Performance of Work Conserving Schedulers and Scheduling of Some Synchronous Dataflow Graphs

Udayan Kanade
Codito Technologies Pvt. Ltd.

Abstract

We know a lot about competitive or approximation ratios of scheduling algorithms. This, though, cannot be translated into direct bounds on the schedule produced by a scheduling algorithm, because often the optimal solution is intractable. We derive a methodology to find absolute bounds on the scheduling of jobs with precedence constraints on parallel identical machines. Our bounds hold for a large class of online and offline scheduling algorithms: the "work conserving" scheduling algorithms. We apply this methodology to prove that an important class of synchronous dataflow graphs – the parallelized pipelines – has very good performance characteristics when scheduled by a work conserving scheduler. Real time guarantees and granularity design for these dataflow graphs are discussed. We argue that parallelized pipelines should be dynamically scheduled on multiprocessor architectures.

Keywords: Parallel job scheduling, online scheduling algorithms, synchronous dataflow graphs, real time systems, performance analysis of parallel systems.

U. Kanade, Performance of Work Conserving Schedulers and Scheduling of Some Synchronous Dataflow Graphs, *Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS'04)*, pages 521-529, 2004.

Performance of Work Conserving Schedulers and Scheduling of Some Synchronous Dataflow Graphs

Udayan Kanade
Codito Technologies Pvt. Ltd.
udayan@codito.com

Abstract

We know a lot about competitive or approximation ratios of scheduling algorithms. This, though, cannot be translated into direct bounds on the schedule produced by a scheduling algorithm, because often the optimal solution is intractable. We derive a methodology to find absolute bounds on the scheduling of jobs with precedence constraints on parallel identical machines. Our bounds hold for a large class of online and offline scheduling algorithms: the "work conserving" scheduling algorithms. We apply this methodology to prove that an important class of synchronous dataflow graphs – the parallelized pipelines – has very good performance characteristics when scheduled by a work conserving scheduler. Real time guarantees and granularity design for these dataflow graphs are discussed. We argue that parallelized pipelines should be dynamically scheduled on multiprocessor architectures.

Keywords: *Parallel job scheduling, online scheduling algorithms, synchronous dataflow graphs, real time systems, performance analysis of parallel systems.*

1. Introduction

Scheduling of precedence constrained jobs on parallel machines has been the topic of extensive study in the past [8, 10]. Various versions of these problems occur in the study of program parallelization, instruction scheduling in compilers, microinstruction scheduling in superscalar architectures and project management and other operations research problems [7].

The competitive or approximation ratio of a scheduling algorithm is the worst guaranteed ratio of the total time of a schedule produced by the algorithm, to the total time of the optimal schedule. R. Graham proved the competitive ratio of $2 - 1/m$ for List Scheduling in [6], which was shown to be optimal among online algorithms in [4].

Even though competitive analysis is important for algorithm comparison, in many practical scenarios, we would like to have absolute bounds in terms of processing time for a scheduling algorithm. This is not possible using competitive analysis because calculating the optimal schedule time (with which the specific algorithm's schedule is compared) is NP hard. In Section 2 we define an important class of schedulers called "work conserving schedulers". We prove absolute bounds on the schedule produced by any work conserving scheduler for a set of precedence constrained jobs.

In Section 3, we use this result to prove bounds on the scheduling of an important class of Synchronous Dataflow Graphs (SDFs). SDFs are an important abstraction for the specification of distributed computation mechanisms. Pipeline form SDFs, where the data flow through linearly arranged stages, are an important class of SDFs. Each stage can itself be divided into multiple parallel jobs. We show that this kind of synchronous dataflow graph, with single or multiple buffering in between the pipeline stages, is equivalent to a set of jobs with precedence constraints, thus enabling usage of the analysis developed earlier.

In Section 4 we extend the parallelized pipeline results so that they apply "in-process", i.e. we give bounds for the completion of each data item going through the pipeline, considering the "interference" caused by earlier or subsequent packets/frames. In Section 5 we use the main results of this paper to derive granularity heuristics, i.e. heuristics about how much a pipeline stage should be parallelized. Our results indicate, as discussed in Section 6, that applications having the parallelized pipeline form should be dynamically scheduled on multiprocessors. Section 7 mentions interesting unsolved problems brought up by this paper. Our results are summarized in Section 8.

2. Bounds for Work Conserving Schedulers

In this section we investigate the performance of any work conserving scheduler for scheduling any set of jobs with arbitrary precedence constraints.

Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS'04)
1521-9097/04 $ 20.00 IEEE

2.1. Work Conserving Schedulers

In this paper, we shall discuss the performance of work conserving schedulers in scheduling certain patterns of jobs (described in Section 2.2 and Section 3). A **work conserving scheduler** is a scheduler which guarantees that whenever a job is ready to be scheduled, if a machine is free to process the job, the job will be scheduled. At no point of time will it happen that a job is ready and a machine is free, and yet the job is not scheduled on the machine.

An important class of work conserving schedulers are work conserving online schedulers. An online scheduler is one which assumes no prior knowledge of the jobs. A job becomes known to an online scheduler only when all its precedence constraints have been satisfied. Furthermore, an online scheduler assumes no knowledge of the running time of the job. The running time becomes known only when the job runs to completion. (This paradigm models the information available to a modern OS scheduler. There are other online paradigms, as explained in [10].) Scheduling algorithms like List Scheduling [6], First Come First Served and Priority Scheduling are work conserving online schedulers. In a variation of online schedulers, some schedulers do not assume knowledge of the set of jobs in advance, but do assume the knowledge of the running time of a job the moment the job becomes known. Examples are Shortest Job First and Longest Job First.

