Parallel Processor Scheduling with Delay Constraints

Daniel W. Engels* Jon Feldman† David R. Karger‡ Matthias Ruhl§

MIT Laboratory for Computer Science
Cambridge, MA 02139, USA

Abstract

We consider the problem of scheduling unit-length jobs on identical parallel machines such that the makespan of the resulting schedule is minimized. Precedence constraints impose a partial order on the jobs, and both communication and precedence delays impose relative timing constraints on dependent jobs. The combination of these two types of timing constraints naturally models the instruction scheduling problem that occurs during software compilation for state-of-the-art VLIW (Very Long Instruction Word) processors and multiprocessor parallel machines.

We present the first known polynomial-time algorithm for the case where the precedence constraint graph is a forest of in-trees (or a forest of out-trees), the number of machines m is fixed, and the delays (which are a function of both the job pair and the machines on which they run) are bounded by a constant D.

Our algorithm relies on a new structural theorem for scheduling jobs with arbitrary precedence constraints. Given an instance with many independent dags, the theorem shows how to convert, in linear time, a schedule S for only the largest dags into a complete schedule that is either optimal or has the same makespan as S.

1 Introduction

In this paper we consider the problem of scheduling unit-length jobs on m identical parallel machines to minimize the makespan in the presence of *precedence constraints*, *precedence delays* and *communication delays*. Precedence constraints model dependencies between the tasks; if job j depends on job i, then job j must be executed after job i. Precedence delays $l_{i,j}$ impose relative timing constraints; job j cannot begin execution until at least $l_{i,j}$ time steps after job i completes. Communication delays $c_{i,j}$ impose delays across machines; if jobs i and j run on different machines, job j cannot begin execution until at least $c_{i,j}$ time steps after job i completes.

Previous algorithms for scheduling jobs on parallel machines consider either communication delays or precedence delays, but not both. In this paper we generalize both types of delays to a single *separation delay* $\ell_{i,j,a,b}$, where job j running on machine b cannot begin execution until at least $\ell_{i,j,a,b}$ time units after job i completes on machine a. Moreover, we overcome the restriction of previous exact algorithms where delays could only be either 0 or 1.

We give a polynomial algorithm for the case where the precedence graph is a forest[1] and the delays are bounded by a constant D. We also give a useful structural theorem for instances where the precedence graph is a collection of independent dags; we show that any schedule S for the largest dags can be converted, in linear time, into a complete schedule that is either optimal or has the same makespan as S. Our interest in this problem is motivated by the instruction scheduling problem encountered by compilers for emerging system architectures.

Instruction scheduling for parallel machine and VLIW compilation. VLIW (Very Long Instruction Word) architectures have recently begun to appear in a variety of commercial processor and embedded system designs. In these architectures, the processor contains multiple functional units capable of executing basic operations in parallel in one clock cycle. The VLIW processor is controlled by *meta-instructions* that combine the instructions for the individual functional units into one single instruction word, hence the name VLIW.

The VLIW architecture is the basis for Intel's Itanium chip (formerly code-named Merced), which is scheduled for commercial release in 2000. It uses a new instruction set named IA-64 [9], which was developed by Intel and Hewlett-Packard, and is based on EPIC (Explicitly Parallel Instruction Computing) – Intel's adaptation of VLIW. VLIW architectures have also been used in state-of-the-art Digital

---

*E-Mail: `dragon@lcs.mit.edu`
†E-Mail: `jonfeld@theory.lcs.mit.edu`
‡E-Mail: `karger@theory.lcs.mit.edu`
§E-Mail: `ruhl@theory.lcs.mit.edu`

[1] When we say that the precedence graph is a forest, we mean that it is either a collection of in-trees, or a collection of out-trees.

Signal Processor (DSP) designs, such as the popular Texas Instruments TMS320C6x series [15].

The role of the compiler is much more crucial for VLIW architectures than it is for traditional processors. To exploit the inherent hardware parallelism, the compiler must combine basic operations into meta-instructions in an efficient way. When doing so, it has to observe the data dependencies between the operations and the time it takes to transfer data from one functional unit to another. Since hardware based acceleration schemes such as branch prediction or speculative execution become less powerful on these implicitly parallel architectures, it is the compiler that really determines the quality of the resulting code. This quality is especially important in embedded system design, where the code is only compiled once (making even lengthy compilation times acceptable), but an optimal performance is required of the resulting system.

Our scheduling problem exactly fits this model. Each meta-instruction can be thought of as a slice of time, and the functional units correspond to machines. Pipelining allows all jobs to have unit execution time. Precedence constraints encode the data dependencies, and delays encode the latencies: variable pipeline lengths and limited bypassing create variable precedence delays, and data movement between functional units creates communication delays. Since all the functional units are part of the same processor, precedence delays and communication delays are on the same order of magnitude, and should be considered together. Furthermore, fixing the number of machines and imposing a bound on the delays makes sense in this context; these quantities are a function of the physical characteristics of the chip, and are usually small[2].

