An Experimental Study of Algorithms for Weighted Completion Time Scheduling*

Ivan D. Baev† Waleed M. Meleis† Alexandre Eichenberger‡

Abstract

We consider the total weighted completion time scheduling problem for parallel identical machines and precedence constraints, $P|prec|\sum w_iC_i$. This important and broad class of problems is known to be NP-hard, even for restricted special cases, and the best known approximation algorithms have worst-case performance that is far from optimal. However, little is known about the experimental behavior of algorithms for the general problem. This paper represents the first attempt to comprehensively describe and evaluate a range of weighted completion time scheduling algorithms.

We first describe a family of combinatorial scheduling algorithms that optimally solve the single machine problem, and show that they can be used to achieve good performance for the multiple machine problem. These algorithms are efficient and find schedules that are on average within 1.5% of optimal over a large synthetic benchmark consisting of trees, chains, and instances with no precedence constraints. We then present several ways to create feasible schedules from non-integral solutions to a new linear programming relaxation for the multiple machine problem. The best of these linear programming-based approaches finds schedules that are within 0.2% of optimal over our benchmark.

Finally, we describe how the scheduling phase in profile-based program compilation can be expressed as a weighted completion time scheduling problem and apply our algorithms to set of instances extracted from the SPECint95 compiler benchmark. For these instances with arbitrary precedence constraints, best linear programming-based approach finds schedules that are within 5.7% of optimal. Our results demonstrate that careful experimentation can help lead the way to high quality algorithms, even for difficult optimization problems.

Keywords: experimental evaluation, weighted completion time scheduling, precedence constraints, parallel machines, compilers

1 Introduction

A basic scheduling problem is $P|prec|\sum w_iC_i$ [1], the problem of scheduling n jobs under precedence constraints on m identical parallel machines to minimize total weighted job completion time. Each job i has a positive weight w_i and processing time p_i, $i = 1, \ldots, n$. The start time of job i in a schedule is denoted S_i and the completion time is denoted C_i. A directed acyclic graph describes the precedence constraints among the jobs such that an edge from job i to job j in the graph implies that $C_i \leq S_j$. In a feasible schedule no more than one job executes on any machine at any time, each job is scheduled nonpreemptively, and the precedence constraints are satisfied. The goal of the total weighted completion time (WCT) problem is to find a feasible schedule of the n jobs that minimizes $\sum w_iC_i$.

*A preliminary version of this paper appeared in the Proc. Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 1999.

†Department of ECE, Northeastern University, Boston MA, 02115 (baev,meleis@ece.neu.edu).

‡Department of ECE, North Carolina State University (alexe@eos.ncsu.edu).

The WCT problem is NP-hard even for fixed $m = 2$ and the empty precedence graph [2], but it can be solved efficiently if $m = 1$ for certain classes of precedence constraints. If the precedence graph is empty, an optimal schedule is found by sequencing the jobs in nonincreasing order of w_i/p_i, giving a largest ratio first (LRF) schedule [3]. Generalizations of this approach find optimal schedules for chains, trees, and series-parallel precedence graphs in $O(n\,logn)$ time [4, 5, 6, 7], but the single-machine WCT problem with arbitrary precedence constraints is NP-hard [7].

In the face of intractability two general approaches are traditionally employed. Enumerative methods most often use integer linear programming (ILP) formulations of the problem and rely on branch-and-bound techniques for solving it. These techniques do not run in polynomial time and therefore only small problem instances can be optimally solved. For example, for the WCT problem without precedence constraints, $P||\sum w_iC_i$, Belouadeh and Potts [8] show that a branch and bound algorithm based on a Lagrangian relaxation can solve instances for $n \leq 30$. Other studies have shown that ILP can be used to solve hard compilation problems, such as data layout [9], software pipelining [10, 11], register allocation [12, 13], and scheduling [14, 15, 16, 17].

On the other hand, approximation algorithms run efficiently and come with performance guarantees. More specifically, a δ-approximation algorithm is a polynomial-time algorithm that for every problem instance finds a solution whose cost is within a factor δ of optimal. Kawaguchi and Kyan [18] show that the LRF algorithm gives a 1.21-approximation for $P||\sum w_iC_i$, but until recently little was known about approximation algorithms for the general WCT problem. Hall, Shmoys, and Wein [19, 20] present a $(3 - 1/m)$-approximation algorithm for the case of unit job processing times, $P|prec, p_i = 1|\sum w_iC_i$, and a 7-approximation algorithm for $P|r_i, prec|\sum w_iC_i$. Chekuri, Motwani, Natarajan, and Stein [21] give 4- and 2-approximation algorithms for series-parallel and in-tree precedence graphs, respectively.

There has been recent interest in experimentally evaluating the quality of solutions provided by approximation scheduling algorithms. Savelsbergh, Uma, and Wein [22] apply linear programming-based approximation algorithms to the single-machine WCT problem without precedence and with release dates. They show that the best algorithms produce near optimal solutions in $O(n\,logn)$ time. The goal of our study is to evaluate the performance of a class of well-known scheduling algorithms when applied to the general WCT problem. These algorithms are divided into two groups: efficient algorithms that solve single machine WCT problems, and algorithms based on linear programming (LP) relaxations. While several approximation results make use of such single machine schedules, there are no empirical studies of the effectiveness of this approach. We show that using optimal single-machine schedules as priority lists for the multiple-machine problem produces near optimal solutions over a wide range of instances. We compare these schedules to optimal solutions for small instances found using ILP, and to tight lower bounds for larger instances. The simplicity of these algorithms makes them valuable for applications such as compilers.

Next, we use the non-integral solutions to a relaxed LP formulation to create feasible schedules. We investigate two such rounding techniques, α-scheduling and average start-time algorithms. This LP-based approach experimentally outperforms the single-machine based approach. Our work represents the first experimental evaluation of various algorithms for the general WCT problem, and extends the results in [22] to multiple machines and in-tree, chain, and empty precedence constraints. Finally, we show that these techniques can be applied to a practical compiler scheduling problem. We model the scheduling phase of profile-based program compilation as a weighted completion time scheduling problem and apply our algorithms to a set of instances extracted from the SPECint95 compiler benchmark. For these instances with arbitrary precedence constraints, the best linear programming-based approach finds near-optimal schedules.

