Meeting Deadlines Cheaply

Julien Legriel
STMICROELECTRONICS
12, rue Jules Horowitz, 38019 Grenoble, France
Julien.Legriel@st.com

Oded Maler
CNRS-VERIMAG
2, av. de Vignate, 38610 Gieres, France
Oded.Maler@imag.fr

Abstract

We develop a computational framework for solving the problem of finding the cheapest configuration (in terms of the number of processors and their respective speeds) of a multiprocessor architecture on which a task graph can be scheduled within a given deadline. We then extend the problem in two orthogonal directions: taking communication volume into account and considering the case where a stream of instances of the task graph arrives periodically.

1. Introduction

This paper is motivated by several contemporary developments in computer technology, computer applications and algorithmics: 1) The recent shift toward multi-core computer architectures renewed the interest in efficient parallel execution of programs; 2) Mobile platforms such as laptops and phones are becoming central and consequently the problem of reducing power consumption has become crucial. One way of reducing the power consumption of a processor is to reduce its voltage and frequency to the minimal level in which it can still meet performance constraints; 3) New classes of applications characterized by ongoing streams of *structured* computational tasks that come from the outside environment (unlike the classical problems of loop parallelization) and where significant amounts of data are exchanged among tasks; 4) The recent progress in Boolean SAT solving, and in SAT Modulo Theories (SMT) has led to a new generation of powerful tools for solving mixed combinatorial and numerical constraint optimization problems.

This work is part of the MINALOGIC ATHOLE project, focusing on the multi-core architecture XSTREAM developed by STMICROELECTRONICS as a potential next-generation platform targeting mainly mobile and multi-media applications. This architecture is designed around a "streaming fabric" consisting of a set of processors or specialized hardware blocks connected via flow-control mechanisms on top of a network-on-chip (NoC) [7]. To facilitate power saving, the system has means to manage several power modes, for example, to change the state of the different computing nodes: on/off, idle, low-voltage/lower clock rates, etc. Energy-efficient scheduling of applications on this architecture is one of the primary goals of the ATHOLE project: let the system work at the least energy-consuming configuration which can still meet the computational demands of the application.

In this work we formalize this problem as an extension of the well-known task-graph scheduling problem [9, 17, 15], where with each task we associate a *quantity of work* whose translation into *duration* depends on the speed of the processor on which it executes. A configuration of the architecture is characterized by the number of of processors running in each of the frequencies, from which a cost function is derived, reflecting the energy consumption of the configuration. Given such a task graph and an upper bound on its acceptable execution time (deadline) we ask the following question: *what is the cheapest configuration on which the task graph can scheduled for execution while meeting the deadline?* We formulate the question as a mixed logical-numerical constrained optimization problem and solve it using the SMT solver Yices [16]. We develop a binary search algorithm over the cost space which uses the solver to provide solutions with guaranteed distance from the optimum. We manage to handle randomly-generated task-graph problems having around 40-50 tasks on execution platforms consisting of machines with 3 different speeds.

We then extend the problem in two directions. First, we annotate the task graph with quantities of data that have to be communicated among tasks and refine the model of the execution platform to include communication network topology. We then pose the same problem where the communication channels are considered as additional resources occupied for durations proportional to the amount of transmitted data, thus incorporating *data locality* considerations. In this setting we can solve problems with 15-20 tasks on a *spidergon* topology [10] with up to 8 processors. Finally we formulate a periodic extension of the problem where task in-

stances arrive every ω time units and must be finished within a relative deadline $\delta > \omega$. Efficient resource utilization for this problem involves *pipelined* execution, which may increase the number of decision variables and complicate the constraints. We solve this infinite problem by reducing it to finding a schedule for a sufficiently-long finite unfolding. Preliminary experiments show that we can treat the periodic case where the number of unfolded tasks is up to 100.

The rest of the paper is organized as follows. In Section 2 we give the basic definitions of execution platforms, task graphs and feasible schedules. In Section 3 we discuss briefly several approaches for handling mixed constraints in optimization and describe our encoding of the problem using a Boolean combination of linear constraints. We then present our search procedure along with experimental results obtained using the Yices solver. The extension of the model to treat communication is sketched in Section 4 while Section 5 is devoted to the definition and encoding of a periodic version of the problem. Both sections are accompanied with experimental results. Finally in Section 6 we mention some related work, discuss its limitations of our models and suggest future research directions.

2. Problem Formulation

2.1 Execution Platforms

We formalize here the notion of an execution platform, which represents a possible configuration of a multiprocessor system, characterized by the number of active "machines" and their respective speeds. We assume a fixed set of speeds $V = \{v_0, v_1, \ldots, v_m\}$ with $v_0 = 0$ and $v_k < v_{k+1}$, each representing an amount of work (for example, number of instructions) that can be provided per unit of time. We do not bound a priori the number of machines.

DEFINITION 1 (Execution Platform). *An execution platform is a non-increasing function $R : \mathbb{N}_+ \to V$ assigning a speed to each machine, with $R(j) = 0$ indicating that machine j is turned off (or does not exist). A platform is finite if $R(j) = 0$ for every $j > j_*$ for some j_*.*

It is sometimes convenient to view R as a vector $R = (r_1, \ldots, r_m)$ with r_k being the number of machines working at speed v_k. To compare different platforms we use a *static* cost model that depends only on the membership of machines of various speeds in the platform and not on their actual utilization during execution.

DEFINITION 2 (Platform Cost). *The cost associated with a platform $R = (r_1, \ldots, r_m)$ is $C(R) = \sum_{k=1}^{m} c_k \cdot r_k$ where $c_k < c_{k+1}$ are technology-dependent positive constants.*

Below we define some useful measures on platforms related to their work capacity.

- Number of active machines: $\mathcal{N}(R) = \sum_{k=1}^{m} r_k$;
- Speed of fastest machine: $\mathcal{F}(R) = R(1)$;
- Work capacity: $\mathcal{S}(R) = \sum_{k=1}^{m} r_k \cdot v_k = \sum_i R(i)$.

