Fast Optimal Task Graph Scheduling using Satisfiability Solving

Jasper Kjersgaard Juhl and Michael Gade Nielsen

Department of Computer Science
Aalborg University, Denmark, May 2004
jasper@cs.auc.dk, michael@cs.auc.dk

Abstract. In this paper, we apply satisfiability (SAT) solving to optimal scheduling of tasks in parallel on a multiprocessor architecture. We introduce a model for formulating the scheduling problem as a satisfiability problem and apply known bounding techniques to the start time of tasks. Furthermore we present experimental results of reducing and expanding redundant task precedence constraints while keeping optimality. We show that our model is an order of magnitude faster, in practical experiments, than an existing SAT-based model for scheduling in high-level synthesis. Our model also scales better in the number of tasks. Finally, we show that our model is able to solve scheduling problems with up to 200 tasks within 30 minutes.

1 Introduction

Scheduling computational tasks in parallel on a multiprocessor architecture is a widely explored area [2]. Given a set of tasks with precedence constraints and execution times, the scheduling problem assigns each task to a free execution slot on one of the processors, such that the minimum execution time, known as makespan, is obtained when all tasks are done. The difficulty of this problem differs depending on the constraints of the number of processors, task execution time and precedence. With two processors, equal task execution time and arbitrary precedence constraints an optimal solution can be found in polynomial time. The above is known to be NP-hard with arbitrary number of processors. With both arbitrary number of processors and task execution time, the problem is known to be strong NP-hard [1].

Usually there is a need to keep all constraints arbitrary, when dealing with scheduling problems. In this case most approaches of providing a schedule are based on a heuristic solution [2], which does not guarantee that an optimal solution is found. This lack of guarantee is in some cases acceptable. Finding an optimal schedule of tasks is, although, still an important issue for optimization which often arise when scheduling tasks on a multiprocessor architecture or for planning. In such cases, other methods must be applied.

In this paper we treat the problem of *task graph scheduling* on a multiprocessor architecture. A task graph is a directed acyclic graph of tasks, see Figure 1, with arbitrary precedence constraints among tasks and arbitrary execution

time of tasks. A feasible schedule of a task graph with M machines must respect the following conditions: 1) A task can only be executed if all its parents have been executed, 2) Each machine can process at most one task at a time, 3) Tasks cannot be preempted.

We use satisfiability solving to determine an optimal non-preemptive schedule that minimizes the schedule time when the tasks are assigned to M uniform processors. In doing so, we first simplify the problem by considering the case with equal execution time and compare two models for this simplified problem; an existing model and a new two-variable type model. We benchmark these two models, and conclude that the new two-variable type model is an order of magnitude faster in practical experiments. We proceed with expanding this model with execution time and provide experimental results.

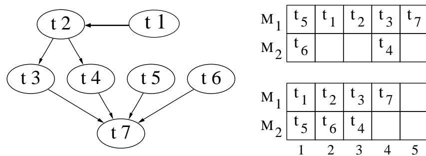

Fig. 1. A task graph with 7 tasks all with equal execution time. The upper most schedule shows a feasible schedule of the task graph with two machines and a schedule time of 5. The schedule below it, shows an optimal schedule with a schedule time of 4 on two machines.

In Section 1.1 we introduce satisfiability solving and in 1.2 we discuss the related work of optimal scheduling and satisfiability solving.

1.1 Satisfiability Solving

The Boolean Satisfiability (SAT) problem is to determine whether a Boolean formula has a satisfying assignment. SAT is known to be NP-complete [3].

The subject of practical SAT solvers have received considerable attention, and numerous solvers have been proposed: GRASP [5], POSIT [6], SATO [7], WalkSAT [8], BerkMin [9] and Chaff [10]. Most solvers operate on Boolean formulas in conjunctive normal form (CNF) and are variants of the Davis-Putnam [11] search algorithm. A CNF formula is a conjunction of clauses, a clause is a disjunction of literals where a literal is a variable v or its negation $\overline{v}$.

The basic Davis-Putnam search algorithm selects a variable that is not currently assigned, and gives it a value, this is known as a *decision*. The Boolean Constraint Propagation (BCP) identifies any variable assignments required by the current variable state to satisfy the Boolean formula. In general, BCP identifies *unit clauses* where one literal must be true to satisfy the formula, this assignment is known as *implication*. A conflict occurs when implications exist for setting the same variable to both true and false. A conflict is handled by invalidating the conflicted implications, this is known as *learning*, and *backtracking*

by flipping the value of a decision. If both values of a decision have been tried, the algorithm backtracks further and tries another decision that has not yet been tried both ways. If no such decision exists, then the formula is not satisfiable (UNSAT). The formula is satisfiable (SAT) if there exists no unassigned variables, at some time during the search.

We use Chaff for our experimental results in the implementation, since it has shown reasonable results against public domain SAT solvers on difficult problems from Electronic Design Automation [10].

