From Machine Scheduling to VLIW Instruction Scheduling

Benoît Dupont de Dinechin

Updated February 2006

Abstract

We discuss the similarities and the differences between machine scheduling problems and instruction scheduling problems on modern VLIW architectures such as the STMicroelectronics ST200. Our motivations are to apply the machine scheduling techniques that are relevant to instruction scheduling in VLIW compilers and to understand how processor micro-architecture features impact advanced instruction scheduling techniques. Based on this discussion, we present our contributions to the field of instruction scheduling, which are fully implemented in the STMicroelectronics ST200 production compiler. We also introduce a new time-indexed formulation to optimally solve the register constrained instruction scheduling problem.

Introduction

Instruction scheduling is a performance critical optimization which takes place when compiling applications for instruction-level parallel processors such as VLIW processors. An instruction scheduling problem is defined by a set of operations[1] to schedule, a set of dependences between these operations and a target processor micro-architecture. The objective of instruction scheduling is to build an operation sequence of minimum length (block instruction scheduling) or of maximum throughput (software pipeline instruction scheduling) on the target processor.

Although instruction scheduling is a mature discipline, its relationship with the field of machine scheduling is often overlooked. This can be explained because the classic results of machine scheduling apply to problems that are too limited to be of practical use in instruction scheduling. For example, these results assume simple precedence constraints on the order of operations in a schedule, instead of precedences with time-lags like those in pipelined processors, and focus on machine models where each operation uses one of m identical processors for its execution.

In this report, we compare the fields of machine scheduling and of instruction scheduling in order to extend the machine scheduling techniques to instruction scheduling on VLIW architectures such as the STMicroelectronics ST200[2]. This report contains three parts.

In the first part, we summarize the main results and the recent advances in machine scheduling that are relevant to VLIW instruction scheduling. In the second part, we introduce the VLIW instruction scheduling problems with regards to the resource-constrained project scheduling problems (RCPSP) and we survey the field of instruction scheduling. In the third part, we present our contributions to VLIW instruction scheduling that have been implemented in several production compilers, in particular the STMicroelectronics ST200 production compiler. We also introduce a new time-indexed formulation for the register-constrained instruction scheduling problems (RCISP), whose resolution in case of large-scale problems is an ongoing research topic.

[1] An operation is an instance of an instruction in a program text. An instruction is a member of the processor ISA.

[2] The ST200 is based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

1 Advances in Machine Scheduling

1.1 Parallel Machine Scheduling Problems

In parallel machine scheduling problems, an operation set $\{O_i\}_{1 \le i \le n}$ is processed on m identical processors. To be processed, each operation O_i requires the exclusive use of one of the m processors during p_i time units, starting at its *schedule date* σ_i. This processing environment is called *non-preemptive*. In the *preemptive* case, the processing of any operation can be stopped to be resumed later, possibly on another processor. In parallel machine scheduling, the *resource constraints* are the limit m on the maximum number of operations that are being processed at any time.

Parallel machine scheduling problems also involve *temporal constraints* on the schedule dates. In case of *release dates* r_i, the schedule date of operation O_i is constrained by $\sigma_i \ge r_i$. In case of *due dates* d_i, there is a penalty whenever $C_i > d_i$, with C_i the *completion date* of O_i defined as $C_i \stackrel{\text{def}}{=} \sigma_i + p_i$. For problems where $C_i \le d_i$ is mandatory, the d_i are called *deadlines*. Temporal constraints may also include *precedence constraints*, given by a partial order between the operations. A precedence constraint $O_i \prec O_j$ requires O_i to complete before O_j starts, that is, $\sigma_i + p_i \le \sigma_j$. In case of *time-lags* or *precedence delays* l_i^j, the precedence constraint becomes $\sigma_i + p_i + l_i^j \le \sigma_j$.

A *feasible schedule* is a mapping from operations to schedule dates $\{\sigma_i\}_{1 \le i \le n}$, such that both the resource constraints and the temporal constraints of the scheduling problem are satisfied.

Machine scheduling problems are denoted by a triplet notation $\alpha|\beta|\gamma$ [33], where α describes the processing environment, β specifies the operation properties and γ defines the optimality criterion. For the parallel machine scheduling problems, the common values of α, β, γ are:

α : 1 for a single processor, P for parallel processors, Pm for the given m parallel processors.

β : r_i for release dates, d_i for deadlines (if $\gamma = \bullet$) or due dates. Additional β fields include:

- $prec(l_i^j)$ Precedences which have time-lags l_i^j.
- $prec(l_i^j = l)$ All the l_i^j have the same value l (applies to pipelined processing).
- $inTree$ The precedence graph is an in-tree (each operation has one successor).
- $outTree$ The precedence graph is an out-tree (each operation has one predecessor).
- $chains$ The precedence graph is made of independent chains.
- $intOrder(mono\ l_i^j)$ The dependence graph is a monotone interval order (see § 1.4).
- $p_i = 1$ All the processing times are 1 (*unit execution time* or UET operations).
- $p_i = p$ All the processing times have the same value p.

γ : $\bullet$ for the scheduling problem feasibility, C_{max} or L_{max} for the minimization of these objectives. $C_{max} \stackrel{\text{def}}{=} \max_i C_i$ is the *makespan* and $L_{max} \stackrel{\text{def}}{=} \max_i L_i : L_i \stackrel{\text{def}}{=} C_i - d_i$ is the *maximum lateness*.

A $\gamma = L_{max}$ scheduling problem is more general than a $\gamma = C_{max}$ scheduling problem, as $d_i \stackrel{\text{def}}{=} 0$ reduces L_{max} to C_{max}. A $\gamma = L_{max}$ scheduling problem without release dates is also equivalent to a $\gamma = C_{max}$ scheduling problem with release dates r_i, by inverting the precedence graph and by taking $r_i \stackrel{\text{def}}{=} \max_j d_j - d_i$. Finally, solving a $\gamma = \bullet$ scheduling problem with deadlines solves the corresponding $\gamma = L_{max}$ scheduling problem with due dates d_i, by dichotomy search of the smallest L_{max} such that the $\gamma = \bullet$ scheduling problem with deadlines $d_i + L_{max}$ is feasible.

Some significant complexity results of parallel machine scheduling are summarized in Table 1 for the polynomial-time solvable problems and in Table 2 for the NP-hard problems. These tables

Problem	Result	Source
$1\|\|L_{max}$	Jackson 1955	[24]
$1\|prec; r_i\|C_{max}$	Lawler 1973	[33]
$1\|prec\|L_{max}$	Lawler 1973	[33]
$1\|r_i; p_i = 1\|L_{max}$	Baker et al. 1974	[24]
$1\|prec; r_i; p_i = p\|L_{max}$	Simons 1978	[33]
$1\|prec(l_i^j = 1); r_i; p_i = 1\|L_{max}$	Bruno et al. 1980	[49]
$1\|prec(l_i^j = 1)\|C_{max}$	Finta & Liu 1996	[49]
$1\|prec(l_i^j \in \{0,1\})\|C_{max}$	Leung et al. 2001	[49]
$1\|prec(l_i^j \in \{0,1\}); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	[49]
$P2\|prec; p_i = 1\|C_{max}$	Coffman & Graham 1972	[24]
$P2\|prec; p_i = 1\|L_{max}$	Garey & Johnson 1976	[24]
$P2\|prec; r_i; p_i = 1\|L_{max}$	Garey & Johnson 1977	[24]
$P2\|prec(l_i^j \in \{-1,0\}); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	[49]
$P\|tree; p_i = p\|C_{max}$	Hu 1961	[24]
$P\|r_i; p_i = 1\|L_{max}$	Blazewicz 1977	[24]
$P\|outTree; r_i; p_i = p\|C_{max}$	Brucker et al. 1977	[24]
$P\|inTree; p_i = p\|L_{max}$	Brucker et al. 1977	[24]
$P\|inTree(l_i^j = l); p_i = 1\|L_{max}$	Bruno et al. 1980	[49]
$P\|outTree(l_i^j = l); r_i; p_i = 1\|C_{max}$	Bruno et al. 1980	[49]
$P\|intOrder(mono\ l_i^j); p_i = 1\|L_{max}$	Palem & Simons 1993	[49]
$P\|intOrder(mono\ l_i^j); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	[49]

Table 1: Polynomial-time solvable parallel machine scheduling problems.

illustrate the advances achieved in parallel machine scheduling, especially the recent discovery of several polynomial-time solvable (PTS) machine scheduling problems. Although these new PTS parallel machine scheduling problems include precedence constraints, they are restricted to either constant processing time ($p_i = p$), or to UET ($p_i = 1$).

1.2 Parallel Machine Scheduling Extensions and Relaxations

A first class of extensions of parallel machine scheduling problems is the replacement of the precedence graph with time-lags by a *dependence graph*. Unlike the precedence graph, which represents a partial order, a dependence graph may include circuits. Each dependence arc represents the temporal constraint $\sigma_i + \theta_i^j \leq \sigma_j$, with the *dependence latency* θ_i^j unrelated to p_i or p_j and of arbitrary sign. In dependence graphs, it is often convenient to introduce the dummy start and stop operations O_0 and O_{n+1}. In particular, the release dates r_i can be expressed as dependences between O_0 and O_i with the latencies $\theta_0^i \stackrel{\text{def}}{=} r_i$. The deadlines d_i can also be expressed as dependences between O_i and O_0, by using the (negative) latencies $\theta_i^0 \stackrel{\text{def}}{=} p_i - d_i$. The dummy stop operation O_{n+1} is used for C_{max} minimization by using dependences between O_i and O_{n+1} of latency $\theta_i^{n+1} \stackrel{\text{def}}{=} p_i$.