The last measure gives an upper bound on the quantity of work that the platform may produce over time when it is fully utilized.

2.2 Work Specification

The work to be done on a platform is specified by a variant of a *task graph*, where the size of a task is expressed in terms of work rather than duration.

DEFINITION 3 (Task Graph). *A task graph is a triple $G = (P, \prec, w)$ where $P = \{p_1, \ldots p_n\}$ is a set of tasks, $\prec$ is a partial order relation on P with a unique[1] minimal element p_1 and a unique maximal element p_n. The function $w : P \to \mathbb{N}$ assigns a quantity of work to each task. When a task p is executed on a machine working in speed v, its execution time is $w(p)/v$.*

The following measures give an approximate characterizations of what is needed in terms of time and work capacity in order to execute G.

- The width $\mathcal{W}(G)$ which is the maximal number of $\prec$-incomparable tasks, indicates the maximal parallelism that can be useful;
- The length $\mathcal{L}(G)$ of the longest (critical) path in terms of work, which gives a lower bound on execution time;
- The total amount of work $\mathcal{T}(G) = \sum_i w(p_i)$ which together with number of machines gives another lower bound on execution time.

A schedule for the pair (G, R) is a function $s : P \to \mathbb{N} \times \mathbb{R}_+$ where $s(p) = (j, t)$ indicates that task p starts executing at time t on machine j. We will sometime decompose s into s_1 and s_2, the former indicating the machine and the latter, the start time. The duration of task p under schedule s is $d_s(p) = w(p)/R(s_1(p))$. Its execution interval (we assume no preemption) is $[s_2(p), s_2(p) + d_s(p)]$. A schedule is feasible if the execution intervals of tasks satisfy their respective precedence constraints and if they do not violate the resource constraints which means that they are disjoint for all tasks that use the same machine.

DEFINITION 4 (Feasible Schedule). *A schedule s is feasible for G on platform R if it satisfies the following conditions: 1)* Precedence*: If $p \prec p'$ then $s_2(p) + d_s(p) \leq s_2(p')$; 2)* Mutual exclusion*: If $s_1(p) = s_1(p')$ then $[s_2(p), s_2(p) + d_s(p)] \cap [s_2(p'), s_2(p') + d_s(p')] = \emptyset$. The total duration of a schedule is the termination time of the last task $\ell(s) = s_2(p_n) + d_s(p_n)$.*

The singular[2] deadline scheduling predicate $\text{SDS}(G, R, \delta)$ holds if there is a feasible schedule s for G on R such that

[1] This can always be achieved by adding fictitious minimal and maximal tasks with zero work.

[2] To distinguish it from the periodic problem defined in Section 5

$\ell(s) \leq \delta$. The problem of finding the cheapest architecture R where this holds is then the constrained optimization problem $\min C(R)\ s.t\ .\text{SDS}(G, R, \delta)$. The harder part of the problem is to check whether, for a given architecture R, $\text{SDS}(G, R, \delta)$ is satisfied when δ is close to the duration of the optimal schedule for G on R. Since we will be interested in approaching the cheapest platform we will often have to practically solve the optimal scheduling problem which is NP-hard.

Let us mention some observations that reduce the space of platforms that need to be considered. First, note that if $\text{SDS}(G, R, \delta)$ is solvable, then there is a solution on a platform R satisfying $\mathcal{N}(R) \leq \mathcal{W}(G)$, because adding processors beyond the potential parallelism in G does not help. Secondly, a feasible solution imposes two lower bounds on the capacity of the platform: 1) the speed of the fastest machine should satisfy $\mathcal{F}(R) \geq \mathcal{L}(G)/\delta$, otherwise there is no way to execute the critical path before the deadline. 2) The total work capacity should satisfy $\mathcal{S}(R) \geq \mathcal{T}(G)/\delta$, otherwise even if we manage to keep the machines busy all the time we cannot finish the work on time

3. Constrained Optimization Formulation

3.1 Background

The problem of optimizing a linear function subject to linear constraints, also known as linear programming [23], is one of the most studied optimization problems. When the set of feasible solutions is *convex*, that is, it is defined as a *conjunction* of linear inequalities, the problem is easy: there are polynomial algorithms and, even better, there is a simple worst-case exponential algorithm (simplex) that works well in practice. However, all these nice facts are not of much help in the case of scheduling under resource constraints. The mutual exclusion constraint, an instance of which appears in the problem formulation for every pair of unrelated tasks executing on the same machine, is of the form $[x, x'] \cap [y, y'] = \emptyset$, that is, a *disjunction* $(x' \leq y) \vee (y' \leq x)$ where each disjunct represents a distinct way to solve the potential resource conflict between the two tasks. As a result, the set of feasible solutions is decomposed into an exponential number of disjoint convex sets, a fact which renders the nature of the problem more combinatorial than numerical. Consequently, large scheduling problems cannot benefit from progress in relaxation-based methods for mixed integer-linear programming (MILP).

Techniques that are typically applied to scheduling problems [5] are those originating from the field known as constraint logic programming (CLP) [21]. These techniques are based on heuristic search (guessing variable valuations), constraint propagation (deducing the consequences of guessed assignments and reducing the domain of the remaining variables) and backtracking (when a contradiction is found). A great leap in performance has been achieved during the last decade for search-based methods for *the* generic discrete constraint satisfaction problem, the satisfiability of Boolean formulae given in CNF form (SAT). Modern SAT solvers [26] based on improvements of the DPLL procedures [12] can now solve problems comprising of hundreds of thousands of variables and clauses and are used extensively to solve design and verification problems in hardware and software.

