Hardware/Software Partitioning and Static Task Scheduling on Runtime Reconfigurable FPGAs using a SMT Solver

Mingxuan Yuan, Xiuqiang He and Zonghua Gu
Dept. of Computer Science and Engineering
Hong Kong University of Science and Technology

Abstract

FPGAs are often used together with a CPU as hardware accelerators. A runtime reconfigurable FPGA allows part of the FPGA area to be reconfigured while the remainder continues to operate without interruption, so that hardware tasks can be placed and removed dynamically at runtime. In this paper, we formulate and solve the problem of optimal hardware/software partitioning and static task scheduling for a hybrid FPGA/CPU device, with the optimization objective of minimizing the total schedule length, in the framework of Satisfiability Modulo Theories (SMT) with Linear Integer Arithmetic.

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, 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. (In this paper, we use the term *FPGA* to refer to *PRTR FPGA*.) The task scheduler and placer are needed to find empty space to place a new task, and recycle the occupied area when a task is finished while making sure all task deadlines are met.

Current commercial FPGA technology, e.g., Xilinx Virtex-4, supports both *1D reconfiguration*, where each task occupies a contiguous set of columns, and *2D reconfiguration*, where each task occupies a rectangular area with a width and height in terms of number of CLBs on the two dimensions. Real-time scheduling for 1D reconfigurable FPGAs shares many similarities with global scheduling on identical multi-processors, where all processors in the system have identical processing speed, and different task invocation instances may run on different processors. Similarly, a task can be relocated to a different position on the FPGA at runtime, with the associated reconfiguration overhead. But FPGA scheduling is a more general and difficult problem than multi-processor scheduling, since each HW task may occupy a different area size on the FPGA while a SW task always occupies one and only one CPU. In fact, we can view multiprocessor scheduling as a special case of HW task scheduling on a 1D reconfigurable FPGA where all tasks have unit width.

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 that is proportional to size of the area being reconfigured. Several techniques have been proposed to reduce the impact of reconfiguration overhead. In this paper, we consider *configuration prefetch* [1], a 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 computation 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 computation stage.

FPGAs may be used in place of ASICs as co-processors, also called HW accelerators, for computation-intensive kernels in the application. As HW co-processors, 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 co-processors accelerate specific functions, co-processors based on FPGAs can be applied to the speedup of arbitrary SW programs with some distinctive characteristics (e.g., programs with parallelizable bit-level operations [2]).
- 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 co-processors.

There are some commercial products to support development of FPGA-based co-processors as HW accelerators, e.g., Altera has developed HW and tools for a flexible co-processor

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.39

IEEE computer society

Figure 1. One example configuration of Altera FPGA co-processor, taken from the Altera website [3].

architecture. Fig. 1 shows one possible configuration using Altera's FPGA co-processor with Texas Instruments' digital signal processor, where the co-processor 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 of an external processor. Combined with PRTR feature of modern FPGAs, this approach provides a flexible and high-performance approach to HW acceleration.

In this paper, we address the *HW/SW partitioning* problem, where the FPGA is paired with a CPU, which may be a *softcore CPU* carved out of the configurable logic in the FPGA, e.g., MicroBlaze for Xilinx FPGAs, or a separate independent CPU connected to the FPGA via a bus. Each task has both a HW implementation that can run on the FPGA, and a SW implementation that can run on the CPU. Communication between the FPGA and CPU incurs a fairly large delay. The application is given 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. The HW/SW partitioning problem is to 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). We make the following assumptions in this paper:

- The FPGA is 1D reconfigurable.
- The FPGA has a single reconfiguration controller, which is the state-of-the-art in industry practice. The reconfiguration controller is a global shared resource, and reconfiguration stages of multiple tasks must be serialized while their execution stages can be concurrent as long as they can all fit on the FPGA. This is a major source of complexity of real-time scheduling on FPGAs.
- The entire FPGA area is uniformly reconfigurable, and each dynamic task can be flexibly placed anywhere on the FPGA area as long as there is enough empty space. In practice, it is common to pre-configure parts the FPGA area for dedicated purposes as static components where dynamic tasks cannot be placed. This situation can be handled easily in our task model by denoting these columns as always occupied. It is also possible to allocate an area of logical shared memory that spans the entire width of the FPGA and acts as a global shared communication medium [4], which reduces the height of the FPGA columns that are available to dynamic tasks but does not otherwise affect our task model.