1.2 Related Work

The scheduling problem is closely related to high-level synthesis scheduling, which have been studied by Memik and Fallah [4]. They formulate the resource constraint scheduling of control/data flow graphs as a satisfiability problem. They present experimental results of the performance on Chaff, and present domain specific and general optimizations for scheduling control/data flow graphs. We describe their model in Section 3.1.

Satisfiability solving is also applied in the field of model checking. In *bounded model checking* [12] a Boolean formula is constructed that is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reaches certain states of interest. If such a path segment cannot be found at length k, the search continues at larger k. The applications of model checking often result in large state transition systems. Hence, the SAT solver in bounded model checking deals with large SAT instances, which motivates our choice of applying SAT solving to obtain optimal schedules of large task graphs.

Optimal scheduling of task graphs have also been studied by Abdeddam, Kerbaa and Maler by using Timed Automata [13]. They present a framework where the scheduling problem admits a state-space representation and an optimal schedule corresponds to a shortest path in the timed automaton.

The rest of this paper is organized as follows. In Section 2 we define a task graph, a feasible and optimal schedule. In Section 3 we introduce the two models and compare them. Section 4 describes how we expand our model with execution time and Section 5 shows experimental results on this model. Section 6 concludes our work and Section 7 discusses future work.

2 Task Graphs and Schedules

A task graph $G = (\mathcal{T}, \rightarrowtail, e)$ is a directed acyclic graph, where $\mathcal{T}$ is the set of tasks $\{\tau_1, \ldots, \tau_n\}$. $\rightarrowtail$ is a relation between $\tau_1, \tau_2 \in \mathcal{T}$, denoted $\tau_1 \rightarrowtail \tau_2$, representing the precedence constraints among tasks, where τ_1 is the parent of τ_2. The execution time of a task is given by the function $e\colon \mathcal{T} \to \mathbb{N}$.

In the following we define the immediate children and parents of a task, and all the children of a task:

Definition 1 (π). *The immediate children of task τ, $\pi(\tau) = \{\tau' \in \mathcal{T} \mid \tau \rightarrowtail \tau'\}$.*

Definition 2 **(Π).** *The immediate parents of task τ, $\Pi(\tau) = \{\tau' \in \mathcal{T} \mid \tau' \rightarrowtail \tau\}$.*

Definition 3 **(π^*).** *All the children of task $\tau_1 \in \mathcal{T}$, $\pi^*(\tau_1) = \{\tau_n \in \mathcal{T} \mid$ there exists a sequence $\tau_1 \rightarrowtail \tau_2, \tau_2 \rightarrowtail \tau_3, \ldots, \tau_{n-1} \rightarrowtail \tau_n$ for some $\tau_2, \ldots, \tau_{n-1} \in \mathcal{T}\}$.*

From the three conditions described in Section 1, a feasible schedule and an optimal schedule are defined formally as:

Definition 4 **(Feasible Schedule).** *A feasible schedule for a task graph G and M machines is a function $st\colon \mathcal{T} \rightarrow \mathbb{N}$ (indicating the start time of each task), where:*

1. *For every $\tau \in \mathcal{T}, st(\tau) \geq \max_{\tau' \in \Pi(\tau)}(st(\tau') + e(\tau'))$.*
2. *Every $k \in \mathbb{N}$ belongs to at most M intervals from the set $\{[st(\tau), st(\tau)+e(\tau)[: \tau \in \mathcal{T}\}$.*

The schedule time of a feasible schedule st is $\lambda(st) = \max_{\tau \in \mathcal{T}}(st(\tau) + e(\tau))$

Definition 5 **(Optimal Schedule).** *Let F be the set of all feasible schedules of a task graph, $st \in F$ is optimal if every $st' \in F$, $\lambda(st) \leq \lambda(st')$.*

In the next Section we describe two different SAT-based models for obtaining feasible schedules of task graphs.

3 SAT-Based Models

In the following we present two different models for formulating the scheduling problem as a satisfiability problem. Both models only consider tasks with equal execution time, i.e. $e\colon \mathcal{T} \rightarrow 1$. The first formulation is that of Memik and Fallah [4], simply modified such that the resources, i.e. the machines are of the same capability. We analyze the problem and arrive at a motivation for building another formulation. In section 3.2 we present this formulation and in Section 3.4 we benchmark the two formulations, and show that our model is an order of magnitude faster at solving the scheduling problem in practical experiments.

The approach in both models is to create a bounded SAT instance, that is satisfiable if and only if a feasible schedule at that bound exists. An optimal schedule is then found by searching a bound interval of the task graph, which is described in Section 3.3.

In describing both models, we use I to denote the bound, T to denote the number of tasks, i.e. $|\mathcal{T}|$, and M to denote the number of machines which the tasks may be allocated on. When describing the constraints of each model, we provide correlation to what condition each constraint fulfills in Definition 4.