Recently, efforts have been made to leverage this success to solve satisfiability problems for Boolean combinations of predicates, such as numerical inequalities, belonging to various "theories" (in the logical sense), hence the name *satisfiability modulo theories* (SMT) [18, 2]. SMT solvers combines techniques developed in the SAT context (search mechanism, unit resolution, non-chronological backtracking, learning, and more) with a theory-specific solver that checks the consistency of truth assignments to theory predicates and infers additional assignments. The relevant theory for standard scheduling problems is the theory of *difference constraints*, a sub theory of the theory of linear inequalities, but in order to cover costs and speeds we use the latter theory. To this end we use the powerful solver Yices [16] which excels on SMT problems that involves linear constraints. In the sequel we describe the problem encoding.

3.2 Problem Encoding

Solutions to the problem are assignments to decision variables $\{u_j\}$, $\{e_i\}$ and $\{x_i\}$ where variable u_j ranging over V, indicates the speed of machine j, integer variables e_i indicates the machine on which task p_i executes and variable x_i is its start time. Constants $\{v_k\}$ indicates the possible speeds, $\{w_i\}$ stand for the work in tasks and $\{c_k\}$ is the cost contributed by a machine running at speed v_k. The above-mentioned lower bounds on the speed of the fastest machine and on the platform capacity are b_1 and b_2, respectively. We use auxiliary (derived) variables $\{d_i\}$ for the durations of tasks based on the speed of the machine on which they execute and C_j for the cost of machine j in a given configuration. The following constraints define the problem.

- The speed of a machine determines its cost:

$$\bigwedge_j \bigwedge_k (u_j = v_k \Rightarrow C_j = c_k)$$

- Every task runs on a machine with a positive speed and this defines its duration:

$$\bigwedge_i \bigwedge_j ((e_i = j) \Rightarrow (u_j > 0 \wedge d_i = w_i/u_j))$$

- Precedence: $\bigwedge_i \bigwedge_{i' : p_i \prec p_{i'}} x_i + d_i \leq x_{i'}$
- Mutual exclusion:

$$\bigwedge_i \bigwedge_{i' \neq i} ((e_i = e_{i'}) \Rightarrow ((x_i + d_i \leq x_{i'}) \vee (x_{i'} + d_{i'} \leq x_i))$$

- Deadline: $x_n + d_n \leq \delta$
- Total architecture cost: $C = \sum_j C_j$.

We use additional constraints that do not change the satisfiability of the problem but may reduce the space of the feasible solutions. They include the above mentioned lower bounds on architecture size and a "symmetry breaking" constraint which orders the machines according to speeds, avoiding searching among equivalent solutions that can be transformed into each other by permuting the indices of the machines.

3.3 Implementation and Experimental Results

We have implemented a prototype tool that takes a task graph as an input, performs preliminary preprocessing to compute width, critical path and quantity of work and then generates the constraint satisfaction problem in the solver input language. The solver has no built-in optimization capabilities and we can pose only queries of the form $\text{SDS}(G, R, \delta) \wedge C(R) \leq \mathbf{c}$ for some cost $\mathbf{c}$. We use $\psi(\mathbf{c})$ as a shorthand for this query. The quest for a cheap architecture is realized as a sequence of calls to the solver with various values of $\mathbf{c}$. Performing a search with an NP-hard problem in the inner loop is very tricky since, the closer we get to the optimum, the computation time becomes huge, both for finding a satisfying solution and for proving unsatisfiability (from our experience, the procedure may sometimes get stuck for hours near the optimum while it takes few seconds for slightly larger or smaller costs). We have implement the following search algorithm. We fix a time budget θ beyond which we do not wait for an answer (currently 5 minutes on a modest laptop). The outcome of the query $\psi(\mathbf{c})$ can be either

$(1, c)$	:	There is a solution with cost $c \leq \mathbf{c}$
0	:	There is no solution
\$	:	The computation is too costly

At every stage of the search we maintain 4 variables: $\underline{c}$ is the maximal value for which the query is not satisfiable, $\underline{\$}$ is the minimal value for which the answer is \$, $\overline{\$}$ is the maximal value for which the answer is \$, and $\overline{c}$ is the minimal solution found (see Figure 1). Assuming that computation time grows monotonically as one approaches the optimum from both sides, we are sure not to get answers if we ask $\psi(\mathbf{c})$ with $\mathbf{c} \in [\underline{\$}, \overline{\$}]$. So we ask further queries in the intervals $[\underline{c}, \underline{\$}]$ and $[\overline{\$}, \overline{c}]$ and each outcome reduces one of these interval by finding a larger value of $\underline{c}$, a smaller value of $\underline{\$}$, a larger value of $\underline{\$}$ or a smaller value of $\overline{c}$. Whenever we stop we have a solution $\overline{c}$ whose distance from the real optimum is bounded by $\overline{c} - \underline{c}$. This scheme allows us to benefit from the advantages of binary search (logarithmic number of calls) with a bounded computation time. We are working on a more sophisticated scheme where progressively-larger values of θ are used.

We did experiments with this algorithm on a family of platforms with 3 available speeds $\{1, 2, 3\}$. The costs associated with the speeds are, respectively, 1, 8 and 27, reflecting the approximate cubic dependence of energy on speed. We have experimented with numerous graphs generated by TGFF tool [14] and could easily find solutions for problems with 40-50 tasks. Figure 2 illustrates the influence of the deadline constraints on the platform and the schedule for a 10-task problem of width 4. With deadline 100 the problem can be scheduled on the platform $R = (0, 0, 3)$, that is, 3 slow processors, while when the deadline is reduced to 99, the more expensive platform $R = (0, 1, 1)$ is needed.

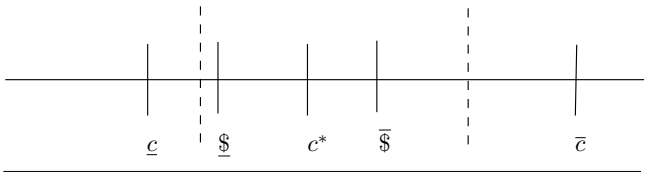

Figure 1. Searching for the optimum. The dashed line indicate the interval toward which the algorithm will converge, the best estimation of the optimum for time budget θ.

