A Decomposition-based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors

Nadathur Satish Kaushik Ravindran Kurt Keutzer
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA, USA
{nrsatish, kaushikr, keutzer}@eecs.berkeley.edu

Abstract

We present a decomposition strategy to speed up constraint optimization for a representative multiprocessor scheduling problem. In the manner of Benders decomposition, our technique solves relaxed versions of the problem and iteratively learns constraints to prune the solution space. Typical formulations suffer prohibitive run times even on medium-sized problems with less than 30 tasks. Our decomposition strategy enhances constraint optimization to robustly handle instances with over 100 tasks. Moreover, the extensibility of constraint formulations permits realistic application and resource constraints, which is a limitation of common heuristic methods for scheduling. The inherent extensibility, coupled with improved run times from a decomposition strategy, posit constraint optimization as a powerful tool for resource constrained scheduling and multiprocessor design space exploration.

1. Introduction

Static compile time task allocation and scheduling is an important step in deploying concurrent applications on multiprocessors. When the application workload and parallel tasks are known at compile time, it is viable to determine an application mapping statically. Signal processing applications derived from static data flow specifications can be statically scheduled on multiprocessors [1]. Static scheduling is also relevant to rapid design space exploration (DSE) for microarchitectures and systems [2].

Kwok and Ahmed present a comprehensive taxonomy of static scheduling algorithms to map a directed acyclic task graph to a network of processors [3]. Their survey indicates that the fastest methods are heuristics based on list scheduling. However, heuristics are inherently "short-sighted" and provide no guarantee of optimality. But besides speed and optimality, another important metric is the extensibility of an approach: a measure of how easy it is to accommodate practical implementation and resource constraints. For instance, in a DSE framework, the result of a certain mapping imposes new restrictions on the application and microarchitecture to guide exploration. Unfortunately, most heuristic methods are not easily extensible. They are customized for a specific problem and will have to be reworked each time new assumptions or constraints are imposed. Tompkins corroborates this observation that "(approximation heuristics) lack the complexity necessary for modeling any real world scheduling problem" [4].

Constraint optimization formulations, such as Mixed Integer Linear Programming (MILP) and Constraint Programming (CP), are naturally extensible. For example, Thiele proposes using MILP to solve a scheduling problem with complex resource constraints on memory size, communication bandwidth and access conflicts to memories and buses [5]. However, the drawback of using a generic solver is the significant computation cost. In previous work related to DSE frameworks, we also adopted MILP on account of its extensibility [6]. However, the solver failed to find even a single non-optimal solution different from a trivial one for many medium-sized problems with fewer than 30 tasks.

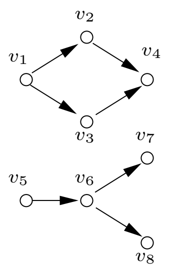
In this paper, we present a decomposition strategy to speed up constraint optimization that is applicable to many variants of the static scheduling problem with diverse implementation and resource constraints. The basic idea is to divide the scheduling problem into a constraint optimization "master" problem and a fast graph-theoretic "sub" problem and solve them iteratively. The master solves a simplified optimization problem and generates trial solutions. The sub problem analyzes these solutions and in turn learns constraints to incrementally prune inferior parts of the solution space. The master problem preserves the generality of constraint optimization, while the sub problem computation is specialized to quickly analyze partial solutions, derive tight lower bounds, and efficiently direct search in the master problem. This approach is inspired by the more general *Benders decomposition* technique of "learning from one's mistakes" [7].

In order to demonstrate the effectiveness of our decomposition strategy for constraint optimization, we apply it to a representative static scheduling problem. We evaluate its performance against two competitive approaches: an MILP formulation [4] and a list scheduling heuristic [8]. We show that our approach can robustly handle large problem instances with over 100 tasks, outdoing MILP and other existing constraint formulations. We also discuss how practical implementation and resource constraints can be enforced, and why these extensions may be difficult to accommodate in heuristic methods. Thus, the decomposition strategy retains the extensibility of constraint optimization and improves solver performance. We advance this as a general solution technique that has significant potential for many variants of the scheduling problem with realistic implementation and resource constraints.

This paper is organized as follows. Section 2 states the representative scheduling problem. Section 3 describes existing MILP and list scheduling approaches to solve this problem. Section 4 details our decomposition strategy. Section 5 presents results and evaluates the performance and extensibility of these approaches. Section 6 summarizes our work.