Offline schedulers are algorithms which assume complete knowledge of the set of jobs, their precedence constraints and their running times, to schedule the whole set of jobs. Work conserving schedulers can be online as well as offline.

Work conserving schedulers can be both preemptive and non-preemptive. Modern OSes use preemptive work conserving schedulers like Round Robin or preemptive Priority Scheduling.

For multiprocessor scheduling problems, almost all the algorithms devised are work conserving. Non work conserving methodologies have been used for more complex problems like problems which include setup times, or problems having slightly delayed introduction of jobs, or problems where smoothing out of the service rate is important.

2.2. Scheduling Precedence Constrained Jobs

The scheduling problem we are investigating is as follows. There is a set of jobs $\{J_i\}$ to be scheduled on m identical machines. A **job** J_i is a serial program consuming total time $t(J_i)$. There are arbitrary precedence constraints between the jobs, given by a partial order '$\prec$'on the jobs. We write $J_i \prec J_j$ to mean that J_i has to be finished before J_j can be started. The jobs and the precedence constraints together form a partially ordered set (**poset**) $J = (\{J_i\}, \prec)$.

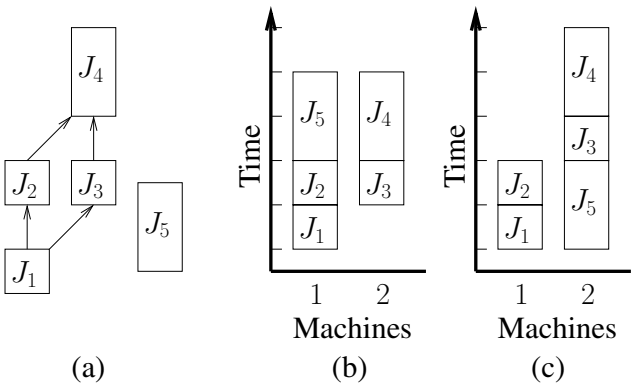

Figure 1. (a) An example partial order of jobs. (b) An optimal schedule for it. (c) A work conserving schedule for it.

Figure 1(a) shows a set of jobs with precedence constraints depicted as a Directed Acyclic Graph (DAG). Arcs are drawn from preceding to succeeding jobs. The transitive closure of the DAG is the partial order. In the diagrams in this section, the height of the rectangle is proportional to the time taken by the job.

The aim of the scheduling algorithm is to minimize the makespan. The **makespan** of a schedule produced by a scheduling algorithm is the total running time of the schedule, i.e. the difference between the time the earliest job starts to the time the last job finishes. This problem is denoted $P|prec|C_{\max}$ in the three field problem description notation introduced by [9]. Figure 1(b) shows an optimal schedule for the job poset in Figure 1(a).

Our aim in this section is to give an upper bound on the makespan of a schedule produced by *any* work conserving scheduler (defined in Section 2.1) in scheduling a precedence constrained set of jobs. A job being "ready" in this context means that all its precedence constraints have been satisfied. Thus, a work conserving scheduler, for the above problem, is a scheduler which keeps machines busy as long as there are unprocessed jobs with all their precedence constraints satisfied.

It is interesting to note that if one is constrained to using non-preemptive work conserving schedules, none of the minimum makespan schedules may be work conserving. For example, the optimal work conserving schedule for Figure 1(a) is Figure 1(c), which has a worse makespan than the optimal schedule shown in Figure 1(b). On the other hand, if preemption is allowed, at least one optimal schedule is work conserving.

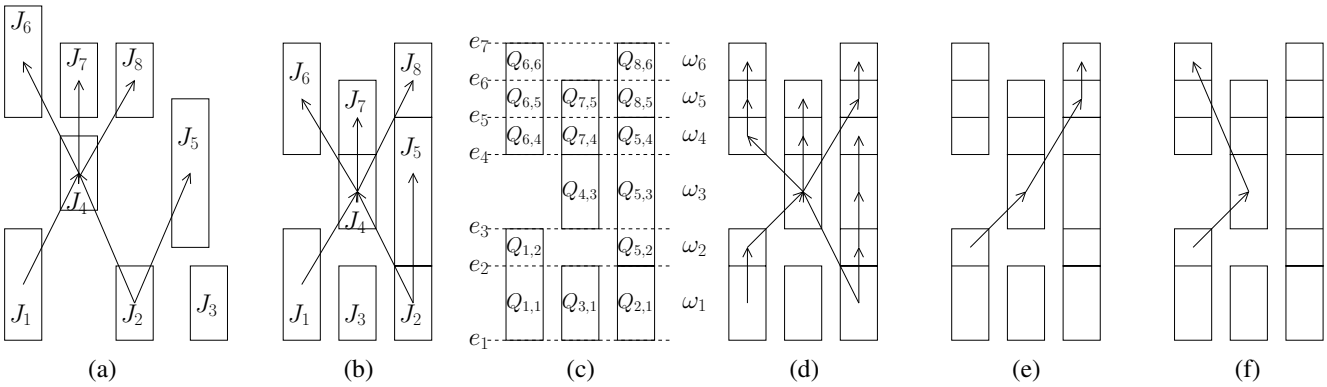

Figure 2. (a) An example set of jobs with precedence constraints. (b) A 3-processor work conserving schedule for it. (c) Event boundaries, schedule intervals and job sections. (d) Equivalent precedence constraints in job sections. (e) Working of Lemma 1: $Q_{8,6}$ depends on something in ω_2. (f) Working of Theorem 1: chain covering partly idle intervals.