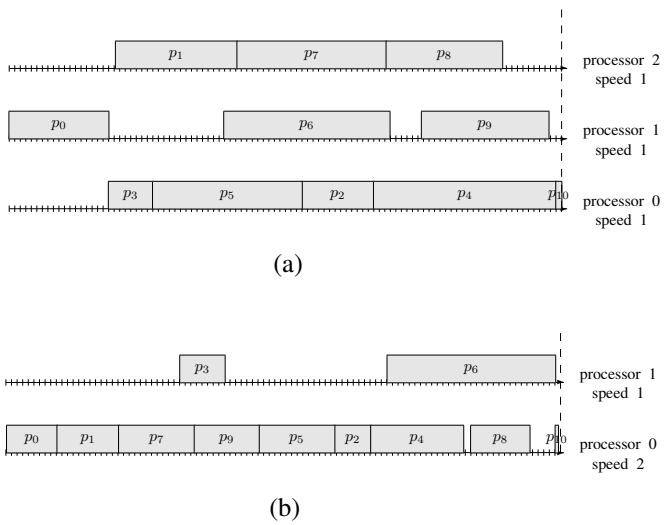

Figure 2. The effect of deadline: (a) a cheap architecture with deadline 100; (b) a more expensive architecture with deadline 99.

4. Adding Communication

The model described in the preceding section neglects communication costs. While this may be appropriate for some traditional applications of program parallelization, it is less so for modern streaming applications that have to pass large amounts of data among tasks and there is a significant variation in communication time depending on the *distance* between the processors on which two communicating tasks execute. To this end we extend the models as follows.

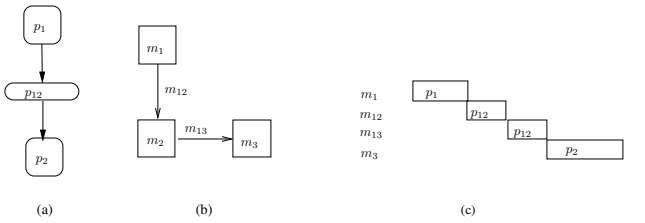

Figure 3. (a) a part of a task-data graph with data transmission from p_1 to p_2 denoted by p_{12}; (b) a part of an architecture where communication from m_1 to m_3 is routed via channels m_{12} and m_{23}; (c) a part of a schedule, including communication, where p_1 executes on m_1 and p_2 on m_3.

On the application side we annotate any edge $p \prec p'$ in the precedence graph with a number indicating the *quantity* of data that p transmits to p'. We assume that the data items sent by a task to each of its successors are disjoint. Furthermore, we assume that a task waits for the arrival of *all* its data before execution and that it transmits them *after* termination. On the architecture side we assume a network topology defined by a strongly-connected *connectivity graph*, a subgraph of the full directed graph whose nodes are the processors existing in the configuration (those with positive speed). We assume all these channels have the same speed normalized to 1, hence transmitting data of quantity Q on such a channel will occupy it for Q time. The contribution of an existing channel to the (static) cost of the architecture is a positive constant. Clearly, the model can be extended to admit channels with different speeds and hence different costs as we did with processors. We assume a *routing function* that maps any pair (m, m') of distinct processors to a loop-free path on the connectivity graph leading from m to m'. Hence when a task p and its successor p' are mapped to m and m' respectively, the termination of p should be followed by the execution of additional communication tasks that have to be scheduled successively on the channels on the path from m to m' before p' may be executed.[3] We assume the communication time between two tasks running on the same machine to be negligible. We spare the reader from the formal definitions of the extended task-data graph, the connectivity graph and the routing function and just illustrate them in Figure 3.

We have coded this problem and ran experiments with TGFF-generated graphs to find cheap and deadline-satisfying configuration of an architecture with up to 8 processors, $\{0, 1, \ldots, 7\}$ equipped with a *Spidergon* network topology developed in ST [10] which is used in the XSTREAM architecture. A spidergon network has links of the form $(i, i + 1 \ mod \ 8)$, $(i, i - 1 \ mod \ 8)$ and $(i, i + 4 \ mod \ 8)$ for every i and there is a path of length at most 2 between any pair of processors. In this setting we could solve easily problems with 15-20 tasks. Figures 4 shows the effect of the deadline on the architecture for a task-date graph with 16 tasks. For a deadline of 25 the task graph can be scheduled on an architecture with 3 machines while for deadline 24, 4 machines are needed. Note that the schedule for the first case is very tight and is unlikely to be found by heuristics such as list scheduling.

One important parameter that partly determines the shape of optimal solutions for task-data graph scheduling problems is the computation/communication to ratio [19]. Computation and communication are antagonist in nature since, while computation calls for parallelism, communication is reduced by sequential execution of two communicating tasks on the same processor. This is illustrated in the schedules of Figure 5 for two task-data graph which are identical except for the fact that all quantities of communication in second are doubled and this increases the amount of communication to the point that parallel execution becomes infeasible and a sequential solution on a faster machine is the only possibility.

5. Periodic Scheduling

In this section we extend the problem to deal with a *stream* of instances of G that arrive periodically.

DEFINITION 5 (Periodic Scheduling Problem). *Let ω (arrival period) and δ be positive integers. The periodic deadline scheduling problem* $\text{PDS}(G, R, \omega, \delta)$ *refers to scheduling an infinite stream $G[0], G[1], \ldots$ of instances of G such that each $G[h]$ becomes available for execution at $h\omega$ and has to be completed within a (relative) deadline of δ, that is, not later than $h\omega + \delta$.*

Let us make some simple observations concerning the relation between SDS and PDS for fixed G and R.