2. Problem Description

We consider a representative static scheduling problem for which we shall demonstrate our approach. The objective is to schedule a task dependence graph with communication delays on a multiprocessor to minimize schedule length, or *makespan*. The input task graph is a directed acyclic graph (DAG) $G = (V, E)$, where vertexes $V = \{v_1, \dots, v_n\}$ are computation tasks, and directed edges $E \subseteq V \times V$ denote dependence constraints and data transfers between tasks. The target multiprocessor, denoted by $P = \{p_1, \dots, p_m\}$, is a fully connected network of m identical processors. Each task is executed sequentially without preemption on a single processor. The execution time for task $v \in V$ is given by $w(v)$. The communication delay $c((v_1, v_2))$ along any edge $(v_1, v_2) \in E$ corresponds to the latency due to data transfer when tasks v_1 and v_2 are executed on different processors. When both tasks are assigned to the same processor no latency is incurred. Each processor is assumed to contain dedicated communication hardware so that computation can be overlapped with communication. An example task dependence graph with execution time and communication delay annotations is displayed in Figure 1.



	$w(v)$
v_1	3
v_2	3
v_3	3
v_4	7
v_5	2
v_6	3
v_7	5
v_8	6

	$c(e)$
(v_1, v_2)	9
(v_1, v_3)	5
(v_2, v_4)	7
(v_3, v_4)	4
(v_5, v_6)	3
(v_6, v_7)	10
(v_6, v_8)	4

Figure 1: An example task dependence graph with annotations for execution time of each task and communication delay of each edge (source: [8]).

Given a task dependence graph $G = (V, E)$ and a set of processors P, a *valid allocation* A is a function $A : V \to P$ that assigns every task in V to a single processor in P. For a valid allocation, a *valid schedule* S is a function $S : V \to \Re^+$ that assigns a non-negative start time to every task and satisfies the following three conditions:

$\forall (v_1, v_2) \in E$ (dependence constraints),

$(a)\ S(v_2) \geq S(v_1) + w(v_1)$

$(b)\ A(v_1) \neq A(v_2) \ \Rightarrow\ S(v_2) \geq S(v_1) + w(v_1) + c((v_1, v_2))$

$\forall v_1, v_2 \in V, v_1 \neq v_2$ (ordering constraints),

$$(c)\ A(v_1) = A(v_2) \ \Rightarrow \\ S(v_1) \geq S(v_2) + w(v_2) \ \vee\ S(v_2) \geq S(v_1) + w(v_1)\,.$$

The *makespan* of the schedule S is defined as: $\max_{v \in V} S(v) + w(v)$. The objective of the scheduling problem is to compute a valid allocation A and schedule S with *minimum makespan*.

3. Related Approaches

Various heuristic algorithms, branch-and-bound methods, evolutionary algorithms and constraint formulations have been devised for the representative scheduling problem in Section 2 [3]. We review two different, yet competitive approaches, which we later use to evaluate our solution.

3.1. MILP Formulations

The static scheduling problem has lent itself to a variety of MILP formulations [5] [6]. These are typically "single-pass" formulations (the problem constraints are presented at the start of execution) intended for a commercial MILP solver like ILOG CPLEX [9], and do not advocate any decomposition or branching strategies. From experiments, we observed that an MILP model due to Tompkins based on overlap variables was superior to others over a large set of examples [4]. In Section 5, we evaluate our approach relative to this formulation.

3.2. A List Scheduling Heuristic using Dynamic Levels

List scheduling assigns a priority to each task and schedules them in descending order of priority. Sih and Lee proposed a dynamic level scheduling algorithm (DLS), which accounted for communication delays in list scheduling [8]. The DLS algorithm uses an attribute called *dynamic level* to assess the priority of every task-processor pair and selects the pair with the highest dynamic level at each scheduling step. Independent benchmarking efforts for static scheduling algorithms due to Kwok and Ahmad [3] and Davidović and Crainic [10] have endorsed the effectiveness of the DLS algorithm for scheduling task graphs onto a bounded number of processors. The algorithm runs on problem instances with over 200 tasks in seconds, and the results are usually close to the minimum makespan.

4. Decomposition-based Constraint Optimization

The objective is to improve the performance of constraint optimization formulations for the static scheduling problem in Section 2. Our solution scheme is a problem decomposition with two components which are solved iteratively: a constraint optimization "master" problem, and a graph-theoretic "sub" problem. Instead of solving a complex optimization problem in one pass, the decomposition approach iteratively solves simpler versions of the same problem and learns constraints at each iteration to prune the solution space.