A second class of extensions of parallel machine scheduling problems is to consider *multi-processor tasks*. In this extension, denoted by $size_i$ in the β field, an operation O_i requires $size_i$ processors for p_i time units to execute. Multi-processor scheduling is NP-hard even for the basic UET problem $P|p_i = 1; size_i|C_{max}$ [8]. Two-processor multi-processor scheduling is more tractable, as $P2|prec; p_i = p; size_i|C_{max}$ and $P2|r_i; p_i = 1; size_i|L_{max}$ are polynomial-time solvable [8].

Problem	Result	Source
$1\|r_i\|L_{max}$	Lenstra et al. 1977	[24]
$1\|prec(l_i^j); p_i = 1\|C_{max}$	Hennessy & Gross 1983	[49]
$1\|inTree(l_i^j = l); r_i; p_i = 1\|C_{max}$	Brucker & Knust 1998	[24]
$1\|outTree(l_i^j = l); p_i = 1\|L_{max}$	Brucker & Knust 1998	[24]
$P2\|\|C_{max}$	Karp 1972	[24]
$P2\|prec; p_i \in \{1,2\}\|C_{max}$	Ullman 1975	[49]
$P2\|chains\|C_{max}$	Du et al. 1991	
$P\|prec; p_i = 1\|C_{max}$	Ullman 1975	[24]
$P\|inTree; r_i; p_i = 1\|C_{max}$	Brucker et al. 1977	[24]
$P\|outTree; p_i = 1\|L_{max}$	Brucker et al. 1977	[24]
$P\|\|C_{max}$	Garey & Johnson 1978	[24]

Table 2: NP-hard parallel machine scheduling problems.

The multi-processor scheduling model is further generalized in the *resource-constrained project scheduling problems* (RCPSP) [7], where the m processors are replaced by a set of *renewable resources* or *cumulative resources*, whose availabilities are given by an integral vector $\vec{B}$. Each operation O_i is also associated with an integral vector $\vec{b_i}$ of resource requirements and the resource constraints become: $\forall t, \sum_{i \in I_t} \vec{b_i} \leq \vec{B}$, with $I_t \stackrel{\text{def}}{=} \{i \in [1,n] : \sigma_i \leq t < \sigma_i + p_i\}$. That is, the cumulative use of the resources by all the operations executing at any given time must not be greater than $\vec{B}$.

A *relaxation* is a simplified version of a given scheduling problem, such that if the relaxation is infeasible, then the original problem is infeasible too. Relaxations are obtained by allowing preemption, by assuming a particular structure of the precedence constraints, by ignoring some or all resource constraints, or by removing the integrity constraints in problem formulations based on integer linear programming (see §1.5). Relaxations are mainly used to prune the search space of schedule construction procedures, by detecting infeasible problems early and by providing bounds on the objective value and the schedule dates of the feasible problems.

A widely used relaxation of machine scheduling problems is to ignore the resource constraints, yielding a so-called *central scheduling problem* that is completely specified by the dependence graph of the original problem. Optimal scheduling of a central scheduling problem for C_{max} or L_{max} takes $O(ne)$ time where e is the number of dependences [3], as it reduces to a single-source longest path computation from operation O_0. The schedule dates computed this way are the *earliest start* dates of the operations. By assuming an upper bound D on the schedule length and by computing the backward longest paths from operation O_{n+1}, one similarly obtains the *latest start* dates.

1.3 List Scheduling Algorithms and List Scheduling Priorities

The *list scheduling algorithms* (LSA) are the workhorses of machine scheduling [46], as they are able to construct feasible schedules with low computational effort for the RCPSP with general temporal constraints, as long as the dependence graph has no circuits[3]. Precisely, an LSA is a greedy scheduling heuristic where the operations are ranked by priority order in a list. Greedy scheduling means that the processors or resources are kept busy as long as there are available operations to schedule. Two variants of list scheduling must be distinguished [55]:

[3] In case of dependence graphs with circuits, feasibility of the scheduling problem with resources is *NP*-complete.

Problem	Result	Performance Ratio
$P\|prec\|C_{max}$	Graham 1966	$2 - \frac{1}{m}$
$P\|prec(l_i^j \in [0, l]); p_i = 1\|C_{max}$	Palem & Simons 1993	$2 - \frac{1}{m(l+1)}$
$1\|prec(l_i^j)\|C_{max}$	Munier et al. 1998	$\min(2 - \frac{1}{1+\rho}, 1 + \frac{\rho}{2})$
$P\|prec(l_i^j)\|C_{max}$	Munier et al. 1998	$2 - \frac{1}{m(1+\rho)}$

With $\rho \stackrel{\text{def}}{=} \frac{\max l_j^k}{\min p_i}$.

Table 3: Performance guarantees of the GLSA with arbitrary priority.

Graham List Scheduling (parallel schedule generation scheme [46]) Scheduling is performed by scanning the time slots in increasing order. For each time slot, if a processor is idle, schedule the highest priority operation available at this time.

Job-Based List Scheduling (serial schedule generation scheme [46]) Scheduling is performed by scanning the priority list in decreasing order. For each operation of the list, schedule it at the earliest time slot available. This requires that the priority list order be compatible with the operation precedences.

In general, the Graham list scheduling algorithms (GLSA) and the job-based list scheduling algorithms are incomparable with respect to the quality of the schedules they build [45]:

- The job-based LSA generates *active schedules*, where none of the operations can be started earlier without delaying some other operation.
- The Graham LSA constructs *non-delay schedules*, where, even if operation preemption is allowed, none of the operations can be started earlier without delaying some other operation.
- The set of non-delay schedules is a subset of the set of active schedules. It has the drawback that it might not contain an optimal schedule with a regular performance measure[4].

In the important case of UET operations, the set of active schedules equals the set of the non-delay schedules. Moreover, given the same priorities, the two LSA construct the same schedule.

The main results available on list scheduling are the performance guarantees of the GLSA for the C_{max} or L_{max} objectives [55], that is, the ratio between the worst case values of C_{max} or L_{max} and their optimal value. The performance guarantees of the GLSA, irrespective of the choice of the priority function, appear in Table 3. It is interesting to note that the performance guarantee of Graham 1966 does not apply to instruction scheduling problems, as these problems have non-zero time-lags l_i^j (see §2.2). Rather, the bound of Munier et al. 1998 should be considered and is valid only in cases of a GLSA applied to a parallel machine scheduling problem (not to a RCPSP).

The performance guarantees of the GLSA for a specific priority function are given in Table 4. We observe that all the priority functions that ensure optimal scheduling by the GLSA are either the earliest deadlines first, also known as *Jackson's rule*, or the earlier modified deadlines d'_i first. A modified deadline for operation O_i is a date d'_i such that for any feasible schedule $\sigma_i + p_i \leq d'_i$. For example, in case of Hu's algorithm, since the precedences are an in-tree and given the UET operations, the highest in-tree level first is also a modified deadlines priority.

[4]An optimality criterion γ such that $\gamma(C_1, \ldots, C_{n+1}) < \gamma(C'_1, \ldots, C'_{n+1}) \Rightarrow \exists j : C_j < C'_j$ (like C_{max} or L_{max}).

Problem	Result	Performance Ratio	Priority
$1\|\|L_{max}$	Jackson 1955	1 (optimal)	earliest d_i first
$P\|inTree; p_i = 1\|C_{max}$	Hu 1961	1 (optimal)	highest levels first
$P\|\|C_{max}$	Graham 1969	$\frac{4}{3} - \frac{1}{3m}$	non-increasing p_i
$1\|r_i; p_i = 1\|L_{max}$	Baker et al. 1974	1 (optimal)	earliest d_i first
$P\|\|C_{max}$	1976*	$1 + \frac{1}{k} - \frac{1}{km}$	non-increasing p_i
$P2\|prec; p_i = 1\|L_{max}$	Garey et al. 1976	1 (optimal)	earliest d'_i first
$1\|prec; p_i = 1\|L_{max}$	Garey et al 1981	1 (optimal)	earliest d'_i first
$P\|r_i; p_i = 1\|L_{max}$	Blazewicz 1977	1 (optimal)	earliest d_i first
$1\|prec(l_i^j \in \{0, 1\}); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	1 (optimal)	earliest d'_i first
$P2\|prec(l_i^j \in \{-1, 0\}); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	1 (optimal)	earliest d'_i first
$P\|intOrder(mono\ l_i^j); r_i; p_i = 1\|L_{max}$	Leung et al. 2001	1 (optimal)	earliest d'_i first
$P\|inTree(l_i^j = l); p_i = 1\|L_{max}$	Leung et al. 2001	1 (optimal)	earliest d'_i first

*With k the number of operations executed on the last active processor in the schedule.

Table 4: Performance guarantees of the GLSA with a specific priority.