1. $\text{PDS}(\omega, \delta) \Rightarrow \text{SDS}(\delta)$. This is trivial because if you cannot schedule one instance of G in isolation you cannot schedule it when it may need to share resources with tasks of other instances.
2. If $\delta \leq \omega$ then $\text{PDS}(\omega, \delta) \Leftrightarrow \text{SDS}(\delta)$ because in this case each instance should terminate *before* the arrival of the next instance and hence in any interval $[h\omega, (h+1)\omega]$ one has to solve one instance of $\text{SDS}(\delta)$. Thus we consider from now on that $\omega < \delta$.
3. Problem $\text{PDS}(\omega, \delta)$ is solvable only if $W(G) \leq \omega \mathcal{S}(R)$. The quantity of work demanded by an instance should not exceed the amount of work that the platform can supply within a period. Otherwise, backlog will be accumulated indefinitely and no finite deadline can be met.
4. When $\text{SDS}(\omega)$ is not solvable, $\text{PDS}(\omega, \delta)$ can only be solved via *pipelined* execution, that is, executing tasks belonging to successive instances simultaneously.

We first encode the problem using infinitely many copies of the task related decision variables where $x_i[h]$ and $e_i[h]$, denotes, respectively, the start time of instance h of task p_i and the machine on which it executes, which together with

[3] Here too, the model can be refined to have task-dependent routing.

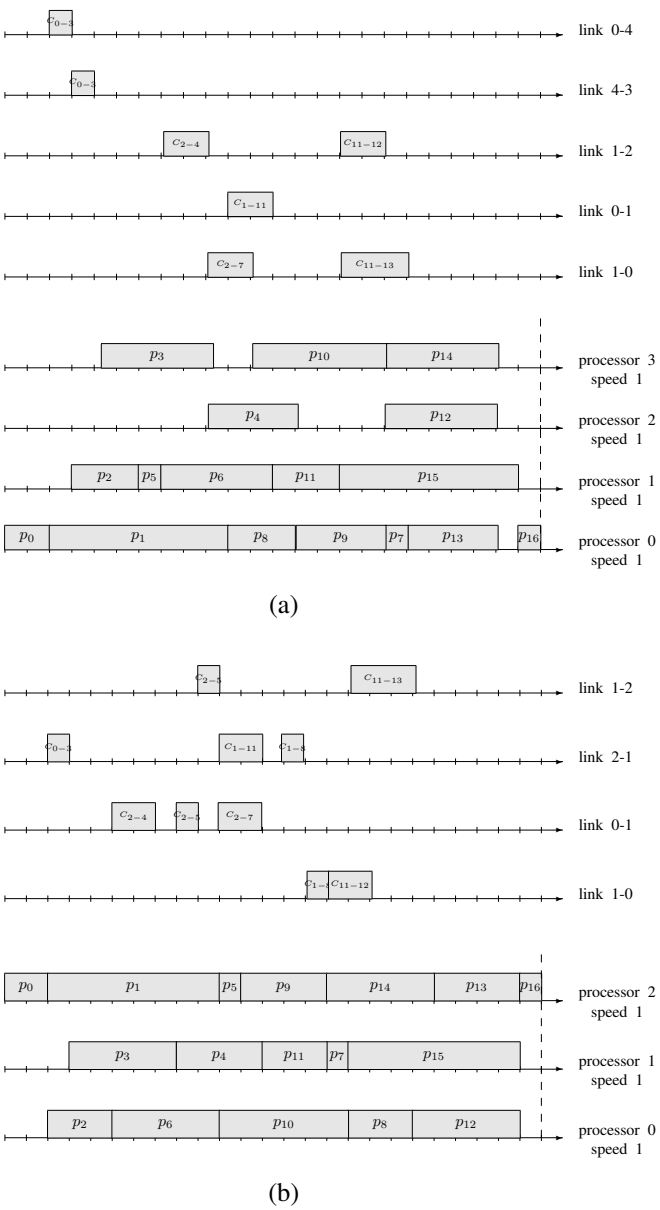

Figure 4. Scheduling with communication: (a) $\delta = 24$; (b) $\delta = 25$.

the speed of that machine determines its duration $d_i[h]$. The major constraints that need to be added or modified are:

- The whole task graph has to be executed between its arrival and its relative deadline:

$$\bigwedge_{h\in\mathbb{N}} x_1[h] \geq h\omega \ \wedge \ x_n[h] + d_n[h] \leq h + \delta$$

- Precedence:

$$\bigwedge_{h\in\mathbb{N}} \bigwedge_{i} \bigwedge_{i':p_i \prec p_{i'}} x_i[h] + d_i[h] \leq x_{i'}[h]$$

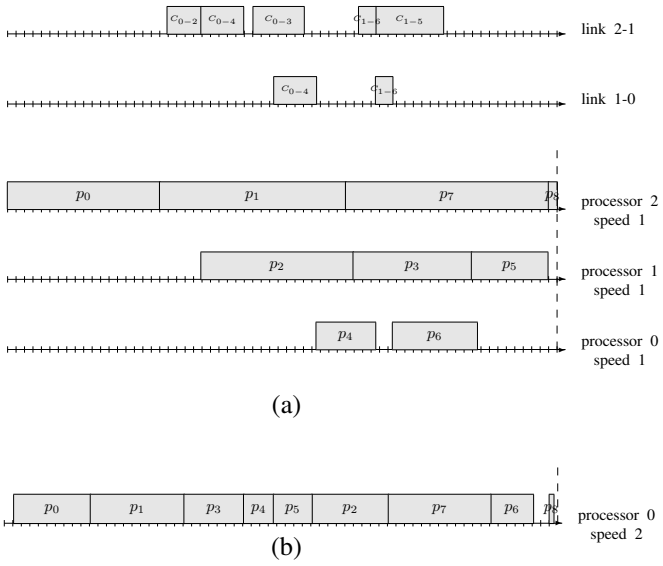

Figure 5. The effect of changing the computation to communication ratio: (a) a schedule for a task graph G with parallelism; (b) A sequential schedule for G' constructed from G by doubling the amount of all communications.