Algorithm 1 DA$(G, w, c, P) \to\ mspan$

```
1   $mspan = \infty$
2   $\phi$ = empty CNF formula
3   BASECONSTRAINTS$(G, w, c, P, \phi)$
4   while $(true)$
        // master problem
5       $x_{SAT}$ =SATSOLVE$(\phi)$
6       if $(x_{SAT}$ = UNSAT$)$     return $mspan$
        // sub problem
7       $G'$ = UPDATEGRAPH$(G, x_{SAT})$
8       if ($G'$ contains a cycle)
9           CYCLECONSTRAINTS$(G', x_{SAT}, \phi))$
10      else
11          $mspan = \min\{mspan,$ MAKESPAN$(G')\}$
12          PATHCONSTRAINTS$(G', w, c, mspan, x_{SAT}, \phi)$
```

The basic flow of our decomposition approach is outlined in Algorithm 1. The constraints for the master problem are formulated in conjunctive normal form (CNF) (line 3). Our choice of a CNF-based encoding is motivated in part by the ease of incorporating learned constraints from the sub problem as Boolean clauses.

We use a satisfiability (SAT) solver to solve the constraint optimization master problem (line 5) [11]. A satisfiable solution to the master problem allocates tasks to processors and orders all tasks allocated to the same processor. The sub problem inspects the satisfying assignment and updates the dependence graph to reflect task orderings within each processor (line 7). One of two possible scenarios occur next. (a) There are cyclic dependencies involving tasks assigned to the same processor. These may occur due to the task ordering selected by the satisfying assignment. In this case, the sub problem records these cycles as constraints for the master problem to avoid revisiting solutions containing these cycles in future iterations (line 9). (b) No cyclic dependencies exist between tasks. In this case, the result of the master problem is a *valid allocation* and a *valid schedule*. The sub problem computes the makespan of this schedule and conditionally updates the best makespan (line 10). It then adds constraints to the master problem to prune parts of the solution space that are guaranteed to contain inferior schedules with makespans greater than the best makespan (line 11). The constraints learned in scenarios (a) and (b) prune out superfluous solutions and reduce the solution space that the master problem must search to find the optimum. Alternatively, if the master problem itself is unsatisfiable, then the best makespan seen thus far is the minimum makespan for the static scheduling problem (line 6).

4.1. Master Problem Formulation

There are three sets of Boolean variables in the CNF formulation of the master problem:

$\forall v \in V, \forall p \in P,$

$$x_a(v,p) = \begin{cases} 1 & : \text{ if task } v \text{ assigned to processor } p \\ 0 & : \text{ else} \end{cases}$$

$\forall (v_1, v_2) \in E,$

$$x_c(v_1,v_2) = \begin{cases} 1 & : \text{ if tasks } v_1 \text{ and } v_2 \text{ are assigned} \\ & \quad \text{to different processors} \\ 0 & : \text{ else} \end{cases}$$

$\forall v_1, v_2 \in V, \quad (v_1, v_2) \notin E^T, (v_2, v_1) \notin E^T,$

$$x_d(v_1,v_2) = \begin{cases} 1 & : \text{ if task } v_1 \text{ precedes task } v_2 \\ 0 & : \text{ else} \end{cases}$$

Allocation variables x_a indicate task assignment to processors. Communication variables x_c indicate if the communication delay is incurred between two dependent tasks (this occurs when the tasks are assigned to different processors). Ordering variables x_d indicate an assumed precedence between two *non-dependent* tasks when assigned to the same processor. We denote by $G^T = (V^T, E^T)$, the transitive closure of the directed graph $G = (V, E)$. Then two tasks v_1 and v_2 are *non-dependent* if $(v_1, v_2) \notin E^T$ and $(v_2, v_1) \notin E^T$. No dependence can be inferred between v_1 and v_2 in G, hence the master problem selects an ordering between v_1 and v_2 when they are assigned to the same processor.