$1\|prec(l_i^j \in \{0, 1\}); r_i; d_i; p_i = 1\|\bullet$	$P2\|prec(l_i^j \in \{-1, 0\}); r_i; d_i; p_i = 1\|\bullet$
$P\|intOrder(mono\ l_i^j); r_i; d_i; p_i = 1\|\bullet$	$P\|inTree(l_i^j = l); d_i; p_i = 1\|\bullet$

Table 5: Problems solved by the algorithm of Leung, Palem and Pnueli [49] steps 1 and 2.

1.4 The Scheduling Algorithm of Leung, Palem and Pnueli

Leung, Palem and Pnueli proposed in 2001 the following algorithm for scheduling UET operations on a parallel machine [49]:

1. For each operation O_i, compute an upper bound d'_i, called "modified deadline", such that for any feasible schedule $\sigma_i < d'_i \leq d_i$.
2. Schedule with the GLSA using the earliest d'_i first as priorities, then check that the resulting schedule does not miss any deadlines.
3. In case of minimizing L_{max}, binary search to find the minimum scheduling horizon such that the scheduling problem is feasible.

The steps 1 and 2 of this algorithm solve the problems of Table 5 in $O(n^2 \log n\alpha(n) + ne)$ time.

As introduced by Papadimitriou & Yannakakis [60], an *interval-order* is defined by an incomparability graph that is chordal. An interval-order is also the complement of an interval graph [60]. Thus, in an interval order graph, each node can be associated to a closed interval on the real line and there is an arc between two nodes only if the two intervals do not intersect.

A *monotone interval-order* graph is an interval-order graph (V, E) with a weight function w on the arcs such that, given any $(v_i, v_j), (v_i, v_k) \in E : w(v_i, v_j) \leq w(v_i, v_k)$ whenever the predecessors of vertex v_j are included in the predecessors of vertex v_k.

In order to compute the modified deadlines in step 1, Leung et al. apply a technique of *optimal backward scheduling* [59], where a series of relaxations are optimally solved by the GLSA in order to find, for each operation, its latest schedule date such that the relaxations are feasible. Precisely, the implementation of optimal backward scheduling is as follows:

- Iterate in backward topological order on the operation set to build a series of scheduling sub-problems $S_i \stackrel{\text{def}}{=} \{O_i, succ_i \cup indep_i, \{r'_j\}, \{d'_j\}\}$. Here $succ_i$ is the set of successors of O_i, $indep_i$ is the set of operations that are independent from O_i, $\{r'_j\}$ are the dependence-consistent release dates and $\{d'_j\}$ are the current modified deadlines.

- For each scheduling sub-problem S_i, perform a binary search for the latest schedule date p of O_i such that the constrained sub-problem $(r'_i = p) \wedge S_i$ is feasible. If there is a solution for such p, define the modified deadline of O_i as $d'_i \stackrel{\text{def}}{=} p + 1$. Else the original scheduling problem is infeasible.

- To find if a constrained sub-problem $(r'_i = p) \wedge S_i$ is feasible, convert the transitive dependence latencies from O_i to all the other O_j of S_i into release dates; then, forget the dependences. This yields a relaxation, which is the simpler scheduling problem $P|r_i; d_i; p_i = 1|\bullet$.

- Optimally solve this $P|r_i; d_i; p_i = 1|\bullet$ relaxation using the GLSA with the earliest d_i first priority (Jackson's rule). This gives the feasibility status of the relaxation.

Thanks to a fast list scheduling implementation based on union-find data-structures, solving a $P|r_i; d_i; p_i = 1|\bullet$ relaxation takes $O(n\alpha(n))$ time. Here $\alpha(n)$ is the inverse Ackermann function and e is the number of dependence arcs. The binary search introduces a $\log n$ factor and there are n scheduling sub-problems S_i to consider, each one implying a $O(n + e)$ transitive latency computation. Thus, the total time complexity of backward scheduling is $O(n^2 \log n\alpha(n) + ne)$. Because the final GLSA has a time complexity of $O(n^2)$, the full algorithm takes $O(n^2 \log n\alpha(n) + ne)$ time either to build a schedule or to report that the original scheduling problem is infeasible.

The main theoretical contributions of this work are the proofs that the feasible schedules computed this way are in fact optimal for all the cases listed in Table 5 and also the unification of many earlier modified deadlines techniques under a single framework [49]. On the implementation side, besides using the technique of fast list scheduling in $O(n\alpha(n))$ time for solving the $P|r_i; d_i; p_i = 1|\bullet$ relaxations, they prove that iterating once over the operations in backward topological order of the dependence graph yields the same set of modified deadlines as the brute-force approach of iterating the deadline modification process until a fixed-point is reached.

1.5 The Non-Preemptive Time-Indexed RCPSP Formulation

Due to its ability to describe the RCPSP with general temporal constraints and other extensions, the *non-preemptive time-indexed formulation* introduced by Pritsker, Watters and Wolfe in 1969 [62] provides the basis for many RCPSP solution strategies. In this formulation, T denotes the time horizon and $\{0, 1\}$ variables $\{x_i^t\}$ are introduced such that $x_i^t \stackrel{\text{def}}{=} 1$ if $\sigma_i = t$, else $x_i^t \stackrel{\text{def}}{=} 0$. In particular $\sigma_i = \sum_{t=1}^{T-1} t x_i^t$. The following formulation minimizes the schedule length C_{max} by minimizing the completion time of the dummy operation O_{n+1}:

$$\text{minimize} \sum_{t=1}^{T-1} t\, x_{n+1}^t \quad : \tag{1}$$

$$\sum_{t=0}^{T-1} x_i^t = 1 \quad \forall i \in [1, n+1] \tag{2}$$

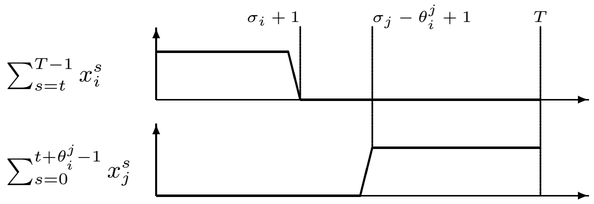


Figure 1: The time-indexed dependence inequalities of Christofides et al. [11]

$$\sum_{s=t}^{T-1} x_i^s + \sum_{s=0}^{t+\theta_i^j-1} x_j^s \quad \leq \quad 1 \quad \forall t \in [0, T-1], \forall (i,j) \in E_{dep} \tag{3}$$

$$\sum_{i=1}^{n} \sum_{s=t-p_i+1}^{t} x_i^s \, \vec{b}_i \quad \leq \quad \vec{B} \quad \forall t \in [0, T-1] \tag{4}$$

$$x_i^t \quad \geq \quad 0 \quad \forall i \in [1, n+1], \forall t \in [0, T-1] \tag{5}$$

$$x_i^t \quad \in \quad \mathbb{Z} \quad \forall i \in [1, n+1], \forall t \in [0, T-1] \tag{6}$$

The equations (2) ensure that any operation is scheduled once. The inequalities (3) describe the dependence constraints with T inequalities per dependence, as proposed by Christofides et al. [11]. While the original formulation [62] introduced only one inequality per dependence $\sum_{t=0}^{T-1} t(x_j^t - x_i^t) \geq \theta_i^j \Leftrightarrow \sigma_j - \sigma_i \geq \theta_i^j$, it was reported to be more difficult to solve in practice. The explanation for the dependence inequalities (3) appears in Figure 1: ensuring the sum of the terms $\sum_{s=t}^{T-1} x_i^s$ and $\sum_{s=0}^{t+\theta_i^j-1} x_j^s$ is no greater than one implies $\sigma_i + 1 \leq \sigma_j - \theta_i^j + 1$.

Last, the inequalities (4) enforce the cumulative resource constraints for $p_i \geq 1$. The extensions of the RCPSP with time-dependent resource availabilities $\vec{B}(t)$ and resource requirements $\vec{b}_i(t)$ are described in this formulation by generalizing (4) into (7):

$$\sum_{i=1}^{n} \sum_{s=0}^{t} x_i^s \, \vec{b}_i(t-s) \quad \leq \quad \vec{B}(t) \quad \forall t \in [0, T-1] \tag{7}$$

Although fully descriptive and extensible, the main problem with the time-indexed RCPSP formulations is the large size of the resulting integer programs. Indeed, Savelsbergh et al. [70] observed that such formulations could not be solved for problem instances involving more than about 100 jobs (operations), even with the column-generation techniques of Akker et al. [4].

2 VLIW Instruction Scheduling Problems

2.1 Instruction Scheduling and Register Allocation in a Code Generator

In a compiler, instruction scheduling is applied in the last stages of optimizations before assembly language output, inside the so-called *code generator*. The code generator is in charge of four major tasks: code selection, control optimizations, register allocation and instruction scheduling [22]. Code selection translates the computations from the upper stage of the compiler into efficient target processor operation sequences. Control optimizations reduce the run-time penalty of branches by predicating code [54] and by reordering the basic blocks [61]. The register allocator maps the operation operands to the processor registers or memory locations. The instruction scheduler reorders the operations to reduce the run-time processor stalls. In modern compilers such as SGI's Open64 [58], the code generator is also in charge of several loop transformations and scalar optimizations that are applied before code selection.