3.1 Control/Data Flow Graph Model

In this section we introduce the control/data flow graph model of Memik and Fallah [4]. Their model is with heterogeneous resources whereas ours are with

homogeneous. To be able to compare the two models we removed this feature from their model, which amounts to a separate constraint.

Their model uses a variable type *allocated* $a_{t,i,m}$, where $1 \leq t \leq T$ represents a task index, $1 \leq i \leq I$ denotes an iteration, and $1 \leq m \leq M$ denotes a machine index. $a_{t,i,m}$ is true if and only if task τ_t is allocated at iteration i on machine m. The model is divided into three constraints, where these should be conjuncted to yield a feasible schedule:

1. Constraint. Each task must be allocated on exactly one machine at exactly one iteration (Definition 4.1). Thus, exactly one of the $I \cdot M$ variables should be true. This is guaranteed by two sets of clauses. The first set guarantees having *at most* one variable is true.

$$\bigwedge_{t=1}^{T} \bigwedge_{m=1}^{M} \bigwedge_{m'=m}^{M} \bigwedge_{i=1}^{I} \bigwedge_{i'=i+1}^{I-1} \left(a_{t,i,m} \rightarrow \overline{a_{t,i',m'}}\right) \tag{1}$$

The second set guarantees having *at least* one variable is true:

$$\bigwedge_{t=1}^{T} \left(\bigvee_{i=1}^{I} \bigvee_{m=1}^{M} a_{t,i,m}\right) \tag{2}$$

2. Constraint. This constraint implies that if $\tau_{t'} \in \pi(\tau_t)$ then $\tau_{t'}$ cannot start before τ_t has been scheduled (Definition 4.1):

$$\bigwedge_{m=1}^{M} \bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \bigwedge_{i'=1}^{i} \bigwedge_{\tau_{t'} \in \pi(\tau_t)} \bigwedge_{m'=1}^{M} \left(a_{t,i,m} \rightarrow \overline{a_{t',i',m'}}\right) \tag{3}$$

3. Constraint. At any iteration, the number of tasks running on each machine can be at most one (Definition 4.2):

$$\bigwedge_{i=1}^{I} \bigwedge_{m=1}^{M} \bigwedge_{t=1}^{T-1} \bigwedge_{t'=t+1}^{T} \left(a_{t,i,m} \rightarrow \overline{a_{t',i,m}}\right) \tag{4}$$

In measurement of the hardness of practical SAT instances, the tendency is to measure the number of clauses generated. Although, for random generated formulas the hardness if often measured by the clause to variable ratio [14]. We deal with practically hard SAT instances, hence it is reasonable to measure the number of clauses and variables generated. The following describes an upper bound for the number of clauses and provides correlation between this upper bound and a practical example.

The clauses generated by the SAT formulation of the above scheduling constraint is at most, $f(E,T,I,M) = \frac{1}{2} \cdot T \cdot I \cdot M \cdot (I \cdot M - 1) + T + E \cdot (I \cdot M)^2 + T \cdot I \cdot M$ [4], where E represents the number of edges. For instance, a task graph with 50 tasks, 105 edges and solved with 4 machines at bound 25 has an upper bound of 1 302 550 clauses. In practice, this task graph generates 834 652 clauses.

The SAT formulation uses one variable type with three indices, which gives $T \cdot I \cdot M$ variables in the formulation. The number of clauses can be reduced by using two variable types instead of one. In the next Section, we introduce a task graph model that uses two variable types which reduces the clauses generated, by the above task graph, to an upper bound of 637 725.

3.2 Task Graph Model

In this section we present our SAT-based model for formulating the scheduling problem. The model uses two variable types for specifying the variables in the formulation. The variable type *allocated* $a_{t,i,m}$, is reused in our formulation with the same semantic meaning as in Section 3.1. The variable type *finished* $f_{t,i}$, is true if and only if task τ_t, where $1 \leq t \leq T$, in iteration $0 \leq i \leq I$ is finished. If $a_{t,i,m}$ is true then task τ_t is finished in the i'th and the next iterations, i.e. $f_{t,i}, \ldots, f_{t,I}$ are true. In the following we present each constraint of the model:

Start constraint. The start constraint guarantees that no task is finished at iteration zero (partly Definition 4.1):

$$\bigwedge_{t=1}^{T} \overline{f_{t,0}} \tag{5}$$

Run constraint. The run constraint specifies that task τ_t in iteration i may begin execution if its parents are finished and it can be allocated on a machine, or it may remain in the same state as in the prior iteration:

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \Big(f_{t,i} \leftrightarrow \big(f_{t,i-1} \vee (\mathcal{M} \wedge \mathcal{P}) \big) \Big), \tag{6}$$

where

$$\mathcal{P} = \bigwedge_{\tau_{t'} \in \Pi(\tau_t)} f_{t',i-1} \tag{7}$$

guarantees that all the parents of task τ_t should be finished in the prior iteration (Definition 4.1), such that task τ_t can be executed. And

$$\mathcal{M} = \bigvee_{m=1}^{M} a_{t,i,m} \tag{8}$$

specifies that task τ_t should be allocated on at least one machine (partly Definition 4.2).

