign actors to pipeline stages, with the objective of maximizing throughput. Kudlur and Mahlke [2008] removed the constraint in Lin et al. [2007] on the maximum number of simultaneous DMA operations, which enabled them to use a greedy algorithm for pipeline stage assignment instead of building it into the ILP formulation.

Ahn et al. [2006] discuss two categories of algorithms for mapping an application onto a Coarse-Grained Reconfigurable Array (CGRA), that is, *spatial mapping*, where each processing element has a fixed configuration, and *temporal mapping*, where the configuration of each processing element is changed at runtime to use time-multiplexing to reduce the number of processing elements required. Piperench [Goldstein et al. 2000] is a coarse-grained reconfigurable fabric, an interconnected network of configurable logic and storage elements, designed for stream-based multimedia applications. As an example

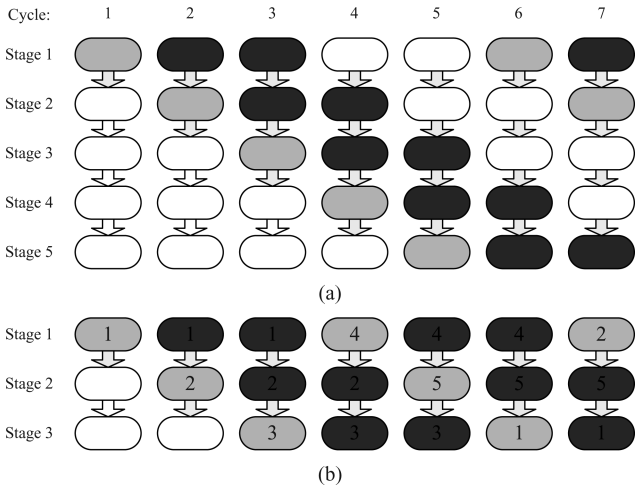


Fig. 12. HW virtualization is used in Piperench to map a 5-stage application pipeline to a HW device with 3 processing elements: (a) the virtual pipestage; (b) the physical pipeline stage (the numbers in the ovals refer to the virtual pipeline stage).

of temporal mapping, Figure 12, taken from Goldstein et al. [2000], shows that letting each pipeline stage occupy a separate HW processing element results in resource underutilization, indicated by the idle times of the 5 processing elements at pipeline steady state. Letting multiple pipeline stages share a single processing element with time-multiplexing helps to eliminate all idle time and make all 3 processing elements fully utilized at pipeline steady state, leaving the other 2 processing elements available to other applications. It is a form of *HW virtualization* [DeHon et al. 2006], where a HW-oblivious application is implemented with limited HW resources using time-multiplexing "behind the back" of the application programmer, who is given the illusion of unlimited HW resources.

Different from pipelined scheduling on multiprocessors and CGRAs, pipelined scheduling of HW tasks on FPGAs brings unique challenges due to the unique characteristics of HW task reconfiguration on the FPGA (unique reconfiguration controller, configuration prefetch and reuse to hide large reconfiguration delays), and has not been adequately addressed in the literature.

4.3 Work on Reducing Reconfiguration Delay

The need for dynamic reconfiguration arises from insufficient FPGA area resources to place all tasks simultaneously, thus making it necessary to use time-multiplexing to share the same FPGA area among multiple tasks. If the FPGA area is large enough to place all tasks, then obviously it is not necessary to resort to dynamic reconfiguration. With the increasing size and complexity of today's embedded systems applications, we believe there is an increasing need for using dynamic reconfiguration to support either a single large application, or multiple different applications on a limited-size FPGA.

One natural concern is whether dynamic reconfiguration is feasible considering the large reconfiguration delay of FPGAs. Especially, the pipelined scheduling algorithm allows multiple pipeline stages to share a common reconfigurable area, hence some columns are time-multiplexed between multiple tasks, making it necessary to perform frequent reconfigurations during application execution. (The reconfiguration delay issue is less problematic for the HW/SW partitioning problem for minimizing schedule length, which has been extensively addressed in related work, since the task graph may be executed only once, or periodically executed with a large period.) In order to make pipelined scheduling feasible, HW task reconfiguration times have to be small enough (compared to their execution times) to justify the cost of frequent reconfiguration. Next, we discuss some recent research advances that can reduce reconfiguration delay significantly.

HW vendors have been improving HW design to increase reconfiguration speed. The configuration bitstream is downloaded to the FPGA via a bus to its Internal Configuration Access Port (ICAP), so reconfiguration delay is determined by the bitstream size, the bus speed, and the ICAP throughput. Virtex-II Pro provides an ICAP with an 8-bit wide interface working at 50MHz, while Virtex-4 provides an ICAP with a 32-bit wide interface working at 100MHz. This means that the theoretical upper limit of reconfiguration throughput has been increased from 50KB/ms to 400KB/ms from Virtex-II Pro to Virtex-4. What prevents us from achieving the theoretical upper limit of today's high-speed ICAP is data transfer from memory to ICAP. Recent research advances demonstrate that the reconfiguration delay can be reduced significantly for standard commercial FPGAs compared to what was achievable a few years ago. The traditional ICAP controller is a slave attachment on the low-speed OPB (On-Chip Peripheral Bus), which can be a throughput bottleneck. Claus et al. [2007] developed a framework for reducing reconfiguration delays by using the *combitgen* tool to generate efficient bitstreams, and a new ICAP controller connected directly to the high-speed PLB (Processor Local Bus) as a master and equipped with DMA (Direct Memory Access). As a result, the configuration speed can be 20 times faster compared to the OPBHWICAP from Xilinx, where the ICAP controller is attached to the OPB. Cuoccio et al. [2008] proposed a customized ICAP controller based on that in Claus et al. [2007] to further reduce the configuration delay. It stores part of the configuration bitstream in the internal memory of the ICAP controller, so the ICAP can start the bitstream transfer process immediately while the rest of the bitstream is retrieved from main memory across the bus. Liu et al. [2009] proposed two additional ICAP-based reconfiguration techniques: one, MST_HWICAP, is to use an integrated bus master with burst transmission support instead of DMA, hence the bus master can actively fetch the bitstream from memory to avoid the communication overhead between DMA and ICAP. The other, BRAM_HWICAP, is to use a dedicated Block RAM on the ICAP to store the bitstreams (similar to Cuoccio et al. [2008]), hence avoiding the need to transfer the bitstream from memory to ICAP. This design requires that the bitstream size is small enough to fit in the BRAM. The measured maximum reconfiguration speed using BRAM_HWICAP is 371.4MB/s on a Xilinx Virtex-4 FX20 FPGA, very close to the theoretical

maximum of 400MB/s imposed by the physical limitation of the ICAP interface (32-bit, 100 MHz). Since the typical HW task bitstream size is in the range of a few hundred KBs, the reconfiguration delay can be potentially less than 1ms, which is smaller than the CPU context-switch latency of some operating systems. Besides the these approaches, a complementary approach to reducing reconfiguration delay is to *reduce HW task bitstream size* with *difference-based* partial reconfiguration or *bitstream compression*, which we will not elaborate here.