The major component of the program representation in a code generator is the *code stream*, which contains a list of *basic blocks*. Each basic block is itself a list of the target processor operations. A basic block that can only be entered at the first operation and contains no control-flow operations (branch or call) except possibly for the last operation. The most important property of a basic block is that all its operations are executed when the program reaches its first operation. The program execution flow is itself represented by a *control flow graph*, whose nodes are the basic blocks and whose arcs materialize the possible control flow transfers between the basic blocks.

In the compiler code generator, instruction scheduling problems are of several types. A first parameter is the shape of the program regions that compose each instruction scheduling problem, or *scheduling regions*. While the simplest scheduling regions are the basic blocks, the *super blocks* are preferred [40] because they enlarge the instruction scheduling problem scope. A super block is a string of basic blocks where each basic block has a single predecessor, except possibly for the first basic block. The second parameter of instruction scheduling is whether the scheduling region is a whole inner loop body or not. If so, *cyclic scheduling* may be applied instead of the *acyclic scheduling* discussed so far. The third parameter of instruction scheduling is whether it runs before register allocation (*pre-pass scheduling*) or after (*post-pass scheduling*).

In the code generator, the operations exchange values through *virtual registers*. Register allocation is the process of mapping the virtual registers in use at any program point either to the architectural registers or to memory locations. When a virtual register is mapped to a memory location, it needs to be *reloaded* before use. When space is exhausted in the register file, one or more virtual registers must be *spilled* to memory locations. In some cases, it is cheaper to recompute a virtual register value from the available register operands than reloading it, an action called *rematerialization*. The register allocation may also *coalesce* some register live ranges to eliminate COPY operations and, conversely, may have to *split* a register live range by inserting a COPY operation in order to relax the register assignment constraints.

The *local register allocation* operates one program region at a time and deals with the virtual registers that are referenced in or live through that region. In practice, the program regions managed by the local register allocation are the basic blocks. Conversely, the *global register allocation* deals with the virtual registers that are live across the local register allocation regions. With state of the art compiler technology such as SGI's Open64 [58], global register allocation between the basic blocks is performed first, followed by local register allocation.

Most results and techniques for minimizing the penalty of register allocation assume a given operation order and insert extra operations order to spill, reload, rematerialize, or split the register live ranges. In this report, our focus is to minimize the penalty of register allocation by extending the pre-pass instruction scheduling problems with specific constraints or objectives.

In early compilers, the operation order used by register allocation was given to the code generator and instruction scheduling was only performed after register allocation. However, on RISC-like architectures with large register files, the register dependences created by the reuse of the architectural registers significantly restrict the instruction-level parallelism [36, 10]. The practice in modern production compilers is thus to run a pre-pass instruction scheduler, the register allocation, then a post-pass instructions scheduler, to obtain the final code.

Although an improvement over earlier solutions, instruction scheduling twice does not solve the phase-ordering problem between instruction scheduling and register allocation. In particular, when the pre-pass instruction scheduling creates high register pressure, the register allocator may insert so much spill code that the resulting performance is degraded, compared to a less aggressive or even no pre-pass instruction scheduling. The strategies to address this issue are:

- The *lifetime-sensitive instruction scheduling problems* (LSISP) seek to minimize the register lifetimes during the pre-pass instruction scheduling. A *register lifetime* is a time interval in the instruction schedule that starts at the schedule date of the first operation that defines the register and stops at the schedule date of the last operation that uses the register.
- The *register pressure instruction scheduling problems* (RPISP) seek to minimize the maximum register pressure during the pre-pass instruction scheduling. The *register pressure* is defined for a given date as the number of register lifetimes that contain this date. The maximum register pressure is taken over all the dates spanned by the instruction schedule.
- The *register-constrained instruction scheduling problems* (RCISP) ensure that a given maximum register pressure is never exceeded during the pre-pass instruction scheduling. When it does not fail, this ensures the local register allocation has minimal or no impact on the schedule.
- The *register allocation instruction scheduling problems* (RAISP) combine the register spilling, reloading, rematerialization and splitting with the instruction scheduling.

2.2 The Block and Pipeline VLIW Instruction Scheduling Problems

We shall illustrate instruction scheduling problems with the ST220 processor, a single-cluster implementation of the Lx core [29] produced by STMicroelectronics. This processor executes up to 4 operations per cycle with a maximum of one control operation (goto, jump, call, return), one memory operation (load, store, prefetch), and two multiply operations per cycle. All arithmetic instructions operate on integer values with operands belonging either to the General Register file (64×32-bit), or to the Branch Register file (8×1-bit). In order to eliminate some conditional branches, the ST200 architecture also provides conditional selection. The processing time of any operation is a single cycle, while the latencies between operations range from 0 to 3 cycles.

The resource constraints of the ST220 processor are accurately modeled with cumulative resources. In Figure 2, **Width** represents the issue width, **IMX** represents the extended immediate value generator, **MUL** represents the multiply operators, **MEM** represents the memory access unit and **CTL** represents the control unit. The artificial resource **ODD** represents the operation alignment constraints. For instruction scheduling purposes, the resource requirements of the operations are abstracted by *resource classes*, which in case of the ST220 processor are { **ALU**, **ALUX**, **MUL**, **MULX**, **MEM**, **MEMX**, **CTL** }. The resource classes **ALU**, **MUL**, **MEM** and **CTL** correspond respectively to the single-cycle, multiply, memory and control operations. Similarly, the resource classes **ALUX**, **MULX** and **MEMX** abstract the operations with an extended immediate operand.

We define a *VLIW instruction scheduling problem* as a RCPSP with dependence constraints, where the dependence latencies θ_i^j are non-negative integers. Most VLIW instruction scheduling problems also have UET operations. Indeed, in modern processors, all the functional units are pipelined, meaning there is an apparent processing time p_i of one cycle and a fixed non-negative latency θ_i^j, after which the result of an operation is available in a destination register. In addition to these so-called read after write (RAW) register dependences, the dependences of instruction scheduling problems include write after read (WAR), write after write (WAW), register dependences, memory dependences and control dependences.

Resource	Width	IMX	MUL	MEM	CTL	ODD
Availability	4	2	2	1	1	2
Resource Class Requirements	**Width**	**IMX**	**MUL**	**MEM**	**CTL**	**ODD**
ALU	1					
ALUX	2	1				1
MUL	1		1			1
MULX	2	1	1			1
MEM	1			1		
MEMX	2	1		1		1
CTL	1				1	1

Figure 2: The ST220 cumulative resource availabilities and resource class requirements.

```
int
prod(int n, short a[], short b) {
  int s = 0, i;
  for (i = 0; i < n ; i++) {
    s += a[i]*b;
  }
  return s;
}
```

```
L?__0_8:
  LDH_1       g96 = 0, G92
  MULL_2      g97 = G91, g96
  ADD_3       G94 = G94, g97
  ADD_4       G93 = G93, 1
  ADD_5       G92 = G92, 2
  CMPNE_6     b100 = G83, G93
  BR_7        b100, L?__0_8
```

Figure 3: Source code and the inner loop body code generator representation.

We define a *block instruction scheduling problem* (BISP) as an acyclic instruction scheduling problem whose objective is the minimization of L_{max}, or C_{max} when there are no deadlines on the operations. In a BISP, release dates and deadline dates arise because of the interactions between the different scheduling regions [1]. A BISP may apply to a basic block, a super block or other acyclic scheduling region and can be either pre-pass or post-pass. Conversely, we define a (software) *pipeline instruction scheduling problem* (PISP) as a cyclic instruction scheduling problem whose objective is the minimization of the average time between the execution of two successive loop iterations, also called the loop *initiation interval* [65].

These two specializations of instruction scheduling problems are illustrated on the sample C source code in Figure 3, which also displays the translation of the inner loop body in ST220 operations. When considering this inner loop body as a BISP, the ST200 LAO instruction scheduler (see §3.4) generates an instruction schedule of 7 cycles, displayed in Figure 4, which starts at cycle `[0]` and stops at cycle `[6]`. In the right part of Figure 4, we display the *resource table*, the data structure of the instruction scheduler that tracks the resource uses of each scheduled operation (in the resource table, each operation is represented by its resource class).

When considering the inner loop of Figure 3 as a PISP, the result is a 8-cycle *local schedule*, displayed in Figure 5. However, this local schedule allows the execution of the loop operations to be overlapped, or *software pipelined*, with a constant period of 2 cycles, as illustrated by the resource table in Figure 5. The software pipelining technique used by the ST200 LAO instruction scheduler to solve the PISP is *insertion scheduling* (§3.4), a variant of *modulo scheduling* (§2.4).