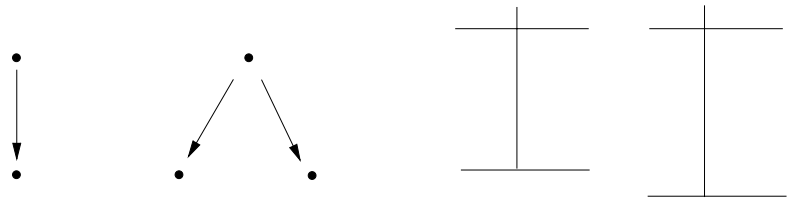

Figure 1: Delays may be necessary in optimal schedules for out-trees

2 Algorithms

The algorithms considered in this study each consist of two stages. They first assign priorities to the jobs, and then use *list scheduling* to assign each job a start time. A job i is *ready* at any point in the schedule if every job that has precedence over i has completed execution. When a machine becomes free, the highest priority ready job that has not begun or completed execution is assigned to that machine. In this section we describe a series of list scheduling algorithms. We begin by discussing the class of single-machine optimal algorithms and then describe LP-based approaches.

The schedules found by list scheduling-based algorithms do not contain delays, where a delay is defined as a time period during which a machine is idle even though there is a ready job available. Optimal solutions to the WCT problem with arbitrary precedence constraints may contain delays, as shown in Figure 1 for the case of out-trees. However the following lemma demonstrates that no delays are necessary when the precedence graph is an in-tree or a set of chains.

Lemma 1. *Given an instance of the WCT scheduling problem where every job in the precedence graph is an in-tree, an optimal schedule exists without delays.*

Proof. Given a minimum cost schedule that contains a delay, we show that it is possible to create another schedule containing no delays whose cost is not larger. We start with an optimal schedule containing at least one delay and consider the last time t at which a job i is issued on a machine immediately following a delay. We let m_i be this machine.

If i does not depend on a job that completes at time t, then i can be scheduled at time $t-1$ without violating the precedence constraints or increasing the cost of the schedule. This transformation eliminates the delay between time $t-1$ and t on machine m_i, and possibly creates up to one new delay after job i. Therefore the number of delays does not increase.

If i does depend on such a job, then let k_c be the number of jobs that complete at time t and k_s be the number of jobs that start execution at t that depend on one of the k_c jobs. We then have $k_s \leq k_c$ since at most one job can depend on each of the k_c jobs. If more than k_s jobs begin execution at time t, there must exist a job j on machine m_j that does not depend on any of the k_c jobs that complete at time t. We exchange j and all jobs following it on machine m_j with i and all jobs following it on machine m_i, and move job j one time period earlier. This transformation does not violate the precedence constraints or increase the cost of the schedule. The delay between time $t-1$ and t on machine m_i has been eliminated, and possibly up to one new delay is created after job j. Therefore the number of delays does not increase.

Now consider the case where exactly k_s jobs begin execution at time t. At time t, there are k_c+1 machines available to start executing jobs, and only $k_s < k_c+1$ jobs that actually begin executing.

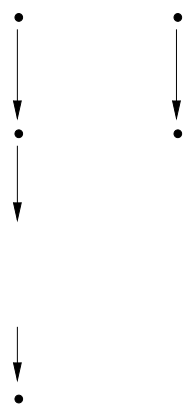

Figure 2: A worst-case LRF instance with precedence

Therefore at least one machine is idle at time t, and this machine must also be idle from time $t+1$ until the end of the schedule because by assumption there are no delays after t. We assign i and all jobs following i on machine m_i to that machine, eliminating the delay between time $t-1$ and t on machine m_i, and creating no new delays. This transformation does not violate the precedence constraints or increase the cost of the schedule.

This process is repeated until no delays remain by again considering the last time at which a job is issued on a machine immediately following a delay. The process must terminate because at each step either one job is moved earlier in the schedule and the number of delays does not increase, or jobs are reassigned to another machine and the number of delays decreases. In this way we produce a schedule without delays without increasing the cost. □

We conclude that by using an appropriate priority assignment, list scheduling-based algorithms can find optimal solutions to WCT problems when the precedence graph is an in-tree or a set of chains.

2.1 Single-machine optimal algorithms

Single machine scheduling problems for WCT are well understood, and efficient algorithms exist for a range of special cases. We describe algorithms that apply when the precedence graph is either empty, a set of chains, or a tree, and give the worst-case performance ratio for each.

The LRF algorithm finds a minimum cost schedule when the precedence graph is empty and $m = 1$ [3]. This can be seen by the following argument: if there is a job i that is immediately followed in the schedule by job j such that $w_i/p_i < w_j/p_j$, then interchanging i and j in the schedule reduces the cost of the schedule by $w_j p_i - w_i p_j = p_i p_j (w_j/p_j - w_i/p_i)$, which is positive.

The LRF algorithm

1. The priority of every job i is $\rho_i = w_i/p_i$.
2. Apply list scheduling.

The algorithm does not necessarily find a minimum cost schedule if $m > 1$ or if there are precedence constraints. Furthermore, the cost of the schedule found can be arbitrarily many times worse than the optimal schedule. Consider the single machine WCT instance shown in Figure 2, where $s \geq 2$

Figure 3: ρ-max algorithm for chains compared to optimal algorithm

is some fixed constant. In the optimal schedule the jobs in Chain 2 begin execution at time 0 and 1, giving a cost of $3s^2 + 5s + 1$. The LRF algorithm schedules Chain 1 first, giving a cost of s^3+3s^2+2s+1. The ratio of the cost of the LRF schedule to the cost of the optimal schedules tends to infinity as s increases. The LRF algorithm is greedy in the sense that it gives higher priority to the job with the highest ratio without considering the effects of precedence. Scheduling job $s+1$, which has a small ratio, allows job $s+2$ with a large ratio to be scheduled earlier. It is this poor worst-case performance that makes the LRF algorithm unappealing for solving the general WCT problem. Furthermore, our experiments show that the LRF algorithm gives the worst results of the algorithms we consider (see Section 3.1).