Machine constraint. There are two parts of the machine constraint. The first part, equal to Equation 4, specifies that at any iteration, the number of tasks running on each machine can be at most one (Definition 4.2):

$$\bigwedge_{i=1}^{I} \bigwedge_{m=1}^{M} \bigwedge_{t=1}^{T-1} \bigwedge_{t'=t+1}^{T} \Big(a_{t,i,m} \rightarrow \overline{a_{t',i,m}} \Big) \tag{9}$$

The second part guarantees that a task can only be allocated on a machine once. Hence, allocation of a task implies that it cannot be allocated again:

$$\bigwedge_{t=1}^{T} \bigwedge_{i=1}^{I} \bigwedge_{\substack{i'=1 \\ i' \neq i}}^{I} \bigwedge_{m=1}^{M} \bigwedge_{m'=1}^{M} \left(a_{t,i,m} \rightarrow \overline{a_{t,i',m'}} \right) \tag{10}$$

This part is auxiliary since, if left out, the other constraints still yield a feasible schedule. But the feasible schedule may contain reallocation of completed tasks, if an available machine exists that is not utilized to reach optimality. Hence, this constraint avoids noise in the schedule but most importantly it helps the SAT solver. The SAT solver can more effectively cut off reallocation branches that it otherwise would have explored. Preliminary tests show a 121% increase in speed of the task graph scheduler.

End constraint. The end constraint specifies that all tasks should be finished at the last iteration (Definition 4.1):

$$\bigwedge_{t=1}^{T} f_{t,I} \tag{11}$$

The constraints must be in CNF, hence only the **run** constraint must be converted into CNF. The conversion yields

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \Big((\overline{f_{t,i}} \vee f_{t,i-1} \vee \mathcal{M}) \wedge (\mathcal{P} \vee \overline{f_{t,i}} \vee f_{t,i-1}) \tag{12}$$

$$\wedge (f_{t,i} \vee \overline{f_{t,i-1}}) \wedge (\overline{\mathcal{M}} \vee \overline{\mathcal{P}} \vee f_{t,i}) \Big) = \tag{13}$$

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \Big((\overline{f_{t,i}} \vee f_{t,i-1} \vee \mathcal{M}) \wedge (\bigwedge_{p \in \Pi(\tau_t)} (f_{p,i-1} \vee \overline{f_{t,i}} \vee f_{t,i-1})) \tag{14}$$

$$\wedge (f_{t,i} \vee \overline{f_{t,i-1}}) \wedge (\bigwedge_{m=1}^{M} (a_{t,m,i} \vee \overline{\mathcal{P}} \vee f_{t,i})) \Big) \tag{15}$$

by using the distributive and associative laws of Boolean Logic equality. Next, we analyze the clauses generated by the constraints in CNF format. The **start** (Eq. 5) and **end** (Eq. 11) constraints generate $2T$ clauses. The first part of the **machine** (Eq. 9) constraint generates $I \cdot M \cdot \frac{1}{2} \cdot T^2$ clauses. The second part of the **machine** (Eq. 10) constraint generates $(I \cdot M)^2 \cdot T$ clauses. The **run** (Eq. 14- 15) constraint in CNF generates at most $I \cdot (T \cdot (2 \cdot M) + E)$ clauses. Hence, the number of clauses is bounded by $h(E,T,I,M) = 2T + I \cdot M \cdot \frac{1}{2} \cdot T^2 + I \cdot (T \cdot (2 \cdot M) + E) + (I \cdot M)^2 \cdot T$.

For instance, a task graph with 50 tasks, 105 edges and solved with 4 machines at bound 25 has an upper bound of 637 725 clauses. In practice this task graph generates 627 633 clauses.

This SAT formulation uses two variable types with two and three indices, which give $T \cdot I + T \cdot I \cdot M$ variables. Hence this model uses $T \cdot I$ more variables than the control/data flow graph model, described in Section 3.1.

A schedule is build by looking at which truth value the SAT solver assigns to a variable. If for instance $a_{1,10,4}$ is true then task τ_1 should be scheduled at iteration/time 10 on machine 4.

3.3 Searching for an Optimal Schedule

The SAT solver must search a bound interval to determine the optimal bound, i.e. the optimal schedule time, see Definition 5.

For a task graph G an upper bound, $U(G)$, can be determined by using a heuristic scheduler. There exist various heuristic algorithms, which may give different schedule times. On large task graphs, the SAT solver may use much more time to prove either satisfiability (SAT) or unsatisfiability (UNSAT), hence a tight bound is very important. Therefore, it is reasonable to run different heuristic algorithms and use the tightest upper bound.