4.4 SMT vs. SAT or ILP

Some authors have formulated boolean encodings for the High-Level Synthesis (HLS) problem, that is, finding the shortest schedule length of a control/dataflow graph on a limited set of HW resources, in order to use a SAT solver to obtain optimal solutions. Memik and Fallah [2002] used the SAT solver Chaff [Moskewicz et al. 2001] to solve the HLS problem, and showed that Chaff outperforms the ILP solver CPLEX in terms of CPU time by as much as 59 fold. Cabodi et al. [2005] developed a Bounded Model-Checking (BMC) formulation of the HLS problem for control-intensive control/dataflow graphs, and used the BerkMin SAT engine to solve the BMC problem. These SAT-based encoding techniques add a new set of boolean variables at each time-step, hence the number of boolean variables can grow quite large if the schedule length (number of time-steps) is large. This approach is often adequate for HLS, where the maximum schedule length is typically quite small in terms of the number of clock cycles, but it is not very scalable for problems involving large timing attributes. By handling integer or real arithmetic directly instead of using a boolean encoding, SMT solvers are not sensitive to absolute time attributes and can be much more efficient than SAT solvers for such problems.

ILP is a widely used technique for solving NP-hard combinatorial optimization problems. It is also possible to use ILP instead of SMT in this article. Here we briefly compare their pros and cons.

—SMT solvers can handle both disjunctive ($\vee$) and conjunctive ($\wedge$) constraints efficiently due to integration with a SAT solver, while ILP does not support disjunctive constraints natively. Therefore, SMT is more expressive than ILP, and allows us to express constraints more naturally and concisely. We can transform a logical formula with both disjunctive and conjunctive constraints into another equivalent formula with only conjunctive constraints supported by the ILP solver using the *Big-M* method [Papadimitriou and Steiglitz 1998], the size of the ILP constraint set may be much larger than the SMT constraint set. For example, Suhendra et al. [2006] used a lot of "infinity" constants in their ILP formulation of the pipelined scheduling problem to express disjunctive constraints, which can be avoided if SMT is used instead of ILP. However, the number of variables is a poor metric to characterize the complexity of the problem instance, so we cannot completely attribute the performance advantage (if any) of SMT solvers only to the reduced number of variables. Rather, any possible performance advantage of SMT over ILP may be due to clever SAT algorithms, including nonchronological backtracking, a

priori simplification to reduce the problem instance size, and efficient lemma learning, etc.

—One advantage of ILP is that it can be used to encode and solve optimization problems directly, while SMT is a decision procedure that does not support optimization directly, but only provides a yes/no answer to the feasibility of a given set of constraints, and a model when the constraints are feasible. Therefore, using SMT, we need to use binary search, for example, Algorithm 1, to find the optimal value of a given optimization objective. However, this turns out not to be a major performance bottleneck for our problems.

The SMT encoding in this article can be reformulated equivalently with ILP. However, the main contribution of this article is orthogonal to whether SMT or ILP is used, so we do not present the ILP encoding or perform any performance comparisons.

5. PERFORMANCE EVALUATION

We use the tool TGFF (Task Graphs For Free) [Dick et al. 1998] to generate random task graphs for our experiments. Optimization performance depends on the search space size, which in turn depends on many factors including number of tasks, task graph shape, number of messages, and task assignment. Generally, task graphs that are "tall and skinny" tend to have a smaller number of possible execution paths than task graphs that are "short and fat." For our experiments, we keep the task graph shape to be relatively constant by setting both the maximum input and max output degrees of each task node to be 2, so we can compare the relative performance with different numbers of tasks. In each generated task graph, the number of start nodes is between 1 and 3; each task's execution time is between 4 and 12; its width (the number of columns it occupies) is between 4 and 14; its reconfiguration delay is assumed to be numerically equal to its width. We assume the FPGA has 20 columns. The experiments are run on a Linux workstation with 4 AMD Opteron 844 (1.8 GHz) CPUs and 8GB RAM. We use the utility tool *memtime* to measure the running time of Yices. The binary search process in Algorithm 1 typically converges in 7–9 iterations to the final optimal result, and the optimization algorithm running time for Yices refers to the *total time* taken to obtain the optimal solution including all binary search iterations.

5.1 Performance Evaluation of HW/SW Partitioning

Figure 13 shows how algorithm running time increases with increasing task graph size (number of tasks). (Since the peak memory usage grows quite slowly compared to running time, CPU time is the bottleneck factor that limits its scalability instead of memory usage, hence we omit the memory usage figures.) Since the running time is dependent on not only the number of tasks, but also on other task graph characteristics, such as its shape and task execution times, we generated multiple different task graphs and plotted their different running times for each task graph size. In general, 22–23 tasks seems to be the upper bound on the size of the task graph that can be handled within reasonable time.

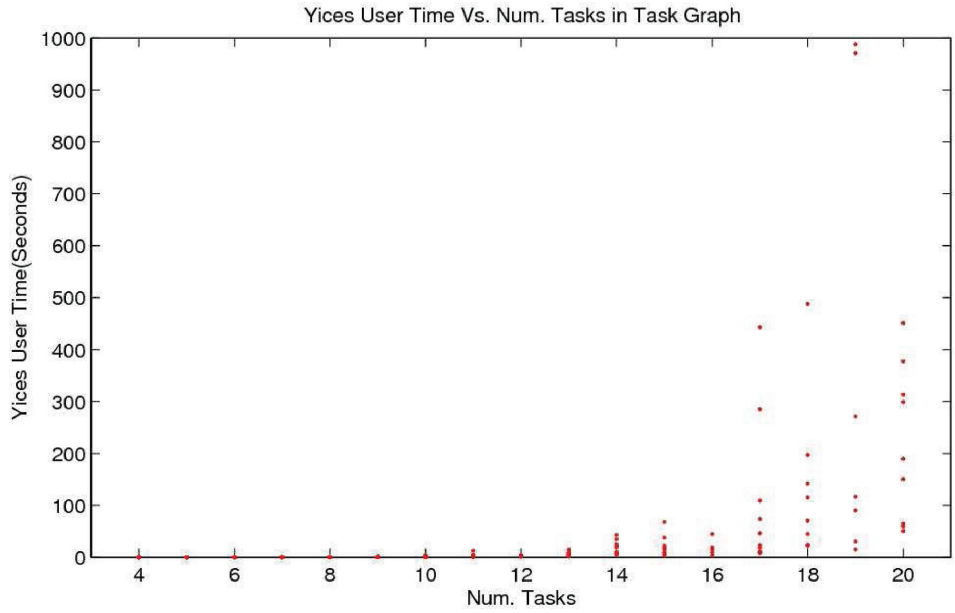


Fig. 13. SMT-based optimization algorithm running time for HW/SW partitioning on a hybrid FPGA/CPU device.