2.3. A Few Definitions

An event boundary e_k in a schedule made by a scheduler for a set of jobs, is a time instant when at least one job starts or stops or gets preempted on a machine. e_k is the k^{th} event boundary in the schedule. The time period between two consecutive event boundaries e_k and e_{k+1} is called **schedule interval** ω_k. Interval ω_k runs for time $t(\omega_k) = e_{k+1} - e_k$.

A job J_i is chopped up into multiple sections by the event boundaries. Let $Q_{i,k}$ be the **job section** of job J_i scheduled in schedule interval ω_k (if J_i was scheduled in ω_k). $Q_{i,k}$ runs for time $t(Q_{i,k}) = t(\omega_k) = e_{k+1} - e_k$.

We impose precedence constraints on the job sections $\{Q_{i,k}\}$ equivalent to the precedence constraints on jobs $\{J_i\}$. $Q_{i_1,k_1} \prec Q_{i_2,k_2}$ if and only if either $J_{i_1} \prec J_{i_2}$ or $J_{i_1} = J_{i_2}$ and $k_1 < k_2$. That is, one job gets divided into multiple job sections, keeping all the precedence constraints with other jobs and adding precedence constraints for the job sections within the job.

Now we define some parameters of a job poset which are used in bounds derived in this paper. Let S be the total job time.

$$S = \sum_{J_i} t(J_i) = \sum_{Q_{i,k}} t(Q_{i,k}) \tag{1}$$

Also, let H be the time taken by the longest (by time) chain of jobs. The longest chain of jobs is obviously equal to the longest chain of job sections. Thus,

$$H = \max_{\substack{\Lambda \text{ is a chain} \\ \text{of jobs}}} \sum_{J_i \in \Lambda} t(J_i) = \max_{\substack{\Lambda \text{ is a chain of} \\ \text{job sections}}} \sum_{Q_{i,k} \in \Lambda} t(Q_{i,k}) \tag{2}$$

The longest chain is also called the "critical path" in Operations Research terminology [7]. The critical path can easily be identified using dynamic programming techniques. The Critical Path Method (CPM) uses such a technique.

2.4. A Bound on the Makespan

Though the main result in this section is based on Graham's competitive ratio analysis [6], we provide an independent proof. Our reasons for doing so are – (a) Our bound is not a competitive bound. In many cases it may provide for sharper analysis of the schedule (e.g. the analysis in Section 3). (b) Graham has proved the bound for what has come to be known as List Scheduling. Though all work conserving non-preemptive algorithms can be thought of as List Scheduling, proving our results through this fact is quite cumbersome. Also, it would be hard to include preemptive algorithms, which we do include in our analysis.

The following lemma is used to prove the central theorem of this section – Theorem 1.

Lemma 1. *If at least one machine is free during a schedule interval ω_k of a schedule made by a work conserving scheduler, then all future job sections $Q_{\bullet,l}$, $l > k$ are dependant on at least one job section that executes in ω_k.*

Proof. Take any job section $Q_{i_1,l}$ such that $l > k$. Since we are using a work conserving scheduler, $Q_{i_1,l}$ did not execute in ω_k implies it was not ready during ω_k. Thus it became ready later, say due to the execution of Q_{i_2,l_2}, where $Q_{i_2,l_2} \prec Q_{i_1,l}$ and $k \leq l_2 < l$. If $l_2 \neq k$, we can continue in the same fashion to find a Q_{i_3,l_3} (such that $Q_{i_3,l_3} \prec Q_{i_2,l_2}$ and $k \leq l_3 < l_2$) and so forth, finding

jobs from earlier intervals till we find a job $Q_{i_\bullet,k}$ such that $Q_{i_\bullet,k} \prec \ldots Q_{i_2,l_2} \prec Q_{i_1,l}$. □

Theorem 1. *The makespan (total running time) T of the schedule for m identical machines produced by any work conserving scheduler for the set of jobs $\{J_i\}$ with precedence constraints $\prec$ is bounded by*

$$\max\left(\frac{1}{m}S, H\right) \leq T \leq \frac{1}{m}S + \left(1 - \frac{1}{m}\right)H \qquad (3)$$

Proof. The lower bound $S/m \leq T$ is obvious. The lower bound $H \leq T$ is also obvious. For the upper bound, we calculate the total busy time and idle time of machines separately. All the machines together will be busy exactly for a total time of S. Now consider the set of intervals $\{\omega_{n_1}, \omega_{n_2}, \ldots \omega_{n_r}\}$, $n_1 < n_2 < \ldots n_r$ when at least one machine is idle. Pick a job section Q_{i_r,n_r}. By Lemma 1, there exists a job section $Q_{i_{r-1},n_{r-1}}$ such that $Q_{i_{r-1},n_{r-1}} \prec Q_{i_r,n_r}$. Similarly, there exists a job section $Q_{i_{r-2},n_{r-2}}$ such that $Q_{i_{r-2},n_{r-2}} \prec Q_{i_{r-1},n_{r-1}}$. Continuing this line of reasoning, we find that there is a chain of job sections $Q_{i_1,n_1} \prec Q_{i_2,n_2} \prec \ldots Q_{i_r,n_r}$ "covering" the intervals where at least one machine is idle. An upper bound on the total time when at least one machine is idle is thus H. Since at most $m-1$ machines can be idle at any time, the upper bound on the total idle time of machines is $(m-1)H$. Thus, the total time spent by all the machines together is bounded above by $S + (m-1)H$. Dividing by m, the number of machines, we get the required upper bound. □

The above theorem can be used to give the competitive ratio proved by Graham for List Scheduling [6] and shown to be optimal [4] among online scheduling algorithms.