Several algorithms are known to solve the single-machine WCT problem for special classes of non-empty precedence constraints [4, 5, 6, 7]. The decomposition approach of Sidney subdivides the precedence graph into modules and then uses optimal schedules of each module to construct the final schedule. We selected this approach because it is widely applicable and easy to implement.

We first describe an algorithm that applies when the precedence constraints form a set of r chains. In this case every job has no more than one immediate predecessor and no more than one immediate successor. Let n_i be the number of jobs in the ith chain, and let $i:j$ represent a job where i is the chain number and j is the job's position within the chain. Then for any $i:j$ such that $j \in [2, n_i]$, job $i:j-1$ is an immediate predecessor of job $i:j$.

We also define $Pred(i:j) = \{i:k | 1 \leq k \leq j\}$. That is, $Pred(i:j)$ equals the job $i:j$ and all of its predecessors. We let H_i equal the set of jobs in the ith chain. For a set of jobs U we define the ratio $\rho(U) = \sum_{i \in U} w_i / \sum_{i \in U} p_i$.

The ρ-max algorithm for chains

1. Begin with an empty priority list β.
2. Repeat the following until the chains are all empty:
   (a) For every nonempty chain H_i, let $\delta_i = \max_j\{\rho(Pred(i:j))\}$.
   (b) Let $\delta = \max_i\{\delta_i\}$, and let i^* and j^* be the largest values such that $\rho(Pred(i^*:j^*)) = \delta$.
   (c) Add the jobs in $Pred(i^*:j^*)$ in order to the end of β, and remove these jobs from H_{i^*}.
3. Apply list scheduling, using β as the priority list.

This algorithm can be implemented to run in $O(n\,logn)$ time [7]. The set of consecutive jobs selected in step 2(b), referred to as a *module*, minimizes the ratio over all other such sets. A schedule is called *module-intact* if every module is scheduled consecutively on a particular machine. Sidney [5] shows that for arbitrary precedence constraints, an optimal module-intact single-machine schedule always exists. However in this general case finding modules and scheduling within a module can be difficult. When the precedence graph is a set of chains, both steps can be implemented efficiently, as shown in the ρ-max chain algorithm.

When $m = 1$, the priority list and the final schedule are identical and the cost is minimized. However the chain algorithm does not necessarily find minimum cost schedules for $m > 1$, as shown in Figure 3. When the algorithm is applied to this instance with $m = 2$, the modules consist of jobs $\{1, 2\}$, $\{3, 4\}$, and $\{5, 6\}$, which are each scheduled consecutively. However in an optimal schedule these jobs are not scheduled consecutively. We have therefore shown that there exist instances of the WCT problem that do not have module-intact solutions, even when the precedence graph is a set of chains. In this paper, the single-machine optimal algorithms we apply to the general WCT problem find module-intact schedules. No theoretical or experimental results exist that predict the extra cost incurred by restricting the search space in this way. In Section 3.3 we show that the average difference in cost between optimal module-intact schedules and optimal schedules is very small.

A variation of this chain algorithm can also be used to find optimal schedules when the precedence graph is a tree where each job has at most one immediate successor (i.e. an in-tree). Jobs that have more than one immediate predecessor are called *branch jobs*, and a branch job that does not have a direct or indirect predecessor that is a branch job is called an *initial branch job*. Note that initial branch jobs depend only on chains, and if there are no branch jobs the precedence graph must be a single chain. Here we let $Pred(i)$ equal all predecessors of job i.

The ρ-max algorithm for trees

1. Repeat the following until there are no remaining branch jobs:
   (a) Select an initial branch job i.
   (b) Use the chain algorithm to optimally order the jobs in $Pred(i)$ and set β equal to this order.
   (c) Add a precedence arc (j, k) to the precedence graph if job j precedes job k in β.
2. Create a priority list using the total ordering of jobs, and apply list scheduling.

The tree algorithm applies the chain algorithm to initial branch jobs, replacing two chains with a single chain composed of a series of modules. This entire procedure can be implemented to run in $O(n\,logn)$ time [7]. Chekuri et al. [21] show that the worst-case performance ratio for the ρ-max algorithms applied to chains and trees is 2.

2.2 Linear programming-based algorithms

There has been recent interest in using optimal solutions to linear programming relaxations of scheduling problems as the basis for approximation algorithms [19, 20, 21]. These techniques typically solve an easier, less constrained version of a scheduling problem. That schedule is then used as a priority list for a list scheduler that produces the final schedule. Savelsbergh, Uma, and Wein [22] show that such algorithms can find high quality solutions for the single machine WCT with release dates, but without precedence. We use a group of similar ideas to produce solutions to the general WCT problem from solutions to the LP relaxation.

We formulate the WCT problem as follows

$$\text{minimize} \sum_{i=1}^{n} \sum_{t=1}^{T} w_i \cdot x_{i,t} \cdot t \tag{1}$$

subject to

$$\sum_{t=1}^{T} x_{i,t} = 1 \qquad (i = 1 \ldots n) \tag{2}$$

$$\sum_{i=1}^{n} \sum_{t'=t}^{t+p_i-1} x_{i,t'} \leq m, \qquad (t = 1 \ldots T) \tag{3}$$

$$\sum_{t=1}^{T} (x_{i,t} \cdot t) \leq \sum_{t=1}^{T} (x_{j,t} \cdot t) - p_j \qquad ((i,j) \in R) \tag{4}$$

$$x_{i,t} \in \{0, 1\} \tag{5}$$

The binary variables $x_{i,t}$ indicate whether the ith job completes at time t, and parameters p_i and w_i give the processing time and weight of the ith job. Constraints (2) and (3) ensure that every job completes exactly once and that no more than m jobs are executing at any time. There is an arc (i, j) in R for each precedence arc from job i to job j, and for each such arc constraint (4) ensures that the completion time of i is at least time p_j before the completion time of j. The formulation minimizes the objective function (1) which gives the weighted sum of the completion times of the jobs. This formulation has approximately $n \cdot T$ variables, where T is the sum of the latencies of all the jobs. As a result, memory restrictions prevent the model from solving large WCT instances on available computing resources.