Determining a minimum makespan schedule for arbitrary instruction dependencies is a long-standing open problem (see section 1.1). We therefore focus on scheduling forests, which often occur in practice, for example, when processing expression trees or divide-and-conquer algorithms.

Problem statement. We are given a set of n jobs and m machines on which to execute the jobs, where m is a constant. Each job has unit processing time. There exists a directed acyclic precedence graph $G = (V, E)$ on the jobs V. With each precedence-constrained job pair $(i, j) \in E$, and pair of machines (a, b), there is an associated nonnegative delay $\ell_{i,j,a,b}$ bounded by a constant D. The output is a schedule assigning a job to each processor and time slot. A schedule is legal iff it includes all jobs, and for all precedence-constrained job pairs $(i, j) \in E$, if job j runs on machine b at time t, job i must be scheduled on some machine a *before* time $t - \ell_{i,j,a,b}$ (i.e., there must be $\ell_{i,j,a,b}$ time units *between* them).

We denote the completion time of job j as C_j. We are concerned with minimizing the makespan, $C_{\max} = \max_j C_j$. Let $C^*_{\max}$ be the optimal value of $C_{\max}$. Extending the notation introduced by Graham et al. [8], we can denote the problems considered in this paper as $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$.

We can also allow multiple instances of the same job to be scheduled on different machines; this is called *job duplication*. Allowing job duplication can make a difference in the makespan of a schedule when computing the same value twice is more efficient than transferring the value across machines (see section 4.1).

Our contribution. We give a polynomial-time algorithm for the problem where the precedence graph G is a forest: $Pm \mid tree; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$. The algorithm works with or without job duplication allowed on a job-by-job basis.

Our result is more general than previous known polynomial algorithms in both the precedence delay and the communication delay communities for optimally scheduling trees on a fixed number of processors. Previous results assumed at most unit time delays: Varvarigou, Roychowdhury and Kailath [17] solve $Pm \mid tree; p_j = 1; c_{i,j} = 1 \mid C_{\max}$. Bernstein and Gertner [1] solve $1 \mid tree; p_j = 1; l_{i,j} \in \{0, 1\} \mid C_{\max}$. Our algorithm solves both these problems as special cases. Another important contribution of this paper is the Merge theorem:

THEOREM 1.1. (THE MERGE THEOREM) *Consider an instance of* $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$ *where the precedence graph* G *contains at least* $2m(D+1) - 1$ *independent dags. Given a schedule with makespan* T *for only the jobs from the largest* $2m(D+1) - 1$ *dags, one can construct in linear time a schedule for all jobs with makespan* $\max\{\lceil \frac{n}{m} \rceil, T\}$.

Since this theorem holds for *any* dag, not just trees, it shows that any heuristic or approximation algorithm for scheduling only the jobs from large dags can be extended into an algorithm for scheduling all jobs. The theorem might also be applied to single dags after they have been broken into independent pieces. Furthermore, since a schedule of length $\lceil \frac{n}{m} \rceil$ is clearly optimal, the new algorithm will have the same performance guarantee as the original algorithm with only a linear time additive cost in running time.

1.1 Related Work

Polynomial algorithms: precedence delays. Precedence delays have been used to model single-processor latencies that arise due to pipelined architectures. Bernstein and Gertner [1] use a modification of the Coffman-Graham algorithm [3] to solve $1 \mid prec; p_j = 1; l_{i,j} \in \{0, 1\} \mid C_{\max}$.

---

[2] As an example, Intel's Itanium chip has six functional units, and Texas Instruments' TMS320C6x has eight.

Finta and Liu [5] give a polynomial time algorithm for the more general $1 \mid prec; p_j; l_{i,j} \in \{0,1\} \mid C_{\max}$. Both of these algorithms crucially depend on assuming unit-delays between jobs.

Polynomial algorithms: communication delays. In the classical models of parallel computation, communication delays are orders of magnitude larger than precedence delays, so algorithms for scheduling on parallel machines have generally ignored precedence delays. A survey by Chrétienne and Piccoleau [2] gives an overview of the work in this area.

All previous polynomial-time algorithms for a bounded number of machines work only for the special case of unit communication delays. Varvarigou, Roychowdhury and Kailath [17] show that $Pm \mid tree; c_{ij} = 1; p_j = 1 \mid C_{\max}$ is solvable in time $O(n^{2m})$ by converting the tree into one without delays. This conversion relies heavily on the fact that that the delays are unit-length. The special case $m = 2$ was shown to be solvable in $O(n^2)$ time by Picouleau [14], and was later improved to linear time by Lenstra, Veldhorst and Veltman [11], using a type of list scheduling.

Finta and Liu [6] give a quadratic algorithm for $P2 \mid SP1; p_j = 1; c_{ij} = 1 \mid C_{\max}$, where $SP1$ are *series-parallel-1* graphs, a subclass of series-parallel graphs. There has also been some work on approximation algorithms for an arbitrary number of machines. Möhring and Schäffter [12] give a good overview of this area.

Several authors (e.g. [10, 13]) have considered related problems where the number of processors is unbounded, i.e. the schedule can use as many processors as desired. However, that model is fundamentally different from the one we study, since optimal schedules usually make extensive use of the unlimited parallelism.