There are three types of base constraints for the master problem (see adjacent column). Constraints $A1$ and $A2$ are allocation constraints which assign each task to exactly one processor. Constraint $A2$ specifically emphasizes that a conflict occurs if a task is assigned to two processors. Constraint $C1$ determines when a communication delay is incurred between two dependent tasks. The value of the communication variable $x_c(v_1, v_2)$ depends entirely on the values of the allocation variables for v_1 and v_2. Nevertheless, using x_c as a primary variable enables the sub problem to record compact constraints with these variables. Constraints $D1$ and $D2$ enforce the selection of a valid order between two *non-dependent* tasks assigned to the same processor. These constraints create a total order among all tasks in a processor.

$$\begin{aligned}
&\forall v \in V, \\
&\quad (A1) \quad \Big(\bigvee_{p \in P} x_a(v,p) \Big) \\
&\forall v \in V, \forall p_1, p_2 \in P, p_1 \neq p_2, \\
&\quad (A2) \quad x_a(v,p_1) \wedge x_a(v,p_2) \Rightarrow 0
\end{aligned}$$

$$\begin{aligned}
&\forall (v_1, v_2) \in E, \forall p_1, p_2 \in P, p_1 \neq p_2, \\
&\quad (C1) \quad x_a(v_1,p_1) \wedge x_a(v_2,p_2) \Rightarrow x_c(v_1,v_2)
\end{aligned}$$

$$\begin{aligned}
&\forall v_1, v_2 \in V, (v_1,v_2) \notin E^T, (v_2,v_1) \notin E^T, \forall p \in P, \\
&\quad (D1) \quad x_a(v_1,p) \wedge x_a(v_2,p) \Rightarrow x_d(v_1,v_2) \vee x_d(v_2,v_1) \\
&\quad (D2) \quad x_d(v_1,v_2) \wedge x_d(v_2,v_1) \Rightarrow 0
\end{aligned}$$

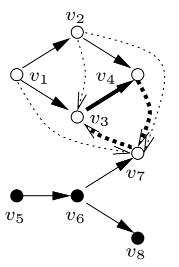
4.2. Sub Problem Decomposition Constraints

A satisfiable solution x_{SAT} to the master problem assigns values to the x_a, x_c and x_d variables consistent with the base constraints and additional learned constraints from previous iterations. The x_a variables assigned to 1 generate a *valid allocation* A, which allocates each task to exactly one processor. The x_d variables assigned to 1 denote assumed dependence edges between *non-dependent* tasks in G that are assigned to the same processor under A. Thus, the solution x_{SAT} of the master problem is a *valid allocation* A and an updated task graph, $G' = (V, E')$, where:

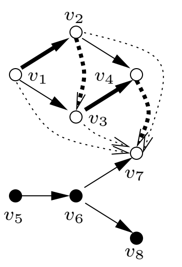
$$E' = E \cup \left\{ \begin{array}{ll} (v_1, v_2) \mid & x_d(v_1,v_2) \wedge \\ & \big(\exists p \in P : \; x_a(v_1,p) \wedge x_a(v_2,p)\big) \end{array} \right\}.$$

The sub problem analyzes the allocation A and the updated graph G' and learns constraints to direct search in the master problem. As an example, consider the problem of statically scheduling the task graph in Figure 1 on a 2-processor system. Figure 2 presents two different solutions derived from the master problem. The assignments to the x_a and x_c variables are identical in (a) and (b). They share the same valid allocation A, as listed in Figure 2. Solutions (a) and (b) differ only in their assignments of the variables $x_d(v_3, v_7)$ and $x_d(v_7, v_3)$. Consequently, the direction of the assumed dependence edge between v_3 and v_7 is different in (a) and (b).

Solution (a) contains a cycle involving tasks v_3, v_4 and v_7, which prohibits derivation of any *valid schedule*. The sub problem detects this cycle and encodes it as a constraint for the master problem to avoid any solution containing this cycle in future iterations. Specifically, the cycle constraint in (a) is: $x_d(v_4, v_7) \wedge x_d(v_7, v_3) \Rightarrow 0$. Since the original graph G is a DAG, a cycle in G' arises only due to assumed dependence edges, which correspond to the x_d variables assigned to 1. More generally, if



(a) Cycle: (v_3, v_4, v_7)



(b) Critical path: $(v_1, v_2, v_3, v_4, v_7)$
Makespan = 21

$A(v)^{(a,b)}$		$S(v)^{(b)}$	
v_1	p_1	v_1	0
v_2	p_1	v_2	3
v_3	p_1	v_3	6
v_4	p_1	v_4	9
v_5	p_2	v_5	0
v_6	p_2	v_6	2
v_7	p_1	v_7	16
v_8	p_2	v_8	5

Figure 2: Two solutions to the master problem for scheduling the task graph in Figure 1 on a 2-processor system. The dotted edges are the assumed dependence edges due to x_d variables. The thicker edges are part of the cycle in (a) and the critical path in (b).