We use this formulation to find exact solutions for small instances of the WCT problem. We compute a lower bound on the optimal value of the objective function for large instances using a relaxation of this formulation where the variables $x_{i,t}$ can take on values between 0 and 1. We show in Section 3 that this bound is very tight and we use it to evaluate the quality of the solutions found by the scheduling heuristics.

The non-integral solutions to the WCT problem found by the relaxed formulation are not necessarily solutions to the exact formulation. The relaxed formulation allows multiple jobs to execute simultaneously on a machine. In this case, constraint (2) ensures that for each job i, the sum of $x_{i,t}$ over all time periods t equals 1. However the completion time of the ith job may now be distributed over many time periods. Consider for example some job i with $p_i = 3$, and $x_i = (0, 0, .1, 0, 0, .1, 0, .8)$ in a solution to the relaxed formulation. This indicates that .1 of job i is executing from time 1 to time 3, .1 is executing from time 4 to time 6, and .8 is executing from time 6 to time 8. Notice that each fractional component of job i has the same processing time, and that two of the components overlap at time 6. By constraint (3), the total number of fractional components executing at any time does not exceed the number of machines, and constraint (4) ensures that for each arc (i, j) in R the average completion time of job i does not exceed the average start time of job j. The average completion time $\sum_{t=1}^{T}(x_{i,t} \cdot t) = 7.3$ is the sum of the completion times of the fractional components of job i, weighted by the fractional size of each component. The average start time is then computed by subtracting the processing time of job i.

An alternate relaxation of the WCT scheduling problem allows jobs to execute preemptively while only allowing one job to execute on a machine at any time. This formulation has been successfully used to develop approximation algorithms for the single processor WCT problem without precedence. The preemptive relaxation is appealing in this case because the preemptive WCT scheduling

problem under these conditions is solvable in polynomial time. The exact solution to the preemptive problem is therefore a good starting point for solving the nonpreemptive problem. However, the general WCT problem with preemption remains intractable.

We use a non-integral solution to the WCT problem to construct priority lists that are then list scheduled to give a feasible schedule. There are several ways that the priority lists can be constructed. For some fixed $\alpha \in [0, 1]$, an *α-schedule* is created using the following algorithm.

The α-scheduling algorithm

1. Solve the relaxed integer-programming formulation
2. For each job i, select the earliest time t_i such that $\sum_{t=1}^{t_i} x_{i,t} \geq \alpha$
3. Set the priority of the ith job equal to $t_i - p_i$ and apply list scheduling

The second step selects for each job the earliest time where at least an α fraction of that job has completed executing. If $\alpha = .01, .5$, or 1.0, then the respective completions times are $t = 3, 8$, or 8 for the job considered above. Note that setting $\alpha = 0$ always gives a priority of $t = 0$. The third step creates a priority list based on the corresponding job start times $t = 0, 5$, or 5. The α parameter allows the scheduler to produce a family of related feasible schedules that are derived from a single solution to the relaxed formulation. For example, $\alpha = .01$ gives each job a priority that closely approximates the earliest time when that job executes in the relaxed formulation. Similarly, $\alpha = 1$ gives each job a priority equal to the latest time when that job executes in the relaxed formulation. Values of α between these extremes allow job priorities to be determined by different fractions of job start times. Finally, list scheduling creates a feasible m-machine schedule.

In this paper we evaluate the quality of α-schedules for a range of fixed α values between .01 and 1.0. We also evaluate an algorithm, *Best-α*, that returns the minimum cost α-schedule. A related scheduling algorithm lets a job's priority be its average start time in the non-integral schedule, as described below.

The average start-time algorithm

1. Solve the relaxed integer programming formulation
2. Set the priority of the ith job equal to $\sum_{t=1}^{T}(x_{i,t} \cdot t) - p_i$ and apply list scheduling

The priorities computed by the average start-time algorithm are similar to those found by the α-scheduler with $\alpha = .5$. However the average start-time algorithm computes a true average start time for each job, (7.3 in the above example) while the α-scheduler with $\alpha = .5$ finds the earliest time such that half the job has been scheduled (8 in the example). The α-scheduler selects some priority equal to t such that $x_{i,t}$ is non-zero, while the average start-time algorithm can select a time when a job is not executing at all or a non-integral time.

3 Experiments

We evaluate the algorithms described in the previous section by applying them to a set of 440 synthetic WCT instances. An instance consists of a set of n jobs, each with a processing time and a weight, a precedence graph on the jobs, and a set of m machines. The instances whose precedence graphs are chains (chain instances) are divided into small instances with $n = 10, 15, 20, 25, 30$, and large instances with $n = 40, 60, 80, 100, 120, 140$, and 160. The instances whose precedence graphs are empty are divided into the same groups. The instances whose precedence graphs are a tree (tree instances) are divided into a group of small instances, with $n = 10, 15, 20$, and large instances with $n = 40, 60, 80, 100, 120, 140$, and 160. The processing time of each job is uniformly distributed

prec	No precedence			Chains			Trees		
m	2	4	6	2	4	6	2	4	6
mean	0.00	0.00	0.00	0.06	0.05	0.03	0.40	0.02	0.00
max	0.00	0.02	0.00	1.04	0.44	0.51	2.30	0.52	0.00

Table 1: Quality of LP lower bound vs. optimal value for small instances (1.0 = 1%).

in $[1, 5]$, and the weight of each job is uniformly distributed in $[1, 100]$. Twenty random instances are generated for each n, with $m = 2, 4$, or 6 for each.