We have implemented the heuristic algorithm for HW/SW partitioning in Banerjee et al. [2005], which is based on the well-known KLFM (Kernighan-Lin-Fiduccia-Matheyses) heuristic that iteratively improves a HW/SW partitioning solution by moving tasks from FPGA to CPU or vice versa. The quality of a move is evaluated by a heuristic list scheduler, and each task's priority is calculated as $f = -A * columns - B * EST + C * pathlength - D * EFT$, where *columns* denotes the task's size in terms of the number of FPGA columns it occupies; *EST* and *EFT* denote the earliest possible start and finish time points, respectively; and *pathlength* denotes the length of the longest path through the task graph. This formula has 4 parameters, A, B, C, and D, as weights of the 4 terms. The task with the largest f value is scheduled first during the list scheduling process. In our implementation, we set the parameter values to be: $A = C = 1$, $B = D = 3$, and fix the number of task moves to 10. Of course, different parameter settings may lead to different results. As expected, the heuristic algorithm runs very fast, but produces suboptimal results. Figure 14 shows the comparison of the schedule lengths between SMT and the heuristic algorithm. We can see that SMT provides an exact solution reasonably quickly for small-scale problems within its scalability limit, and can result in significant reduction in schedule length compared to the heuristic algorithm.

5.2 Performance Evaluation of Pipelined Scheduling

The task graph deadline can have a large impact on the pipelined scheduling throughput. It is obvious that the minimum task graph schedule length *SL* for nonpipelined scheduling obtained using the SMT model in Section 2.2 (on

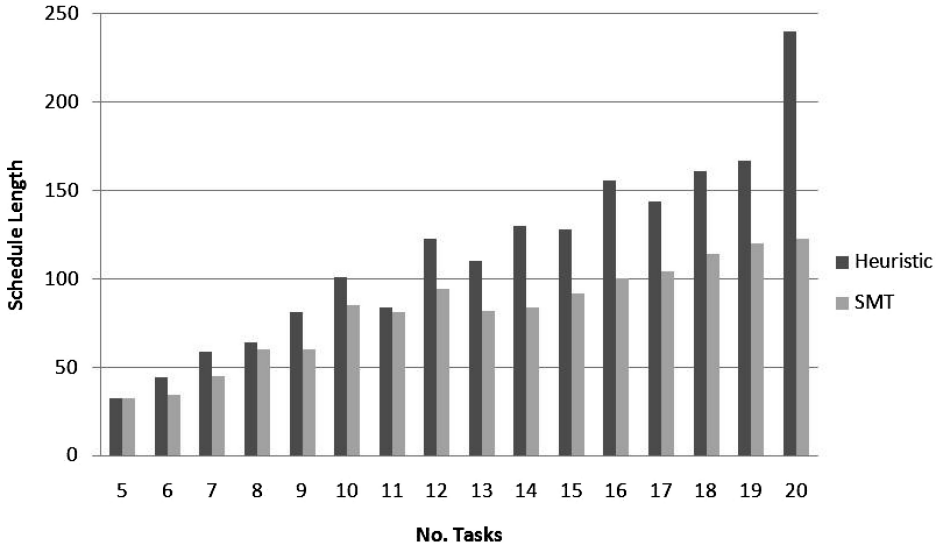


Fig. 14. Comparison of schedule length between Yices and Banerjee et al.'s [2005] heuristic algorithm.

a FPGA without the CPU) is the minimum feasible task graph deadline for pipelined scheduling. We set the task graph deadline to be $1.25 * SL$, $1.5 * SL$ or $1.75 * SL$ for the SMT encoding in Section 3. (It is also possible to remove the deadline constraint at the cost of increased search space size.)

For the Kernel Recognition (KR)-based heuristic algorithm, we set $w_1 = w_2 = w_3 = 1$ in Eq. (20) in the first set of simulation experiments. Next, in order to find potentially better assignment of these weight values, we used Simulated Annealing [Kirkpatrick et al. 1983] to optimize them. SA is a global optimization method that tries to find the global optimal point in the design space by jumping over local optimal points. SA works as follows: The temperature T is gradually decreased as SA runs.[2] At each step of SA, we make a random move, and calculate the *energy function* E_{new} for the new solution obtained via one of the random moves, then compare it to the energy function of the current solution E. If $E_{new} < E$, then the new solution has an improved energy function, so accept it; otherwise, if $e^{(E-E_{new})/T} >$ random(0, 1), then accept the new solution despite the energy increase; otherwise, reject the new solution and attempt another random move. When the temperature is high, the probability of accepting a higher energy solution is high, to enable it to jump out of any possible local minima; as temperature drops, the system gradually settles down to a stationary state, possibly a local or global minima. The SA algorithm behaves similarly to random search at high temperatures, and to hill-descent at low temperatures. Important components of SA include the definition of the

[2]Different cooling schedules may be adopted, including geometric cooling, nonmonotonic cooling, etc. We adopt the simple constant-rate cooling schedule, since it shows reasonably good performance compared to other cooling schedules.