$E_c = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$ are the edges comprising a cycle in G', then the associated cycle constraint is:

$$\Big(\bigwedge_{(u,v)\in E_c - E} x_d(u, v)\Big) \;\Rightarrow\; 0\,.$$

Solution (b) in Figure 2 does not contain cycles, hence a *valid schedule* S can be computed that assigns a non-negative start time to every task. The objective is to minimize schedule length, therefore an obvious value for $S(v)$ is the delay of the longest path from any source vertex in the DAG G'. This adheres to the dependence and ordering constraints requisite for a valid schedule. The schedule S is computed in topological order of the vertexes in G': if v is a source vertex, $S(v) = 0$, otherwise

$$S(v) = \max_{(u,v)\in E} S(u) + w(u) + x_c(u, v)\; c(u, v)$$

The makespan of S is: $\max_{v\in V} S(v) + w(v)$. The algorithm keeps track of the best makespan from all past iterations and updates it if the makespan of the current schedule S is lower than the incumbent makespan.

Coming back to the example, the *valid schedule* S for solution (b) is listed in Figure 2. The resulting makespan is 21 due to the critical path $(v_1, v_2, v_3, v_4, v_7)$. Furthermore, any solution to the master problem containing this critical path cannot yield a makespan lower than 21. The sub problem records this fact as a constraint to eliminate all solutions containing the critical path. Specifically, the constraint that encodes the critical path in (b) is: $x_d(v_2, v_3) \wedge x_d(v_4, v_7) \Rightarrow 0$.

Note that the algorithm is not restricted to learning only the critical path as a constraint. Any path in G' with delay greater than or equal to the best makespan provides a valid path constraint. In example (b), if we assume that the incumbent makespan is 20, then the path (v_5, v_6, v_7) also bounds the makespan and is a valid constraint (the path has delay 20 since edge (v_6, v_7) incurs a communication delay). This can be encoded as: $x_c(v_6, v_7) \Rightarrow 0$. The delay of any path is due only to the assumed dependence edges (x_d values) and the original graph edges on which a communication delay is incurred (x_c values). More generally, if $E_p = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$ are the edges comprising a path in G' with delay greater than or equal to the best makespan at an iteration, then the associated path constraint is:

$$\Big(\bigwedge_{(u,v)\in E_p - E} x_d(u, v)\Big) \wedge \Big(\bigwedge_{(u,v)\in E_p\cap E,\, A(u)\neq A(v)} x_c(u, v)\Big) \;\Rightarrow\; 0.$$

4.3. Algorithmic Improvements

In this section, we briefly describe some algorithmic extensions to boost performance of the decomposition strategy. In Algorithm 1, the sub problem is invoked only after the master fully solves the satisfiability problem and generates an assignment for all primary variables. However, valid cycle and path constraints can be derived from partial solutions at intermediate nodes in the search tree of the SAT solver. This provides a technique to fathom partial solutions early in the SAT search tree and prune potentially large subsets of inferior solutions. Tight lower bounds on the makespan of partial solutions reinforce the pruning condition. Two obvious lower bounds on the makespan are: (a) the delay of the longest path in the task graph, and (b) the ratio of the total execution time of unassigned tasks to the number of processors. A more complex lower bound, due to Gerasoulis and Yang, propagates the minimum schedule length of primitive *fork* and *join* graph structures in the presence of communication delays [12].

An useful side effect of inspecting partial solutions is that the sub problem can recommend a decision variable for the master problem to branch on. For example, if two tasks assigned to the same processor violate ordering constraints, the sub problem can determine the next x_d variable that must be set to resolve this conflict. Additionally, the sub problem can apply a priority metric, such as *dynamic levels* from Section 3.2, and select a task-processor allocation for the master problem. Thus, the technique of inspecting partial solutions enables the sub problem to closely direct search and prune the solution space quickly.

4.4. Related Decomposition Approaches

The use of a decomposition strategy to speed up constraint optimization is not new. *Benders decomposition* is a problem solving strategy that has been successfully applied to MILP formulations. A pivotal factor that determines its effectiveness is the derivation of *Benders cuts* to exclude superfluous solutions. It is essential to choose an encoding, based on some insight into the problem structure, that eases the incorporation of *Benders cuts*.

Hooker and Ottoson formally proved how *Benders decomposition* can be generalized for any class of constraint optimization problems besides MILP [7]. Following this, problem decompositions using hybrid MILP and Constraint Programming (CP) models were shown to achieve run time improvements over "single-pass" MILP or CP models. In the context of static scheduling, Benini, et al. considered a problem similar to the one presented in this paper, but with additional constraints on the memory per processor [13]. The authors applied an MILP/CP decomposition and showed that decomposition was superior to solving either model separately.