2.3 Cyclic, Periodic and Pipeline Scheduling Problems

A *cyclic machine scheduling problem* [38] is given by a set of generic operations $\{O_i\}_{1\leq i\leq n}$ with generic dependences between them. This set of operations is to be executed repeatedly in a given

```
[0] LDH_1       g96 = 0, G92        [0] ALU_5   MEM_1
[0] ADD_5       G92 = G92, 2        [1]
[2] ADD_4       G93 = G93, 1        [2] ALU_4
[3] MULL_2      g97 = G91, g96      [3] ALU_6   MUL_2
[3] CMPNE_6     b100 = G83, G93     [4]
[6] ADD_3       G94 = G94, g97      [5]
[6] BR_7        b100, L?__0_8       [6] ALU_3   CTL_7
```

Figure 4: The block scheduled loop body and the block schedule resource table.

```
[0] LDH_1       g96 = 0, G92        [0] ALU_5.1 MEM_1.1
[0] ADD_5       G92 = G92, 2        [1]
[2] ADD_4       G93 = G93, 1        [2] ALU_4.1 ALU_5.2 MEM_1.2
[3] MULL_2      g97 = G91, g96      [3] ALU_6.1 MUL_2.1
[3] CMPNE_6     b100 = G83, G93     [4] ALU_4.2 ALU_5.2 MEM_1.3
[6] ADD_3       G94 = G94, g97      [5] ALU_6.2 MUL_2.2
[7] BRF_7       b100, .LAO001       [6] ALU_3.1 ALU_4.3 ALU_5.3 MEM_1.4
                                    [7] ALU_6.3 MUL_2.3 CTL_7.1
                                    [8] ALU_3.2 ALU_4.4 ALU_5.4 MEM_1.5
                                    [9] ALU_6.4 MUL_2.4 CTL_7.2
```

Figure 5: The software pipeline local schedule and the software pipeline resource table.

machine environment and all the instances $\{O_i^k\}^{k>0}$ of a given generic operation O_i have the same resource requirements. In a feasible cyclic schedule[5] $\{\sigma_i^k\}_{0\leq i\leq n+1}^{k>0}$, the total resource requirements of the operation instances executing at any time is not greater than the resource availabilities and the dependences between the operation instances are respected. The common model of generic dependences in cyclic scheduling problems is the *uniform dependence* model [38]:

$$O_i \xrightarrow{\theta_i^j,\omega_i^j} O_j \Longrightarrow \sigma_i^k + \theta_i^j \leq \sigma_j^{k+\omega_i^j} \quad \forall k > 0$$

The quality of a cyclic schedule is measured by its asymptotic throughput $R_\infty \stackrel{\text{def}}{=} \lim_{k\to\infty} \frac{k}{\sigma_{n+1}^k}$.

Periodic schedules are cyclic schedules defined by the property that the execution of the successive instances of a particular operation O_i are separated by a constant period $\lambda \in \mathbb{Q}$:

$$\forall i \in [0, n+1], \forall k > 0 : \sigma_i^k = \sigma_i^0 + k\lambda$$

Periodic schedules are a particular case of the K-periodic schedules, defined by $\sigma_i^k + K\lambda = \sigma_i^{k+K}$:

$$\forall i \in [0, n+1], \forall k > 0 : \sigma_i^k = \sigma_i^{k \bmod K} + K\lambda \left\lfloor \frac{k}{K} \right\rfloor$$

Given a cyclic machine scheduling problem with uniform dependences, there is always a K such that the K-periodic schedules contain an optimal solution [39], but this K is not guaranteed to equal one. The asymptotic throughput of a periodic schedule equals λ, so the *periodic scheduling problem* is the construction of a K-periodic schedule of minimum λ for a cyclic scheduling problem.

[5]We assume dummy operations O_0^k and O_{n+1}^k such that $\sigma_0^k \stackrel{\text{def}}{=} \min_{1\leq i\leq n} \sigma_i^k$ and $\sigma_{n+1}^k \stackrel{\text{def}}{=} \max_{1\leq i\leq n} \sigma_i^k$.

The lower bounds on λ for a K-periodic schedule are obtained using two types of relaxations. In the first class of relaxations, the *free schedule* ignores all the resource constraints. A refinement of this relaxation is the basic cyclic scheduling problem (BCSP) introduced by Hanen & Munier [38], where in addition to the original uniform dependences, the operations are constrained to be non-reentrant ($\sigma_i^k + p_i \leq \sigma_i^{k+1}$) through additional uniform dependences. In both cases [39]:

$$\lambda \geq \lambda_{rec} \stackrel{\text{def}}{=} \max_C \frac{\sum_C \theta_i^j}{\sum_C \omega_i^j} : \quad C \text{ circuit of the dependence graph}$$

The maximum ratio λ_{rec} is reached in particular on simple circuits, the *critical cycles*. The discovery of the maximum ratio λ_{rec} and the identification of a critical cycle is called the optimum cost to time ratio problem, for which a number of efficient algorithms exist [14]. Solving the BCSP for any $\lambda \geq \lambda_{rec}$ then reduces to a simple longest path computation [34].

In the second class of relaxations, all the dependences are ignored and the operations are scheduled for a minimum length λ_{res}, which provides another lower bound on λ. In case of parallel machine environment, UET generic operations and acyclic dependence graph, the solution of this relaxation can always be converted into an optimal periodic schedule with $\lambda = \lambda_{res}$ [65]. This technique was generalized by Eisenbeis & Windheiser to non-UET generic operations [28].

In the pipeline scheduling techniques, a local schedule $\{\sigma_i^1\}_{1 \leq i \leq n}$ is created first; then, K copies of this schedule are overlapped in a way to achieve the maximum average throughput, thus creating a K-periodic schedule of constrained structure. These *pipeline scheduling problems* originated from hardware pipeline optimization [44] and were later adapted to the software pipelining of loops. Trying to adapt the local schedule so as to maximize the software pipeline throughput led to the discovery of the more effective modulo scheduling techniques discussed in §2.4.

One heuristic technique to solve the K-periodic instruction scheduling problems is to schedule successive instances of the set of generic operations in acyclic context, until a repeating pattern of K successive instances appears in the schedule [6, 2]. In order to converge, this method requires that the span of any local schedule be bounded, a condition that is enforced by adding a span-limiting dependence from O_{n+1} to O_0. Although these techniques may suffer from slow convergence (large values of K) or from the sequential effect of the span-limiting dependence, they do not require the loop body to be a single basic block or super block.

The techniques that optimally solve the K-periodic instruction problems originate in work by Feautrier [30], where the D-periodic schedules are defined by numbers $A, \{B_i\}_{1 \leq i \leq n}, D \in \mathbb{Z}$ and:

$$\forall i \in [0, n+1], \forall k > 0 : \sigma_i^k = \left\lfloor \frac{Ak + B_i}{D} \right\rfloor \quad \text{with } A \text{ and } D \text{ relatively prime and } \lambda \stackrel{\text{def}}{=} \frac{A}{D}$$

Unlike the K-periodic schedules discussed earlier, this definition ensures that the σ_i^k are integral, yet it does not prevent them from reaching optimal solutions of the cyclic scheduling problems.

For uniform dependences and exclusive resources (unit availability), Feautrier shows [30]:

$$O_i \xrightarrow{\theta_i^j, \omega_i^j} O_j \implies B_i + D\theta_i^j - A\omega_i^j \leq B_j \tag{8}$$

$$O_i \text{ and } O_j \text{ share a resource} \implies A(q_i^j + 1) - B_j + B_i \geq Dp_j \wedge B_j - B_i - Aq_i^j \geq Dp_i \tag{9}$$

Here q_i^j is an integer variable. By guessing the value of D from the structure of the dependence circuits and by fixing the value of A in each step of an enumerative search for the minimum, these equations can be optimally solved by integer linear programming to compute the $\{B_i\}_{1 \leq i \leq n}$.

In a series of contributions, Fimmel & Müller refine this work, first to manage register constraints [31], then to find the optimal values of A and D without having to enumerate them [32].

2.4 Modulo Instruction Scheduling Problems and Techniques

Modulo scheduling is a framework for building 1-periodic schedules, where only the generic operations and the generic dependences are considered. The original modulo scheduling technique was introduced by Rau & Glaeser in 1981 [65], improved with *modulo expansion* by Lam in 1988 [47] and later refined by Rau in 1992 under the name *iterative modulo scheduling* [68].

In applications to cyclic instruction scheduling, the restriction to 1-periodic schedules with integer λ is not a problem, as loop unrolling is applied before instruction scheduling in order to enable further optimizations [15]. Indeed, the K-periodic scheduling techniques mentioned so far only perform *loop unwinding*, that is, replicating K complete copies of the original loop body. With loop unrolling, code is factored across the K copies of the original loop body, critical paths can be reduced [48] and further memory access bandwidth optimizations are enabled [21].

To describe modulo scheduling problems, we extend the $\alpha|\beta|\gamma$ notation with the β field $\pi_i = \lambda$, which requires that all the operations cyclically execute with the same processing period λ [23]. Formally, the modulo scheduling problem is defined as follows:

- Uniform dependence constraints $O_i \xrightarrow{\theta_i^j, \omega_i^j} O_j$: for each such dependence, a valid modulo schedule satisfies $\sigma_i + \theta_i^j - \lambda\omega_i^j \leq \sigma_j$. The *latency* θ_i^j and the *distance* ω_i^j of the dependences are non-negative integers. The *carried* dependences are such that $\omega_i^j > 0$. In addition, the dependence graph without the carried dependences is a directed acyclic graph (DAG).

- Cumulative modulo resource constraints: each operation O_i requires $\vec{b}_i \geq \vec{0}$ resources for all the time intervals $[\sigma_i + k\lambda, \sigma_i + k\lambda + p_i - 1], k \in \mathbb{N}$ and the total resource use at any time must not exceed $\vec{B}$. The positive integer value p_i is the processing time of operation O_i.