Hardness results. Even without any delays, the problem is NP-hard if the precedence relation is arbitrary and the number of machines is part of the input. This is the classic result of Ullman [16], showing NP-hardness of $P \mid prec; p_j = 1 \mid C_{\max}$. Lenstra, Veldhorst and Veltman [11] show the problem is still NP-hard when the precedence graph is a tree and there are unit communication delays ($P \mid tree; c_{ij} = 1; p_j = 1 \mid C_{\max}$).

Engels [4] proves NP-hardness for the single-machine case when the precedence constraints form chains, and the delays are restricted to be either zero or a single input value, i.e., he shows $1 \mid chain; p_j = 1; l_{i,j} \in \{0,d\} \mid C_{\max}$ to be strongly NP-hard, where d is an input to the problem.

When the processing times are not unit, the problem is also NP-hard. Engels [4] shows that scheduling chains with job processing times of either one or two and constant precedence delays, i.e., $1 \mid chain; p_j \in \{1,2\}; l_{i,j} = D \geq 2 \mid C_{\max}$, is strongly NP-hard.

Thus the only natural gap between our result and NP-hard problems is the generalization to arbitrary precedence structures on a fixed number of machines, i.e., the problem $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0,1,\ldots,D\} \mid C_{\max}$. However, this gap comes as no surprise, since the famous 3-processor scheduling problem ([7], problem [OPEN8]) is a special case. It turns out that even an algorithm for the one-processor version where all delays are equal to three ($1 \mid prec; p_j = 1; \ell_{i,j} = 3 \mid C_{\max}$) could be used to solve instances of 3-processor scheduling ($P3 \mid prec; p_j = 1 \mid C_{\max}$). The reduction is straightforward.

1.2 Organization

The remainder of the paper is organized as follows. In section 2 we give the proof of the Merge Theorem. In section 3 we use the Merge Theorem as the foundation for a scheduling algorithm that solves the single processor case, where the precedence graph is a collection of chains. We present the full algorithm in section 4. We conclude in section 5 with a brief discussion of our results.

2 Proof of the Merge Theorem

We begin by proving the Merge Theorem for the case where we have chain precedence constraints and only one processor (G is a collection of k independent paths, $m = 1$). This proof establishes all of the techniques used for the general case and is less obscured by details. We then sketch the natural generalization to dags, parallel processors and general separation delays.

2.1 Special Case: chain precedence constraints, one processor

Our goal is as follows: Given a schedule of the $2D+1$ largest chains that finishes at time T, we must construct a complete schedule for all k chains that finishes at time $\max\{n, T\}$. As a running example consider the instance shown in figure 1. This example consists of 7 chains with a total of $n = 21$ jobs. The maximum precedence delay is $D = 2$. Figure 1a shows a feasible schedule for the $2D+1 = 5$ largest chains with makespan $C^*_{\max} = n$. We will construct the new schedule in four steps.

Step 1: Truncating. Let n_i be the number of jobs in the i-th largest chain. We begin by removing the last n_{2D+1} jobs in each of the scheduled chains from the current schedule (as in figure 1b, where $2D+1 = 5$ and $n_5 = 2$). We call these deleted sub-chains the *tails*. Note that we have removed $2D+1$ tails with exactly n_{2D+1} jobs in each tail.

Step 2: Shifting operations. Next, we modify the schedule with the tails removed by shifting jobs so that they are executed as early as possible. Beginning at the first time slot, we traverse the schedule through time T. Whenever we encounter a hole (time slot that does not have a job