We randomly select precedence constraints for each instance such that every unique graph is equally likely to occur. Selecting a dependence graph that is a chain is equivalent to selecting a random partition of a set of n elements. For example, the partition of the set $\{1, 2, 3, 4, 5\}$ into $\{\{1, 2, 3\}, \{4\}, \{5\}\}$ corresponds to the dependence graph containing jobs $1, 2$, and 3 in the first chain, and jobs 4 and 5 in separate chains. Without loss of generality we assume that two partitions are identical if the sets of component sizes for each partition are identical. The algorithm to select a random partition is described in [23]. This approach computes $p(n, k)$, the number of partitions of n whose largest component is exactly k, and $p(n)$, the number of partitions of n. Then the largest component is selected to be of size k with probability $p(n, k)/p(n)$, and this process is repeated with successively smaller components.

We use a random binary tree generation algorithm [24, 25] that ensures that every distinct tree is equally likely to be selected. A binary tree containing n nodes can be uniquely represented by a sequence of n ordered pairs of binary values (x, y) such that exactly $n - 1$ of the values equals one. If x equals 1 then the corresponding node has a left parent, while if x equals 0 then the node has no left parent. The same holds for y and its right parent. The pairs are listed in the order in which the nodes are visited in a preorder traversal of the tree. Not all such sequences correspond to valid binary trees, but every sequence contains exactly one valid sequence among all its rotations. An invalid sequence can be transformed into a unique valid sequence by considering all rotations. Therefore there is a one-to-one mapping from sequences to binary trees. A random binary tree with n nodes is generated by creating a random sequence containing $n - 1$ ones, rotating it to create a valid sequence, and mapping it to the equivalent binary tree.

We begin by computing optimal solutions to the small instances using the IP formulation described in Section 2.2. The formulation is coded in AMPL and solved using a CPLEX solver running on a 166MHz Sun 4 Workstation with 128 Mb of memory. The largest of these instances ($n = 30$ for chains and $n = 20$ for trees) represent the largest WCT problems that the formulation solved to completion, and required several hours to complete. While it is likely that more aggressive approaches will be able to solve larger instances our focus is on evaluating the quality of efficient heuristics and we show that these results and our lower bound are sufficient to achieve this end.

We use the relaxed (nonintegral) version of the IP formulation to compute a lower bound on the cost of the optimal schedule for both the small and the large instances. The average percentage difference between the lower bound and the optimal value for the small instances is shown in Table 1 for each m. Without precedence, the average difference is 0.00% and the largest difference is 0.02%. For chains, the average difference is 0.046%, and the largest difference is 1.04%. For trees, the average difference is 0.127% and the largest difference is 2.30%. These results suggest that the relaxed formulation gives a very tight lower bound, within a fraction of a percent of the optimal

value, over our entire benchmark. For a particular tree instance, the bound may understate the optimal value by several percent, but over a wide range of instances sizes we can reliably use the bound to evaluate the quality of our heuristics.

3.1 Single machine optimal algorithms

We apply the LRF and ρ-max algorithms to the small instances and show the average percentage difference between this result and the optimal value for each small instance in Table 2. The average percentage difference between the heuristics and the lower bound is shown for each large instance in Table 3.

The results indicate that without precedence, the LRF algorithm finds schedules that are on average 0.33% from the optimal value for the small instances, and 0.03% from the lower bound for the large instances. When this algorithm is applied to chains and trees, the average difference rises to 6.17% and 2.73%, respectively, for the small instances, and 8.56% and 12.87% for the large instances. In contrast the equivalent differences for the ρ-max algorithms are 1.26% and 1.54% for the small instances, and 0.40% and 5.40% for the large instances. The ρ-max algorithm, which gives optimal results for the WCT problem with $m = 1$, reduces the average cost for chain instances by 83% and the average cost of tree instances by 55% compared with LRF over the entire benchmark.

The performance of these WCT algorithms for multiple machines is related to the algorithms' single machine performance. In the absence of precedence constraints the LRF algorithm, which is optimal for a single machine, gives very good performance on multiple machines. With even the simplest precedence constraints (chains) the LRF algorithm has an unbounded worst-case performance ratio, and its performance on multiple machines is quite poor. The ρ-max algorithm, which is optimal for chains and trees for a single machine, gives much better performance on multiple processors.

3.2 LP-based algorithms

We compute α schedules for 10 values of α between 0.1 and 1.0 in increments of 0.1. We also use $\alpha = 0.01$ to give priorities equal to the earliest starting time for each job. The best-α algorithm then selects the best α-schedule for each instance. The average start-time algorithm computes a weighted average of the fractional start times for each job.

The average percentage differences between the costs of the best-α schedules and average start-time schedules, and the cost of the optimal schedules for the small instances, are shown in Table 2. The corresponding average percentage differences from the lower bound for large instances are shown in Table 3. The average start-time algorithm finds schedules that are on average 0.16% from the optimal value for the small instances and 0.39% from the lower bound for the large instances. Some of these differences can be attributed to the gap between the lower bound and the optimal value. The best-α algorithm finds schedules that are on average 0.06% from the optimal value for the small instances, and 0.33% from the lower bound for the large instances. Varying α introduces small perturbations in the priorities which allows the scheduler to find a locally optimal schedule that outperforms the schedule found for any fixed α. It is significant that the average start-time algorithm does an order of magnitude less work to transform non-integral solutions into feasible schedules, and finds schedules that are only slightly worse than those found by the best-α algorithm.