However, these works reported experimental results for problems containing fewer than 20 tasks. We contend that the use of CP solvers for the sub problem is less efficient. CP models are generic and do not sufficiently exploit the underlying graph structure of the scheduling problem. In contrast, the success of list scheduling methods is due to their ability to operate directly on the task dependence graph and derive schedules using longest path delay computations. Based on this insight, our approach uses a fast graph-theoretic sub problem algorithm to learn compact *Benders cuts* and quickly prune the solution space.

5. Results and Evaluations

In this section, we present results of our experiments to evaluate the decomposition strategy against the DLS list scheduling heuristic [8] and the MILP formulation using overlap variables [4] for the scheduling problem in Section 2. We created a prototype implementation of the decomposition approach on top of the MiniSAT SAT solver [11]. The MILP instances were solved using the ILOG CPLEX v10.1 solver [9]. The experiments were conducted on a PentiumIV 2.4 GHz processor with 1GB RAM running Linux. We stipulated a timeout of 5 minutes for all our runs.

Two benchmark sets were used in our experiments. The first benchmark consisted of task graph instances derived from two practical applications from the multimedia and networking domains, viz. MJPEG decoding and IPv4 packet forwarding. The task execution times and communication delays were profiled for the Xilinx MicroBlaze soft processor and the point-to-point fast simplex links (FSL) available with the Xilinx Embedded Development Kit (EDK) [14]. The base task dependence graph can be replicated to exploit the coarse grained parallelism available in these applications. For MJPEG decoding, each replicated copy processes a different frame. For IPv4 packet forwarding, the number of replications corresponds to the number of network input ports. We generated 10 problem instances for each application for different replications of the base task graph. The second benchmark was a set of random task graph instances proposed by Davidović et.al [10]. These problems were designed to be unbiased towards any particular solver approach and are reportedly harder than other existing benchmarks for scheduling task dependence graphs.

Table 1 reports the results of the different scheduling approaches for MJPEG decoding and packet forwarding applications. Column 1 gives the number of tasks in the task dependence graph. The subsequent columns report the makespan computed by the DLS, MILP and decomposition approach (abbreviated "DA") for scheduling the task graph on 2, 4, and 6 processors. The makespan results that were proved to be optimal by MILP and DA are also highlighted. The non-bold entries are the best solutions at the end of the 5-minute timeout.

#	# Processors = 2			# Processors = 4			# Processors = 6		
Tasks	DLS	MILP	DA	DLS	MILP	DA	DLS	MILP	DA
13	20198	**19328**	**19328**	14352	**14352**	**14352**	14352	**14352**	**14352**
24	36702	n/a	36636	22386	21684	**21618**	19112	**19112**	**19112**
35	54264	n/a	54162	30588	n/a	29886	24578	n/a	24190
46	72438	n/a	71640	39148	n/a	38856	30590	n/a	**29822**
57	89576	n/a	89576	48144	n/a	47602	37310	n/a	36770
68	107178	n/a	107178	56738	n/a	56738	42462	n/a	42462
79	124190	n/a	124190	65482	n/a	65172	47982	n/a	47982
90	142102	n/a	142102	74044	n/a	74044	54068	n/a	54068
101	159244	n/a	159244	87278	n/a	87278	59948	n/a	59948

#	# Processors = 2			# Processors = 4			# Processors = 6		
Tasks	DLS	MILP	DA	DLS	MILP	DA	DLS	MILP	DA
11	194	194	**194**	194	194	**194**	194	194	**194**
20	246	238	**238**	194	194	**194**	194	194	**194**
29	366	358	358	206	206	**206**	194	194	**194**
38	486	n/a	**476**	246	n/a	**238**	194	194	**194**
47	606	n/a	596	310	n/a	302	226	n/a	218
56	714	n/a	**714**	368	n/a	368	246	n/a	**238**
65	834	n/a	834	432	n/a	430	302	n/a	292
74	962	n/a	962	488	n/a	484	336	n/a	336
83	1082	n/a	1082	542	n/a	542	368	n/a	368
92	1190	n/a	**1190**	606	n/a	606	412	n/a	412

Table 1: Makespan results for the DLS, MILP and decomposition approach (DA) on task graphs derived from MJPEG decoding (top) and IPv4 packet forwarding (bottom) scheduled on 2, 4 and 6 processors.