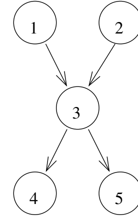
Many scheduling problems in design automation are NP-complete. Typically, efficient heuristic algorithms are devised to obtain near-optimal solutions. Researchers have also used various techniques to obtain exact solutions to these scheduling problems, including Integer Linear Programming (ILP) solvers, Constraint Programming (CP) solvers, Binary Decision Diagram (BDD) packages, Satisfiability (SAT) solvers, 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 is 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, e.g., equality with uninterpreted function symbols, linear integer arithmetic, linear real arithmetic, integer difference logic and real difference logic. In our problem formulation, we use the Linear Integer Arithmetic (LIA) theory, which can express linear inequalities involving integer variables and constants. Integer Difference Logic (IDL) is a specialized fragment of LIA that can express inequalities that constrain the difference between pairs of integer variables. (If we adopt a continuous view of time, then we can declare all time-related variables as real numbers instead of integers in the SMT solver, which can handle real arithmetic as well as integer arithmetic. Our experience shows that Yices exhibits slightly worse performance if we use real variables than if we use integers, so we declare all variables to be integers in this paper.)

Early attempts for solving SMT 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 ad-

dition. This observation led to the development of a number of SMT solvers that tightly integrate the boolean reasoning of a DPLL-style search 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 on to 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. In this paper, we use the SMT solver Yices [5] from Stanford Research Institute (SRI) to solve the HW/SW partitioning problem, as an alternative to more conventional optimization techniques such as ILP. Yices handles linear arithmetic constraints by using a Simplex-based linear arithmetic solver that is integrated efficiently in the DPLL(T) framework. The solver gains efficiency with a number of features such as fast backtracking, a priori simplification to reduce the problem size, and an efficient form of theory propagation. We refer the interested reader to [5] for details of the algorithm.

This paper is structured as follows: we first introduce the HW/SW partitioning problem in Section 2, then discuss the details of modeling with SMT and ILP in Section 3. We give performance evaluation results in Section 4, and discuss related work in Section 5. Finally, we draw conclusions in Section 6.



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

Table 1. Task graph example 1.

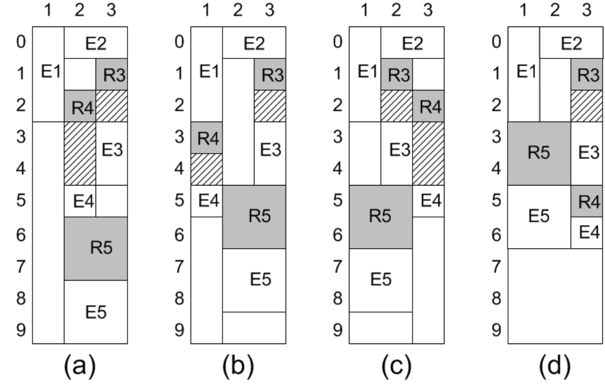


Figure 2. Some possible schedules of the task graph in Fig. 1 on a FPGA with 3 columns. The vertical axis is time, and the horizontal axis is the column position on the FPGA. Dark boxes denote the reconfiguration stage; white boxes denote the execution stage; striped boxes denote the gap between a task's reconfiguration stage and its execution stage.

2 Problem Formulation

2.1 HW Task Scheduling

A *task graph* is a directed acyclic graph where each vertex represents a task, and each edge represents precedence relationship between two tasks. Each task T_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, i.e., 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 computation stage with length E_i. A task's computation stage can start when all its predecessors have finished their computation stages, i.e., precedence relationships in the task graph constrain task computation stages, not reconfiguration stages, which can occur in arbitrary order. Communication delays between tasks can be ignored in comparison to reconfiguration delays and computation times.

Definition 1 *An **optimal schedule** of a task graph on a FPGA is the static schedule with the shortest possible length that satisfies all task precedence relationships.*

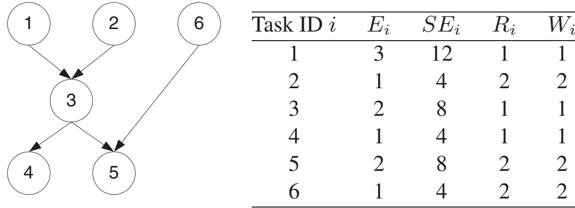
The optimal schedule may not be unique since multiple schedules may have the same length but have different task invocation sequences. Consider the problem of finding an optimal schedule of a task graph on a FPGA with 3 columns. If each task occupies 3 columns, then it's identical to a single-processor scheduling problem, treating the entire FPGA as a CPU. If each task occupies 1 column, then it's identical to a multiprocessor scheduling problem, treating each column as a CPU, and we can use existing algorithms to find the minimum-length schedule. But if each task can occupy 1, 2 or 3 columns, then it becomes a more general and difficult problem than an optimal task graph scheduling problem for multi-processors. Consider the problem of allocating the task graph in Table 1 on a FPGA with 3 columns. Fig. 2 shows four possible schedules. When a task T_i's columns have been reconfigured, they are reserved for T_i and should not be allocated to other tasks, as indicated by the shaded areas. However, T_i may not be able to start its computation stage immediately since it have to wait for all its predecessors to finish. For example, in schedule (d), reconfiguration of column C_3 for Task T_3 is finished at time 2, but T_3's computation stage E_3 cannot start until time 3, since T_1's computation stage E_1 must finish before E_3 can start, according to the task graph specification in Table 1. Shaded areas denote the gaps between a task T_i's reconfiguration and computation stages caused by precedence constraints, within which no other

task T_j should execute. Otherwise, T_i's configuration will be overwritten and it will needed to be reconfigured again.