The modulo scheduling problems are in fact equivalent to a particular class of acyclic scheduling problems, the regular $\bar{\rho}$-unwinded schedules [23]. This equivalence yields results on the complexity of modulo scheduling problems. In particular, the problems $P|r_i; d_i; p_i = 1; \pi_i = \lambda|\bullet$ and $P|circuit(\theta_i^j, \omega_i^j); \sum \omega_i^j = 1; r_i; d_i; p_i = 1; \pi_i = \lambda|\bullet$ can be solved in pseudo-polynomial time [23]. The β field $circuit(\theta_i^j, \omega_i^j)$ denotes a dependence graph that is a single circuit.

We define a *modulo instruction scheduling problem* (MISP) as an instruction scheduling problem with modulo resource constraints and uniform dependences whose latencies θ_i^j and distances ω_i^j are non-negative. Compared to the BISP, the main difficulties of the MISP are the possibly negative dependence latencies $\theta_i^j - \lambda\omega_i^j$, the modulo resource constraints and the fact that these constraints are parametric with λ. The MISP objective is to minimize the λ and then C_{max}.

In the classic modulo scheduling framework [65, 47, 68], a dichotomy search for the minimum λ that yields a feasible schedule is performed, starting from $\lambda = \max(\lambda_{rec}, \lambda_{res})$. At each search step, heuristic scheduling is attempted, using a list scheduling algorithm extended to handle the modulo resource constraints. Because of the dependences with negative latencies, heuristic scheduling may fail even if the modulo scheduling problem is feasible at the current λ. To address this issue, limited backtracking is implemented with satisfactory results [42, 68, 69, 73]. When failure persists, the dichotomy search attempts heuristic scheduling at a higher λ.

A different heuristic approach for modulo scheduling was pioneered by Gasperoni & Schwiegelshohn [34]. The principle of *decomposed software pipelining* [71] is to decompose the modulo schedule dates into a quotient and remainder by λ, that is $\forall i \in [0, n+1] : \sigma_i = \lambda\phi_i + \tau_i, 0 \leq \tau_i < \lambda$. Given an operation O_i, ϕ_i is its *column number* or *stage number* and τ_i is its *row number*. The modulo scheduling problem is then approximately solved as follows [34]:

- build a resource-free schedule $\{\sigma_i^*\}_{1\leq i\leq n}$ with $\lambda^* \stackrel{\text{def}}{=} \lambda_{rec}$;
- compute the column numbers of the operations as $\phi_i \stackrel{\text{def}}{=} \left\lfloor \frac{\sigma_i^*}{\lambda^*} \right\rfloor$;
- delete the dependences that appear to be violated with the row numbers $\{\tau_i^* \stackrel{\text{def}}{=} \sigma_i^* \bmod \lambda^*\}_{1\leq i\leq n}$, that is, when $\tau_i^* + \theta_i^j > \tau_j^*$, yielding an acyclic modified dependence graph;
- schedule the operations with this modified acyclic dependence graph, this time taking into account the resource constraints;
- the resulting schedule dates are the final row numbers $\{\tau_i\}_{1\leq i\leq n}$ and the resulting initiation interval λ is the length τ_{n+1} of this schedule.

The main idea is that fixing the column numbers reduces the modulo scheduling problem with UET operations to a minimum length scheduling problem involving only the row numbers $\{\tau_i\}_{1\leq i\leq n}$ and ordinary resource constraints. This works because an apparent dependence violation on the row numbers implies that the difference between the column numbers is large enough: $\sigma_i + \theta_i^j - \lambda\omega_i^j \leq \sigma_j \wedge \tau_i + \theta_i^j > \tau_j \Rightarrow \phi_i < \omega_i^j + \phi_j$ (see Corollary 1 in §3.3). Ignoring these dependences while recomputing the row numbers at a higher λ is safe in a variety of cases.

The idea of decomposing the modulo scheduling problem has been generalized by Wang & Eisenbeis to compute either the column numbers first, or the row numbers first [71]. In particular, given the row numbers, they prove that dependence-preserving column numbers exist iff [71]:

$$\sum_{(O_i,O_j)\in C} \left\lceil \frac{\theta_i^j + \tau_i - \tau_j}{\lambda} \right\rceil - \omega_i^j \quad \leq 0 \quad \forall C \text{ circuit of the dependence graph}$$

Another generalization that still computes the column numbers first in resource-free context, then the row numbers with acyclic scheduling, is the loop shifting and compaction of Darte & Huard [13]. We also developed the idea of rescheduling while keeping the column numbers invariant in insertion scheduling [18], a modulo scheduling technique that neither assumes UET operations, nor requires a decomposition of the modulo scheduling problem (see §3.3 and §3.4).

2.5 Register Instruction Scheduling Problem Solving Approaches

The simplest attempt to solve a LSISP is to perform pre-pass instruction scheduling backward. This works because many expression DAG look like in-trees, while the instruction scheduling heuristics based on the LSA are greedy. As a result, backward instruction scheduling keeps the operations that define registers close to their use on average, resulting in shorter register lifetimes, thus indirectly contributing to lower register pressure.

However, minimizing the register lifetimes sometimes requires that some operations be scheduled as early as possible, in particular with modulo scheduling for some of the register lifetimes that carry values from one iteration to the other. These observations led to the formulation of lifetime-sensitive instruction scheduling [42], hyper-node reduction modulo scheduling [50], minimum cumulative register lifetime modulo scheduling [17] and swing modulo scheduling [51]. The general idea of these techniques is to schedule operations with scheduling slack near their early or late start dates, depending on the anticipated effects on the register lifetimes.

The RPISP are motivated by the fact that the amount of spill code generated by the register allocation is primarily dependent on the maximum register pressure *MaxLive* achieved in the instruction schedule; hence, *MaxLive* minimization should be part of the objective besides the schedule length or the modulo scheduling period λ. Unlike the register lifetimes, *MaxLive* is not linear with the schedule dates, so its minimization is difficult to integrate even in resource-free instruction scheduling problems.

For the block RPISP, an integrated technique was proposed by Goodman & Hsu in 1988 [36], where the instruction scheduling objective switches between the minimum length and the minimum register usage, whenever a threshold of register pressure is crossed. The block RPISP is now optimally solved with the effective enumeration procedure of Kessler [43]. For the modulo RPISP, several heuristic approaches have been proposed, based on the *stage scheduling* of Eichenberger & Davidson [26]. In stage scheduling, once a modulo schedule is computed, the stage (column) numbers of the operations are further adjusted to reduce the maximum register pressure.

The most advanced techniques for solving the RPISP are based on integer linear programming. Using the minimum buffers equations defined by Ning & Gao [57] for the resource-free modulo scheduling problems, Govindarajan et al. [37] introduce a time-indexed formulation for the resource-constrained modulo scheduling problems that minimizes the number of buffers, a coarse approximation of *MaxLive*. The modern formulation of the modulo RPISP, discussed in §2.6, was given in 1995 by Eichenberger, Davidson and Abraham [25].

The RCISP are related to the RPISP, with the main difference that *MaxLive* is now constrained by a hard bound. This has two advantages: first, this allows instruction scheduling to focus on the single objective of minimizing the schedule length (of the local schedule in case of modulo scheduling) and this objective is achieved by the schedule as far as the hard bound on *MaxLive* is not reached; second, by setting the hard bound value lower than the number of allocatable registers, the register allocation is guaranteed to succeed without spilling any register.

Precisely, in case of acyclic scheduling of scheduling regions without control-flow merge points (like basic blocks, super blocks, trees of blocks, etc.), register assignment reduces to the problem of minimum node coloring the interference graph of the register live ranges, which is a chordal graph, since it is an intersection graph of tree paths. This problem is polynomial-time solvable and the number of registers needed is exactly *MaxLive*, the maximum clique size of this graph.

In case of cyclic scheduling with basic block or super block regions, the interference graph of the register live ranges is a cyclic interval graph [41], whose minimum node coloring problem is *NP*-hard. As observed by Hendren et al. [41], splitting some of the register live ranges (by inserting COPY operations) yields an interval graph that is colorable with *MaxLive* registers as in the acyclic case. Alternatively, by heuristically coloring the original interference graph, it is observed that most loops can be register assigned with *MaxLive*+1 registers [66].

An heuristic for the block RCISP was proposed by Natarajan & Schlansker in 1995 [56]. Their approach is to schedule the operations for the minimum length and in case the maximum register pressure exceeds its hard bound, the dependence graph (which is a DAG) is split in two and each part is re-scheduled with the same technique. The main contribution is how to split the DAG to decrease the maximum register pressure reached in the sub-DAG in a resource-free schedule.

For the periodic RCISP, Fimmel et al. [31] introduce time-indexed variables ("binary decomposition"), with register pressure equations that are similar to those proposed by Eichenberger et al. [25]. For solving the modulo RCISP, we propose a new time-indexed formulation in §3.6, which, unlike those approaches, does not require the λ-decomposition of the schedule dates. This modulo RCISP formulation also applies to the block RCISP by assuming a large enough λ.