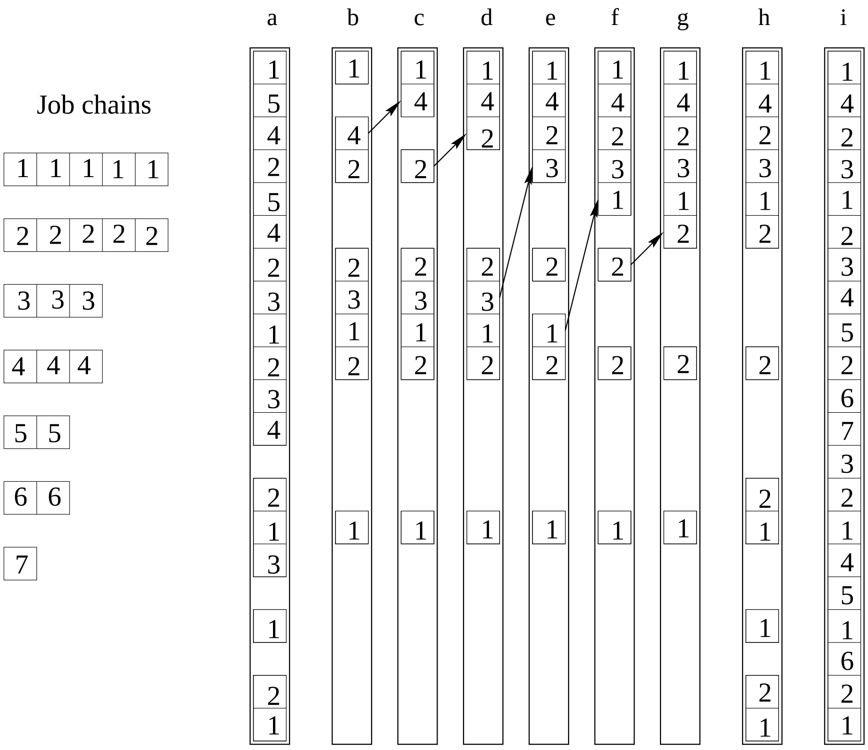

Figure 1: Problem instance (on left) and construction of an optimal schedule (on right), for $D = 2$. The instance is composed of two chains with 5 jobs, two chains with three jobs, two chains with two jobs and one chain with one job. All delays between consecutive jobs in a chain are 2. Constructing the schedule: **a)** A schedule for the large chains. **b)** Step 1, deleting the tails of the large chains. **c-g)** Step 2, shifting jobs earlier in the schedule until at most D chains remain active. **h)** Step 3, Putting the tails of the active chains back into the schedule. **i)** An optimal schedule after inserting the remaining jobs using the round-robin of Step 4.

scheduled in it) in the schedule, we try to fill that hole by moving a job earlier in the schedule (as in figure 1c-g).

We can always fill a hole with a job that is currently scheduled later, if, at the position of the hole, at least $D+1$ of the chains are *active*, i.e., they have not yet been scheduled up to the point at which they were truncated. To see why this is possible, note that if $D+1$ chains are still active, at least one of these chains has not been executed during the last D time steps before the hole. Therefore, if we move the next job of that chain into the current hole, it will be executed at least D time units after its predecessor. The precedence delay is satisfied after this move since the delay is at most D.

After repeatedly moving jobs to fill holes, we will either finish shifting all of the truncated chains or reach the first hole that we cannot fill without violating a delay constraint (as in figure 1g). The resulting schedule is tight before that hole (i.e. there are no holes before it), and there are at most D of the truncated chains active at that position (recall that we can always move a job if more than D chains are still active). In the example (figure 1g), chains 1 and 2 are still active at the first hole.

Step 3: Re-inserting some of the tails. We now reinsert the tails of the (at most D) chains that are still active at the first hole (as in figure 1h). We reinsert these jobs at their positions given by the original schedule. These positions in the schedule are still unoccupied, since jobs were only moved to time slots earlier than the first hole. Moreover, the makespan of the total schedule is still at most T.

Step 4: A Round-Robin for Scheduling Tails and Short Chains. We are now left with the tails of at least $D+1$ chains, each containing exactly n_{2D+1} jobs, whose truncated versions finished before the first hole (call these tails the **blue** chains) and $k-(2D+1)$ short chains, each containing at most n_{2D+1} jobs (call these the **red** chains). The red chains are the ones that were not among the $2D+1$ largest. In the example, the sub-chains consisting of the last $n_{2D+1} = 2$ jobs in chains 3, 4 and 5 are blue, and chains 6 and 7 are red.

Completing the schedule is done by filling holes with the remaining jobs in a round-robin fashion, i.e., we cycle through the chains (both the red and blue chains) in some fixed order, inserting the next job of each one, until they are all scheduled.

We have to be a bit careful about the first D holes we fill in this process, since the blue chains cannot start too close to their predecessors from their original chain.

This problem can be solved by systematically choosing the order we cycle through the chains. Since there are at least $D+1$ blue chains, one of their predecessors has not been executed during the last D steps, so we can safely schedule that chain first. Among the remaining blue chains, one has not been executed in the last $D-1$ steps, and therefore it can be scheduled second, and so on. We fix this order of the blue chains (in the example, we let this order be 3,4,5), and then follow it with any order of the red chains (6,7 in the example).

Since all blue chains have the same length, they all finish on the same round. Furthermore, the red chains finish on or before this round, since they are no longer than the blue chains. Therefore, every round consists of at least $D+1$ different chains, and we can fill every hole until the round-robin ends.

Thus, we have scheduled all jobs, obeying the chain precedence constraints and the precedence delays (as in figure 1i). If this step 4 did not fill all the holes that existed after step 3, then we know that our schedule still has makespan at most T. Otherwise, the new schedule has no idle time, and has makespan n. Also, the running time of each step of this construction can be made linear in the number of jobs.

2.2 Dags, parallel processors, and general separation delays

There is a natural generalization of the above construction to dags, parallel processors and general separation delays. We sketch the necessary changes, and leave the details for the full version of the paper.

Given a schedule with makespan T for the largest $2m(D+1)-1$ dags, we must construct a schedule for all the dags with makespan $\max\{\lceil \frac{n}{m} \rceil, T\}$. We follow the same four basic steps as before.