More detailed α-scheduling results are shown in Figure 4 and in Tables 4 and 5. For each fixed value of α, we show the average percentage difference between the cost of the α-schedules and the cost of the optimal schedules over the all the small and large instances with precedence in Figure 4.

	prec	No precedence			Chains			Trees		
	m	2	4	6	2	4	6	2	4	6
LRF	mean	0.12	0.31	0.57	8.11	5.54	2.85	6.23	1.46	0.01
	max	1.33	2.43	2.60	28.27	26.55	25.61	26.04	13.16	0.32
ρ-max	mean	0.12	0.31	0.57	1.25	1.50	0.59	3.76	0.78	0.00
	max	1.33	2.43	2.60	10.20	12.12	5.07	17.17	12.27	0.12
average start-time	mean	0.00	0.00	0.00	0.22	0.27	0.12	0.66	0.14	0.00
	max	0.00	0.18	0.00	2.29	3.18	4.76	5.47	3.34	0.00
best-α	mean	0.00	0.00	0.00	0.12	0.04	0.09	0.26	0.00	0.00
	max	0.00	0.05	0.00	2.29	0.89	1.79	3.16	0.27	0.00

Table 2: Quality of LRF, ρ-max and LP-based algorithms vs. optimal value for small instances ($1.0 = 1\%$).

	prec	No precedence			Chains			Trees		
	m	2	4	6	2	4	6	2	4	6
LRF	mean	0.01	0.03	0.06	8.80	8.62	8.25	11.20	14.34	13.07
	max	0.09	0.31	0.42	18.54	18.22	18.43	31.25	30.80	26.50
ρ-max	mean	0.01	0.03	0.06	0.19	0.45	0.55	2.33	6.55	7.31
	max	0.09	0.31	0.42	9.57	6.66	4.59	11.32	17.45	16.76
average start-time	mean	0.00	0.00	0.00	0.11	0.15	0.11	1.29	1.05	0.76
	max	0.00	0.03	0.04	1.56	1.96	1.03	7.04	3.46	3.67
best-α	mean	0.00	0.00	0.00	0.05	0.08	0.07	1.19	0.95	0.67
	max	0.01	0.04	0.04	0.81	0.70	0.58	4.28	4.30	3.66

Table 3: Quality of LRF, ρ-max and LP-based algorithms vs. lower bound for large instances ($1.0 = 1\%$).

The minimum average gap assuming fixed α is achieved for $\alpha = 0.4$ over the small instances and for $\alpha = 0.3$ over the large instances. As α increases or decreases, the size of the gap increases sharply. Notice however that the minimum gap for fixed α is still significantly larger than the gap achieved by the best-α algorithm. This is because the best-α algorithm attempts to select the optimal α for each instance, while the fixed α algorithm selects a single value for α.

The results in Tables 4 and 5 give the average percentage differences between the costs of the α schedules and the cost of the optimal schedules for the small instances, or the lower bound for the large instances, for every value of α. The minimum average gap is highlighted in each column. Each entry in Tables 4 and 5 gives the average gap over the instances when scheduled with a fixed value of α. The optimal fixed value of α is between 0.3 and .5 for trees, and between 0.01 and 0.3 for chains. For chains, there is little variation in the quality of the schedules as α varies, while for trees the change in the quality of the schedules is more pronounced. In general, the quality of the schedules decreases significantly for large α, and $\alpha = 1$ usually gives the worst schedule.

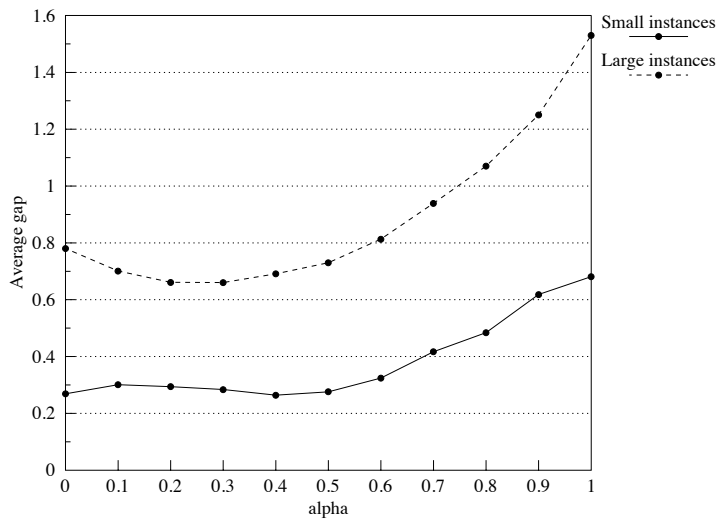

Figure 4: Average quality of α-schedules for fixed α over instances with precedence

3.3 Optimal module-intact schedules

The schedules found by the ρ-max algorithms are module-intact which greatly reduces the space of schedules considered. It is an open question whether this approach limits the effectiveness of multiple processor scheduling algorithms for the WCT problem. We answer this question by finding optimal module-intact schedules for the synthetic chain instances.

We use a variation on the ρ-max algorithm for chains to find the modules for each chain instance. We create a new module-intact instance in which the bth module $\{1, 2, \cdots, k\}$ in the original instance, containing k jobs, is replaced by a new job b, with $w_b = \sum_i w_i$, and $p_b = \sum_i p_i$ summed over the jobs in the module (here we assume that the jobs in the module are numbered consecutively starting with 1). Any module-intact schedule of the original jobs can be replaced by an equivalent schedule of the new jobs, where consecutively scheduled jobs within a module are replaced by a single job. The cost of scheduling jobs $\{1, 2, \cdots, k\}$ consecutively such that job k completes at time t is a constant more than the cost of scheduling job b so that it completes at time t. This constant is independent of t and can therefore be subtracted from the cost of any module-intact schedule of the new jobs, giving the cost of scheduling the original jobs in an equivalent schedule.

The results of using the IP model to find an optimal module-intact schedule for each chain instance in the benchmark are shown in Table 6. Comparison is made with the optimal IP result for small instances, and with the LP lower bound for large instances. We find that on average the cost of optimal module-intact schedules is very close to the cost of optimal schedules, and that 91 out of 100 small chain instances achieve the optimal cost. This suggests that it makes sense to further study scheduling algorithms that do not attempt to split modules. Furthermore, it is much easier to find module-intact schedules because the number of jobs and precedence constraints is smaller. As a result, we are able to find optimal module-intact schedules for $n \leq 160$, and the quality of the solutions is even better than those found by the LP-based algorithm.