A lower bound, $L(G)$, can be determined by calculating the critical path i.e. the longest path determined by the total execution time from a task to all others. A lower bound can also be determined by summing the execution times of all tasks and divide it by the number of machines. This approach assumes full parallelization. The maximum of these bounds is used as the lower bound.

We employ a linear search from the upper bound toward the lower bound, known as an upper bound search. The search completes once the SAT solver yields an UNSAT condition. The SAT solver usually uses more time on showing UNSAT than SAT [4], hence we have not experimented with a lower bound search as this would expose the SAT solver to, most likely, several UNSAT instances. This is also the case for binary search, which although in the worst case would expose the SAT solver to less instances, but would require showing more UNSAT instances.

If a trivial optimal schedule exists, the SAT solver does not need to be used since the heuristic upper bound schedule is optimal:

Definition 6 (Trivial Optimal Schedule). *An optimal schedule of a task graph G is trivial if its upper bound equals its lower bound, $U(G) = L(G)$.*

3.4 Comparison

In this section, we compare the two models with regard to performance and complexity. Both models are tested on the Standard Task Graph Set (STG) [1], which is a set of benchmarks for evaluating multiprocessor scheduling on task graphs. The set is comprised of randomly generated task graphs.

In the following we analyze the complexity of the functions that determine the clauses generated by the two models and conduct practical experiments on the two models.

Complexity In this theoretical part, we compare the worst case growth rate of the functions, $f(E, T, I, M)$ and $h(E, T, I, M)$, that determines the clauses generated by the control/data flow graph model and our task graph model, respectively.

We use the Big-O notation since this is a worst case analysis. E is substituted with $O(T^2)$ since $E = O(T^2)$ and I is substituted with $O(T)$ since $I = O(T)$.

First we consider the function f:

$$O(f) = O(\frac{1}{2} \cdot T^2 \cdot M \cdot (T \cdot M - 1) + T + T^4 \cdot M^2 + T^2 \cdot M) \quad (16)$$

$$= O(T^4 \cdot M^2) \quad (17)$$

Next, we consider the function h:

$$O(h) = O(2T + T \cdot M \cdot \frac{1}{2} \cdot T^2 + T \cdot (T \cdot (2 \cdot M) + T^2) + T^3 \cdot M^2) \quad (18)$$

$$= O(T^3 \cdot M^2) \quad (19)$$

Equation 17 clearly outgrows Equation 19. Hence our model scales better in the number of tasks and thus generates fewer clauses than the control/data flow graph model in the worst case.

Our model uses $T \cdot I + T \cdot I \cdot M$ variables while their uses $T \cdot I \cdot M$ variables, hence in the worst case they both use $O(T^2 \cdot M)$ variables (I is substituted with T). We need to compare the two models in practice by conducting experiments on them, which is the topic of the next section.

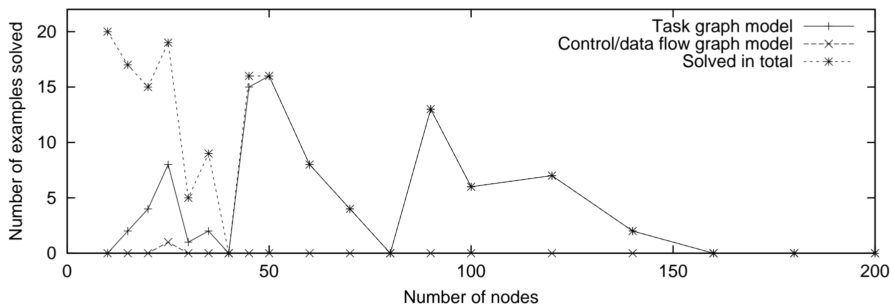

Fig. 2. Test results from various task graphs with $M = 4$. The number of examples solved *exclusively* by each model and the number of examples solved in total are shown.

Practical A test of the two models has been conducted on randomly generated task graphs with 20 graphs of each size $T = \{10, 15 \dots 200\}$ [18].

The two models were implemented in C++ and executed with a time bound of 20 minutes of CPU time. If a model was not able to solve a task graph within

this time, we consider the task graph to be unsolvable within reasonable time by the model. The tests were conducted on a Fire SPARC v880R 900 MHz CPU with 4 GB of memory running Solaris 9.

Figure 2 shows the number of examples solved *exclusively* by each model with $M = 4$. The volatile drops and spikes in the task graph are due to the different edge generation methods of the STG. Only one graph could be exclusively solved by the control/data flow graph model, while several task graphs could only be solved by our model. Their model was able to solve 68 examples while our model solved 156 examples in total.