Previously, for chains, the first step of the construction removed the last n_{2D+1} jobs from the large scheduled chains. Now, in the general case we remove the $n_{2m(D+1)-1}$ jobs from each dag that are *scheduled* last (ties are broken arbitrarily). In step 2 of the chains case, we shifted jobs to earlier in the schedule as long as at least $D+1$ of the chains were still active. To be able to shift jobs in the general case, we now need $m(D+1)$ dags active. Step 3 is identical; we reinsert the jobs from the dags that are still active at the first hole we cannot fill.

Now at step 4 in the general case, there are at least $m(D+1)$ **blue** dags, each containing the same number of jobs, and several smaller **red** dags (the ones which were not in the initial schedule). On step 4 of the chains case (the round-robin fill-in step), notice that we made no assumptions about the delays between the jobs in the red and blue chains other than that they were bounded by D. So for dags, we first topologically sort the dags in an arbitrary way, making them chains. Then we perform the round-robin as before. The red chains finish first, the blue chains all finish on the same round, and we have either finished before time T, or filled every hole. The running time of each step is still linear in the number of jobs. $\square$

3 A Dynamic Program for Chains

We will now state a first simple version of our algorithm for the case where G is a collection of chains, and there is only one processor ($m=1$). In the next section, we give a more general version that works for trees on parallel processors. The algorithm given here is slightly less efficient than we can achieve; it runs in time $O(n^{3D+1})$. We will briefly sketch how to improve this to $O(n^{2D+1})$ at the end of the section. We give this slightly less efficient algorithm because it establishes some of the machinery used for the the general case.

The Merge Theorem shows how to construct an optimal schedule, assuming we know how to optimally schedule the $2m(D+1)-1$ largest chains in the precedence graph. This immediately suggests an algorithm:

1. **Dynamic Program.** Use a dynamic program to optimally schedule the $2m(D+1)-1$ largest chains in the input, setting aside the other chains for the moment.

2. **Merging.** Apply the Merge Theorem to schedule the chains we set aside during the dynamic program.

The dynamic program we will use can be thought of as finding the shortest path through a state space, where state transitions correspond to the scheduling of jobs on a single time step. Every state encodes ‘where we are’ in the current schedule; it records the jobs available to be scheduled on the upcoming time step, as well as a recent history of the current schedule, which we will use to determine when jobs become available in the future. More precisely, states have the form $\langle A, P \rangle$, where

- A is a set we call the *active set*. This is the set of currently *active* jobs, i.e. the jobs which can be scheduled in the next time step.

- P is a vector of length D, whose entries either contain jobs or are empty. These are the *past jobs*, the jobs that have been scheduled within the last D time steps. Essentially, P is a ‘window’ into the last D time steps of the schedule.

The following operations define both the legal transitions between states and the scheduling/status updating done by a search passing through this transition:

1. Schedule a job j in A. Shift the window P one time step forward, yielding P_{new}, whose last entry is j. It is also possible to not schedule any job (this is the only possibility if the active set is empty). In that case, P_{new} will have an empty last entry.
2. Use the information in P to determine the set B of jobs which become available on this new time step (the delays from their parents has just elapsed). Since the delays are bounded by D, the information in P is sufficient to make this determination.
3. Set A_{new} equal to the new set of active jobs, $(A \setminus \{j\}) \cup B$. The new state is $\langle A_{new}, P_{new} \rangle$.

Creating an optimal schedule now corresponds to finding a shortest path from the *start state* $\langle A, P \rangle$ (where A consists of the roots of the $2D+1$ largest chains, and P is an empty vector), to an *end state* (one where A is empty, and all jobs in P have no children that are not also in P).

The above dynamic program is enough to schedule chains on a single processor ($m = 1$) in polynomial time. This is because we can bound the size of the active set A. The set A can contain at most one job per chain, since no two jobs from the same chain can be active at the same time. Since the size of A is therefore limited by $2D+1$, there are only $O(n^{2D+1})$ possible values for A. Since there are $O(n^D)$ possible values for P, the number of states is bounded by $O(n^{3D+1})$. This bound is polynomial, and therefore we can find the optimal schedule for the largest $2D+1$ chains in polynomial time.

The second step (Merging) in our algorithm for chains is quite simple. Suppose the resulting schedule for the largest $2D+1$ chains has length T. We then apply the Merge Theorem to construct a schedule of all jobs of length $\max\{n, T\}$. Since T was a lower-bound on the optimal solution for the whole problem, the schedule must be optimal.

As a side note, we can reduce the size of the state space for chains and one processor to $O(n^{2D+1})$. Each state stores, for each chain, the last job executed and how long ago it was executed. This is enough information to determine A and P as above.

4 The Algorithm for Trees

In this section we give a polynomial time algorithm for scheduling jobs with tree precedence constraints, separation delays, and possible duplication of jobs. We assume that the precedence graph G forms a collection of out-trees. By reversing the time-line, the algorithm can also be used to schedule a collection of in-trees.

4.1 Notes on Job Duplication

Before we turn to the actual algorithm, we will briefly discuss job duplication. When scheduling jobs under separation delay constraints, it sometimes pays to execute a job multiple times on different processors. This is especially true if many other jobs depend on this one job, and it is time-consuming to move data from one processor to another.