Table 2 shows results of the DLS, MILP and DA methods on the second benchmark with randomly generated task graphs. The benchmark instances are classified by the number of tasks and edge density (the percentage ratio of the number of edges in the task graph to the maximum possible number of edges). Each entry in Table 2 is an average over 5 runs for a different number of processors. The optimal solutions for these instances was known *a priori* [10]. Columns 3-5 report the average percentage difference of the DLS, MILP and DA results from the optimal solution. Under the DA results, we also indicate in parenthesis the number of instances for which the algorithm proved optimality of the final solution and terminated before the timeout (DLS does not provide any proof of optimality).

# Tasks	Edge Density	DLS	MILP	DA (# optimal)
52	10	14.9	n/a	10.7 (0)
52	30	16.2	n/a	9.1 (1)
52	50	14.0	n/a	5.4 (3)
52	70	6.3	n/a	0 (5)
52	90	4.2	n/a	0 (5)
102	10	28.5	n/a	28.5 (0)
102	30	15.7	n/a	15.7 (0)
102	50	7.5	n/a	7.5 (0)
102	70	4.5	n/a	4.3 (1)
102	90	1.2	n/a	1.2 (3)
Avg. Difference		11.3	-	8.2
# Optimal Solutions		0 / 50	0 / 50	18 / 50

Table 2: Average percentage difference of DLS, MILP and decomposition approach (DA) from the optimal solution for random task graph instances (benchmark source: [10]) scheduled on 2, 4, 6, 9 and 12 processors.

We observe from Table 1 that the direct MILP formulation solved using the CPLEX solver [9] does not find any feasible solution on problem instances with more than 30 tasks (indicated by the "n/a" annotation). This trend seems to be invariant of the application task graph structure or the number of processors. Indeed, on many instances the solver does not go beyond the preprocessing steps involved in calculating a lower bound for the problem. The results in Table 2 further attest to the limitations of using an MILP formulation. In contrast, constraint optimization using our decomposition approach robustly handles problems with 80-100 tasks, and proves optimality of the final solution in about 35% of the cases in both Tables 1 and 2. This is an improvement over previous constraint optimization techniques using MILP formulations [4] or hybrid MILP/CP decomposition approaches [13], which were limited to instances with fewer than 20-30 tasks.

However, the constraint optimization results (MILP and DA) pale in comparison to the DLS results in Tables 1 and 2. This is consistent with earlier studies that indicated the efficacy of DLS and similar list scheduling heuristics for the representative scheduling problem [3] [10]. We observed from many experiments that the DLS solution was usually within 5-10% of the optimum on realistic task graphs and problem instances with over 200 tasks were solved in seconds. Our DA method quickly finds the DLS solution on all instances, since the sub problem internally uses *dynamic levels* to recommend task-processor allocations for the master problem. It is reasonably successful in improving the DLS makespan or proving its optimality on instances with 80-100 tasks. But on larger problems, DA seldom improves the DLS makespan. Nevertheless, the utility of MILP and DA, and constraint optimization methods in general, becomes evident if the problem to be solved imposes additional application and resource constraints on valid schedules.

5.1. Extensibility of Constraint Optimization Methods

Heuristic approaches like DLS are not easily extensible to include specialized implementation and resource constraints in the problem. For example, practical multiprocessor architectures are connected in specific topologies such as ring or mesh topologies. Bambha and Bhattacharyya proposed an extension to the DLS list scheduling heuristic for irregular multiprocessor topologies [15]. However, we observed from our experiments that extending DLS with topology constraints compromised the original quality of its results. Table 3 shows the results of running the DLS and DA algorithms on a few instances of the MJPEG decoding application on processors arranged in ring and mesh topologies. On the average, DLS results were about 15-20% inferior compared to the DA results (previously DLS results were inferior only by 2-5% of the DA results for the basic problem without topology constraints).

	#	# Processors = 4		# Processors = 6		# Processors = 9	
Topology	Tasks	DLS	DA	DLS	DA	DLS	DA
Ring	46	41350	40368	49280	33898	48834	38958
	79	67888	67888	62380	53362	62310	58568
	101	87834	84398	75246	68120	80834	75422
Mesh	46	41350	40368	43108	31972	43108	28632
	79	67888	67888	59484	53732	63724	45842
	101	87834	84398	68470	68470	72162	63304

Table 3: Makespan results for the DLS and DA approaches on task graphs derived from MJPEG decoding scheduled on 4, 6 and 9 processors arranged in ring and mesh topologies.