Even though we only show one iteration of the task graph's execution, it is likely that the entire task graph is repeatedly executed periodically. *Initial tasks* T_1 and T_2 start at time 0 and do not suffering any reconfiguration delays, because we can pre-configure them on the FPGA before the start of each periodic execution. This requires us to leave enough slack between the total schedule length and end of the execution period to pre-configure the initial tasks. For example, if the schedule in Fig. 2 (c)is repeated periodically, then the task period cannot be smaller than 10, 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.

2.2 HW/SW Partitioning

Table 2. Task graph example for HW/SW partitioning.



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

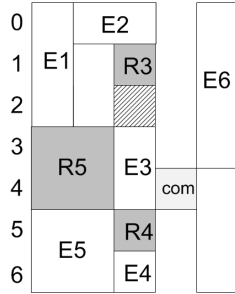


Figure 3. A schedule of the task graph in Table 2 on a hybrid FPGA/CPU device.

Fig. 3 shows one possible HW/SW partition and schedule for the task graph in Fig. 2. Each task has an additional attribute SE_i for its execution time on the CPU. 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 sends a message to HW task 5.

2.3 Other Task Models

Some variations on the task model discussed above are possible, but we will not discuss them further due to space limitations.

- **Typed Task Graphs**. Each task has a type, and tasks of the same type have identical bitstream content, so it is not necessary to reconfigure the task area if a task executes at the same location after another task of the same type finishes. This is a fairly realistic task model [6], as many applications may invoke the same functional module several times during its execution, e.g., in a *for* loop.
- **Column-Wise Reconfiguration**. The configuration controller reconfigures a task that spans multiple columns in a column-by-column manner, i.e., when one column of a task is being reconfigured, the other columns of the same task can be used for execution of other tasks. Reconfiguration of each column takes the same amount of time.

3 Modeling with SMT and ILP

As a brief review, a *Linear Programming* (LP) problem is one of maximizing or minimizing a linear function subject to linear equality and inequality constraints, with this general form:

$$\begin{gathered}
\text{Maximize } c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
\text{Subject to} \\
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\
\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\
l_1 \leq x_1 \leq u_1 \\
l_2 \leq x_2 \leq u_2 \\
\vdots \\
l_n \leq x_n \leq u_n
\end{gathered}$$

where $x_1, x_2, \cdots, x_n$ are variables and the remaining elements are input data. *Maximize* can alternatively be *Minimize*, and each $\leq$ operator can alternatively be an $=$ or a $\geq$ operator. If the unknown variables are required to be all integers, then the problem becomes an *Integer Linear Programming* (ILP) problem.

The linear arithmetic in SMT is more expressive than ILP, as it support all boolean logic operators such as *And* ($\wedge$), *Or* ($\vee$), *Not* ($\neg$) and *Implies* ($\Longrightarrow$), while ILP only supports the *And* ($\wedge$) operator in conjunctive normal form. To transform a SMT model to an equivalent ILP model, it is necessary to add additional boolean variables to cope with ILP's limited expressiveness. Next, we describe techniques for transforming a SMT model into an equivalent ILP model by turning a general boolean logic formula into conjunctive normal form with only the *And* ($\wedge$) operator acceptable to the ILP solver.

- A set of disjunctive predicates, e.g., $(x_1 \leq a_1 \vee x_2 \leq a_2 \vee \cdots \vee x_n \leq a_n)$, can be transformed into conjunctive predicates as follows:

$$\begin{aligned} x_1 - a_1 &\leq \mathit{LargeN} * v_1 \\ x_2 - a_2 &\leq \mathit{LargeN} * v_2 \\ &\vdots \\ x_n - a_n &\leq \mathit{LargeN} * v_n \\ 0 \leq v_1 &\leq 1 \\ 0 \leq v_2 &\leq 1 \\ &\vdots \\ 0 \leq v_n &\leq 1 \\ v_1 + v_2 + \cdots + v_n &\leq n - 1 \end{aligned} \tag{1}$$

  where $LargeN$ is a large integer (larger than any other value in the model), and $(v_1, v_2, \ldots, v_n)$ are newly-introduced boolean variables. $(v_1 + v_2 + \cdots + v_n \leq n-1)$ guarantees that at least one of these variables must be 0, which in turn guarantees that at least one predicate $(x_i \leq a_i, 1 \leq i \leq n)$ must be true, and the original disjunction is true.

- The expression $l_1 \vee l_2$, where one or both of l_1 or l_2 are equalities, e.g., $(x = 2 \vee y \geq 3)$, is equivalent to $(x \leq 2 \vee y \geq 3) \wedge (x \geq 2 \vee y \geq 3)$, and then each of $(x \leq 2 \vee y \geq 3)$ and $(x \geq 2 \vee y \geq 3)$ can be transformed into conjunctive form.