The simplest example is an out-tree consisting of three nodes: a root with two children. The delay between the root and its children is 0 if they run on the same processor, and 10 otherwise. Suppose we want to schedule this instance on two processors. Clearly, without duplication, the shortest solution uses three time steps (schedule all three jobs on one processor). However, if we execute the root on both processors, we can execute both children in the next time step, resulting in a schedule of length two.

While duplication is clearly useful, it does not appear in completely arbitrary ways in a schedule. In fact, there always exists an optimal schedule in which no two copies of a job are executed more than D time steps apart. To see this, consider a job that is executed twice, where the second execution is more than D time steps after the first. In that case we can just delete the second one, since all its children were already available at the time the second copy was executed.

4.2 Overview of the Algorithm

We now turn to the scheduling algorithm for trees. The algorithm consists of the same two phases as the algorithm for chains given in the previous section: a dynamic program and a merging step. The states in our dynamic program will be similar to the ones in the previous section. They are of the form $\langle A, P \rangle$, where A contains jobs available on all processors and P contains a 'window' into the past D time steps of the schedule.

The transitions given in the previous section are not general enough to schedule trees, since the number of concurrently active jobs in A may grow without bound, e.g, if a job has many children that all become available at the same time. If the size of A is not bounded, the size of our state space will not be polynomial in size. To overcome this problem, we limit the maximum number of jobs in A to be $2m(D+1)-1$. Whenever a transition increases the number of active jobs above that number, we *set aside* the jobs from all but the largest $2m(D+1)-1$ trees rooted at these potentially active jobs. In the Merging step we will include the jobs from these set aside trees into the schedule.

To simplify the presentation, we introduce the notion of the *status* of a job. This status is not explicitly stored in the state, but is useful when we think about how the dynamic program creates a schedule. We say a job is:

- ***active***, if it can be scheduled right away on *all* processors, since all delays from its predecessor have elapsed,
- ***waiting***, if it has not been scheduled, and there is a processor on which it cannot run yet (because its

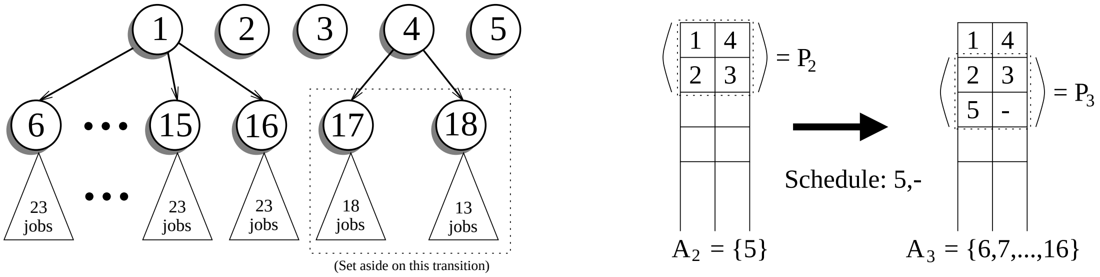

Figure 2: Example of an input tree and a state transition from $\langle A_2, P_2\rangle$ to $\langle A_3, P_3\rangle$. The maximum delay D is 2 and all delays $\ell_{i,j,a,b}$ are equal to 2. There are two machines ($m = 2$). The active set A_2 consists of only job 5, as it is the only one available. The transition schedules job 5 on the first machine, and nothing on the second machine. Jobs 6 through 18 all become available, but only $2m(D+1)-1 = 11$ can be in A_3, so jobs 17 and 18, the ones with the fewest number of jobs in their subtree, are set aside, along with the jobs in their subtrees. The new active set A_3 is $\{6, 7, \dots, 16\}$.

predecessor has not been executed yet, or not long enough ago),

- ***scheduled***, if it has already been scheduled on some processor, or
- ***set aside***, if the dynamic program has decided to ignore it, and will be scheduled only later in the Merging step.

4.3 A new dynamic program

The state space contains all pairs $\langle A, P\rangle$, where A is the active set, limited to $2m(D+1)-1$ jobs, and P is an $m \times D$ *matrix* recording the last D time steps of the schedule. This means that we have $O(n^{3mD+2m-1})$ states in the dynamic program, making finding a shortest path possible in polynomial time.

The state transitions are more complex than in the algorithm from the previous section. An example state transition can be found in figure 2. If we are at a state $\langle A, P\rangle$, we can go to a new state $\langle A_{new}, P_{new}\rangle$, as follows:

1. Choose jobs $j_1, j_2, \dots, j_m$ to be executed on the m processors. Set their status to *scheduled*. Each job j_i can be one of the following:
   - nothing (no job scheduled)
   - any job in the set A
   - any job in the matrix P that is executable on processor i at the current time step (job duplication)
   - any child of a job in matrix P that is executable on processor i (but not all processors) at the current time step (partially available job)
2. The new matrix P_{new} is P shifted forward by one row, with the new last row $(j_1, j_2, \dots, j_m)$. All jobs that were in the first row of P (the one that got shifted out) that are still in P_{new} (due to job duplication) are removed from P_{new}.
3. Using the information in P, determine the set of jobs B that on *this* step become available on *all* processors, and have not been executed before, and set A_{new} to $(A \setminus \{j_1, j_2, \dots, j_m\}) \cup B$.
4. If A_{new} has more than $2m(D+1)-1$ elements, remove all but the $2m(D+1)-1$ 'largest' jobs from the set, where 'largest' is measured in terms of the size of the sub-tree rooted at the job. These removed jobs, along with all the jobs contained in their sub-trees, are *set aside*. They will be dealt with in the Merging phase.