Corollary 1. *The makespan of* any *work conserving scheduler has an approximation/competitive ratio of* $2 - 1/m$.

Proof. Obviously $T_{\mathrm{OPT}} \geq S/m$. Also obviously $T_{\mathrm{OPT}} \geq H$. Using Theorem 1, we get $T \leq \frac{S}{m} + (1 - \frac{1}{m})H \leq (2 - \frac{1}{m})T_{\mathrm{OPT}}$. □

Thus, Theorem 1 cannot give an approximation ratio of better than $2 - 1/m$. But, as we will see in Section 3, it can be used to give better absolute (non-relative) bounds.

Though the algorithms that Theorem 1 applies to maybe online, the analysis dictated by Theorem 1 may only be practically carried out offline, for the knowledge of S and H is not available online. (This applies even to competitive analysis. The difference is – given a poset of jobs, S and H are much simpler to compute than T_{OPT}, and they give better absolute bounds.)

Corollary 2. *The upper bound of Theorem 1 holds after any relaxation of precedence constraints. The upper bound of Theorem 1 holds even after removing a few jobs or reducing the time taken by some jobs.*

Proof. After relaxing precedence constraints, S remains as it is, and H can only become smaller. If a few jobs are removed or their times are reduced then both S and H become smaller. □

Thus, Theorem 1 also gives bounds for Richard's paradoxes [6]. The rest of this paper applies these bounds to particular cases.

3. Bounds for Parallelized Pipelines

In the remainder of this paper, we shall consider a particular class of SDFs called parallelized pipelines. In these SDFs (depicted in Figure 3(a)) data flows through linearly arranged stages. Each stage is itself divided into multiple parallel jobs.

Such pipelines occur frequently in various multimedia, DSP, control, network and graphics applications. Various audio and video codecs (MPEG-II, H.264, mp3, etc.) may be modeled in this way. Pipeline stages in a typical video encoder are motion prediction, transform, quantization and variable length coding. [1] talks about a nine-stage motion estimation pipeline.

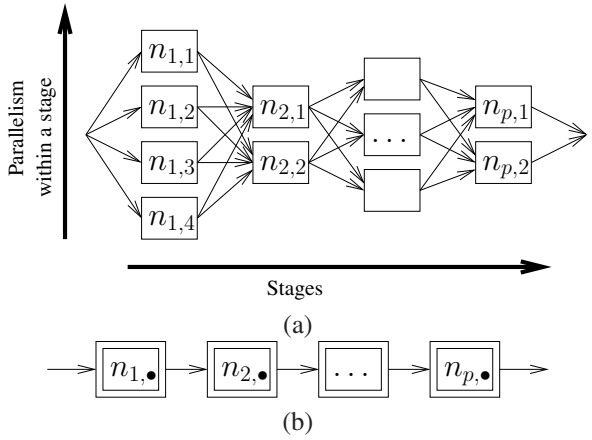

Figure 3. (a) A parallelized pipeline. (b) Concise depiction.

[1, 5] give algorithms and analysis of statically allocated parallel pipelines, where multiple processors are assigned to each stage. The algorithms in [3] can be used to find an offline schedule which is then run repeatedly online. None of these static scheduling methodologies are work conserving. Dynamic scheduling methodologies, on the other hand, can

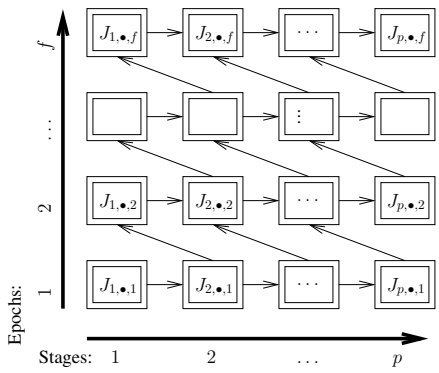

Figure 4. Job DAG for pipeline with precedence constraints.

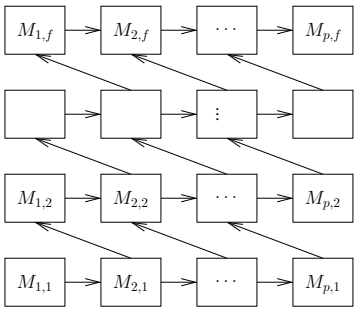

Figure 5. DAG with only the largest jobs.

easily be work conserving (more about this in Section 6). In this section, we will derive bounds on the running of parallelized pipelines using any work conserving scheduler.

A parallelized pipeline, (depicted in Figure 3(a)) is a pipeline of p stages, each stage i consisting of multiple execution nodes (compute jobs) $n_{i,j}$. Successive items pass through all the stages of the pipeline one by one. A node $n_{i,j}$ takes time $t(n_{i,j})$ for *each* item going through the pipeline. All the nodes of a particular stage have to finish processing before any node of the next stage can start. We depict parallelized pipelines as shown in Figure 3(b), where the double rectangle stands for multiple execution nodes.