Other resource constraints may be more difficult to express in the list scheduling heuristic. An example of such a constraint is limits on the total amount of memory available on each processor. The problem objective is to find a valid schedule with minimum makespan subject to the constraint that the memory consumption of all tasks assigned to a processor is within the prescribed limit. List scheduling heuristics like DLS sequentially fix tasks to processors based on some local cost function. However, after a few greedy choices, the heuristic can reach a configuration from which no valid solutions are possible. In that case, the heuristic must undo its choices until a feasible configuration is reached. For such constraints, a greedy heuristic is not even guaranteed to find a single valid solution.

Practical extensions such as (a) task release times and deadlines, (b) preferred allocations of tasks, (c) mutual exclusion between execution periods of certain tasks, (d) synchronization requirements between tasks, and (e) constraints on task groupings are generally difficult to impose in DLS-like heuristics. Constraint optimization methods, on the other hand, provide an easy way to encode resource constraints and efficiently integrate them into the problem. They leverage the advantage of a generic search tool that does not impose many restrictions on the nature of the constraints. In this context, our contribution is a decomposition strategy to improve the performance of constraint optimization that is applicable to many variants of the static scheduling problem.

6. Summary

In this paper, we presented a decomposition approach to speed up constraint optimization for statically scheduling task dependence graphs to multiprocessors. List scheduling heuristics are effective for this problem, but difficult to extend with specialized implementation and resource constraints. Constraint optimization methods, such as MILP, are naturally extensible but their success is limited by prohibitive run times even on medium-sized problem instances. Our decomposition strategy addresses this limitation in a manner similar to the more general *Benders decomposition* technique. The main elements of our strategy are: (a) fast master and sub problem iterations, (b) compact sub problem constraints to record inferior parts of the solution space, (c) tight lower bounds to fathom partial solutions, and (d) variable selection to guide search. While an MILP formulation for the scheduling problem was ineffective on instances with over 30 tasks, our approach was robust on instances with over 100 tasks. The performance improvement owing to our decomposition strategy places constraint optimization as a viable tool for practical resource constrained multiprocessor static scheduling problems.

In direction of future work, we intend to investigate the following techniques to improve solver performance: (a) different master problem formulations for specialized solvers (b) improved lower bounds for special graph structures, and (c) symmetry representation to restrict the solution space. The goal is to integrate our solution method into a practical design space exploration tool for multiprocessor platforms.

References

[1] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[2] M. Gries, "Methods for Evaluating and Covering the Design Space during Early Design Development," *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, 2004.

[3] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, 1999.

[4] M. F. Tompkins, "Optimization Techniques for Task Allocation and Scheduling in Distributed Multi-Agent Operations," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2003.

[5] L. Thiele, "Resource Constrained Scheduling of Uniform Algorithms," *VLSI Signal Processing*, vol. 10, pp. 295–310, Aug 1995.

[6] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES'05)*, pp. 273–278, ACM Press, 2005.

[7] J. Hooker and G. Ottosson, "Logic-Based Benders Decomposition," Dec 1999. http://www.citeseer.ist.psu.edu/hooker95logicbased.html.

[8] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, 1993.

[9] "ILOG CPLEX v10.1." http://www.ilog.com/products/cplex/.

[10] T. Davidović and T. G. Crainic, "Benchmark-Problem Instances for Static Scheduling of Task Graphs with Communication Delays on Homogeneous Multiprocessor Systems," *Computers & OR*, vol. 33, pp. 2155–2177, 2006. http://www.mi.sanu.ac.yu/∼tanjad/tanjad_pub.htm.

[11] N. Een and N. Sörensson, "An Extensible SAT-solver [ver 1.2]," in *Lecture Notes in Computer Science* (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of *SAT*, pp. 502–518, Springer, 2003.

[12] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs onto Multiprocessors," *J. Parallel Distrib. Computing*, vol. 16, pp. 276–291, Dec 1992.

[13] L. Benini, D. Bertozzi, A. Guerri, and M. Milano, "Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation," in *Principles and Practice of Constraint Programming, 11th International Conference*, pp. 107–121, 2005.

[14] Xilinx, Inc., *Embedded Systems Tools Guide*, Xilinx Embedded Development Kit, EDK version 6.3i ed., June 2004.

[15] N. K. Bambha and S. S. Bhattacharyya, "System Synthesis for Optically Connected Multiprocessors on Chip," in *In Proc. of the International Workshop for System on Chip*, 2002.