In the RAISP, a hard bound on *MaxLive* lower than the number of allocatable registers is assumed like in the RCISP, but, in case the original scheduling problem is not feasible under such constraint or its solution is not satisfactory, operations that spill, restore, or split the register live ranges are inserted in the schedule. The main objective is to perform the local register allocation of the scheduling region while scheduling. Solving the RAISP is a motivating area of research for compiler code generator design [73]. In case of the software pipelined loops, an effective heuristic is implemented in the MIPS-PRO production compiler [69]. For the acyclic scheduling regions, an effective heuristic technique is implemented in the Multiflow Trace Scheduling compiler [53].

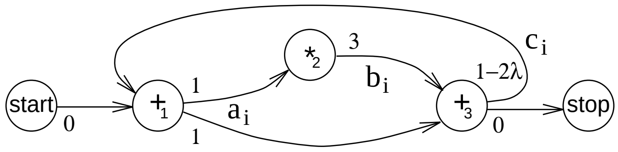
2.6 Time-Indexed Formulation for the Modulo RPISP

The modern formulation of the modulo RPISP is from Eichenberger et al. [25, 27]. This formulation takes a λ-decomposed view of the modulo scheduling problem, that is, the time horizon is $[0, \lambda-1]$ and the time-indexed variables represent the row numbers: $\{\tau_i \stackrel{\text{def}}{=} \sum_{t=0}^{\lambda-1} t x_i^t\}_{1\leq i\leq n}$. On the other hand, the column numbers $\{\phi_i\}_{1\leq i\leq n}$ are directly used in the formulation. The objective is to minimize the register pressure integer variable P:

$$
\begin{array}{rcll}
\text{minimize } P & : & & (10)\\
\sum_{t=0}^{\lambda-1} x_i^t & = & 1 \quad \forall i \in [1, n] & (11)\\
\sum_{s=t}^{\lambda-1} x_i^s + \sum_{s=0}^{(t+\theta_i^j-1) \bmod \lambda} x_j^s + \phi_i - \phi_j & \leq & \omega_i^j - \lfloor \frac{t+\theta_i^j-1}{\lambda} \rfloor + 1 \quad \forall t \in [0, \lambda-1], \forall (i,j) \in E_{dep} & (12)\\
\sum_{i=1}^{n} \sum_{r=0}^{p_i-1} x_i^{(t-r) \bmod \lambda} \vec{b}_i & \leq & \vec{B} \quad \forall t \in [0, \lambda-1] & (13)\\
\sum_{s=0}^{t} x_i^s - \sum_{s=0}^{t-1} x_j^s + \phi_j - \phi_i - v_i^t & \leq & -\omega_i^j \quad \forall t \in [0, \lambda-1], \forall (i,j) \in E_{reg} & (14)\\
\sum_{i=1}^{n} v_i^t - P & \leq & 0 \quad \forall t \in [0, \lambda-1] & (15)\\
x_i^t & \in & \{0, 1\} \quad \forall i \in [1, n], \forall t \in [0, \lambda-1] & (16)\\
v_i^t & \in & \mathbb{N} \quad \forall i \in [1, n], \forall t \in [0, \lambda-1] & (17)\\
\phi_i & \in & \mathbb{N} \quad \forall i \in [1, n] & (18)
\end{array}
$$

Although this formulation was elaborated without apparent knowledge of the time-indexed RCPSP formulation of Pritsker et al. [62], it is quite similar. By comparison with the formulation in §1.5, equations (11), (12), (13), extend to modulo scheduling the equations (2), (3), (4). Besides the extension of the RCPSP formulation to the λ-decomposed view of modulo scheduling problems, the other major contribution of Eichenberger et al. [25, 27] is the development of the inequalities (14) that compute the contributions v_i^t to the register pressure at date t.

As explained in [25], these equations derive from the observation that the function $D_i(t) \stackrel{\text{def}}{=} \sum_{s=0}^{t} x_i^s$ transitions from 0 to 1 for $t = \tau_i$. Similarly, the function $U_j(t) \stackrel{\text{def}}{=} \sum_{s=0}^{t-1} x_j^s$ transitions from 0 to 1 for $t = \tau_j + 1$. Assuming that $\tau_i \leq \tau_j$, then $D_i(t) - U_j(t) = 1$ exactly for the dates $t \in [\tau_i, \tau_j]$, else is 0. Similarly, if $\tau_i > \tau_j$ then $D_i(t) - U_j(t) + 1 = 1$ exactly for the dates $t \in [0, \sigma_j] \cup [\sigma_i, \lambda-1]$ else is 0. These expressions define the so-called *fractional lifetimes* of a register defined by O_i and used by O_j. When considering the additional register lifetime created by the column numbers, an *integral lifetime* of either $\phi_j - \phi_i + \omega_i^j$ or $\phi_j - \phi_i + \omega_i^j - 1$ must be accounted for. This yields the inequalities (14) that define the v_i^t and the maximum register pressure P is computed by (15).




```
for (i=2; i<n; i++) {
    a[i] = x+c[i-2];
    b[i] = a[i]*f;
    c[i] = a[i]+b[i];
}
```

Figure 6: Original dependence graph.

Like the other classic modulo scheduling techniques [47, 68, 69], this time-indexed formulation takes λ as a constant given in the scheduling problem description. Thus, the construction and the resolution of this formulation is only a step in the general modulo scheduling process, which searches for the minimum value of λ that allows a feasible modulo schedule.

3 Instruction Scheduling for the ST200 Architecture

3.1 Minimum Cumulative Register Lifetime Modulo Scheduling

Since controlling the register pressure during instruction scheduling is a difficult problem, *lifetime-sensitive* modulo schedulers have been proposed [42], which aim at minimizing the cumulative register lifetimes. We now show that, in case of resource-free modulo scheduling problems, the modulo schedule that minimizes the cumulative register lifetimes is easily computed by solving a network flow problem on the dependence graph suitably augmented with nodes and arcs. See §3.4 for an application of this result to modulo scheduling with resource constraints.

We shall illustrate the minimization of the cumulative register lifetimes on the sample dependence graph in Figure 6. This dependence graph has been simplified by removing the memory access operations after converting the uniform dependences into register transfers [9]. We assume that the target architecture is such that the lifetime of a register starts at the time the operation which produces it is issued. When this is not the case, the only difference is a constant contribution to the cumulative register lifetimes that does not impact the minimization.

Taking the dependence graph in Figure 6, let $\vec{\sigma}$ be the schedule dates of the respective operations $+_1, *_2, +_3$ and let $\vec{l}$ be the lifetimes of the register defined by these operations [17]. Then we get the following system of inequalities:

$$\left(\begin{array}{c}\textit{dependence}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}\sigma_2-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_2 & \geq & 3\\ \sigma_1-\sigma_3 & \geq & 1-2\lambda\end{array}\right. \qquad \left(\begin{array}{c}\textit{lifetime}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}l_1 & \geq & \sigma_2-\sigma_1\\ l_1 & \geq & \sigma_3-\sigma_1\\ l_2 & \geq & \sigma_3-\sigma_2\\ l_3 & \geq & \sigma_1+2\lambda-\sigma_3\end{array}\right.$$

This illustrates the equations we introduced in [17] and the Minimum Cumulative Register Lifetimes (MCRL) problem is defined with the objective to minimize $\sum_i l_i$. In [19], we introduced a transformation of the MCRL problem that makes it solvable in a simple and efficient way. The first step is to introduce the variables $\vec{r}$ such that $r_i \stackrel{\text{def}}{=} l_i + \sigma_i$ and to eliminate the variables $\vec{l}$:

$$\left(\begin{array}{c}\textit{dependence}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}\sigma_2-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_2 & \geq & 3\\ \sigma_1-\sigma_3 & \geq & 1-2\lambda\end{array}\right. \qquad \left(\begin{array}{c}\textit{reduced}\\ \textit{lifetime}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}r_1-\sigma_2 & \geq & 0\\ r_1-\sigma_3 & \geq & 0\\ r_2-\sigma_3 & \geq & 0\\ r_3-\sigma_1 & \geq & 2\lambda\end{array}\right.$$

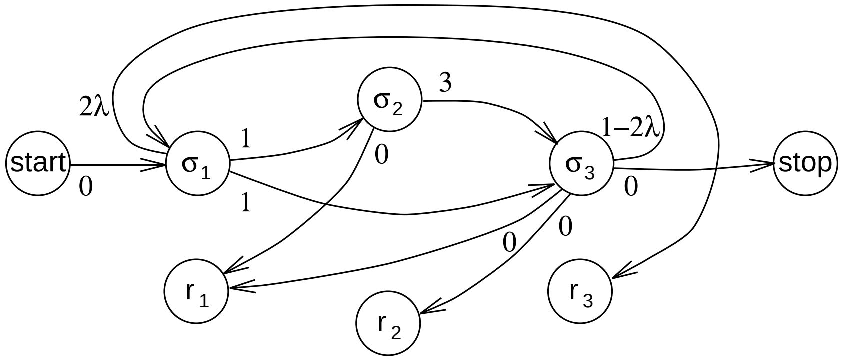


Figure 7: Augmented dependence graph.

This new system of inequalities defines an *augmented dependence graph*, which is derived from the dependence graph as follows: any register lifetime *producer node* is paired with a *lifetime node* and for any lifetime arc between the producer node and a *consumer node*, we add an arc from the consumer node to the lifetime node. This transformation is illustrated in Figure 7 for the dependence graph in Figure 6. Since $l_i + \sigma_i \stackrel{\text{def}}{=} r_i \ \forall i$, the objective to minimize $\sum_i l_i$ becomes $\sum_i r_i - \sigma_i$.

