SCAN: a Heuristic for Near-Optimal Software Pipelining

Florent Blachot[1,2] and Benoît Dupont de Dinechin[2] and Guillaume Huard[1]

[1] ID Laboratory, Grenoble, France[**]
[2] STMicroelectronics, Grenoble, France

Abstract. Software pipelining is a classic compiler optimization that improves the performances of inner loops on instruction-level parallel processors. In the context of embedded computing, applications are compiled prior to manufacturing the system, so it is possible to invest large amounts of time for compiler optimizations.
Traditionally, software pipelining is performed by heuristics such as iterative modulo scheduling. Optimal software pipelining can be formulated as integer linear programs, however these formulations can take exponential time to solve. As a result, the size of loops that can be optimally software pipelined is quite limited.
In this article, we present the SCAN heuristic, which enables to benefit from the integer linear programming formulations of software pipelining even on loops of significant size. The principle of the SCAN heuristic is to iteratively constrain the software pipelining problem until the integer linear programming formulation is solvable in reasonable time.
We applied the SCAN heuristic to a multimedia benchmark for the ST200 VLIW processor. We show that it almost always compute an optimal solution for loops that are intractable by classic integer linear programming approaches. This improves performances by up to 33.3% over the heuristic modulo scheduling of the production ST200 compiler.

1 Introduction

In scientific and multimedia applications, most of the execution time is spent in loops. In case of instruction-level parallel processors such as superscalar and VLIW, instruction scheduling of inner loops can significantly increase performances, in particular with software pipelining [AJLA95]. The mainstream software pipelining technique is called modulo scheduling [RG81,Lam88,Rau94].

Modulo scheduling solves a 1-periodic cyclic scheduling problem with the objective of minimizing the period or *initiation interval* (II). This is achieved by computing first a lower bound $MinII$ on the II. Then cyclic scheduling is attempted for increasing II values starting at $MinII$, until a solution is found. In general it is NP-hard to know what is the minimum II of a modulo scheduling problem, but modulo schedules whose II equals $MinII$ are clearly optimal.

[**] This work has been partially funded by the MOAIS INRIA project

The modulo scheduling problem at a given II can be formulated as an integer programming problem [EDA95,dD05]. The formulation of Eichenberger et al. uses a mix of $\{0, 1\}$ and of integer variables, while the formulation of Dupont-de-Dinechin only uses $\{0, 1\}$ variables (but in a larger number). We implemented the formulation of Dupont-de-Dinechin as it is adapted from the integer linear programming formulation of resource-constrained project scheduling [ADdDA06]. Solving such formulations enables to modulo schedule at the minimum II, an appealing possibility in cases the compilation time is not severely constrained. However, the time required to solve these formulations grows exponentially with the size of the modulo scheduling problem, making classic integer linear programming approaches intractable beyond a few tenths of instructions.

In this article, we propose a new approach for modulo scheduling that enables to benefit from integer linear programming formulations even on problems with a significant number of instructions. The main idea is to restrict the solution space to the areas where the formulation can be solved in reasonable time. Here solved means either computing a solution or proving that no solution exists. For the integer linear programming formulations discussed above, some constants and the number of variables depend on a parameter called the time horizon of the loop. Although a theoretical bound exists for the time horizon, it is usually much higher than required by an optimal solution.

The principle of our approach is to heuristically reduce the time horizon in order to solve the formulation. The issue with this reduction is that it might transform a feasible modulo scheduling problem into an infeasible one. Thus, we have to explore the solution space along two parameters: performance, represented by the II; the time horizon, that must be kept small enough so the formulation can be solved. Our approach takes advantage of an empirical knowledge on the general shape of the solution space, which we deduced from a large set of experiments. We called this approach the SCAN heuristic.

Although reducing the time horizon may eliminate all optimal solutions, our results show that SCAN reaches the optimal performance (II equals $MinII$) for most of the loops of our benchmark, including those that appeared intractable with the original integer linear programming formulations. For the remaining loops, we have no way to know if modulo scheduling with $II = MinII$ is feasible, but the II results of the SCAN heuristic are consistently close to $MinII$ and better than those of the modulo scheduling heuristic of the ST200 production compiler. Overall, our approach results in improvements of up to 33.3% for the most difficult loops as demonstrated by our experiments.