- The expression $x \neq y$, where x and y are both integers, is equivalent to $(x \leq y - 1 \vee x \geq y + 1)$, which can in turn be transformed into conjunctive form.

- The *Implies* ($\Longrightarrow$) operator can be transformed into disjunctive form and then into conjunctive form, e.g., the expression $(x = 2 \implies y \geq 3)$ is equivalent to $(x \neq 2 \vee y \geq 3)$, which is in turn equivalent to $(x \leq 1 \vee x \geq 3 \vee y \geq 3)$, which can then be transformed into conjunctive form.

- All variables are assumed to be non-negative. We add these constraints without explicitly mentioning them in the rest of the paper.

3.1 SMT Formulation of HW Task Scheduling

In this rest of the paper, we follow the convention of using lower-case letters to denote variables, and upper-case letters to denote constants.

Consider a task graph with N tasks. For each task $i, 1 \leq i \leq N$, we define the following constants:

- R_i: HW task i's reconfiguration time on the FPGA.
- E_i: HW task i's execution time on the FPGA.
- W_i: HW task i's size in terms of the number of contiguous FPGA columns it occupies.
- SL: Upper bound of the schedule length of the task graph

As shown in Fig. 4, each HW task occupies a rectangular area in the time-FPGA position chart, which consists of three parts: reconfiguration stage, execution stage, and possibly a gap between them due to configuration prefetching. For each task $i, 1 \leq i \leq N$, we define the following variables:

- pl_i: position of task i's leftmost column on the FPGA.
- tr_i: start time of task i's reconfiguration stage.
- te_i: start time of task i's execution.
- r_i: actual reconfiguration time of task i contributing to the schedule length. For tasks that are not source nodes, $r_i = R_i$. For a task i that is a source node in the task graph (it has no predecessors in the task graph), $r_i = 0$ if task i starts execution at time 0, and $r_i = R_i$ otherwise, as discussed in Section 1 and shown in Fig. 2.

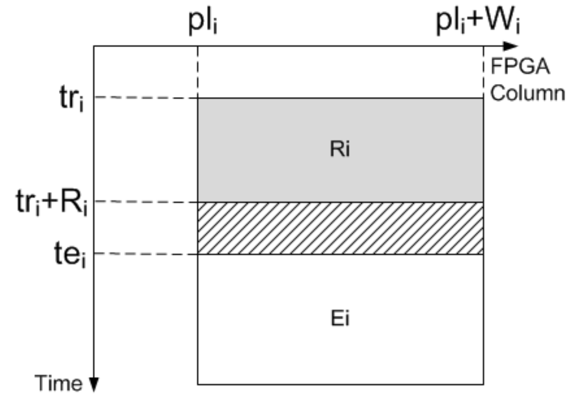


Figure 4. HW task i on the 2D time-FPGA position chart.

We define the following constraints:

- **C1** A task must fit on the FPGA's area:

$$\forall i \quad (pl_i \geq 1 \quad \wedge \quad pl_i \leq N - W_i + 1) \tag{2}$$

- **C2** If a task is a source node in the task graph and it starts at time 0, then it does not need the reconfiguration stage (since the reconfiguration stage should have been finished before time 0, as discussed in Section 1); otherwise, its execution stage can only start after its reconfiguration stage:

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

- **C3** A task can only start its execution stage after its reconfiguration stage. (This holds true even for the source tasks that do not need the reconfiguration stage, since their $r_i = 0$):
$$\forall i \qquad te_i \geq tr_i + r_i \tag{4}$$

- **C4** Two tasks can overlap with each other on either the vertical time axis or the horizontal position axis, but not both:

$$\begin{aligned} \forall i, j, \qquad & te_i + E_i \leq tr_j \\ \vee \quad & te_j + E_j \leq tr_i \\ \vee \quad & pl_i + W_i \leq pl_j \\ \vee \quad & pl_j + W_j \leq pl_i \end{aligned} \tag{5}$$

- **C5** The reconfiguration controller is a shared resource, so reconfiguration stages of different tasks must be serialized:

$$\begin{aligned} \forall i, j, \qquad & tr_i + r_i \leq tr_j \\ \vee \quad & tr_j + r_j \leq tr_i \end{aligned} \tag{6}$$

- **C6** 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. (We do not need to explicitly encode the precondition "there is an edge from task i to task j", since the constraint set is generated automatically from a given task graph, and the knowledge about the task graph is built into the code generator.):

$$te_i + E_i \leq te_j \tag{7}$$

- **C7** Sink tasks must finish before SL:

$$\forall i (\text{task } i \text{ is a sink node}) \quad te_i + E_i \leq SL \tag{8}$$

Equations 2 to 8 form a constraint set that can be fed into a SMT solver to determine feasibility.

Algorithm 1 Top-level binary search algorithm when using SMT.