- Mutual exclusion: execution intervals of two *distinct* task instances that run on the same machine do not intersect.

$$\bigwedge_{j} \bigwedge_{i,i'} \bigwedge_{h,h'} (i \neq i' \vee h \neq h') \wedge e_i[h] = e_{i'}[h'] \Rightarrow$$
$$(x_i[h] + d_i[h] < x_{i'}[h']) \vee (x_{i'}[h'] + d_{i'}[h'] < x_i[h])$$

Note that the first two constraints treat every instance *separately* while the third is machine centered and may involve several instances due to pipelining.

While any satisfying assignment to the infinitely-many decision variables is a solution to the problem, we are of course interested in solutions that can be expressed with a finite (and small) number of variables, solutions which are *periodic* in some sense or another, that is, there is an integer constant β such that for every h, the solution for instance $h + \beta$ is the solution for instance h shifted in time by $\beta\omega$.

DEFINITION 6 (Periodic Schedules).

- *A schedule is β-machine-periodic if for every h and i, $e_i[h + \beta] = e_i[h]$;*
- *A schedule is β-time-periodic if for every h and i, $x_i[h + \beta] = x_i[h] + \beta\omega$;*
- *A schedule is (β_1, β_2)-periodic if it is β_1-machine-periodic and β_2-time-periodic.*

We say that β-periodic solutions are *dominant* for a class of problems if the existence of a solution implies the existence of a β-periodic solution. It is not hard to see that there is some β for which β-periodic solutions are dominant: the deadlines are bounded, all the constants are integers (or can be normalized into integers), hence we can focus on solutions with integer start times. Combining with the fact that

overtaking (executing an instance of a task before an older instance of *the same* task) can be avoided, we end up with a finite-state system where any infinite behavior admits a cycle which can be repeated indefinitely. However, the upper bound on dominant periodicity derived via this argument is too big to be useful.

It is straightforward to build counter-examples to dominance of β-machine-periodic schedules for any β by admitting a task of duration $d > \beta\omega$ and letting $\delta = d$. Each instance of this task has to be scheduled immediately upon arrival and since each of the preceding β instances will occupy a machine, the new instance will need a different machine. However, from a practical standpoint, for reasons such as code size which are not captured in the current model, it might be preferable to execute all instances of the same task on the same processor and restrict the solutions to be 1-machine-periodic. For time periodicity there are no such constraints unless one wants to use a very primitive runtime environment.

We encode the restriction to (β_1, β_2)-periodic schedules as an additional constraint.

- Schedule periodicity:

$$\bigwedge_{h\in\mathbb{N}} \bigwedge_{i} e_i[h+\beta_1] = e_i[h] \wedge x_i[h+\beta_2] = x_i[h] + \beta_2\omega$$

We denote the infinite formula combining feasibility and (β_1, β_2)-periodicity by $\Phi(\beta_1, \beta_2)$ and show that it is equisatisfiable with a finite formula $\Phi(\beta_1, \beta_2, \gamma)$ in which h ranges over the finite set $\{0, 1, \ldots \gamma - 1\}$. In other words, we show that it is sufficient to find a feasible schedule to the finite problem involving the *first* γ instances of G.

CLAIM 1 (Finite Prefix).
Problems $\Phi(\beta_1, \beta_2)$ *and* $\Phi(\beta_1, \beta_2, \gamma)$ *are equivalent when* $\gamma \geq \beta + \lceil \frac{\delta}{\omega} \rceil - 1$. *where* $\beta = lcm(\beta_1, \beta_2)$.

Proof: Intuitively, the prefix should be sufficiently *long* to demonstrate repeatability of a segment where β instances are scheduled, and sufficiently *late* to demonstrate steady-state behavior where resource constraints are satisfied by the maximal number of instances whose tasks may co-exist simultaneously without violating the deadline.

Suppose we scheduled $\gamma = \beta + \lceil \frac{\delta}{\omega} \rceil - 1$ instances successfully. Since $\gamma \geq \beta$ we can extract a pattern that can be repeated to obtain an infinite schedule that clearly satisfies β-periodicity and precedence constraints. We now prove that this periodic schedule satisfies resource constraints. Suppose on the contrary that instance h of a task i is in conflict with instance h' of a task i', $h \leq h'$ and let $h = (k\beta + k')$, $k' \in \{0 \ldots \beta - 1\}$. The deadline constraint, together with the fact that task i and i' overlap in time, limits the set of possible values for h' to be of the form

$$h' = h + \Delta = k\beta + k' + \Delta$$

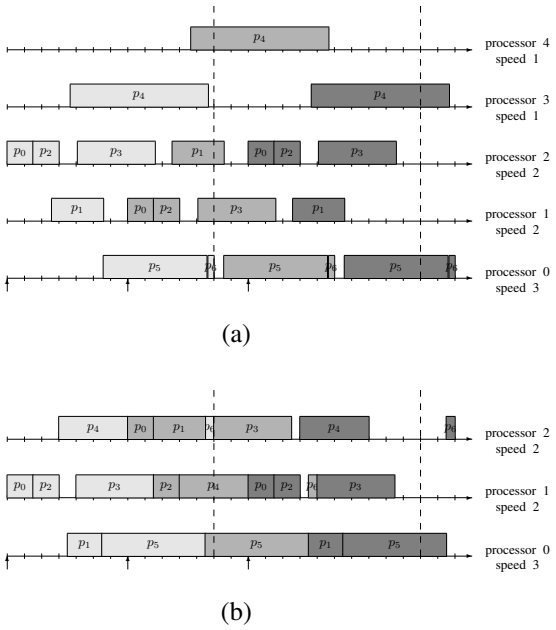

Figure 6. The effect of time periodicity: (a) A $(2, 1)$-periodic solution; (b) A cheaper $(2, 2)$ periodic schedule. Tasks p_1 and p_3 are not executed 1-periodically.