Moreover, when both models were able to solve an instance our model was an order of magnitude faster; an average time of 16.5 seconds versus 317.5 seconds. This increase in speed by our model is probably due to a number of combinations all with relation to our use of two variable types. We require a task to remain finished in the iterations after it is finished, this may help the SAT solver to cut off branches. Also the ratio between clauses and variables of our model might be more ideal for the SAT solver, or the reduction of clauses greatly impact the SAT solver. An exact reason requires further detailed study.

4 Introducing Execution Time

In the following we proceed with expanding our task graph model, introduced in Section 3.2, with execution time, i.e. $e\colon \tau \to \mathbb{N}$. Section 4.1 describes how the start time of a task can be bounded which reduces the number of clauses, and thereby the number of variables, in the SAT formulation. Section 4.2 describes two methods for optimizing the number of precedence constraints among tasks, while preserving the same optimal schedule time as if no optimization was applied.

To introduce execution time in our model, presented in Section 3.2, we choose to expand our variable state $f_{t,i}$ with a third index. The third index is the *execution step* s of a task τ_t where $1 \leq s \leq e(\tau_t)$. The state $f_{t,i,s}$ is true if and only if task τ_t in iteration i is finished with its s'th execution step. If $f_{t,i,1}$ is true then task τ_t has been started and is finished once $f_{t,i',e(\tau_t)}$ is true, which is when $i' = i + e(\tau_t)$.

Start constraint. The start constraint is modified such that no task is finished in any of its execution steps at iteration zero:

$$\bigwedge_{t=1}^{T} \bigwedge_{s=1}^{e(\tau_t)} \overline{f_{t,0,s}} \tag{20}$$

Run constraint. The run constraint is divided into three parts: a run first execution step, a run s'th execution step and a part that ensures that tasks are not preempted. The following describes each part.

first execution step. This part amounts to Equation 6 in Section 3.2. Task τ_t may begin execution of its first execution step in iteration i if both its parents are finished and it can be allocated on a machine, or it may remain in the same state as in the prior iteration:

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \Big(f_{t,i,1} \leftrightarrow \big(f_{t,i-1,1} \vee (\mathcal{M} \wedge \mathcal{P}) \big) \Big) \tag{21}$$

where

$$\mathcal{P} = \bigwedge_{\tau_{t'} \in \Pi(\tau_t)} f_{t',i-1,e(\tau_{t'})} \tag{22}$$

guarantees that the parents of task τ_t should be finished in the prior iteration. $\mathcal{M}$ was defined in Section 3.2 (Eq. 8) which guarantees that task τ_t can be allocated on at least one machine.

***s*'th execution step.** The s'th execution step of task τ_t may begin its execution if it can be allocated on a machine and its $(s-1)$'th execution step is finished, or it may remain in the same state as in the prior iteration:

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \bigwedge_{s=2}^{e(\tau_t)} \Big(f_{t,i,s} \leftrightarrow \big(f_{t,i-1,s} \vee (\mathcal{M} \wedge f_{t,i-1,s-1}) \big) \Big) \tag{23}$$

This constraint allows a task to be preempted, because it is not required that if $f_{t,i,s}$ is true then $f_{t,i+1,s+1}$ is also true, hence the next constraint is needed to guarantee that tasks are not preempted.

no preemption. This part ensures that if a task has been started it may not be preempted. If the first execution step of task t is finished, $f_{t,i-1,1}$, and the whole task has not been completed yet, $\overline{f_{t,i-1,e(\tau_t)}}$, then its next execution step must be allocated on the same machine $a_{t,i,m}$, as it was allocated on in the prior iteration, $a_{t,i-1,m}$:

$$\bigwedge_{i=1}^{I} \bigwedge_{t=1}^{T} \Big((f_{t,i-1,1} \wedge \overline{f_{t,i-1,e(\tau_t)}}) \rightarrow (\bigwedge_{m=1}^{M} a_{t,i-1,m} \leftrightarrow a_{t,i,m}) \Big) \tag{24}$$

Machine constraint. This constraint is identical to Equation 9 in Section 3.2. The performance impact by adding the second part of the **machine** constraint (Eq. 10) is replaced by the **no preemption** constraint (Eq. 24).

End constraint. All tasks should be finished with their last execution step at the last iteration:

$$\bigwedge_{t=1}^{T} f_{t,I,e(\tau_t)} \tag{25}$$

4.1 Bounding the Start Time of Tasks

If the soonest a task τ can possibly be started is i (ASAP) and the latest it possibly can be started is i' (ALAP), then clauses that allow τ to be started outside the interval $[i; i']$ can be omitted from the formulation. Hence, bounding the start time of tasks results in a formulation with less clauses and thereby less variables.

ASAP for a task can be calculated by using the length of the critical path from a domino start task to itself in the task graph. This length indicates the earliest iteration where the task can be scheduled. ALAP for a task can be calculated by using the length of the critical path cp from a domino terminal task to itself minus the upper bound I, hence $I - cp$ indicates the latest iteration where the task can be started.