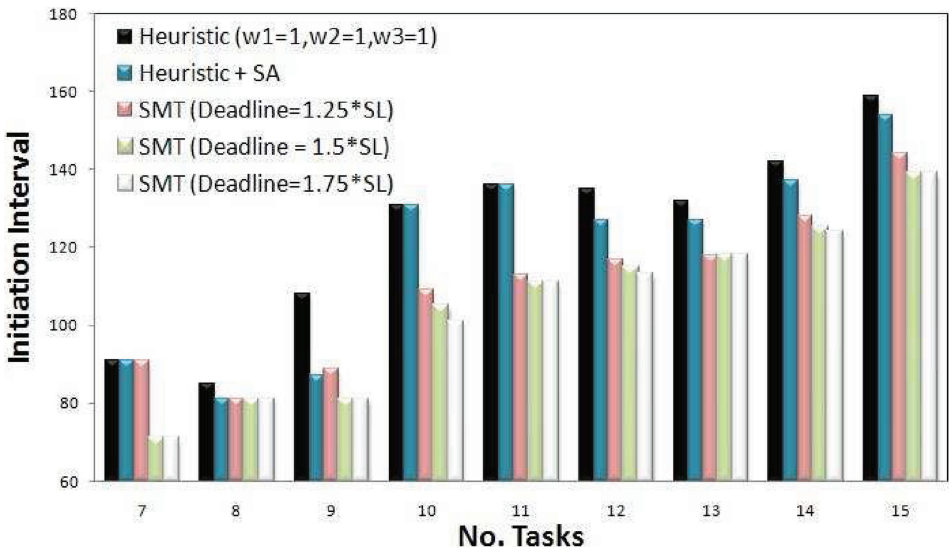


Fig. 15. Comparison of IIs obtained with SMT-based pipelined scheduling algorithm with different deadlines, and the KR-based heuristic algorithm with no deadline constraint.

energy function to be minimized, the *annealing schedule*, and the *neighborhood function* that describes how a new solution is obtained by mutating the current solution.

—The energy function to be minimized is the II of the pipelined schedule.
—The annealing schedule has a large impact on solution quality. The slower the temperature cools down, the solution quality gets better, but the algorithm running time also gets longer. Hence there is a trade-off between optimization quality and running time of the optimization algorithm. In our experiments, we set the initial temperature to be $T_0 = 1000$, and the temperature decrease per step as $\delta t = 2$. (We also tried slower cooling schedules, but did not observe any appreciable difference.)
—For the neighborhood function, we randomly choose among one of the three choices given next with equal probability.
  —Swap any two values among the three variables w_1, w_2, w_3.
  —Randomly assign a new value in the continuous range of [0,10] to one of w_1, w_2, w_3.
  —Randomly assign new values in the continuous range of [0,10] to all 3 variables w_1, w_2, w_3.

Figure 15 compares the IIs obtained with different solution techniques. We make the following observations.

—Increasing the task graph deadline in the SMT-based method can help improve the throughput (reduce the II), since a looser deadline allows the scheduler more freedom to pack the tasks more tightly in the steady state of the

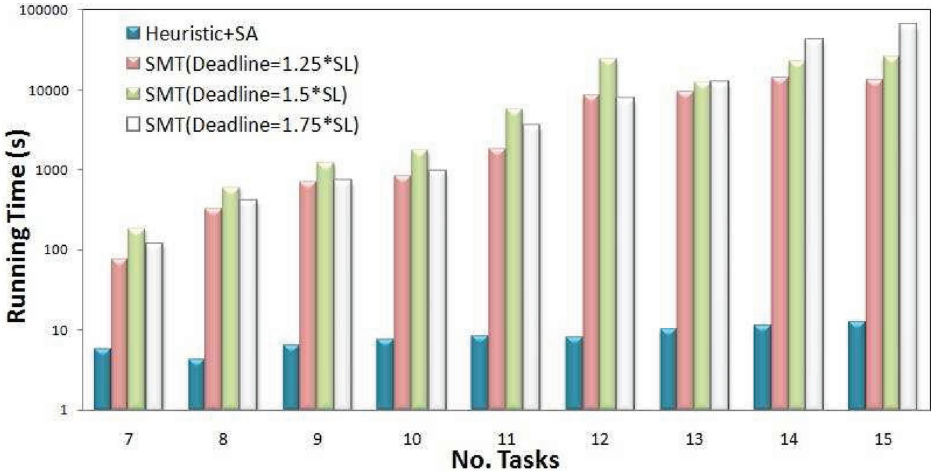


Fig. 16. Comparison of algorithm running times for pipelined scheduling. The y-axis is drawn in log-scale.

pipeline. (As we will show in Figure 19, there is a threshold deadline value, and increasing the deadline above it no longer helps to reduce II.)

—As mentioned in Section 3.3, we assume that no deadlines are associated with the task graph for the KR-based heuristic, hence it is theoretically possible for the heuristic algorithm to achieve higher throughput than the SMT-based method. However, even with this relaxed assumption, the KR-based heuristic algorithm generates worse schedules (with larger II) than the scheduled obtained with SMT. The magnitude of difference is not very large, that is, the KR-based heuristic algorithm achieves reasonably good performance.

—SA can be used to optimize settings of the weight parameters (w_1, w_2, w_3 in Eq. (20)), but the performance improvement over the that for the setting $w_1 = w_2 = w_3 = 1$ is limited. We thus conclude that the heuristic algorithm is not very sensitive to the weight parameter settings (within a certain range), and the setting of $w_1 = w_2 = w_3 = 1$ is a reasonable one.

Figure 16 shows how algorithm running time varies with increasing task graph size (number of tasks) and different task graph deadlines. For the SMT-based method, the maximum task graph size that can be handled within an upper bound of 10000 seconds (about 3 hours) is about 15. Comparing this to the upper bound of 22–23 tasks for HW/SW partitioning, we conclude that the more complex constraint set of the pipelined scheduling problem makes it harder and more time consuming to solve than the HW/SW partitioning problem. The running time of the SA algorithm shows that it is quite fast and scalable (for the chosen cooling schedule). Running times of the KR-based heuristic algorithm are very short and generally fall within 20ms, hence they are not shown in the figure.

5.2.1 *The Unsharp Masking Application.* In this section, we use an application example from Noguera and Badia [2005, 2006] for the performance evaluation of our pipelined scheduling algorithm.

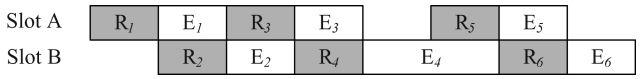


Fig. 17. The 2-slot FPGA task model. R denotes reconfiguration, and E denotes execution. All tasks are assumed to be different from each other, so there is no configuration reuse.