Successive firings of a node (for successive data items) are said to be for successive "epochs". Each such firing constitutes a "job" in the scheduling sense as used in Section 2. A job $J_{i,j,k}$ stands for the firing of the node $n_{i,j}$ for the epoch (item number) k. $t(J_{i,j,k}) = t(n_{i,j})$. Since a node can be processing only one epoch at a time, we have the obvious precedence constraint $J_{i,j,k} \prec J_{i,j,k+1}$ for every $n_{i,j}$. Since the nodes of a particular stage have to complete work for a particular epoch before any nodes of the next stage can start work for the same epoch, $J_{i,j_1,k} \prec J_{i+1,j_2,k}$.

Apart from the above precedence constraints, we add buffering constraints: assume that there is a single buffer between every two adjacent stages for intermediate results. A stage has to wait for the next stage to complete processing the item of the previous epoch before it starts processing. Thus, $J_{i,j_1,k} \prec J_{i-1,j_2,k+1}$. (Adding these constraints is realistic, and these constraints will also help us prove in-process bounds in the next section. Effects of multiple-buffering and no buffering constraints are discussed at the end of this section.)

We assume that the pipeline has to be scheduled for a total of f epochs, numbered from 1 to f. This gives a job poset which is depicted in Figure 4. (Each arrow stands for many arcs – in fact all the possible ones.)

To use the Theorem 1, we need the total job time S and longest chain H. Suppose the time consumed in one particular epoch is $C = \sum_{(i,j)} t(n_{i,j})$. Then the total job time $S = fC$.

In a parallel group of jobs, the longest job is the only one that matters. We formalize this intuition as follows. Let m_i be the most time consuming among the i^{th} stage execution nodes $n_{i,j}$. Let the corresponding longest job for the k^{th} epoch be $M_{i,k}$. (If there are multiple longest jobs, we can choose any one as $M_{i,k}$.)

Lemma 2. *For every i, k, out of the jobs $J_{i,\bullet,k}$, the one that can appear in the longest chain has execution time equal to $t(M_{i,k})$.*

Proof. None of the $J_{i,\bullet,k}$ jobs depend on each other. They have all the same successors and predecessors. Thus, in any chain, we can always replace a $J_{i,j,k}$ with $M_{i,k}$. Thus, in the longest chain, if one of $J_{i,j,k}$ appeared, it will have execution time equal to $t(M_{i,k})$. □

Thus, (without loss of generality) we only need consider the reduced set of the jobs $M_{i,k}$, shown in Figure 5, to find the longest chain.

Let the largest adjacent-stage-pair time be $h_2 = \max_i(t(m_i) + t(m_{i+1}))$. Let the single-epoch pipeline latency be $L = \sum_i t(m_i)$.

Lemma 3. *The longest chain in the poset of jobs due to a parallelized pipeline has length $H = L + (f-1)h_2$.*

Proof. Let the largest adjacent-stage-pair be m_u and m_{u+1}. Then the chain $M_{1,1} \prec M_{2,1} \prec \ldots M_{u,1} \prec M_{u+1,1} \prec M_{u,2} \prec M_{u+1,2} \prec \ldots M_{p,f}$ has length $L + (f-1)h_2$. This chain is depicted in Figure 6. Now we prove that no chain can be longer.

In the reduced job set of Figure 5, $M_{1,1}$ is the predecessor of every job, and $M_{p,f}$ is the successor of every job. The longest chain hence has to start at $M_{1,1}$ and end at $M_{p,f}$. Furthermore, every successor of $M_{i,k}$ is either $M_{i+1,k}$ or $M_{i-1,k+1}$ or one of their successors. Thus, if the longest chain has the job $M_{i,k}$ it will either have the next stage

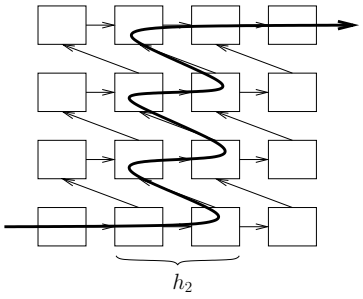

Figure 6. The longest chain

job $M_{i+1,k}$, or the previous stage next epoch job $M_{i-1,k+1}$. Every time the chain takes the previous stage, it jumps an epoch. It does so exactly $f-1$ times. It takes the next stage exactly $p+f-2$ times, the length of the pipeline $(p-1)$ plus compensation for the $(f-1)$ reverse jumps calculated above. Thus, the total number of jumps is exactly $(f-1)+(p+f-2)=p+2f-3$. The total number of jobs in the longest chain are thus $p+2f-2$.

Now, let us call the longest chain $M_{(1)}, M_{(2)}, \ldots M_{(p+2f-2)}$. Let the corresponding sequence of execution nodes be $\Phi = [m_{(1)}, m_{(2)}, \ldots m_{(p+2f-2)}]$. The sequence Φ will have repeated nodes unless $f=1$. In this sequence, the first time a node appears, we color it red. Every other instance of the node is colored blue. The red subsequence has a total time of L, and takes up p nodes. The blue subsequence then has $2f-2$ nodes. Take a maximal contiguous blue-colored run in Φ. Suppose the red node just before it is m_i, of the i^{th} stage. The maximal contiguous blue subsequence starts from m_{i-1}, and goes back-and-forth one stage at a time till it finally reaches m_i. Then, this subsequence has an even number of nodes, each subsequent node being adjacent to the one just covered. Thus, it is made up of adjacent-stage-pair nodes. Thus, the whole blue subsequence of $2f-2$ nodes is made up of $f-1$ adjacent stage pairs. This is obviously bounded above by $(f-1)h_2$. Thus the total time of Φ is bounded above by $L+(f-1)h_2$. □