This article is organized as follows. In section 2, we present the modulo scheduling problem and we review the integer linear programming (ILP) formulation currently used by the SCAN heuristic. In section 3, we develop our findings about the time horizon and its relations with the ILP formulation. We characterize the search space of the initiation interval and the time horizon values. Then we present how the SCAN heuristic searches for the best tractable initiation interval by adjusting the time horizon accordingly. Finally, section 4 reports the experimental results of the SCAN heuristic on a multimedia benchmark.

2 The Cyclic Scheduling Problem

Cyclic scheduling, also known as software pipelining in the case of instruction scheduling, is a widely studied problem [AJLA95]. Modulo scheduling [Rau94] is a class of software pipelining techniques that build 1-periodic schedules. This section introduces our notations for modulo scheduling and presents the integer linear programming formulation of modulo scheduling we use.

2.1 Cyclic Scheduling Problem Formulation

Consider the problem of scheduling a loop with a possibly large number of iterations. The loop can be represented by a finite, directed multigraph $G = (I, E_{dep}, \theta, \omega)$. The vertex set I contains the instructions of the loop body and each instruction $I_i \in I$ generates a set of instruction instances $\{I_i^k | k \in \mathbb{N}\}$, one for each iteration of the loop.

The directed edges E_{dep} model the dependence constraints between instructions: each edge $I_i \longrightarrow I_j \in E_{dep}$ is labeled with a pair $(\theta_i^j, \omega_i^j) \in \mathbb{N} \times \mathbb{N}$ where θ_i^j is the dependence latency and ω_i^j the dependence distance. Such dependence expresses the fact that the execution of I_i^k (instance of I_i at iteration k) must start θ_i^j cycles before the execution of $I_j^{k+\omega_i^j}$ (instance of I_j at iteration $k + \omega_i^j$).

Each instruction I_i is also associated with a execution time p_i and a resource requirements vector $\overrightarrow{b_i}$ of size r. The total availability of the processor resources is also given by a vector $\overrightarrow{B}$. Typical resources are issue width, functional units and memory ports. Execution time of fully pipelined instructions is $p_i = 1$.

The cyclic scheduling problem is to determine a schedule $\sigma : I \times \mathbb{N} \rightarrow \mathbb{N}$ for the instruction instances I_i^k that respects the dependence constraints:

$$\forall I_i \longrightarrow I_j \in E_{dep}, \forall k \geq 0 : \sigma_i^k + \theta_i^j \leq \sigma_j^{k+\omega_i^j} \tag{1}$$

and the resource constraints: at any clock cycle, the sum of resources used cannot be greater than the available resources $\overrightarrow{B}$.

Among all cyclic schedules, the 1-periodic schedules are especially interesting in instruction scheduling as they enable simple code generation. Such schedules, also known as modulo schedules, are defined by:

$$\exists II \in \mathbb{N}, \forall I_i \in I, \forall k \in \mathbb{N} : \sigma_i^k = \sigma_i^0 + k \times II \tag{2}$$

The period of the schedule, usually called the initiation interval in the literature, is denoted II. This is the performance metric of modulo schedules: the lower the initiation interval, the greater the execution throughput.

The initiation interval of any modulo schedule is limited by a lower bound $MinII$ defined as $\max(MIIRec, MIIRes)$ [Rau94], where $MIIRec$ (recurrence minimum initiation interval) is related to dependence circuits and $MIIRes$ (resource minimum initiation interval) is related to resource uses:

$$MIIRec \stackrel{def}{=} \max_{C\ circuit\ in\ G} \left\lceil \frac{\sum\limits_{I_i \to I_j \in C} \theta_i^j}{\sum\limits_{I_i \to I_j \in C} \omega_i^j} \right\rceil \tag{3}$$

$$MIIRes \stackrel{def}{=} \max_{r} \left\lceil \frac{\sum\limits_{I_i \in I} p_i b_i^r}{B^r} \right\rceil \tag{4}$$