The *start state* of the dynamic program is $\langle A_0, P_0\rangle$, where A_0 consists of the roots of the $2m(D+1)-1$ largest trees, and P is the empty matrix. The *end states* have the form $\langle A, P\rangle$ where A is empty, and all jobs in P either have no children, or their children are also in P.

As we traverse the path from a start state to an end state, the status of each job evolves as in figure 3. It is not hard to see that at the end of the path, every job is classified as either *scheduled* or *set aside*.

4.4 Merging and Correctness

A path of length T from a start state to an end state in the state-space defined above gives a schedule of length T for part of the tree. We need to show how the jobs *set aside* by the path can be merged back into the schedule. In the

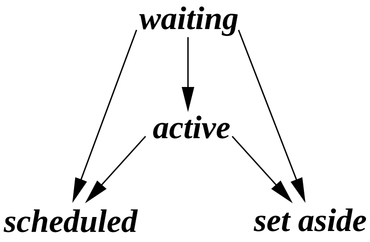

Figure 3: The life of a job.

remainder of this section, we will show two lemmas. The first lemma will establish that we can find a path in the state space that can be converted into an optimal schedule via Merging. The second lemma will show how to perform this Merging step.

Before stating the lemmas, we need three definitions. First, we define the set U_q for a state q, which contains all the jobs which *must* appear after state q in any legal schedule (these are the jobs which are *available* or *waiting* at that state). This set is completely determined by the information contained in $\langle A,P\rangle$.

DEFINITION 4.1. (DEPENDENT JOBS) *For a state $q = \langle A,P\rangle$, let U_q contain all jobs in A, all descendants of jobs in A, and all descendants of jobs in P that are not yet available on all processors, and that are not in P themselves.* □

Now we define the deadline of state q to be the latest possible point on a path where q can appear so that all the dependent jobs of q can still fit into the schedule without making it longer than $C^*_{\max}$:

DEFINITION 4.2. (DEADLINE) *Let $q = \langle A,P\rangle$ be a state. The* deadline *of q is the value* $\left\lfloor C^*_{\max} - \frac{|U_q|}{m} \right\rfloor$. □

In any path in state space that corresponds to an optimal schedule, every state must appear before its deadline. We formalize this in a definition.

DEFINITION 4.3. (ADMISSIBLE PATH) *A path in the state space from a start state to an end state is called* admissible *iff for all x from 0 to $C^*_{\max}$, the x-th state on the path has a deadline of at least x.* □

We will now show that an admissible path always exists, that it can be found in polynomial time, and how to convert it into an optimal schedule.

LEMMA 4.1. (DYNAMIC PROGRAM CORRECTNESS) *There always exists an admissible path that can be found in polynomial time.*

Proof: An admissible path, if it exists, can easily be found by breadth-first search through the state space of the dynamic program we just constructed. The deadline of each state can be determined beforehand[3]. At depth x of the search, we extend the search only to states with a deadline of at least $x+1$.

Now we show that such a path always exists. We show this by constructing an admissible path $(q_0, q_1, ..., q_{C^*_{\max}})$ using an optimal schedule S as a template. We assume that S has no unnecessary job duplications (jobs whose removal from the schedule would maintain feasibility).

We will proceed along the schedule, and at the x-th step take the state transition from $q_{x-1} = \langle A,P\rangle$ to q_x that corresponds to executing the jobs in the x-th time slice of S that are in $P \cup U_{q_{x-1}}$. There must be such a transition, because for every job in $P \cup U_{q_{x-1}}$ that is executed in S at that time slice, it is either in A, or its parent appears in P at the same position as it appears in S (easily shown by induction).

It remains to show that the so constructed path is admissible. Note that when we are at state q_x along the path, then all jobs in U_{q_x} have to appear after time slot x in the schedule S. Because we are executing 'down' the trees, and we never add to a set U_{q_x} to obtain $U_{q_{x+1}}$, we have $U_{q_x} \subseteq U_{q_y}$ if $x > y$. So, if $x > y$, and a job in $U_{q_{y-1}}$ appears in S at time step y (and so is not in U_{q_y} by construction), it will not be in U_{q_x}. This means that none of the jobs in U_{q_x} can appear at or before the x-th time step in S, and therefore all appear after it. But this implies $\lceil |U_{q_x}|/m \rceil \leq C^*_{\max} - x$, which shows that the path is admissible. □

Now that we have a schedule for part of the tree, we need to merge the jobs we set aside back into the schedule. Here is where we use the Merge Theorem.