1: $l = LB$
2: $u = UB$
3: **while** $l < u - 1$ **do**
4: $\quad SL = (l + u)/2$
5: $\quad hasSolution$:=InvokeSMTSolver(GenSMTModel(SL))
6: $\quad$ **if** $hasSolution$ **then**
7: $\quad\quad u := SL$, and record the task schedule.
8: $\quad$ **else**
9: $\quad\quad l := SL$
10: $\quad$ **end if**
11: **end while**
12: return u as the minimum schedule length, along with the corresponding task schedule.

One drawback of SMT compared to ILP is that it does not support optimization directly, but only provides a yes/no answer to the feasibility of a given constraint set, so we need to 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, i.e., the longest delay path from the source tasks 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 our automatic code generator that takes as input the schedule length SL currently being checked for feasibility, and generates the SMT problem instance as input to Yices.

3.2 ILP Formulation of HW Task Scheduling

In this section, we present the ILP formulation of the 7 constraints discussed in Section 3.1. We need to transform the constraint set in SMT to an equivalent constraint set in ILP. We focus on constraints **C2**, **C4** and **C5**, whose SMT formulation involves the *Or* ($\vee$) operator not supported by ILP, while the other four constraints can be directly supported by ILP without modification.

- **C2**(Equation 3) We introduce a boolean variable $bInitial_i$ for each task i to indicate if task i is an initial task, i.e., it starts its execution at time 0. The constraint $(r_i = 0 \vee r_i = R_i)$ can be transformed into $r_i = R_i * bInitial_i$. The constraint $(r_i = 0 \implies te_i = 0)$ can be first transformed into $(r_i \geq 0 \vee te_i \leq 0)$, (since all variables are assumed to be non-negative integers), which is equivalent to the constraint set:

$$\begin{aligned} te_i &\leq LargeN * br_{i1} \\ 1 - r_i &\leq LargeN * br_{i2} \\ br_{i1} + br_{i2} &\leq 1 \end{aligned} \tag{9}$$

- **C4** (Equation 5): we introduce new boolean variables bt_{ij}, bt_{ji}, bp_{ij}, bp_{ji}:

$$\begin{aligned} te_i + E_i - tr_j &\leq LargeN * bt_{ji} \\ te_j + E_j - tr_i &\leq LargeN * bt_{ij} \\ pl_i + W_i - pl_j &\leq LargeN * bp_{ji} \\ pl_j + W_j - pl_i &\leq LargeN * bp_{ij} \\ bt_{ji} + bt_{ij} + bp_{ji} + bp_{ij} &\leq 3 \end{aligned} \tag{10}$$

- **C5** (Equation 6): we introduce new boolean variables br_{ij}, br_{ji}:

$$
\begin{aligned}
tr_i + r_i - tr_j &\leq LargeN * br_{ji} \\
tr_j + r_j - tr_i &\leq LargeN * br_{ij} \\
br_{ji} + br_{ij} &\leq 1
\end{aligned} \tag{11}
$$

Since ILP supports optimization directly, we no longer need to use the binary search algorithm 1, but can use the ILP solver to obtain SL, the minimum schedule length directly. Instead of a constant in the SMT formulation, SL is now a variable to be minimized as our optimization goal. The constraint set can be fed into an ILP solver to solve for SL.

3.3 SMT Formulation of HW/SW Partitioning

In this section, we present an extension of the SMT model in Section 3.1 for HW/SW partitioning on a hybrid FPGA/CPU device. (Since the equivalent ILP model can be derived from the SMT model, we omit the discussion of the ILP model.) We need to define some additional constants and variables in addition to those defined in Section 3.1. For convenience, we list all of them instead of only the additional ones.

For each task $i, 1 \leq i \leq N$, we define the following constants:

- R_i: HW task i's reconfiguration time on the FPGA.
- E_i: HW task i's execution time on the FPGA.
- W_i: HW task i's size in terms of the number of contiguous FPGA columns it occupies.
- SE_i: SW task i's execution time on the CPU.

In addition, we define a new constant $ComDelay$ to represent the communication delay between the FPGA and CPU.

For each task $i, 1 \leq i \leq N$, we define the following variables:

- pl_i: position of task i's leftmost column on the FPGA.
- tr_i: start time of task i's reconfiguration stage.
- te_i: start time of task i's execution.
- tf_i: finish time of task i's execution.
- r_i: actual reconfiguration time of task i contributing to the schedule length.
- hs_i: a boolean variable. If $hs_i = 0$, task i is a HW task on the FPGA; if $hs_i = 1$, task i is a SW task on the CPU.

The new formulation of the SMT constraints in the context of HW/SW partitioning differ from those for task graph scheduling by adding enabling conditions for each existing constraint based on whether the task is a HW task on the FPGA, or the SW task on the CPU, and adding two new constraints **C8** annd **C9**.