The *time horizon* is defined as the maximum number of cycles between the execution of two instruction instances of the same iteration:

$$H = \max_{I_i, I_j \in I} (\sigma(I_i, k) - \sigma(I_j, k)) \forall k \in \mathbb{N} \tag{5}$$

2.2 Modulo Scheduling by Integer Linear Programming

Integer linear programming (ILP) is a well known technique to formulate and solve combinatorial problems such as scheduling and routing. Several ILP formulations have been proposed for the modulo scheduling problem [EDA95,ED97]. Based on the efficient ILP formulations used in resource-constrained project scheduling, B. Dupont-de-Dinechin recently introduced another formulation for modulo scheduling [dD05], which we use for the SCAN heuristic. Compared to the latest formulation of Eichenberger et al. [ED97], this new formulation only uses $\{0, 1\}$ variables and has stronger linear programming relaxations [ADdDA06].

This new ILP formulation can be summarized as follows. For the sake of simplicity, we removed the objective function, the equations related to registers pressure and we assume that instructions have unit execution time, which is the case for our target processor. The complete formulation is available in [dD05].

$$\sum_{t=0}^{H-1} x_i^t = 1 \quad \forall i \in [1, n] \tag{6}$$

$$\sum_{s=t}^{H-1} x_i^s + \sum_{s=0}^{t+\theta_i^j - II\omega_i^j - 1} x_j^s \leq 1 \quad \forall t \in [0, H-1], \forall (i,j) \in E_{dep} \tag{7}$$

$$\sum_{i=1}^{n} \sum_{k=0}^{\lfloor \frac{H-1}{II} \rfloor} x_i^{t+k\times II} \overrightarrow{b_i} \leq \overrightarrow{B} \quad \forall t \in [0, II-1] \tag{8}$$

$$x_i^t \in \{0, 1\} \quad \forall i \in [1, n], \forall t \in [0, H-1] \tag{9}$$

Let n denote number of instructions. Each x_i^t is a $\{0, 1\}$ variable that is 1 if the instruction I_i is scheduled at time t, else it is 0. In this formulation, the equations correspond to: unique scheduling dates (6), dependence constraints (7) and resource constraints (8).

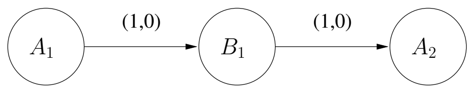
Any solution of this ILP formulation yields a valid modulo schedule at initiation interval II whose time horizon is at most H. By searching iteratively for the minimum II and with a large enough H, we eventually find an optimal solution of the modulo scheduling problem. Unfortunately, the resolution time of such integer linear program grows exponentially with the number of instructions.

3 The SCAN Heuristic

The ILP formulation introduced in section 2.2 depends on two parameters: the initiation interval II, which we want to minimize; the time horizon H, which bounds the span of the schedule of a given loop iteration. In this section, we describe how the SCAN heuristic drastically reduces the time to solve the the integer linear programs, based on a characterization of the solution space on the parameters H and II. It relies on the observation that, usually, a modulo schedule exists at a given II with a small time horizon.

3.1 The Search for the Time Horizon

The time horizon H of a solution to a modulo scheduling problem instance at a given II is not known in advance. A trivial lower bound is deduced from the longest path in the dependence graph. An upper bound is given in [ES96], which roughly equals the number of instructions times the sum of the initiation interval and the maximal dependence distance: $O(n \times (II + max(\omega_i^j)))$.



(a) Simple instructions graph

resources/cycle	1	2	3	4	5	6	7	8	9	10
A	$A_{1,1}$		$A_{1,2}$	$A_{2,1}$	$A_{1,3}$	$A_{2,2}$	$A_{1,4}$	$A_{2,3}$	$A_{1,5}$	$A_{2,4}$
B		$B_{1,1}$		$B_{1,2}$		$B_{1,3}$		$B_{1,4}$		$B_{1,5}$

(b) cyclic schedule of initiation interval of 2 and time horizon of 4

resources/cycle	1	2	3	4	5	6	7	8	9	10
A	$A_{1,1}$		$A_{1,2}$		$A_{1,3}$	$A_{2,1}$	$A_{1,4}$	$A_{2,2}$	$A_{1,5}$	$A_{2,3}$
B		$B_{1,1}$		$B_{1,2}$		$B_{1,3}$		$B_{1,4}$		$B_{1,5}$