Alternatively, ASAP and ALAP can be calculated by assuming full parallelization, i.e. using the sum of the execution times of its parents or children, respectively, and dividing it by the number of machines. Although these and more advanced techniques [15, 16] exist, we have chosen the critical path approach because it is essentially simple and provides reasonable results.

4.2 Reduction and Expansion of Precedence Constraints

In this section we describe methods for reducing and expanding the precedence constraints of a task graph i.e. its edges.

Definition 7 (A Redundant Edge). *An edge from task τ_1 to τ_2 is redundant if $\exists \tau' \in \pi(\tau_1) \backslash \tau_2$, $\tau_2 \in \pi^*(\tau')$.*

A task graph G can be reduced by removing all redundant edges and G can be expanded by adding redundant edges such that $\pi(\tau) = \pi^*(\tau)$ for all tasks $\tau \in \mathcal{T}$. Both methods preserve the same optimal schedule time as if no optimization was applied, but it may result in obtaining an optimal schedule faster. Expanding a task graph results in more clauses while the opposite is true when reducing a task graph. Since the constrainedness [17] is affected by the number of clauses and their length, there exist motivation for applying both methods to task graphs and choosing the best method, which is the topic of the next section.

5 Experimental Results

This section describes the experimental results of our task graph model, with execution time and ASAP/ALAP, on reduced and expanded task graphs. We submit the model to large task graphs and increase the number of machines M, to see how well the model scales.

As described in Section 3.3 some graphs are trivially solved. In order to test our model on these graphs we require the SAT solver to find a satisfying assignment, although this is not necessary in order to get an optimal schedule.

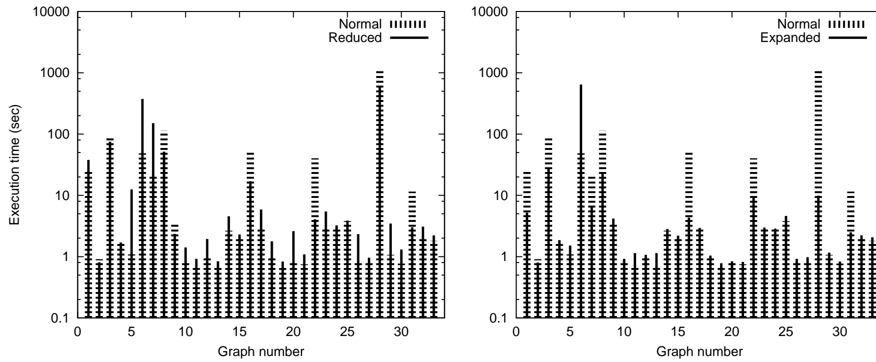

Fig. 3. The left side compares the CPU time, on a logarithmic scale, used to find an optimal schedule on normal and reduced graphs. Normal graphs are on average solved faster, hence the right side compares normal graphs with expanded graphs.

We tested our model on 90 different graphs with each precedence constraint method, i.e. normal (unmodified), reduced and expanded, to test whether it is reasonable to apply any precedence constraint methods before searching for an optimal schedule. The tests were performed with $M = 4$ machines and each test was given a time bound of 30 minutes of CPU time. The graphs originate from the STG (number $0-89$) and had 50 tasks. The same 33 of the 90 graphs were solved by each of the three methods, see Figure 3 which compares the three methods. Normal and expanded could solve one additional task graphs than reduced, while there were two additional task graphs that only expanded could solve. In the 33 cases the average time for normal, reduced and expanded graphs were 47.9, 42.0 and 23.5 seconds, respectively. Since expanding task graphs resulted in obtaining an optimal schedule significantly faster and could solve more graphs, this method was used in further tests.

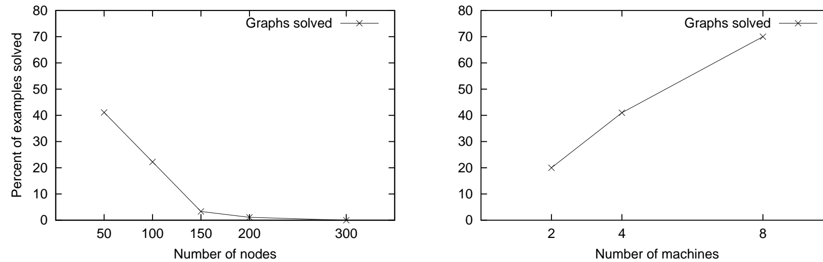

Fig. 4. The left side shows the number of examples our model is able to solve as a function of the graph size. The right side shows the number of examples our model is able to solve as a function of machines.