Now that we know both S and H, application of Theorem 1 gives us

Theorem 2. *The makespan T of a single-buffered parallelized pipeline scheduled using any work conserving scheduler is bounded by*

$$\max\left(\frac{fC}{m}, L+(f-1)h_2\right) \le T \tag{4}$$

$$T \le \frac{fC}{m} + \frac{m-1}{m}\left(L+(f-1)h_2\right) \tag{5}$$

□

Let us call the upper bound we have derived above $\mathcal{U}_T(f)$.

$$\mathcal{U}_T(f) = \frac{fC}{m} + \frac{m-1}{m}(L+(f-1)h_2) \tag{6}$$

This is an affine function of f, of the form $rf+c$. We call r the worst-case processing rate per epoch, and c the worst-case setup latency.

$$r = \frac{d\mathcal{U}_T}{df} = \frac{C}{m} + \frac{m-1}{m}h_2 \tag{7}$$

$$c = \mathcal{U}_T(0) = \frac{m-1}{m}(L-h_2) \tag{8}$$

The pipeline does C work per epoch, thus requiring an amortized C/m from each machine. From Equation 7, we see that the wasted time due to bottlenecks can be at the most $\frac{m-1}{m}h_2$. Thus, the wastage ratio, the ratio of utilized to wasted machine time, is at worst $(m-1)(h_2/C)$. E.g. if the whole work in the pipeline has been divided into 200 equal-time nodes, $h_2/C = 1\%$. The overhead will then be $(m-1)1\%$.

For infinitely divisible work, we can make h_2 go as low as we wish. Practically, nothing is infinitely divisible. Also, as nodes go on becoming smaller, the practical problem of thread-switch overhead becomes more and more relevant. We will tackle the second problem in Section 5.

A pipeline having more than one buffer between stages can be converted into a single-buffered pipeline for purposes of applying the above analysis. For example, if there is double buffering between two stages, we introduce a dummy stage between these two stages, moving one buffer to the input and one to the output of this dummy stage. This stage has a single node taking zero time, which copies its input buffer to its output (in no time - actually the second stage itself switches pointers). We add as many dummy stages between stages as there are extra buffers between them. Thus we convert a multiple-buffered pipeline to a single-buffered pipeline, which we can analyze as given in Theorem 2. The result below is an obvious special case.

Corollary 3. *If there are at least two buffers between every two stages, the bound on makespan is*

$$T \le \frac{fC}{m} + \frac{m-1}{m}(L+(f-1)h_1) \tag{9}$$

where h_1 is the largest node. $h_1 = \max_i t(m_i)$. □

The h_1 based bound above also holds for infinite-buffered pipelines (no buffering constraints). (A formal proof needs to find the longest chain in the job poset without buffering constraints.)

4. In-Process Bounds

The bounds in Section 3 hold for the makespan of the whole process, i.e. when the final-epoch item will be produced. They do not tell us when each intermediate-epoch item will be produced. In this section, we provide bounds for a continuously running pipeline. The buffering constraint is of prime importance here, because if it didn't exist (infinite buffers), then some work conserving schedules may bunch up actual item production very late in the schedule.

Theorem 3. *The item of epoch f will be produced by a continuously running parallelized pipeline with buffering constraints scheduled using any work conserving scheduler on m machines by the time*

$$\mathcal{U}_T(f+p-2) = (f+p-2)\frac{C}{m} + \frac{m-1}{m}(L+(f+p-3)h_2) \tag{10}$$

Proof. The epoch f item is produced when all the $J_{p,\bullet,f}$ jobs have finished running. Before this happens, none of the $J_{p-1,\bullet,f+1}$ can be started, which means none of the $J_{p-2,\bullet,f+2}$ jobs can be started and so forth, till we find that none of the $J_{1,\bullet,f+p-1}$ jobs can be started. Thus, the $(f+p-1)^{\text{th}}$ item doesn't even start its travel through the pipeline. Now, suppose we were to "cut off" the schedule produced by any work conserving scheduler the moment the f^{th} epoch item was completely produced. (Any programs that are half-completed at that moment are assumed to have completed with reduced job times.) What we get is a work conserving schedule of a sub-poset of the job poset up to the epoch $(f+p-2)$, with some job times reduced. Applying Corollary 2 and Theorem 2, the result is proved □

The rate of item production is still at worst

$$r = \frac{C}{m} + \frac{m-1}{m}h_2 \tag{11}$$

But because of the interference of subsequent epochs in production of an epoch, the latency has increased to

$$c' = (p-2)\frac{C}{m} + \frac{m-1}{m}(L+(p-3)h_2) \tag{12}$$

The first epoch can be as late as $c'+r$, each subsequent epoch f will have been produced by $c'+rf$. Furthermore,

Corollary 4. *If the item of epoch f_1 has been produced at time t_1, the item of a subsequent epoch f_2 will be produced by the time $t_1 + \mathcal{U}_T(f_2 - f_1 + p - 2)$.*

Proof. If we "cut" the schedule at time t_1 (and keep the subsequent events), the poset we get is a sub-poset of the poset starting from epoch f_1+1 (every job up to epoch f_1 has definitely finished). (Programs which were half completed are assumed to have just started, and completed with reduced job times.) The poset starting from epoch f_1+1 is equivalent to the poset starting from epoch 1. Renumbering the epochs, epoch f_1+1 becomes epoch 1, thus making epoch f_2 now epoch $f_2 - f_1$. Applying Theorem 3 and using the same sub-poset argument used in Corollary 2, we get the required result. □