(c) cyclic schedule of initiation interval of 2 and time horizon of 6

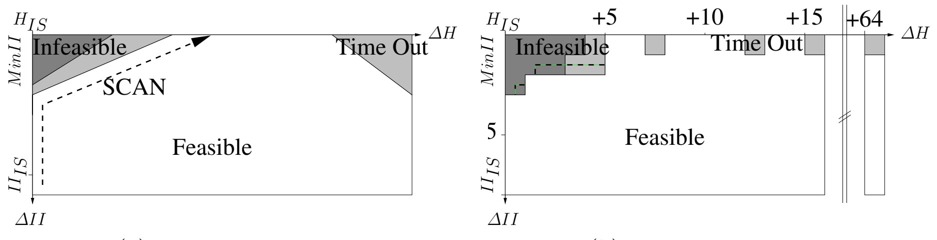
Fig. 1. A simple instance of software pipelining. Graph of three instructions (1(a)) and two cyclic schedules (1(b)) and (1(c)) with time horizon 4 and 6.

A main issue is that the number of variables of the ILP formulation of section 2.2 directly depends on the value of the time horizon. Given that in most cases the time horizon of the optimal solution is close to its lower bound [ES96], the idea of using the ILP formulation with a small time horizon is natural. The difficult part is to determine which value of H is sufficient to keep the problem solvable at a given II. Unfortunately, this value is highly dependent on the interference between dependence and resource constraints.

Possible candidate values are difficult to guess as illustrated in figure 1: in this example, three instructions form a dependence chain and require two type of resources (type A for instructions A_1 and A_2 and type B for instruction B). Each resource type is limited to one instruction at a time. Because of this limitation and of the dependences, an optimal initiation interval of 2 (the resources lower bound) is achievable with a time horizon of 4 and 6 but not 5.

3.2 Characterization of the Search Space

An inappropriate choice for the value of H makes the integer linear program either infeasible or intractable. As appropriate choices of H for a given II are difficult to guess, we conducted experiments on all our benchmark loops to determine the shape of this search space. We tried all the possible (H, II) values with II ranging from the lower bound of the problem to the II found by a modulo scheduling heuristic and H ranging from the lower bound to the upper bound. For each possible couple of values, we reported if the problem was infeasible, feasible or reached a given timeout (in which case we consider it as intractable).



(a) for the general case (b) for a loop of `transfo`

Fig. 2. Characterization of the search space and the SCAN heuristic

For all the loops, the observed search space has the same general shape: a timeout area for large H values, an infeasible area for small values of both II and H and a timeout area between the infeasible area and the rest which is feasible. This characterization is depicted in figure 2. For some loops, the infeasibility area reduces to an empty part (along with the timeout border) or the slope of the separating line might change (at worst this is an horizontal line: at some point we are not able to find a lower II whatever the H value). This leads to:

– infeasibility usually results from a too small value for H, in this case the solution is simply to increase it.

– intractability is more difficult to handle. It might result from a choice of H which is either too small, in the timeout border on the side of the unfeasible area, or too large, in the large timeout area.

Because of the non-predictable location of the timeout points, it is not possible to use a dichotomy for a given II value to find the appropriate H value. As it will be shown in the section 4, the value chosen for the timeout does not change the shape of the solution space. By choosing a smaller timeout value, the feasible area is just smaller and included in the area found with a larger value.

3.3 The SCAN Heuristic

The main idea of the SCAN heuristic is to change the values for II and H in order to progress along the line that separates the unfeasible area from the feasible one. This enables in most cases to reach the part with the lowest initiation interval while remaining within the feasible area. The algorithm is illustrated by figure 2 and can be described as follows:

1. start from II and H found by a classical modulo scheduling heuristic
2. solve the linear program, there are two cases:
   (a) if the program is unfeasible or stopped by the timeout, $H = H + 1$
   (b) if the program is feasible, $II = II - 1$
3. repeat step 2 until a global timeout or reaches $MinII$
4. return the best solution (lowest H in the set of lowest II feasible solutions)

4 Experimental Results

We performed experiments using our implementation of the SCAN heuristic integrated in the production compiler developed by STMicroelectronics for the ST200 processor. The ST200 is a VLIW processor that executes up to 4 instructions by cycle and has clean pipelines (instructions can be viewed as unit execution time with a latency of either 1 or 3 towards dependent instructions). The optimal scheduler used by the SCAN heuristic is linked with the CPLEX9.0 solver from ILOG for the ILP resolution. The compilation is performed on a cross-compiler running on a Pentium 4 1.8 GHz system with 1 Gb of RAM.