Noguera and Badia [2006] built a HW device based on Xilinx FPGAs called the *Galapagos prototyping platform*. The HW architecture consists of a general-purpose CPU, an array of Dynamically Reconfigurable Processors (DRPs) and shared memory resources: a small on-chip L2 cache memory and a large off-chip DRAM memory. Each DRP is implemented on a Xilinx Virtex-II FPGA using its partial reconfiguration capability. A HW task is placed at the center of the FPGA, which is dynamically reconfigurable; and the load/store units are placed on the left and right sides, which are fixed and not dynamically reconfigurable. This task model is a special case of the more general task model addressed in this article, where each task can be placed on an arbitrary position on the FPGA, except one minor difference in assumptions: the Galapagos platform allows simultaneous reconfiguration of multiple DRPs, since each DRP is a separate FPGA, while we consider commercial FPGAs with a single reconfiguration controller, hence only one HW task can be configured at any give time. We can implement the task model that is (almost) equivalent to the Galapagos platform (except for the different assumptions on the reconfiguration controller) on a single PRTR FPGA by dividing the FPGA area into several equal-sized slots or partitions, each corresponding to a DRP. This more restrictive task model also has the benefit of making it easier to achieve intertask communication, as *bus macros* can be placed at interslot boundaries for communication. As a real example of this task model, Dittmann [2007] implemented a 2-slot FPGA prototype, where HW tasks in the two slots are configured and executed in a pipelined manner so that task reconfiguration delays are hidden, either completely or partially, as shown in Figure 17. The task model used for the experiments in this section can be considered as a generalization of the 2-slot model to multiple slots. (In this section, we use the term "DRP" to be consistent with the terminology in Noguera and Badia [2006], but the terms "slot" or "partition" on the FPGA can also be used in its place.)

Noguera and Badia [2006] explored system-level power-performance trade-offs when implementing streaming applications on the Galapagos platform. If the number of HW tasks exceeds the number of available DRPs, then a HW-supported dynamic scheduler is used to time-multiplex tasks on the available DRPs. Data partitioning is used to split a HW task into multiple data-parallel tasks to reduce size of the data buffers at a cost of increased reconfiguration delay. The input dataset is partitioned into several data blocks of a given size, and the task graph must be iterated as many times as the number of input data blocks. Since off-chip memory consumes more power than on-chip cache, different data block sizes can result in different trade-offs of power consumption versus performance. A smaller-size data block can fit into on-chip memory, so the resulting implementation has lower power consumption. But reconfiguration delay is more significant compared to task execution time, which may force

Table V. Task Reconfiguration Delay (for the XC2V250 model of Xilinx Virtex-II FPGA) and Execution Times on Each DRP for the Task Graph in Figure 18 for Different Block Sizes

	Reconfig Delay	T_1	T_2	T_3	T_4	T_5
64×64	0.949	0.201	0.069	0.135	0.135	0.205
256×256	0.949	3.275	1.092	2.181	2.181	3.278

All time units are in ms.

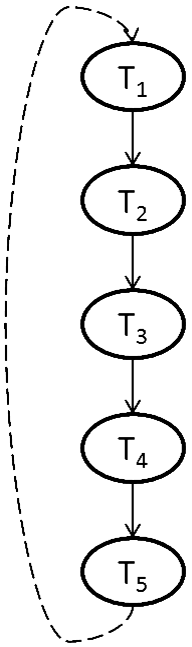


Fig. 18. Task graph for the unsharp masking application. Here are the actual task names shown in parentheses: T_1(RGB2YCrCb), T_2(Blur), T_3(Sub), T_4(Add), T_5(YCrCb2RGB).

some tasks to be mapped to the CPU to avoid dynamic HW reconfiguration. A larger-size data block may need to be stored in off-chip memory, so the system power consumption is higher. Since task execution time is longer, reconfiguration delay is less significant compared to task execution time, making it feasible to use HW dynamic reconfiguration to time-multiplex more tasks on the DRPs. Noguera and Badia [2005] presented some task graph transformations to further improve performance and/or power consumption.

Consider the task graph in Figure 18 with task execution times in Table V for an image processing application called *unsharp masking* in Noguera and Badia [2006]. The authors evaluated power-consumption versus performance trade-offs of implementing the task graph in Figure 18 on the Galapagos platform with 1 CPU plus a number of available DRPs. Task reconfiguration times on each DRP are different for different models of Xilinx Virtex-II FPGA: 0.949ms for XC2V250, 1.087ms for XC2V500, and 1.337ms for XC2V1000.[3] The

[3]The HW reconfiguration times are relatively small, perhaps due to the small size of task bitstreams, which are not specified in the article.

Table VI. Comparison of Total Execution Times Taken to Finish All 9 Task Graph Iterations

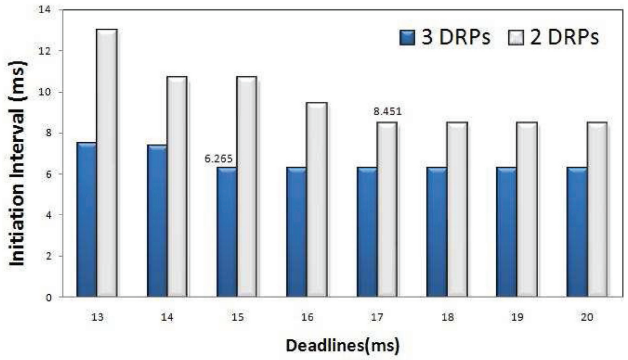
N.D.	(256 × 256, Noguera)	(256 × 256, SMT_P)	(64 × 64, SMT_P)	(256 × 256, SMT_NP)	(64 × 64, SMT_NP)
2	105	84.6	584	116.6	625.0
3	100	65.1	459	108.1	468.9

All time units are in *ms*. N.D. denotes number of DRPs. Columns with label *SMT_P* denote the results from SMT-based pipelined scheduling; columns with label *SMT_NP* denote results from SMT-based nonpipelined scheduling to minimize makespan with the encoding in Section 2.

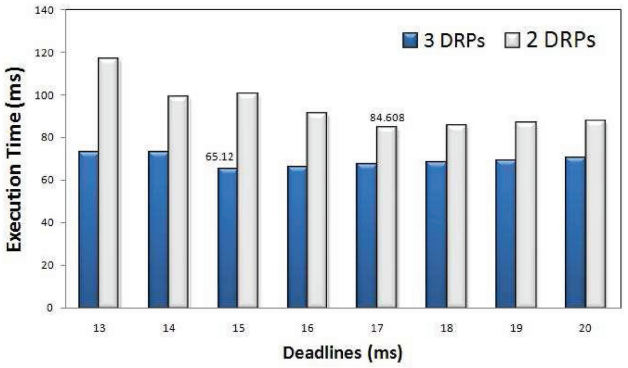
reconfiguration times are different for different FPGA models, since the HW task occupies different area sizes on the different FPGA models; but task execution times are the same, since the task bitstream size is the same, and the DRP runs at the same frequency (60 MHz). Consider an image with size 768x768 (pixels). If the block size processed by each task iteration is 64 × 64, then the task graph is iterated 144 times; if the block size is 256 × 256, then the task graph is iterated 9 times. Although pipelined scheduling is not explicitly mentioned in Noguera and Badia [2005, 2006], the resulting schedule produced by the online scheduling algorithm exhibits a pipeline pattern where multiple pipeline stages may share a single processing resource using dynamic reconfiguration. This application is an good candidate for our pipelined scheduling algorithm with the objective of finishing the "batch job" of all task graph iterations as soon as possible by overlapping execution of different task graph iterations.