	No precedence			Chains			Trees		
α	m = 2	4	6	2	4	6	2	4	6
.01	0.00	0.00	0.00	0.31	0.17	0.14	1.09	0.17	0.00
0.1	0.00	0.00	0.00	0.29	**0.16**	**0.13**	1.26	0.17	0.00
0.2	0.00	0.00	0.00	**0.28**	0.20	0.14	1.11	0.13	0.00
0.3	0.00	0.00	0.00	0.31	0.25	**0.13**	0.99	**0.01**	0.00
0.4	0.00	0.00	0.00	0.34	0.24	0.15	0.68	0.04	0.00
0.5	0.00	0.00	0.00	0.30	0.25	0.17	**0.67**	0.04	0.00
0.6	0.00	0.00	0.00	**0.28**	0.43	0.23	0.69	0.04	0.00
0.7	0.00	0.00	0.00	0.30	0.54	0.31	1.01	0.06	0.00
0.8	0.00	0.00	0.00	0.41	0.59	0.31	1.18	0.10	0.00
0.9	0.00	0.00	0.00	0.45	0.83	0.34	1.69	0.12	0.00
1.0	0.00	0.00	0.00	0.73	0.88	0.42	1.62	0.12	0.00

Table 4: Average quality of α-schedules with fixed values of α for small instances($1.0 = 1\%$).

	No precedence			Chains			Trees		
α	m = 2	4	6	2	4	6	2	4	6
.01	0.00	0.00	0.00	0.10	**0.14**	0.13	1.84	1.83	1.42
0.1	0.00	0.00	0.00	**0.09**	0.15	0.13	1.65	1.58	1.21
0.2	0.00	0.00	0.00	0.10	**0.14**	**0.12**	1.56	1.42	1.09
0.3	0.00	0.00	0.00	0.10	0.15	**0.12**	**1.52**	**1.37**	**1.06**
0.4	0.00	0.00	0.00	0.10	0.15	0.13	1.60	**1.37**	1.09
0.5	0.00	0.00	0.00	0.10	0.16	0.14	1.68	1.42	1.11
0.6	0.00	0.00	0.00	0.12	0.19	0.19	1.85	1.61	1.13
0.7	0.00	0.00	0.00	0.13	0.23	0.25	2.11	1.79	1.36
0.8	0.00	0.00	0.00	0.15	0.31	0.28	2.32	2.08	1.54
0.9	0.00	0.00	0.00	0.18	0.37	0.32	2.62	2.49	1.87
1.0	0.00	0.00	0.00	0.21	0.43	0.35	3.14	3.19	2.36

Table 5: Average quality of α-schedules with fixed values of α for large instances($1.0 = 1\%$).

4 Case study: superblocks

We apply the algorithms described in this paper to a scheduling problem that arises in the design of high-performance compilers. A *basic block* is a maximal sequence of instructions that are guaranteed to execute in sequence. Instructions can be easily rescheduled within basic blocks because the compiler has complete information at compile-time about which instructions within the block will execute if the block is entered. Moving instructions between basic blocks is more difficult because it involves speculatively executing instructions before later branches are resolved. It is this inter-basic block instruction movement that compiler designers use to maximize the utilization of a processor. To facilitate inter-basic block instruction movement, compilers create *superblocks*[26], a generalization of basic blocks. A superblock consists of a sequence of frequently executed, consecutive basic blocks. They are constructed by first identifying an entry basic block, typically the first block of a function or loop. Then basic blocks belonging to the most frequent path originating from the

		Chains		
	m	2	4	6
Small	mean	0.01	0.00	0.00
	max	0.42	0.20	0.01
Large	mean	0.11	0.11	0.11
	max	0.76	0.14	0.21

Table 6: Quality of module-intact schedules for chain instances ($1.0 = 1\%$).

	m	2	4	6
LRF	mean	232.33	148.57	98.97
	max	1134.28	853.90	524.88
average start-time	mean	8.82	7.01	6.44
	max	51.84	37.30	34.39
best-α	mean	7.62	5.42	4.01
	max	37.86	29.76	22.77

Table 7: Quality of LRF and LP-based algorithms vs. lower bound for compiler benchmark ($1.0 = 1\%$).

entry block are added to the superblock until the end of the function or loop is reached, or until the cumulative probability of the path drops below a certain cutoff value. Because superblocks do not allow side entrances (i.e. no branches outside the superblock may target any blocks of the superblock other than the entry block), some code replication may be needed.

Compilers attempt to maximize performance by using program profiling information to schedule superblocks [27, 28]. A particular superblock may consist of branches that are not frequently taken. Postponing such branches may allow more frequently taken branches to be scheduled earlier. Maximum performance is achieved when the scheduler gives higher priority to instructions whose execution is more likely to be necessary. These priorities are represented in the probability that each branch is taken. A high probability for a particular branch indicates that that branch should be scheduled early, possibly at the expense of other branches whose execution is less likely. This probabilistic scheduling model can be easily modeled using the WCT scheduling model considered above. Given a superblock, we let each branch instruction's weight equal the probability that that branch is the last instruction executed in the block. All other instructions have a weight of zero. A solution to this WCT scheduling problem is a schedule that has the smallest expected completion time.

4.1 Experiments

The specific machine latencies used are these of a Tinker-6 machine which can issue up to 6 integer operations at once. These operations execute on 3 integer units, 2 memory units, and 1 branch unit. It is a statically scheduled, very long instruction word (VLIW) architecture. All operations have unit latency except for loads, which take 2 cycles.

The 112 scheduling instances are extracted from the SPECint95 benchmark suite. Classic compiler optimizations are applied to each benchmark program using the IMPACT compiler. The benchmarks are then converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories. Superblock formation is applied within the LEGO research compiler, which has been developed by the TINKER group at North Carolina State University. Dependences among operations consist of all flow dependences, memory dependences between consecutive stores, anti and output dependences due to live values at the side exits of the superblocks. Also, control flow dependences were introduced to prevent operations from moving below branches. However, operations are free to move above branches. The exit probabilities of each branch in the benchmark was measured during runs of the SPECint95 benchmark.