4.1 Space Characterization

The test suite is the multimedia benchmark used internally by STMicroelectronics for its compiler performance validation process. It contains 169 inner loops taken from speech coding, audio and video applications. These loops vary at the structural level (from sequential to highly parallel) as well as in the number of instructions of their body (from 12 to 114 instructions).

We performed an exhaustive search on all the loops of our benchmark to characterize the (H, II) space described in section 3.2. For this search, we used a timeout value of 3000 seconds for each point. Overall, the computation of these results ran for almost two weeks but validated our characterization.

Figure 3 shows the solution space for a difficult loop `dbuffer` with three different timeout values. We notice on this example that the slope of the timeout area frontier is almost horizontal and that the two timeout areas are connected. But it still conforms to our characterization whatever the timeout value.

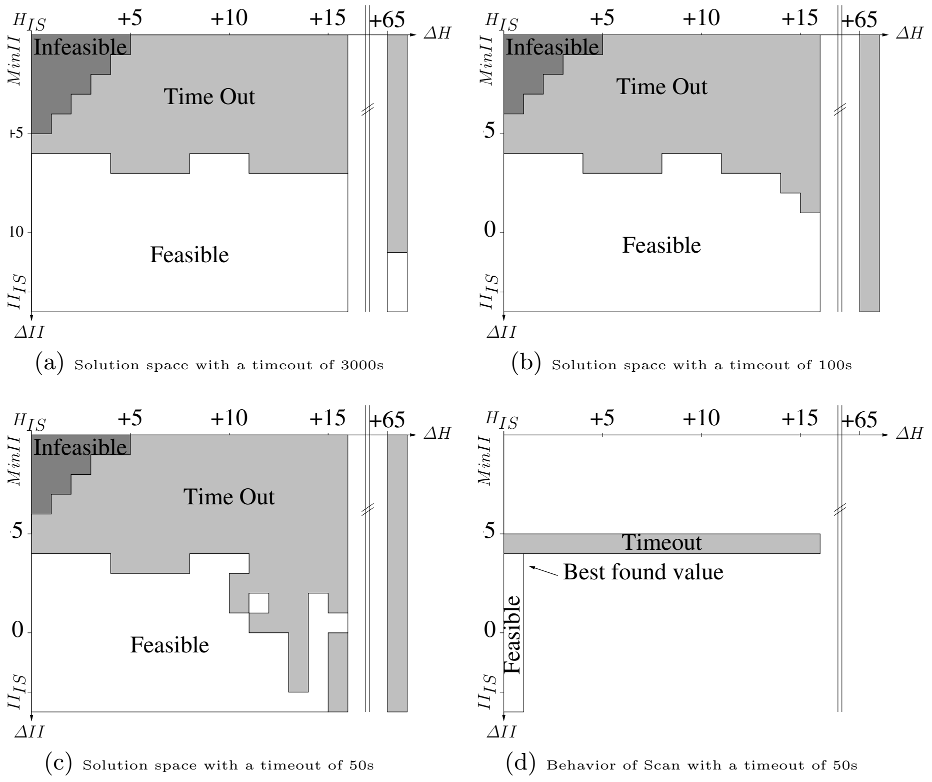


Fig. 3. Different solution spaces for a loop of *dbuffer* and behavior of SCAN

4.2 Performance of the SCAN Heuristic

The ST200 production compiler integrates a heuristic modulo scheduler that schedules 65.4% of the loops at II equal to $MinII$. Precisely, among the 169 inner loops of our benchmark, 108 are scheduled optimally by the heuristic modulo scheduler (106 at the lower bound and 2 proved unfeasible at a lower initiation interval using an exact resolution). Thus 61 loops could be possibly improved after heuristic modulo scheduling. For this 61 loops, an exhaustive search on the initiation interval and horizon was done with a timeout of 3000s. A better solution was found for 47 loops, the other being unsolvable in 3000s. With a timeout of 75s by point, the SCAN heuristic is able to find all these better solutions; thus the SCAN heuristic finds the better known solutions of our benchmark.