- **C1** This only applies to HW tasks:

$$
\forall i \quad hs_i = 0 \implies (pl_i \geq 1 \quad \wedge \quad pl_i \leq N - W_i + 1) \tag{12}
$$

- **C2** This applies to both HW and SW tasks. Compared to Equation 3, this condition says that a HW task that starts at time 0, or a SW task, does not need to suffer the reconfiguration delay.

$$
\begin{aligned}
&\forall i, (\text{task } i \text{ is not a source node}) \quad r_i = R_i \\
&\forall i, (\text{task } i \text{ is a source node}) \\
&\quad (r_i = 0 \vee r_i = R_i) \quad \wedge \\
&\quad (r_i = 0 \implies (te_i = 0 \vee hs_i = 0))
\end{aligned} \tag{13}
$$

- **C3** This only applies to HW tasks:

$$
\forall i \quad hs_i = 0 \implies (te_i \geq tr_i + r_i) \tag{14}
$$

- **C4** This only applies to HW tasks:

$$
\begin{aligned}
\forall i, j, hs_i = 0 \wedge hs_j = 0 \implies & \\
& (tf_i \leq tr_j \\
\vee \quad & tf_j \leq tr_i \\
\vee \quad & pl_i + W_i \leq pl_j \\
\vee \quad & pl_j + W_j \leq pl_i)
\end{aligned} \tag{15}
$$

- **C5** This only applies to HW tasks:

$$
\begin{aligned}
\forall i, j, hs_i = 0 \wedge hs_j = 0 \implies & \\
& tr_i + r_i \leq tr_j \\
\vee \quad & tr_j + r_j \leq tr_i
\end{aligned} \tag{16}
$$

- **C6** This applies to both HW and SW tasks. 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 communication delay between the FPGA and CPU:

$$
\begin{aligned}
hs_i = hs_j &\implies tf_i \leq te_j \\
hs_i \neq hs_j &\implies tf_i + ComDelay \leq te_j
\end{aligned} \tag{17}
$$

- **C7** This only applies to HW tasks, and is the same as Equation 8:

$$
\forall i (\text{task } i \text{ is a sink node}) \quad tf_i \leq SL \tag{18}
$$

- **C8** This applies to both HW and SW tasks to express the relationship between a task's start time te_i and its finish time tf_i:

$$
\begin{aligned}
hs_i = 0 &\implies tf_i = te_i + E_i \\
hs_i = 1 &\implies tf_i = te_i + SE_i
\end{aligned} \tag{19}
$$

- **C9** This only applies to SW tasks to model the shared CPU resource:

$$
\begin{aligned}
\forall i, j, hs_i = 1 \wedge hs_j = 1 \implies & \\
& (tf_i \leq te_j \\
\vee \quad & tf_j \leq te_i)
\end{aligned} \tag{20}
$$

4 Performance Evaluation

In [7], we addressed the optimal task graph scheduling problem with the UPPAAL model-checker [8], but that modeling technique can also be adapted to address the HW/SW partitioning problem. The scalability of UPPAAL is less than satisfactory for this specific problem, which motivated us to look for more scalable solutions. In this section, we conduct performance evaluation experiments to compare Yices with the open source ILP solver lpsolve [9], the commercial ILP solver [10], and the model-checker UPPAAL on the same problem instances.

We use the tool TGFF (Task Graphs For Free) [11] 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 of different tools by varying the number of tasks. For each 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. Whether these numbers are realistic is not that important for our purposes, since our main goal is to compare the *relative* performance of the three tools. The experiments are run on a Linux workstation with $4 \times AMD$ Opteron 844(1.8GHz) CPUs and 8GB RAM. We use the utility tool *memtime* to measure the running time of the Yices, lpsolve, CPLEX and UPPAAL, in order to compare how well they scale up with the number of tasks. When using Yices, the binary search algorithm 1 typically takes 7-9 iterations to converge to the final optimal result, and the "time" entries for Yices refer to the *total time* it takes to obtain the optimal solution for all iterations.

Figs 5 and 6 show the growth of running time and peak memory usage of Yices with increasing number of tasks. (Note that we are using different taskset parameters from those in [7], so the performance results are not directly comparable to those in [7].) We can see that the peak memory usage grows up quite slowly compared to running time, so CPU time is the factor that limits its scalability instead of memory size for model-checking.

Table 3 shows performance comparisons among different tools. O.M. for UPPAAL denotes that UPPAAL has issued an "out of memory" message. O.T. for lpsolve or CPLEX denotes that the tool's running time has exceeded 10 hours, at which point it is terminated. Since CPLEX produces an initial result quickly and incrementally improves upon it, we can still obtain a schedule length (S.L.) value even if it is terminated after 10 hours, although it is not guaranteed to be the optimal solution.