Several implementation alternatives were considered in Noguera and Badia [2006], with different block sizes and number of available DRPs in the HW platform. For performance comparison, we applied our SMT-based pipelined scheduling algorithm to the implementation alternative of mapping all tasks to reconfigurable HW with either 2 or 3 DRPs, with block size of 256 × 256. Since there are 5 tasks in the task graph, dynamic reconfiguration is needed to schedule them on the platform of 2 or 3 DRPs. (We ignore the issue of power consumption, and focus on system performance only.) Table VI shows that our approach *(256 × 256, SMT_P)* can indeed achieve higher throughput, hence smaller total execution times than those in Noguera and Badia [2006] *(256 × 256, Noguera)* for either 2 or 3 DRPs.[4] Obviously, for the same block size, the improvement in total execution times will become more significant with a larger number of task graph iterations. For block size of 64 × 64, the reconfiguration delay is relatively large compared to task execution times, so the SMT-based pipelined scheduling *(64 × 64, SMT_P)* results in much longer total execution time than that block size of 256 × 256. We make the observation that pipelined scheduling that involves frequent dynamic reconfiguration can be feasible if

[4]We achieved better results despite our assumption of a single reconfiguration controller, which means that reconfiguration stages of different tasks cannot overlap, while the HW platform in Noguera and Badia [2006] does not have this restriction. It is also possible to formulate a SMT encoding to minimize the total execution time of the completely unrolled task graph treating all task instances as distinct tasks, but this approach is not scalable with a large number of task graph iterations.



(a) effect of task graph deadline on the II.



(b) effect of task graph deadline on total execution time.

Fig. 19. Effects of task graph deadline on the II and total execution time. The numbers on top of certain columns denote the minimum values among different deadline settings.

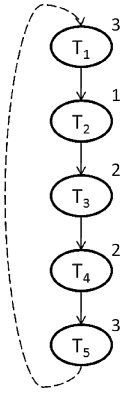
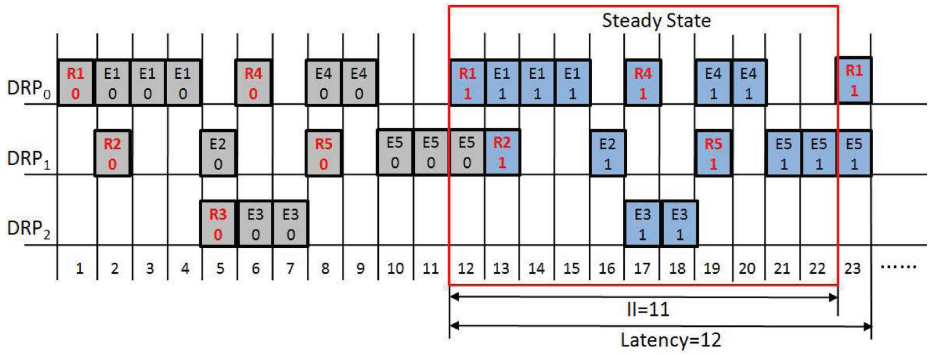
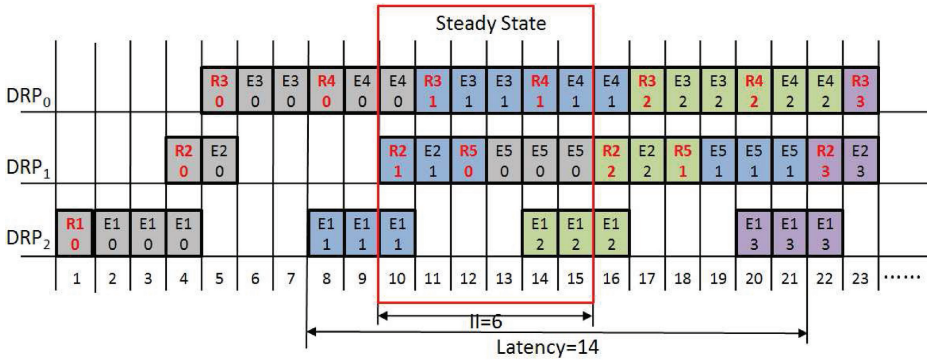


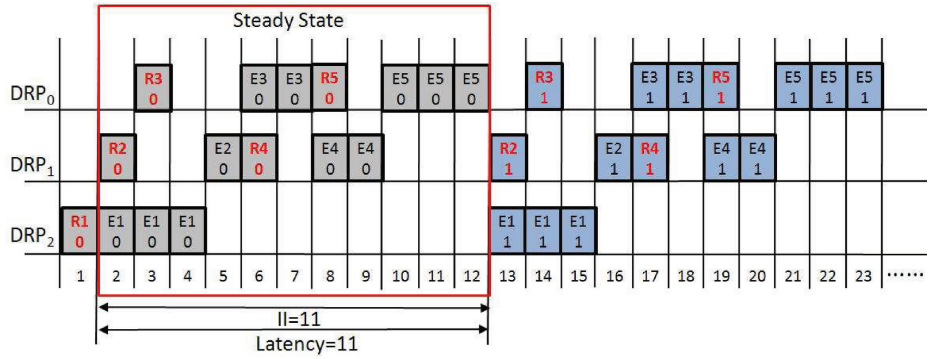
Fig. 20. Linear task graph for the unsharp masking application with task execution times simplified to integers.



(a) schedule from Fig. 2 in (Noguera and Badia 2005) (slightly changed).



(b) schedule obtained from SMT-based pipelined scheduling



(c) schedule obtained from SMT-based non-pipelined scheduling

Fig. 21. Schedules of the task graph in Figure 20 on a HW platform with 3 DRPs. The Label R denotes task reconfiguration, and E denotes task execution. Each number in the box denotes the task graph iteration number, counting from 0. This figure is best viewed in color.