The improvements of the SCAN heuristic over the heuristic modulo scheduler can be significant: for 8 loops the gain is close to 20% and maximum is 33.3%. Among the noticeable results, the main loop of the `fft32x32s` 32-bit fractional radix-4 Fourier transform, which appeared intractable using the ILP formulation because of its 83 instructions, has been improved by 21.4%.

timeout	SCAN vs. HMS	MinII vs. SCAN	time for scan	MinII vs. ILP	time for ILP
5s	4.10%	1.39%	3.36s	2.86%	5.48s
10s	4.21%	1.28%	5.95s	2.78%	8.86s
25s	4.28%	1.20%	9.19s	2.66%	19.75s
75s	4.29%	1.19%	22.10s	2.21%	52.57s
500s	4.29%	1.19%	98.34s	1.59%	277.40s

Table 1. Improvements of the SCAN heuristic for different timeout values.

Table 1 illustrates how the SCAN heuristic improves on average the 169 loops of our benchmark for the different values of the timeout listed in the first column. The second column contains the average II improvements of the SCAN heuristic over the production heuristic modulo scheduler (HMS). The third column contains the average II increase of the SCAN heuristic over $MinII$ and the average time spent per loop in the SCAN heuristic. The fourth column contains the average II increase of the ILP modulo scheduler over $MinII$ and the average time spent per loop in the ILP modulo scheduler. From these figures, the SCAN heuristic appears quite effective even at low timeout values.

5 Conclusions

We presented a heuristic that takes advantage of integer linear programming formulations of modulo scheduling. Such formulations when solved yield optimal software pipelines, but resolution times are worst case exponential. In practice, only loops that comprise less than a few tenths of instructions can benefit from integer linear programming formulations of modulo scheduling.

The SCAN heuristic we propose makes integer linear programming formulations of modulo scheduling applicable to significantly larger loops, by walking on the boundaries of the practically solvable solution space. The solution space we consider is bi-dimensional, one dimension being the software pipeline period II and the other a heuristic restriction on the schedule time horizon H. The SCAN heuristic takes advantage of an empirical characterization of the search space and evolves these parameters towards close to optimal solutions.

We implemented the SCAN heuristic and an integer linear programming formulations of modulo scheduling in the STMicroelectronics production compiler for the ST200 VLIW processor. The experiments we conducted show that our space characterization holds for all of the considered loops. Furthermore, the use of the SCAN heuristic on the difficult loops of a multimedia benchmark produced results up to 33.3% better than the heuristic modulo scheduler of the production compiler. The performance and flexibility of the SCAN heuristic make it perfectly suitable for production use in embedded code compilation.

References

ADdDA06.	S. Azem, B. Dupont de Dinechin, and C. Artigues. Résolution d'un problème d'ordonnancement modulo sur une architecture vliw par la programmation linéaire en nombre entiers. In *ROADEF'2006: actes du 7ème congrès de la Société Francaise de Recherche Opérationnelle et d'Aide à la Décision*, Lille, 2006.
AJLA95.	Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.
dD05.	Benoît Dupont de Dinechin. From machine scheduling to vliw instruction scheduling. *ST Journal of Research*, 1(2), 2005.
ED97.	Alexandre E. Eichenberger and Edward S. Davidson. Efficient formulation for optimal modulo schedulers. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 194–205. ACM Press, 1997.
EDA95.	Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum modulo schedules for minimum register requirements. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 31–40. ACM Press, 1995.
ES96.	Christine Eisenbeis and Antoine Sawaya. Optimal loop parallelization under register constraints. Technical Report RR-2781, INRIA, 1996.
Lam88.	M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328. ACM Press, 1988.
Rau94.	B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM Press, 1994.
RG81.	B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198. IEEE Press, 1981.