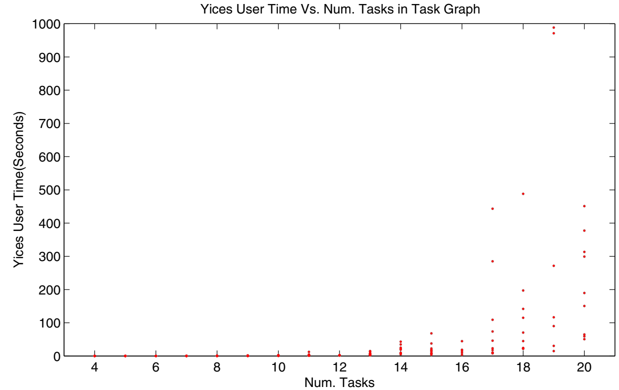


Figure 5. Running time of Yices for HW/SW partitioning on a hybrid FPGA/CPU device, where the FPGA has 20 columns.

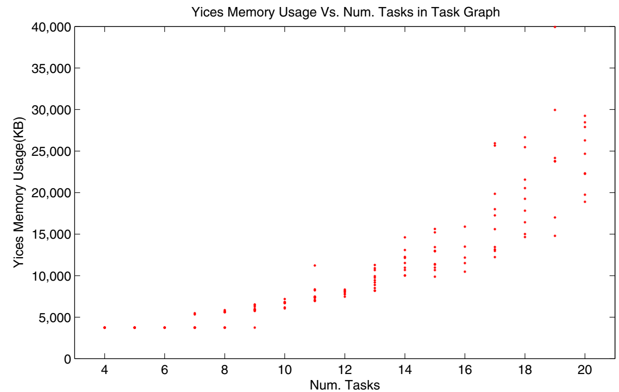


Figure 6. Peak memory usage of Yices for HW/SW partitioning on a hybrid FPGA/CPU device, where the FPGA has 20 columns.

The experiment results show that Yices has significantly better scalability than lpsolve, UPPAAL and CPLEX.

We have implemented the heuristic algorithm for HW-SW partitioning in [12], which is based on the well-know 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; $pathlength$ 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: $A = C = 1$,

N.T.	Yices		lpsolve	UPPAAL	CPLEX	
	Time(s)	S.L.	Time(s)	Time(s)	Time(s)	S.L.
4	0.1	33	0.1	0.6	0.1	33
5	0.1	32	0.4	7.8	0.2	32
6	0.2	56	42.8	2.6	0.2	56
7	0.2	45	256.5	42.7	2.7	45
8	0.3	60	17898.4	61.7	1.5	60
9	0.8	60	O.T.	254.3	2.2	60
10	0.8	63	O.T.	618.4	1273	63
11	1.8	81	O.T.	610.0	107.1	81
12	4.4	92	O.T.	415.7	O.T	92
13	7.0	87	O.T.	10519.8	O.T	87
14	8.1	92	O.T.	19033.5	O.T	93
15	7.4	97	O.T.	O.M.	O.T	97
20	356.2	128	O.T.	O.M.	O.T	132
25	7585.8	165	O.T.	O.M.	O.T	178

Table 3. Performance comparison of Yices, lpsolve, UPPAAL and CPLEX for HW/SW partitioning on a hybrid FPGA/CPU device, where the FPGA has 20 columns. N.T.: Number of Tasks; S.L.: Schedule Length; O.T.: Out of Time; O.M. Out of Memory.

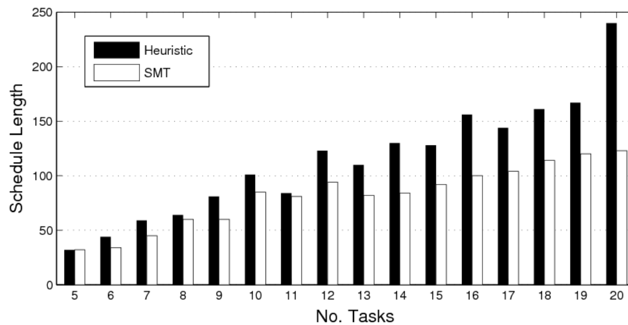


Figure 7. Comparison of schedule lengths between Yices and the heuristic algorithm in [12].

$B = D = 3$, 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 sub-optimal results. Fig. 7 shows the comparison of the schedule lengths between Yices and the heuristic. We can see that exhaustive search techniques such as SMT can lead to a significant reduction in schedule length, especially when the number of tasks grows large.

5 Related Work

Some authors have formulated boolean encodings for the High-Level Synthesis (HLS) problem, i.e., 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 et al [13] used the SAT solver Chaff [14] 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 [15] 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.

HW/SW partitioning is a well-studied problem, but most prior work has not considered PRTR FPGAs, with a few exceptions. Banerjee et al [12] 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 compared with in Section 4. He 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. Besides the fact that our SMT formulation is more compact and efficient than our ILP formulation, both our SMT and ILP formulations turn the task graph scheduling problem into a rectangle placement problem, and are much more efficient than the time-step-based ILP formulation in [12], which adds a new set of boolean variables at every time step, similar to the SAT formulation for the HLS problem [13].