with $\Delta \in \{0 \ldots \lceil \frac{\delta}{\omega} \rceil - 1\}$. Because of β-periodicity we can conclude that task i and i' also experience a conflict in the respective instances k' and $k' + \Delta$. Since $k' < \beta$ and $\Delta \leq \lceil \frac{\delta}{\omega} \rceil - 1$ we have

$$k' + \Delta < \beta + \left\lceil \frac{\delta}{\omega} \right\rceil - 1$$

which contradicts our assumption. ∎

Hence the problem can be reduced to $\Phi(\beta_1, \beta_2, \gamma)$ with γ copies of the task-related decision variables. Note however that in fact there are only β "free" copies of these variables and the values of the other variables are tightly implied by those. The unfolding is required only to add additional resource constraints, not decision variables. Table 1 shows performance results on several graphs with different values of δ/ω, β_1 and β_2. In general we can treat problem where the number of unfolded tasks is around 100. Figure 7 shows a $(2, 1)$-periodic schedule for a task graph 1 of Table 1 with 17 tasks and $\delta/\omega = 4$ which requires 5 unfoldings. Although there are 85 tasks the problem is solved quickly in 3 minutes as there are only 34 decision variables. Figure 6 shows an example where making β_2 larger improves the solution. We consider a task-graph of 5 tasks, $\omega = 7$ and $\delta = 12$ with machine periodicity $\beta_1 = 2$. When we impose 1-time-periodicity we get a solution with 5 machines and cost 45, while when we move to 2-time-periodicity we can do with 3 machines and cost 43.

	tasks	work	cp	ω	δ	$\frac{\delta}{\omega}$	β_2	β_1	platform/cost	time
0	10	78	49	8	24	3	1	2	(1,2,5)/48	1’
							2	2	(1,2,4)/47	2’
							1	3	(1,2,4)/47	2’
					32	4	1	2	(0,3,5)/29	1’
					32	4	2	2	(0,3,4)/28	4’
1	17	77	48	10	30	3	1	2	(0,2,4)/20	4’
							2	2	(0,2,4)/20	5’
					40	4	1	2	(0,1,6)/14	3’
2	21	136	65	20	40	2	1	1	(0,2,3)/19	3’
					60	3			(0,1,5)/13	5’
								2	(0,1,5)/13	6’
3	29	199	89	25	50	2	1	1	(1,2,2)/45	4’
								2	(1,2,2)/45	8’
4	35	187	63	40	80	2	1	1	(0,0,5)/5	6’
				30	60	2			(0,1,5)/13	5’
								2	?	$\perp$
5	40	210	56	35	70	2	1	1	(0,4,4)/(34,36)	11’
6	45	230	45	70	140	2	1	1	(0,0,4)/4	18’

Table 1. Results for the periodic deadline scheduling problem. The columns stand for: number of tasks in G, the quantity of work, the length of the critical path, the graph input period, the deadline, the maximum pipelining degree, the time and machine periodicities, the platform found and its cost and execution time. Platforms are written as $(R(1), R(2), \ldots)$. A pair (c_1, c_2) stands for a solution with cost c_2 whose optimality has not been proved, but for which the strict lower bound c_1 was found. The symbol $\perp$ means that no solution was ever found, i.e. a timeout occurred at the first call to the solver near the upper bound.

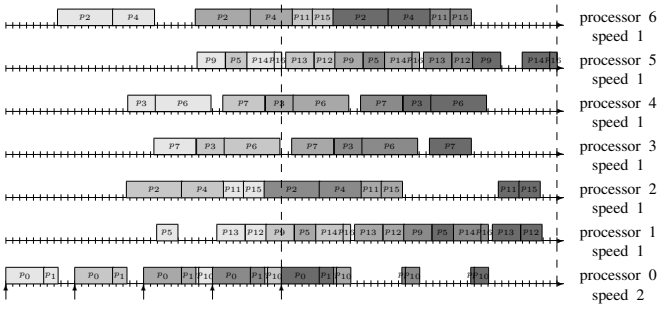

Figure 7. A $(2, 1)$-periodic schedule for a task graph with 16 tasks.

6. Discussion

We have formulated several design questions, involving cost optimization, scheduling, communication and pipelining, inspired by real problems. We have demonstrated how modern solvers for constraint satisfaction problems can handle decent-size instances of these problems. Of course, in order to be used in practice, we will need more refined models that will reflect more closely the reality of applications and architectures. We list below some related work and then discuss limitations of the current models which are subject to ongoing and future work.

The problem of mapping and scheduling of real-time applications on multiprocessor architecture has been subject of numerous publication. We restrict ourselves to few of those that deal with closely-related problems. In [20], a list scheduling technique is developed for design space exploration mapping and scheduling of communicating tasks. It also takes communication resource constraints into account but it is purely heuristic. An adaptation of the list scheduling heuristic has been proposed in [24] in the context of DSP processors. In [25, 6] methods based on ILP/CP decomposition are used to find accurate solutions to mapping/scheduling problems. They take more realistic constraints into account but do not explore pipelining as we do. Concerning the periodic version of the problem, let us remark that we have not found a similar problem formulation in the scheduling literature. In real-time systems [4, 22] one often deals with *independent* tasks that arrive periodically, possibly with distinct periods and deadlines, but not with structured “jobs” consisting of partially-ordered sets of tasks. On the other hand, problems associated with cyclic task graphs as in program loop parallelization [15] or manufacturing, are typically different since no *external* arrival constraints are imposed and a new instance is ready for ex-

ecution once the previous instance has terminated. Let us mention also the recent work [13] which introduces a more general model where a dynamic scheduler has to cope with a stream of requests, chosen non-deterministically by a request generator from a finite sets of task graphs.