We tested the scalability of our model on 5 sets of 90 task graphs with size 50, 100, 150, 200 and 300 tasks. The results are shown in Figure 4. The number of examples our model was able to solve decreases as the number of tasks increases. No task graphs with 300 tasks could be solved and only one task graph could be solved with 200 tasks. Figure 4 also shows the number of examples solved with 50 tasks as the number of machines increases. 20% of the examples could be solved with 2 machines versus 70% with 8 machines. Furthermore the average time was 17.5, 1.5 and 2.19 seconds with 2, 4 and 8 machines, respectively.

6 Conclusion

The task graph model, we presented in this paper, was shown to be better at finding optimal schedules for task graphs than an existing control/data flow model [4]. The task graph model is a satisfiability based model with two variable types and an auxiliary constraint that increased speed by 121%.

The task graph model generates $O(T^3 \cdot M^2)$ clauses, while the control/data flow model generates $O(T^4 \cdot M^2)$ clauses. The task graph model could in a practical experiment solve more task graphs than the control/data flow model, 156 versus 69 task graphs. Moreover, when both models were able to obtain a solution, the task graph model was an order of magnitude faster with an average time of 16.5 versus 317.5 seconds.

We expanded the task graphs by adding redundant edges. We showed that this expansion of task graphs gave the best experimental results for obtaining optimal schedules; an average time of 23.5 versus 47.9 seconds. Further experiments, with expanded task graphs, showed that when increasing the number of machines the model could solve more examples; 20% versus 70% of the examples were solved with 2 and 8 machines, respectively. Furthermore, the experiments showed that the number of examples our task graph model was able to solve decreased as the number of tasks increased, as expected. The task graph model was able to solve task graphs with up to 200 tasks within 30 minutes. The task graph model was not able to find an optimal schedule for task graphs with 300 tasks, which suggests that an upper bound exists between 200 and 300 tasks.

7 Future Work

Based on our work, there are still further aspects of interest:

1. When the bound interval is searched for an optimal schedule, the CNF formula is created again each time the bound decreases. Decremental clause deletion only deletes the clauses necessary to facilitate a bound decrease and remove implications. Also it might be possible to reuse learned clauses from the previous bound.
2. Each bound is solved independently of the overall bound search, hence a multiprocessor architecture could be utilized to assign a SAT instance with different bounds to each processor. The bound search could be greatly advanced when the bound interval is large.

3. The decision strategy of the SAT solver could perhaps be adjusted to fit our domain specific problem by guiding the search i.e. cutting off superfluous branches.
4. Expand our task graph model with communication costs. Tasks pay a cost, measured in time, if they are not executed on the same machine as their precedence constraints.

References

1. Kasahara Laboratory, Waseda University: "Standard Task Graph Set", http://www.kasahara.elec.waseda.ac.jp/schedule
2. Y. Kwok and I. Ahmad, The University of Hong Kong: "Benchmarking and Comparison of the Task Graph Scheduling Algorithms", 1999
3. S. A. Cook: "The Complexity of Theorem Proving Procedures"
4. M. Seda, F. Farzan: "Accelerated SAT-based Scheduling of Control/Data Flow Graphs"
5. Marques-Silva, J.P., and Sakallah, K. A.: "GRASP: A Search Algorithm for Propositional Satisfiability", IEEE Transactions on Computers, vol. 48, 506-521. 1999
6. J. W. Freeman: "Improvements to Propositional Satisfiability Search Algorithms", University of Pennsylvania, 1995
7. Zhang, H.: SATO: "An efficient propositonal prover", International Conference on Automated Deduction, pp. 272-275, July 1997.
8. D. McAllester, B. Selman and H. Kautz: "Evidence for invariants in local search", in Proceedings of AAAI 97, MIT Press, 1997, pp. 321-326
9. E. Goldberg, Y. Novikov: "BerkMin – a Fast and robust Sat-Solver"
10. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik: Chaff: "Engineering an Efficient SAT Solver", Design Automation Conference, 2001
11. M. David, H. Putnam.: "A computing procedure for quantification theory"
12. E. Clarke, A. Biere, R. Raimi and Y. Zhu: "Bounded Model Checking Using Satisfiability Solving".
13. Y. Abdeddam, A. Kerbaa, O. Maler: "Task Graph Scheduling using Timed Automata"
14. D. Mitchell, B. Selman, H. Levesque: "Hard and Easy Distributions of SAT Problems", Tenth National Conference on Artificial IOntelligence (AAAI-92)
15. M. Narasimhan, J. Ramanujam: "Improving the Computational Performance of ILP-based Problems", International Conference on Computer Aided Design, 1998.
16. M. Rim, R. Jain: "Lower Bound Performance Estimation for the High-level synthsis Scheduling Problem"
17. T. Walsh: "The Constrainedness Knife-Edge", University of Strathclyde
18. M. G. Nielsen, J. K. Juhl: "Appendix of Graphs and Detailed Test Results", http://www.cs.auc.dk/~jasper/dat4/