Ghiasi et al [6] developed an optimal polynomial algorithm for minimizing runtime reconfiguration delay for a special type of HW platform: either a multi-FPGA platform composed of K identical fully reconfigurable FPGAs, or equivalently, a FPGA with K identical plug-in locations. This is different from the more general task model considered in this paper.

Fekete et al [16] addressed optimal placement of a task graph on a 2D FPGA by treating each task as a 3D box in space and time and converting the problem into an optimal box packing problem, which is solved using an efficient search algorithm. But if we allow configuration pre-fetch and consider task precedence constraints, then there may be a gap between a task's reconfiguration and computation stages as shown in Fig. 2. As a result, the algorithm in [16] is no longer applicable, since we cannot simply view the static scheduling problem as a box-packing problem.

Perng *et al* [17] developed scheduling algorithms for minimizing the energy consumption or the makespan of a taskset on a *multi-context FPGA* with dynamic voltage scaling, which has several configuration contexts, and different contexts can load or execute tasks independently and simultaneously. Optimal algorithms are developed for the case when a task partition over

contexts is given, and approximation algorithms are developed for the case when no task partition is given.

Instead of offline static HW/SW partitioning, some authors have used efficient heuristic algorithms for dynamic online task allocation (partitioning) for a hybrid FPGA/CPU device. Pellizzoni *et al* [18] developed a pseudo-optimal task allocation algorithm and a relocation scheme for relocatable tasks, derived feasibility conditions for both software and hardware scheduling, and defined an admission control test based on such conditions. Streichert *et al* [19] presented an operating system infrastructure for increasing fault tolerance and flexibility of distributed embedded systems where each node is a FPGA with one or more built-in softcore CPUs, by using online HW/SW partitioning, where *task migration* allows tasks to be moved between different nodes, and *task morphing* (same as relocation in [18]) allows tasks to be moved between HW and SW.

6 Conclusions

In this paper, we formulate and solve the problem of optimal HW/SW partitioning and static task scheduling for a hybrid FPGA/CPU device, with the optimization objective of minimizing the total schedule length, in the framework of Satisfiability Modulo Theories (SMT) with Linear Integer Arithmetic. Performance evaluation shows that SMT is a promising alternative to ILP for solving real-time scheduling problems.

References

[1] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation." in *FPGA*, 2002, pp. 187–195.

[2] G. D. MICHELI and R. K. GUPTA, "Hardware/software co-design," *PROCEEDINGS OF THE IEEE*, vol. 85, no. 3, MARCH 1997.

[3] Altera. [Online]. Available: http://www.altera.com

[4] S. Banerjee, E. Bozorgzadeh, N. Dutt, and J. Noguera, "Selective band width and resource management in scheduling for dynamically reconfigurable architectures," in *DAC*. IEEE, 2007, pp. 771–776.

[5] B. Dutertre and L. M. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 81–94.

[6] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, "An Optimal Algorithm for Minimizing Run-time Reconfiguration Delay." *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 2, pp. 237–256, 2004.

[7] Z. Gu, M. Yuan, and X. He, "Optimal static task scheduling on reconfigurable hardware devices using model-checking," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2007, pp. 32–42.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *STTT*, vol. 1, no. 1-2, pp. 134–152, 1997.

[9] lpsolve. [Online]. Available: http://lpsolve.sourceforge.net/

[10] cplex. [Online]. Available: http://www.ilog.com/products/cplex/

[11] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *CODES*, G. Borriello, A. A. Jerraya, and L. Lavagno, Eds. IEEE Computer Society, 1998, pp. 97–101.

[12] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration." in *Design Automation Conference (DAC)*, 2005, pp. 335–340.

[13] S. O. Memik and F. Fallah, "Accelerated SAT-based Scheduling of Control/Data Flow Graphs," in *ICCD*. IEEE Computer Society, 2002, pp. 395–.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *DAC*. ACM, 2001, pp. 530–535.

[15] G. Cabodi, A. Kondratyev, L. Lavagno, S. Nocco, S. Quer, and Y. Watanabe, "A bmc-based formulation for the scheduling problem of hardware systems," *STTT*, vol. 7, no. 2, pp. 102–117, 2005.

[16] S. P. Fekete, E. Köhler, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints." in *DATE*, 2001, pp. 658–667.

[17] N.-C. Perng, J.-J. Chen, C.-Y. Yang, and T.-W. Kuo, "Energy-efficient scheduling on multi-context fpgas," in *ISCAS*. IEEE, 2006.

[18] R. Pellizzoni and M. Caccamo, "Adaptive allocation of software and hardware real-time tasks for fpga-based embedded systems," in *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, 2006, pp. 208–220.

[19] T. Streichert, D. Koch, C. Haubelt, and J. Teich, "Modeling and design of fault-tolerant and self-adaptive reconfigurable networked embedded systems," *EURASIP Journal on Embedded Systems*, pp. Article ID 42 168, 15 pages, 2006, doi:10.1155/ES/2006/42168.