Corollary 5. *If the item of epoch f_1 is ready for processing at time t_1, the item of a subsequent (or same) epoch f_2 will be produced by the time $t_1 + \mathcal{U}_T(f_2 - f_1 + 2p - 3)$.*

Proof. If $J_{1,\bullet,f_1}$ is ready, it means $J_{2,\bullet,f_1-1}$ have all finished, which means $J_{3,\bullet,f_1-2}$ have all finished, and so on. Continuing this chain of reasoning, we find that $J_{p,\bullet,f_1-p+1}$ have finished, meaning that the $f_1 - p + 1$ epoch item has been produced. Now, using Corollary 4, we get the required result. □

We learn in this section that the worst case rate of production of an item is the same $\frac{1}{m}C + \frac{m-1}{m}h_2$ calculated as the amortized rate in Section 3. The interference due to former and subsequent epochs causes extra additive delays in item production. The best case rate achievable is no better than C/m. The difference between the best and worst rates, fortunately does not translate into larger and larger arbitrary delays between successive items, as shown by the above corollaries.

If we implement a scheduling scheme where a subsequent epoch can never interfere with the production of the current epoch, we will get the smaller latency of Equation 8. (This is like earliest deadline first scheduling.) There is a simple method to implement this. A scheduler should give progressively higher priority to higher stages of the pipeline. Furthermore such priorities should be implemented preemptively. (This does not imply that two epochs will never be processed simultaneously. Whenever an epoch has only one or two ready jobs, the other machines are free to process jobs from subsequent epochs. Such jobs should be preempted the moment the earlier epoch puts more ready jobs in the ready queue.) Thus we have

Corollary 6. *Using preemptive pipeline-progressive priority scheduling, the f^{th} epoch will be produced by $\mathcal{U}_T(f)$. If an item of epoch f_1 is produced at time t_1, the item of epoch f_2 will be produced by time $t_1 + \mathcal{U}_T(f_2 - f_1)$. If an item of epoch f_1 is ready for processing at time t_1, an item of epoch f_2 will be produced by $t_1 + \mathcal{U}_T(f_2 - f_1 + p - 1)$.* □

5. Switching Overhead and Optimal Grain

In the preceding sections, we have seen that a parallelized pipeline produces items at the worst rate of $r =$

$\frac{1}{m}C + \frac{m-1}{m}h_2$. If all pipeline stages can be divided into as many number of jobs as one wishes, then the h_2 term can be brought down to zero. Practically, as we go on increasing the number of threads in an application, more time is spent in thread switching. In this section, we analyze the case where such thread switching (or job switching) takes constant time.

Suppose we have a p stage pipeline, with a total computation requirement of C time per epoch. Suppose we want to break up each stage into jobs, each taking time t (and an adjustment job). We call t the grain of parallelism. Suppose the job switching time is a constant a. What is the optimal grain t?

To analyze this situation, it is enough that we consider the setup time a to be included in the processing time of each job. The pipeline is divided into approximately $C/t+p$ jobs. Adding the switching overhead of all of these gives us an adjusted compute requirement of $C' = C + a(C/t + p)$ per epoch. The worst stage-pair time is $h_2 = 2(a+t)$. From Equation 7, we get that the processing rate is

$$r = \frac{1}{m}\left(C + a\left(\frac{C}{t} + p\right)\right) + \frac{m-1}{m}2(a+t) \qquad (13)$$

As t increases, the linear "bottleneck" term increases. As t decreases the reciprocal term in the total compute time increases. To find the optimal t, we differentiate the above expression with respect to t and equate to zero.

$$t_{\text{opt}} = \sqrt{\frac{aC}{2(m-1)}} \qquad (14)$$

(If instead of using the expression for r in the derivation above, we had used one of the expressions for T, the makespan, our solution would have depended on the number of epochs f, and would have tended in the limit to the above solution as the number of epochs increased.)

Substituting the value of t_{opt} in Equation 13, we get that the maximum guaranteed processing rate is

$$r_{\text{opt}} = \frac{C}{m} + \frac{ap}{m} + \frac{m-1}{m}(4t_{\text{opt}} + 2a) \qquad (15)$$

Here C is the processing time. $(ap + (m-1)2t_{\text{opt}})$ time is wasted due to switching overhead. $(m-1)(2t_{\text{opt}} + 2a)$ time is wasted due to underutilization of the machines. The wastage ratio (the ratio of wasted to utilized time over all the machines) is (in the worst case)

$$2\sqrt{2}\sqrt{\frac{a(m-1)}{C}} + 2\frac{a(m-1)}{C} + \frac{ap}{C} \qquad (16)$$

When C is considerably larger than a, and p and m are not too huge, the wastage ratio is close to $2\sqrt{2}\sqrt{a(m-1)/C}$.

It is important to note that the results in this section may be only used as design heuristics. The t_{opt} derived is optimal under an arbitrary assumption that all pipeline stages have the same granularity. There is no reason for this assumption beyond calculational simplicity. If the machine utilization achieved is not good enough, better granularity assignment for the same problem could be done, by assuming that every stage can have its own granularity, and solving the ensuing computational problem.

For example, even if we just let the odd and even numbered stages have different granularities t_o and t_e, we get $h_2 = 2a + t_o + t_e$. Repeating the same steps as above, we get the optima $t_{o\text{opt}} = \sqrt{aC_o/(m-1)}$ and $t_{e\text{opt}} = \sqrt{aC_e/(m-1)}$, where C_o and C_e are the processing requirements per epoch of all the odd and all the even numbered stages of the pipeline. This gives a wastage ratio of approximately $2((\sqrt{C_o}+\sqrt{C_e})/\sqrt{C})\sqrt{a(m-1)/C}$ which can be up to $\sqrt{2}$ times better than the wastage ratio with equal granularity throughout the pipeline.