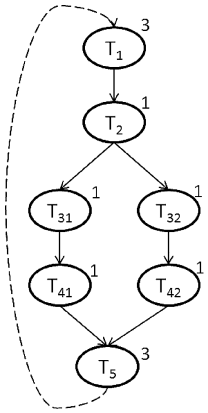
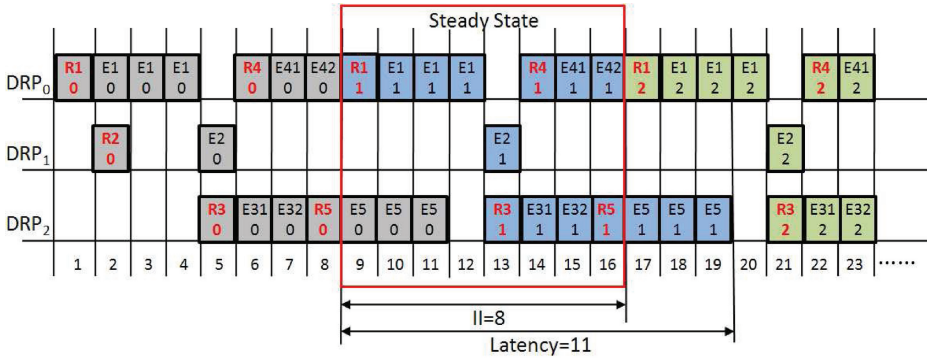


Fig. 22. Task graph for the unsharp masking application, where T_3 and T_4 are each split into 2 data-parallel tasks.

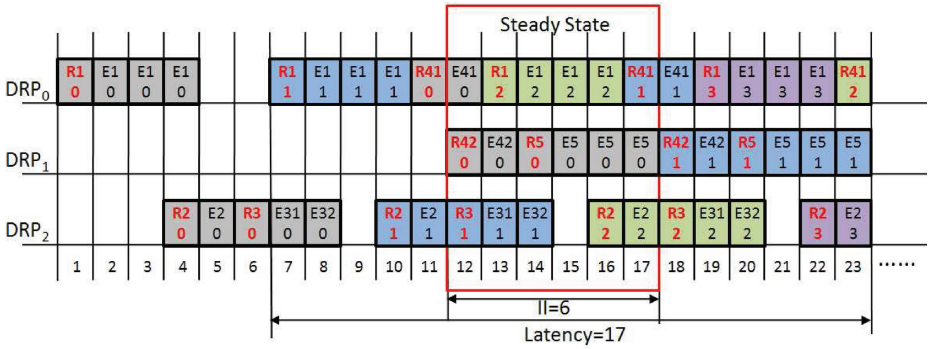
HW reconfiguration delay is relatively small compared to task execution times, for example, for block size of 256×256, but not when HW reconfiguration delay is relatively large, for instance, for block size of 64×64. Hence it is more appropriate for coarse-grained tasks, such as, when each task is a time-consuming signal processing algorithm that processes a large block of data at each iteration, but not for fine-grained tasks, such as, parallel HW implementation of a program loop in C/C++ with a few instructions in the loop body.

For comparison purposes, we also include results from SMT-based nonpipelined scheduling to minimize latency (makespan) of a single iteration with the encoding in Section 2 in columns with label *SMT_NP* in Table VI. Since different task graph iterations do not overlap in the nonpipelined schedule, the total execution time in this case is equal to the task graph latency multiplied by the number of iterations (plus some possible additional reconfiguration delays at initialization time). Compared to pipelined scheduling, we can see that nonpipelined scheduling results in much longer total execution times for block size of 256×256, but only slightly longer total execution times for block size of 64×64. We offer the following explanation: the reconfiguration delay for block size of 64×64 is relatively large compared to task execution times, and the single reconfiguration controller on the FPGA forces the reconfiguration stages of different task instances (in the same or different task graph iterations) to be serialized; therefore, there is limited opportunity for overlapped execution between different task graph iterations, hence the advantages of pipelined scheduling over nonpipelined scheduling become less pronounced.

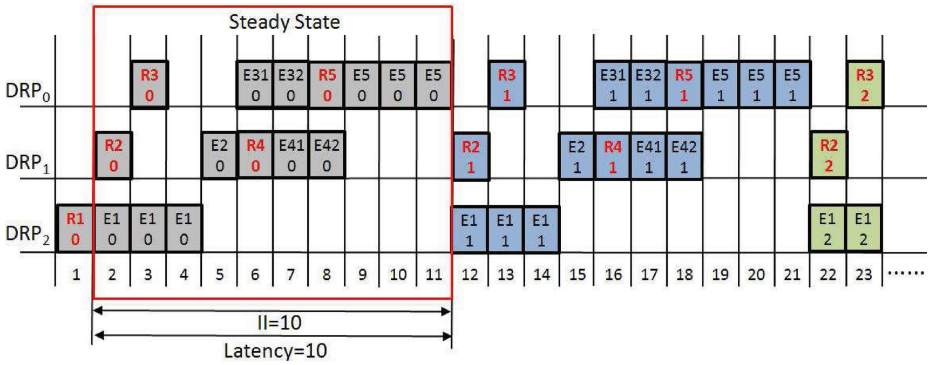
The total execution time taken for finishing execution of a fixed number of task graph iterations is dependent on not only the steady-state throughput (inverse of II), but also the task graph latency, which affects the time taken to "drain the pipeline" when finishing the last iteration of the task graph execution. Figure 19 shows that for a given HW platform, there is a threshold deadline value that results in the minimum II, and perhaps a different deadline value that results in the minimum total execution time. Increasing deadline



(a) schedule from Fig. 3 in (Noguera and Badia 2005) (slightly changed).



(b) schedule obtained from SMT-based pipelined scheduling



(c) schedule obtained from SMT-based non-pipelined scheduling

Fig. 23. Schedules of the task graph in Figure 22 on a HW platform with 3 DRPs. This figure is best viewed in color.

above each threshold value no longer helps reduce II or the total execution time, respectively. (The total execution times in Table VI were obtained with the optimal deadline setting that results in the minimum total execution time.)

Figures 21 and 23 compare the schedules from Noguera and Badia [2005] with the SMT-based pipelined and nonpipelined schedules. (Task execution times are written on the side of tasks. The task execution times and DRP reconfiguration delays are set to integer values to make the figures clearer, but their relative sizes are similar to those of the actual application.) We can see that the SMT solver-based method can indeed achieve higher HW utilization, hence higher system throughput. (The schedules in Figures 21(a) and 23(a) are slightly different from those in Noguera and Badia [2005]: Figure 21(a) is a slightly optimized schedule after removing some unnecessary slack in Figure 2 in Noguera and Badia [2005]; Figure 23(a) is different from Figure 3 in Noguera and Badia [2005] to account for our assumption of a single reconfiguration controller, so that reconfiguration stages of different tasks cannot overlap.)