LEMMA 4.2. (MERGING) *Given an admissible path, an optimal schedule can be constructed in time $O(n^2)$.*

Proof: An admissible path can be directly converted into a schedule S of the same length that contains all but the jobs which were *set aside*. We now show how to incorporate the set aside jobs into the schedule, while making it not longer than the optimal schedule.

We do this by traversing the path from its end to its beginning. When we reach a state q_x at which jobs were set aside, we include them into the schedule as follows. Since trees were set aside at that state, there must be $2m(D+1)-1$ larger trees rooted at the jobs in q_x's active set. The jobs in these 'active' trees are already in the schedule, since either they were scheduled by the admissible path, or they were set aside later, in which case we already merged them into the schedule (recall we are traversing it backwards).

This means we can apply the Merge Theorem to merge the set aside trees into the schedule. Since we started with an admissible path, we know that the number of jobs not yet *scheduled* at q_x does not exceed $m \cdot (C^*_{\max} - x)$, the available room in the schedule. Therefore, merging the set aside trees does not make the schedule longer than the optimal schedule. We repeat this procedure for all states and obtain an optimal schedule.

Since applying the Merge Theorem for every state costs linear time, and there might be up to n states on the path, the total time for the merging operation is $O(n^2)$. □

---

[3] Note that we have to know $C^*_{\max}$ to compute the deadline. But since $C^*_{\max} \leq nD$, we can find the value using binary search with a multiplicative increase of $O(\log n)$ in running time.

5 Conclusion

In this paper we have given the first polynomial-time multiprocessor scheduling algorithm for tree-based precedence constraints that impose precedence and communication delays. As opposed to previous results, separation delays $\ell_{i,j,a,b}$ can depend on jobs and machines, and can have values other than 0 and 1, as long as they are bounded by a constant D. That makes our algorithm more general and applicable to the instruction scheduling for VLIW architectures. The potentially long running time of the algorithm is acceptable to embedded system designers since the software is compiled only once and an optimal performance is required of the resulting system.

The algorithm for trees uses an unconventional dynamic program, where partial paths in state space do not correspond to partial schedules, but rather have to be transformed into a solution during the Merging phase. The running time of our algorithm depends exponentially on the number of processors m and maximum delay D, making it impractical for large values of these constants. However, it is the dynamic programming part of the algorithm that incurs this runtime; the merging step only takes $O(n^2)$ time. This suggests using an heuristic instead of the optimal dynamic program to produce a path through the state space. The Merge Theorem can then be used to incorporate the remaining trees into the schedule. Finding good heuristics, from both a theoretical and an experimental point of view, is a very interesting open problem. We plan to continue our work in this direction.

Another intriguing question is whether our techniques can be extended to the case where G is an arbitrary dag. The Merge Theorem still holds for these inputs. But our dynamic program critically uses the fact that once a branch occurs, the subtrees are completely independent. A more complicated dynamic program might get around this problem without a large increase in the size of the state space. As already mentioned in the introduction, this is very likely a hard problem, since an algorithm for just the single-processor case with $D = 3$ can be used to solve the famous open 3-processor scheduling problem.

Acknowledgments

We would like to thank John Dunagan, Ryan O'Donnell and April Rasala for their helpful comments on this paper.

References

[1] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[2] P. Chrétienne and C. Picouleau. Scheduling with communication delays: A survey. In P. Chrétienne, Jr. E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 65–90. John Wiley & Sons Ltd, 1995.

[3] E.G. Coffman, Jr. and R.L. Graham. Optimal sequencing for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[4] Daniel W. Engels. *Scheduling for Hardware-Software Partitioning in Embedded System Design*. PhD thesis, Massachusetts Institute of Technology, 2000.

[5] Lucian Finta and Zhen Liu. Single machine scheduling subject to precedence delays. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 70, 1996.

[6] Lucian Finta, Zhen Liu, Ioannis Milis, and Evripidis Bampis. Scheduling UET–UCT series–parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323–340, August 1996.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[8] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[9] Intel Corporation. *The IA-64 Architecture Software Developer's Manual*, January 2000.

[10] Hermann Jung, Lefteris Kirousis, and Paul Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proceedings of SPAA*, pages 254–264, 1989.

[11] Jan Karel Lenstra, Marinus Veldhorst, and Bart Veltman. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20(1):157–173, January 1996.

[12] Rolf H. Möhring and Markus W. Schäffter. A simple approximation algorithm for scheduling forests with unit processing times and zero-one communication delays. Technical Report 506, Technische Universität Berlin, Germany, 1995.

[13] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes (extended abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 229–234, May 1988.

[14] C. Picouleau. *Etude de problèmes les systèmes distrubés*. PhD thesis, Univ. Pierre et Madame Curie, Paris, France, 1992.

[15] Texas Instruments. *TMS320C6000 Programmer's Guide*, March 2000.

[16] J. D. Ullman. *NP*-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[17] Theodora A. Varvarigou, Vwani P. Roychowdhury, Thomas Kailath, and Eugene Lawler. Scheduling in and out forests in the presence of communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1065–1074, October 1996.