The size of the 112 superblocks varies from 30 to 339 operations with an average of 92. The weights of all non-branch operations are zero. Branch operations' weights are equal to their frequency of execution and vary from 0 to 10^6. The instances' precedence graphs are directed acyclic graphs, and in general an operation may have more than one successor. The complex structure of the precedence makes scheduling these instances much more difficult than scheduling the synthetic instances considered above. For example, since Lemma 1 does not apply, an optimal schedule may include delays which are never introduced by list scheduling-based algorithms.

The average percentage differences between the costs of the LRF, best-α, and average start-time schedules and the cost of the lower bound are shown in Table 7. These algorithms find schedules that are on average 159.96%, 7.42%, and 5.68%, respectively, from the lower bound. We observe that when applied to the compiler benchmark, the relative performance of the scheduling algorithms remains unchanged when compared with the results from the synthetic benchmark. In both cases the error in the LP-based algorithms is an order of magnitude less than the error in the LRF algorithm. The total computation time for the LP-based algorithms was an average of 2 minutes per instance. This suggests that LP-based scheduling algorithms can be applied in compilers to greatly improve the quality of the schedules found.

5 Discussion

In this paper we have evaluated the quality of schedules produced by a series of list scheduling algorithms that use optimal single machine schedules and LP relaxations to form the priority lists. While these algorithms have been evaluated in other contexts, this represents the first study of their applicability to the multiple machine WCT scheduling problem with precedence. We show that the quality of the multiple machine schedules is closely related to the quality of the single machine priority list, and that it gets harder to achieve good results as the complexity of the precedence increases. We also show that module-intact schedules for chains can closely approximate optimal solutions.

We can use our results to evaluate the correlation between the number of machines, m, and the difference between the cost of schedules found by the LRF, ρ-max, average start-time, and best-α algorithms and the optimal value or lower bound, i.e. the error. In the limit as m increases, any list scheduling algorithm will find schedules that are close to optimal. When the number of machines is greater than or equal to the maximum number of ready jobs at any time during scheduling, any list scheduling algorithm will find an optimal schedule. We can therefore expect the error in list schedules to decrease as m increases. This can be observed in the results for LP-based algorithms applied to the synthetic benchmark. Furthermore this behavior can also be seen in the results from the compiler benchmark.

However this trend may not apply for smaller values of m. The ρ-max algorithms are optimal for $m = 1$, so the error in schedules found for small values of m may be lower than that of schedules for larger m. These two trends, taken together, suggest that the error in single-machine optimal algorithms must achieve a maximum for some $m \geq 2$. This is confirmed by our results. For small instances, the error in the ρ-max algorithm peaks at $m \geq 6$ without precedence, at $m = 4$, for chains, and at $m = 2$ for trees. For large instances, the error in the ρ-max algorithm peaks at $m \geq 6$. The value of m at which the error is maximized tends to decrease as the complexity of the precedence increases, an effect that is most clearly seen in the data for the small instances. More complex precedence decrease the number of ready jobs at any time during scheduling. As a result fewer machines tend to be needed to achieve a given amount of error. Note that this analysis does not fully account for variations in the tightness of the lower bound used to evaluate the schedules for large instances.

The good results produced by the LP-based algorithm suggest that such approaches may be useful for solving other difficult scheduling problems. Extensive research has been done on a range of LP relaxations, and these formulations may allow even better results to be obtained. We are currently extending this study to include the largest class of precedence constraints, series-parallel graphs, that can be optimally solved on a single processor. We are also investigating various algorithms, including module-intact approaches, that apply to the general case of arbitrary precedence, and algorithms that insert delays to produce better schedules.

References

[1] 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.

[2] J. Bruno, E. G. Coffman Jr, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[3] W. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[4] W. A. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal of Applied Mathematics*, 23:189–202, 1972.

[5] J. B. Sidney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23:283–298, 1975.

[6] D. L. Adolphson. Single machine job sequencing with precedence constraints. *SIAM Journal on Computing*, 6:40–54, 1977.

[7] E. L. Lawler. Sequencing jobs to mimimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[8] H. Belouadah and C. N. Potts. Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics*, 48:201–218, 1994.

[9] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 1994.

[10] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimizing register requirements of a modulo schedule via optimum stage scheduling. *International Journal of Parallel Programming*, 2(2):103–132, 1996.

[11] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996.

[12] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software-Practice and Experience*, 26(8):929–965, 1996.

[13] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *8th International Conference on Compiler Construction (CC'99)*, 1999.

[14] S. Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Trans. on Computers*, 34(11):981–995, 1985.

[15] R. Leupers and P. Marwedel. Time-constrained code compaction for dsp's. *IEEE Trans. on VLSI Systems*, 5(1):112–122, 1997.

[16] U. Kremer. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters*, 1997.

[17] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal insruction scheduling using integer programming. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 121–133, 2000.

[18] T. Kawaguchi and S. Kyan. Worst case bound of an LRF schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15:1119–1129, 1986.

[19] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *Proceedings of the 7th Annual ACM-SIAM Simposium on Discrete Algorithms*, pages 142–151, 1996.

[20] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–549, 1997.

[21] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proceedings of the 8th Annual ACM-SIAM Simposium on Discrete Algorithms*, pages 609–618, 1997.

[22] M. W. P. Savelsbergh, R. N. Uma, and J. Wein. An experimental study of LP-based approximation algorithms for scheduling problems. In *Proc. of the 9th Annual ACM-SIAM Symp. on Disc. Algs.*, pages 453–462, 1998.

[23] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms*. Academic Press, 1975.

[24] M. D. Atkinson and J. R. Sack. Generating binary trees at random. *Information Processing Letters*, 41:21–23, 1992.

[25] J. F. Korsh. Counting and randomly generating binary trees. *Information Processing Letters*, 45:291–294, 1993.

[26] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, pages 229–248, 1993.

[27] C.Chekuri, R. Johnson, B. K. Natarajan, B. R. Rau, and M. Schalnsker. An analysis of profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, 1996.

[28] B. L. Deitrich and W. W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 70–79, 1996.