6. CONCLUSIONS

FPGAs are widely used in today's embedded systems design due to their low cost, high performance, and reconfigurability. In this article, we address two problems related to HW task scheduling on Partially RunTime Reconfigurable (PRTR) FPGAs: (1) HW/SW partitioning. Given an application in the form of a task graph with known execution times on the HW (FPGA) and SW (CPU), and known area sizes on the FPGA, find an valid allocation of tasks to either HW or SW and a static schedule with the optimization objective of minimizing: (1) the total schedule length (makespan). (2) Pipelined scheduling. Given an input task graph, construct a pipelined schedule on a PRTR FPGA with the goal of maximizing system throughput while meeting a given end-to-end deadline. A performance evaluation with both synthetic random task graphs and a real application example demonstrates the effectiveness of our techniques.

REFERENCES

AHN, M., YOON, J . W., PAE, K. Y., KIM, Y., KIEMB, M., AND CHOI, K. 2006. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'06)*. G. G. E. Gielen Ed., European Design and Automation Association, 363–368.

AIKEN, A., NICOLAU, A., AND NOVACK, S. 1995. Resource-Constrained software pipelining. *IEEE Trans. Parall. Distrib. Syst. 6*, 12, 1248–1270.

ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. 1995. Software pipelining. *ACM Comput. Surv. 27*, 3, 367–432.

BANERJEE, S., BOZORGZADEH, E., AND DUTT, N. D. 2005. Physically-Aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *Proceedings of the ACM IEEE Design Automation Conference (DAC'05)*. 335–340.

BANERJEE, S., BOZORGZADEH, E., AND DUTT, N. D. 2006. Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration. *IEEE Trans. VLSI Syst. 14*, 11, 1189–1202.

CABODI, G., KONDRATYEV, A., LAVAGNO, L., NOCCO, S., QUER, S., AND WATANABE, Y. 2005. A bmc-based formulation for the scheduling problem of hardware systems. *Int. J. Softw. Tools. Technol. Transfer. 7*, 2, 102–117.

CARPENTER, J., FUNK, S., HOLMAN, P., SRINIVASAN, A., ANDERSON, J., AND BARUAH, S. 2004. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. Chapman and Hall/CRC, 30–1– 30–19.

CLAUS, C., MULLER, F. H., ZEPPENFELD, J., AND STECHELE, W. 2007. A new framework to accelerate virtex-ii pro dynamic partial self-reconfiguration. In *Proceedings of the 14th Reconfigurable Architectures Workshop.*

CUOCCIO, A., GRASSI, P. R., RANA, V., SANTAMBROGIO, M. D., AND SCIUTO, D. 2008. A generation flow for self-reconfiguration controllers customization. In *Proceedings of the IEEE International Symposium on Electronic Design, Test and Applications (DELTA'08).* IEEE Computer Society, 279–284.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Comm. ACM 5*, 7, 394–397.

DEHON, A., MARKOVISKIY, Y., CASPI, E., CHU, M., HUANG, R., PERISSAKIS, S., POZZI, L., YEH, J., AND WAWRZYNEK, J. 2006. Stream computations organized for reconfigurable execution. *Microprocess. Microsyst. 30*, 6, 334–354.

DICK, R. P., RHODES, D. L., AND WOLF, W. 1998. TGFF: Task graphs for free. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES'98),* G. Borriello, A. A. Jerraya, and L. Lavagno Eds., IEEE Computer Society, Los Alamitos, CA, 97–101.

DITTMANN, F. 2007. Methods to exploit reconfigurable fabrics. Ph.D. dissertation, University of Paderborn, Germany.

DUTERTRE, B. AND DE MOURA, L. M. 2006. A DPLL(T). In *Proceedings of the International Conference on Computer Aided Verification (CAV'06)*. T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 81–94.

FEKETE, S. P., KÖHLER, E., AND TEICH, J. 2001. Optimal FPGA module placement with temporal precedence constraints. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'01).* 658–667.

GOLDSTEIN, S. C., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. R. 2000. Piperench: A reconfigurable architecture and compiler. *IEEE Comput. 33*, 4, 70–77.

JIN, Y., SATISH, N., RAVINDRAN, K., AND KEUTZER, K. 2005. An automated exploration framework for fpga-based soft multiprocessor systems. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES+ISSS'05)*. P. Eles, A. Jantsch, and R. A. Bergamaschi Eds., ACM Press, New York, 273–278.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

KLEIN, M., RALYA, T., POLLAK, B., OBENZA, R., AND HARBOUR, M. G. 1993. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Springer.

KUDLUR, M. AND MAHLKE, S. A. 2008. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. R. Gupta and S. P. Amarasinghe Eds., ACM Press, New York, 114–124.

LI, Z. AND HAUCK, S. 2002. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'02)*. 187–195.

LIN, Y., KUDLUR, M., MAHLKE, S. A., AND MUDGE, T. N. 2007. Hierarchical coarse-grained stream compilation for software defined radio. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'07)*, T. Kim, P. Sainrat, S. S. Lumetta, and N. Navarro Eds., ACM Press, New York, 115–124.

LIU, M., KUEHN, W., LU, Z., AND JANTSCH, A. 2009. Run-Time partial reconfiguration speed investigation and architectural design space exploration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'09)*.

MEMIK, S. O. AND FALLAH, F. 2002. Accelerated SAT-based scheduling of control/data flow graphs. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'02)*. IEEE Computer Society, Los Alamitos, CA, 395.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'01)*. ACM Press, New York, 530–535.

NOGUERA, J. AND BADIA, R. M. 2005. Performance and energy analysis of task-level graph transformation techniques for dynamically reconfigurable architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05)*. T. Rissa, S. J. E. Wilton, and P. H. W. Leong Eds., IEEE Press, 563–567.

NOGUERA, J. AND BADIA, R. M. 2006. System-Level power-performance tradeoffs for reconfigurable computing. *IEEE Trans. VLSI Syst. 14*, 7, 730–739.

PAPADIMITRIOU, C. H. AND STEIGLITZ, K. 1998. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications.

SHANG, L., DICK, R. P., AND JHA, N. K. 2007. Slopes: Hardwarecsoftware cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable fpgas. *IEEE Trans. VLSI 26*, 3.

SUHENDRA, V., RAGHAVAN, C., AND MITRA, T. 2006. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. S. Hong, W. Wolf, K. Flautner, and T. Kim Eds., ACM Press, New York, 401– 410.

XILINX. 2005. Xilinx virtex-4 family overview. preliminary specification ds112. Tech. rep., Xilinx Inc.

YUAN, M, HE, X., AND GU, Z. 2008. Hardware/Software partitioning and static task scheduling on runtime reconfigurable fpgas using a smt solver. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 295–304.

Received November 2008; revised November 2009; accepted December 2009