6. On Scheduling Parallelized Pipelines

The parallelized pipelines as described in Section 3, can model various applications in the multimedia, graphics processing, DSP, control and network processing domain. Many multiprocessor architectures are available for these tasks like Cradle, Broadcom's CALISTO and Intel's IXP and IXS series. We claim that in many instances, such pipelines should be implemented using dynamic online (work conserving) scheduling. The benefits of doing so are described below.

The computational throughput achieved is high, and would be almost optimal in many cases, as described in Sections 3 and 4. There are two static scheduling methodologies generally in use. One, statically assigning pipeline stages to processors (MISD processing) In this methodology, all processors except the slowest waste time, since for most applications all stages of the pipeline do not have equal computational requirements. (A variation of this methodology, Pipelined Processor Farms [5] assigns many processors to a single stage. This methodology will become optimal only for a large number of processors, and only if the total computation warrants such a heavy multiprocessor.) Using dynamic scheduling, a heavier stage will be dynamically assigned more processors; even a very few processors can be near-optimally shared.

The other static scheduling methodology is to make all processors work on a single stage for a single epoch (SIMD processing). For optimal utilization, each stage should have a multiple of m jobs to process. This is impossible to achieve, and also inflexible. Also, light stages may not have enough work to be divided into m jobs. Using dynamic scheduling, if a stage does not take enough processors, the

idle processors will be assigned to process other stages.

The epoch jitter is bounded, as shown in Section 4. Of course these bounds are not as low as static scheduling. But for many data stream applications (those that do not require extreme real time response), these bounds are good enough. The jitter, being bounded, may be smoothed out using a jitter buffer. Specific scheduling methodologies having very low jitter may be designed (as in Corollary 6.)

Dynamic scheduling is more resilient to variations in job times. For example, sections of a scene may be more complex than other sections, causing some blocks to take more processing time than others. As another example, for a voice codec application, not everybody would be talking at the same time. Static scheduling cannot benefit much from such stochastic knowledge about an application. Dynamic scheduling "adjusts" for such variations. (We need to derive stochastic bounds on S and H.)

7. Unsolved Problems and Future Directions

This paper opens up many problems, which we list in this section.

The proof of Theorem 1 assumes that for all instants that a machine is free, $m-1$ machines can be free. This need not always be the case. Can we give better bounds for particular job posets? For particular algorithms? (Longest Job First seems beneficial in this context.)

In many applications we have stochastic knowledge about the running time of jobs. What can we derive about S and H under such circumstances?

Can we extend these results to general synchronous data flow graphs?

Given computational requirements of each stage of the pipeline, what is the optimal granularity assignment?

We have explained the use of preemptive pipeline-progressive priority scheduling (Section 4) for achieving lower jitter. In many cases, non preemptive scheduling can be done with a lower overhead than preemptive scheduling. Given the importance of switching overhead (Section 5), what bounds on latency can be achieved with non-preemptive pipeline-progressive priority scheduling?

The in-process bounds of Section 4 have a lot of obvious overestimation. Can sharper bounds be given?

A practical parallelized pipeline is usually not free running, but has specific release and removal times for epochs. E.g. a frame will come in from the camera only once every $1/30^{\text{th}}$ of a second. How are the in-process bounds affected by these timing constraints? (One would expect that as long as the processing requirement remains below $\frac{1}{m}C+\frac{m-1}{m}h_2$, and sufficient latency between input and output is allowed, a work conserving algorithm should be good enough to guarantee performance.)

8. Summary of Results

We showed that any work conserving algorithm takes time close to the "packed optimal" plus the critical path time for scheduling a set of jobs with precedence constraints. We showed that for parallelized pipelines, the rate of processing achieved is close to the "packed optimal" plus a bottleneck term which depends on the slowest pair of adjacent stage jobs. We also showed worst case timing latencies between epochs, which gives worst case real-time guarantees on the production of a certain item. We used these results to derive heuristics for granularity design of a parallelized pipeline. Lastly we gave justifications for dynamic online scheduling of parallelized pipelines. [1]

References

[1] A. Choudhary, B. Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for a class of pipelined computations. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, pages 439–445, 1994.

[2] F. Chudak and D. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[3] M. DiNatale and J. Stankovic. Dynamic end-toend guarantees in distributed real-time systems, 1994.

[4] L. Epstein. Lower bounds for on-line scheduling with precedence constraints on identical machines. *Lecture Notes in Computer Science*, 1444, 1998.

[5] M. Fleury, A. C. Downton, and A. F. Clark. Performance metrics for embedded parallel pipelines. *IEEE Transactions on Parallel and Distributed Systems*, 11(11), 2000.

[6] R. Graham. Bounds on multiprocessor timing anomalies. pages 416–429, 1969.

[7] F. Hiller and G. Lieberman. *Introduction to Operations Research*. McGraw Hill, 6 edition, 1995.

[8] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[9] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S. C. Graves, A. H. G. R. Kan, and P. H. Zipkin, editors, *Handbooks in Operations Research and Management Science*, volume 4, pages 445–552. Elsevier, Amsterdam, 1993.

[10] J. Sgall. On-line scheduling – a survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[11] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[1] **Acknowledgements:** The author would like to thank Dimple Kuriakose for help with the final manuscript, and Mukund Sundararajan, Abhishek Rathod and the Codito team for discussions and references.