The treatment of communication in the present paper suffers from two related shortcomings. First, the model we use may not be the most faithful model for certain situations where the execution of a task is strongly *interleaved* with communication. In such situations, refining the granularity of tasks to a level where the assumption of their separation holds is not realistic due to the huge number of decision variables. We are exploring alternative formulations which do not attempt to fix a precise schedule for these communications but rather attempt to minimize and balance the load on the channels. The disadvantage of such approaches is that in the absence of a well-defined schedule, the evaluation of the mapping solution is left to stochastic simulation. Our communication model is useful for situations were predictability is important such as hard real-time systems [8] and also as a first approximation of feasibility.

Another simplifying aspect of our model is its being *static* in several senses. First, we keep the configuration fixed and do not apply dynamic voltage scaling. The applications that we have in mind admit a more or less regular pattern of activity and the overhead in changing configuration may be too high to implement at the granularity of tasks and should be delegated to higher levels. Secondly we use a static cost model which does not distinguish between periods where a processor is executing instructions and periods where it idles. We believe that this is a good first approximation because the energy consumption of a processor that operates in a given frequency is significantly larger than that of a processor operating in lower frequency or a processor which is turned off. Refining the model in this direction will increase the number of linear constraints. Finally we assumed full information and determinism in the durations of executions and communications and in arrival times. Variability in these parameters may benefit from more adaptive scheduling strategies [1, 13] but the computational and observational overhead in implementing such strategies may turn out to be higher than their gains.

The current model does not capture other factors that affect the feasibility and quality of solutions, most notably those related to memory constraints due to code size and buffers [3]. These aspects can be incorporated into the model while remaining in the domain of linear inequalities. It also restricts itself to what is called task and pipelined parallelism, not touching the issue of data parallelism which is very important for streaming applications. Constraint-based methods can be used to automate the process of splitting and merging data which is often done manually today. Another interesting direction we are currently working on is the extension of the problem to multi-criteria. Rather than keeping the deadline fixed and optimizing the cost, we can adapt the search algorithm to provide a good approximation of the trade-off curve between cost and deadline, a valuable information in the process of design-space exploration. Finally, in terms of the constraint solver, when we invoke it with successive queries with different costs we currently cannot make these invocations "communicate" and cannot use general facts deduced in one invocation to reduce the search space in subsequent queries. A more direct implementation of our procedure inside a dedicated solver [11] will increase the size and complexity of problems that can be solved.

Acknowledgements: We thank Bruno Jego and Gilbert Richard for explaining us the XSTREAM architecture, Aldric Degorre for discussions on periodic scheduling and Scott Cotton for his advice on SAT and SMT.

References

[1] Y. Abdeddaim, E. Asarin and O. Maler Scheduling with Timed Automata , *Theoretical Computer Science* 354, 272-300, 2006.

[2] C. Barrett, R. Sebastiani, S.A. Seshia and C. Tinelli, Satisfiability Modulo Theories, in A. Biere, H. van Maaren,andd T. Walsh (Eds.), *Handbook of Satisfiability*, IOS Press, 2009

[3] S.S Battacharyya, P.K Murthy and E.A. Lee, *Software synthesis from dataflow graphs*, Kluwer, 1996.

[4] G. Bottazzo. *Hard Real-Time Computing Systems: Predictable Schedulding Algorithms and Applications*, Springer, 2005.

[5] Ph. Baptiste , C. Le Pape, W. Nuijten, *Constraint-based Scheduling: Applying Constraint Programming to Scheduling*, Springer, 2001.

[6] L. Benini, D. Bertozzi , A. Guerri and M. Milano, Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation, *CP'05*, 107-121, 2005.

[7] L. Benini and G. De Micheli, Networks on Chips: A New SoC Paradigm, *Computer* 35, 70-78, 2002.

[8] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis and P. Niebert, From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications, *LCTES'03*, 2003.

[9] E.G. Coffman, Computer and Job-Shop Scheduling Theory, Wiley, New York, 1976.

[10] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi and A. Scandurra, Spidergon: a Novel On-chip Communication Network, *System-on-Chip*, 2004.

[11] S. Cotton, *A Study of Some Problems in Satisfiability Solving*, PhD Thesis, University of Grenoble, June 2009.

[12] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving, *CACM* 5, 394-397, 1962.

[13] A. Degorre and O. Maler, On Scheduling Policies for Streams of Structured Jobs, *FORMATS'08*, 141-154, LNCS 5215, 2008.

[14] R.P. Dick, D.L. Rhodes and W. Wolf, TGFF: Task Graphs for

Free, *CODES/CASHE'98*, 97-101, 1998.

[15] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[16] B. Dutertre, L.M. de Moura: A Fast Linear-Arithmetic Solver for DPLL(T), *CAV'06*, 81-94 LNCS 4144, 2006

[17] H. El-Rewini, T.G. Lewis and H.H. Ali *Task Scheduling in Parallel and Distributed Systems*, Prentice-Hall, 1994

[18] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, DPLL(T) Fast Decision Procedures, *CAV04*, 175-188, 2004.

[19] M.I. Gordon, W. Thies, and S. Amarasinghe, Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs, *ASPLOS*, 2006.

[20] J. Hu and R. Marculescu, Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints, *DATE'04*, 234- 239, 2004.

[21] J. Jaffar and M.J. Maher, Constraint Logic Programming: a Survey, *J. of Logic Programming* 19/20, 503-581, 1994

[22] J.W.S Liu, *Real-Time Systems*, Prentice Hall, 2000.

[23] A. Schrijver *Theory of Linear and Integer Programming*, Wiley, 1998.

[24] G. Sih and E.A. Lee, List Scheduling Modifications to Account for Interprocessor Communication Within Interconnection-Constrained Heterogeneous Processor Networks, *Int. Conf. on Parallel Processing*, 1990.

[25] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti and M. Milano, Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip, *DATE'06*, 2006.

[26] L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers, *CAV'02* 17-36, LNCS 2404, 2002.