It is interesting to compare these equations to those of the minimum buffers problem, as defined by Ning & Gao [57]. In the minimum buffers problem, the so-called buffer variables $\vec{b}$ are defined and the objective is to minimize $\sum_i b_i$:

$$\left(\begin{array}{c}\textit{dependence}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}\sigma_2-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_1 & \geq & 1\\ \sigma_3-\sigma_2 & \geq & 3\\ \sigma_1-\sigma_3 & \geq & 1-2\lambda\end{array}\right. \qquad \left(\begin{array}{c}\textit{buffer}\\ \textit{inequalities}\end{array}\right)\left\{\begin{array}{lcl}\lambda b_1 & \geq & \sigma_2-\sigma_1\\ \lambda b_1 & \geq & \sigma_3-\sigma_1\\ \lambda b_2 & \geq & \sigma_3-\sigma_2\\ \lambda b_3 & \geq & \sigma_1-\sigma_3+3\lambda-1\end{array}\right.$$

Obviously, the dependence equations are the same, while the buffer and the lifetime equations are different, although related. To make the minimum buffers problem easier to solve, Ning & Gao [57] make the change of variables $\lambda b_i \stackrel{\text{def}}{=} b'_i$, so the left-hand side of the buffer inequalities become identical to the left-hand side of the lifetime inequalities.

The MCRL problem is efficiently solved on the augmented dependence graph [19]. To see how, let us first assume that all the arcs of the original dependence graph carry a lifetime. Then, if U denotes the arc-node incidence matrix of the original dependence graph, the primal–dual relationships of linear programming can be written as:

$$\left.\begin{array}{c}\min\begin{pmatrix}-\vec{1}\\ \vec{1}\end{pmatrix}^{\mathrm{T}}\begin{pmatrix}\vec{\sigma}\\ \vec{r}\end{pmatrix}\\ \begin{bmatrix}U & 0\\ 0 & -U\end{bmatrix}\begin{pmatrix}\vec{\sigma}\\ \vec{r}\end{pmatrix}\geq\begin{pmatrix}\vec{\theta}-\lambda\vec{\omega}\\ \lambda\vec{\omega}\end{pmatrix}\\ \vec{\sigma},\ \vec{r}\text{ unrestricted in sign}\end{array}\right\}\rightleftharpoons\left\{\begin{array}{c}\max\begin{pmatrix}\vec{\theta}-\lambda\vec{\omega}\\ \lambda\vec{\omega}\end{pmatrix}^{\mathrm{T}}\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix}\\ \begin{bmatrix}U^{\mathrm{T}} & 0\\ 0 & -U^{\mathrm{T}}\end{bmatrix}\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix}=\begin{pmatrix}-\vec{1}\\ \vec{1}\end{pmatrix}\\ \vec{x}\geq\vec{0},\ \vec{y}\geq\vec{0}\end{array}\right.$$

Here the linear program associated with the MCRL problem appears on the left and its dual on the right. Negating both sides of the equal sign in the dual linear program yields a maximum-cost network flow problem on the dependence graph augmented with the lifetime nodes, where each

	0	1	2	3	4
bus1					
bus2	1				
abox		1			
cond			1	1	1
bbox					
ebox					
imul					
iwrt					
fbox					
fdiv					
fwrt					

	0	1	2
bus1			
bus2	1		
abox	1		
cond	1	1	1
bbox			
ebox			
imul			
iwrt			
fbox			
fdiv			
fwrt			

bus1	0
bus2	1
abox	1
cond	3
bbox	0
ebox	0
imul	0
iwrt	0
fbox	0
fdiv	0
fwrt	0

Figure 8: A reservation table, a regular reservation table and a reservation vector.

producer node supplies unit flow and where each lifetime node demands unit flow. In practice, only the nodes of the dependence graph that define a register are paired with a lifetime node.

Beyond the MCRL problems, the generalization of a resource-free scheduling problem, which is solved as a single-source longest path over the dependence graph, into a maximum cost network flow problem over the augmented dependence graph allows the introduction of "soft constraints" in instruction scheduling problems. In the STMicroelectronics ST100 Linear Assembly Optimizer (LAO) code generator [22], increasing some of the flow requirements addressed the non-standard instruction scheduling features of the ST120 decoupled implementation. The augmented dependence graph is also used for the time-indexed integer linear programming formulations of the register pressure instruction scheduling problems (see §3.6).

3.2 Resource Modeling in Instruction Scheduling Problems

The resource constraints for instruction scheduling on a VLIW processor are described in a micro-architecture manual by a set of admissible *bundles*, that is, combinations of operations that may issue without resource conflicts[6]. Precisely, a bundle is a sequence of resource classes, where each resource class is an abstraction for the resource requirements of the operations in the instruction scheduling problem. Going from the bundle specifications of a processor to an accurate model with cumulative resources is not guaranteed to succeed. Techniques to build the cumulative resource model given a processor description are described in [63, 64]. Their idea is to introduce artificial resources where required, based on the identification of critical sets of resource classes.

Even with the introduction of artificial resources like **ODD** for the ST220 in Figure 2, a processor cumulative resource model can still be an approximation because the cumulative constraints can only represent convex regions of resource classes. To illustrate this point, consider a processor with two resource classes **A**, **B** and an issue width of 4. Then, no cumulative resource constraints may allow the combinations $\{\mathbf{A}, \mathbf{A}, \mathbf{A}, \mathbf{B}\}$ and $\{\mathbf{A}, \mathbf{B}, \mathbf{B}, \mathbf{B}\}$ while disallowing the combinations $\{\mathbf{A}, \mathbf{A}, \mathbf{B}, \mathbf{B}\}$. Indeed, this yields the contradiction $3a + b \leq c \wedge a + 3b \leq c \wedge 2a + 2b > c$, assuming that a and b are the resource requirements of **A** and **B** and that c is the resource availability.

Another limitation of the cumulative resources model is that all the resources involved in the execution of an operation O_i are assumed busy for its whole processing time p_i. Thus, when two operations O_i and O_j need to share an exclusive resource, either $\sigma_i + p_i \leq \sigma_j$ or $\sigma_j + p_j \leq \sigma_i$. In order to increase the precision of resource modeling, we need to express the fact that given an exclusive resource, either $\sigma_i + \rho_i^j$ or $\sigma_j + \rho_j^i \leq \sigma_i$, where ρ_i^j and ρ_j^i both depend on O_i and O_j.

[6]Although this term seems to imply EPIC architectures, J. Fisher in particular uses it for VLIW architectures.

The interpretation of ρ_i^j is the minimum number of cycles between σ_i and σ_j that ensures there are no resource conflicts between O_i and O_j, assuming O_i is scheduled before O_j. Such resource constraints can be modeled with *regular reservation tables*.

A *regular reservation table* is the traditional boolean pipeline reservation table of an operation [44, 68], with the added restrictions that the ones in each row start at column zero and are all adjacent. In case of exclusive resources, the regular reservation tables are also equivalent to *reservation vectors*. Figure 8 illustrates the representation of conditional store operation of the DEC Alpha 21064 using a reservation table, a regular reservation table and a reservation vector. The regular reservation tables are accurate enough to cover many pipelined processors. For example, when describing the DEC Alpha 21064 processor, we found [18] that the regular reservation tables were slightly inaccurate only for the integer multiplications and the floating-point divisions.

A main advantage of regular reservation tables is that expressing the modulo resource constraints yields inequalities that are similar to the dependence constraints. Let us assume a resource-feasible modulo scheduling problem at period λ and two operations O_i and O_j that conflict on an exclusive resource. Then, $1 \leq \rho_i^j, \rho_j^i \leq \lambda$ and the dates when operation O_i uses the exclusive resource belong to $\Upsilon_i^j(\lambda) \stackrel{\text{def}}{=} \cup_{k\in\mathbb{Z}}[\sigma_i + k\lambda, \sigma_i + k\lambda + \rho_i^j - 1]$. In the modulo scheduling problem, O_i and O_j do not conflict iff $\Upsilon_i^j(\lambda) \cap \Upsilon_j^i(\lambda) = \emptyset$, that is:

$$\begin{array}{ll}
\forall k, k' \in \mathbb{Z}, & \forall m \in [0, \rho_i^j - 1], \forall m' \in [0, \rho_j^i - 1] : \sigma_i + k\lambda + m \neq \sigma_j + k'\lambda + m' \\
\Longrightarrow & \forall m \in [0, \rho_j^i - 1], \forall m' \in [0, \rho_j^i - 1] : (\sigma_i - \sigma_j) \bmod \lambda \neq (m' - m) \bmod \lambda \\
\Longrightarrow & \max\limits_{m' \in [0, \rho_j^i - 1]} m' < (\sigma_i - \sigma_j) \bmod \lambda < \min\limits_{m \in [0, \rho_i^j - 1]} (\lambda - m) \\
\Longrightarrow & \rho_j^i - 1 < \sigma_i - \sigma_j - \lfloor \dfrac{\sigma_i - \sigma_j}{\lambda} \rfloor \lambda < \lambda - \rho_i^j + 1 \\
\Longrightarrow & \sigma_i - \sigma_j \geq \rho_j^i + k_i^j \lambda \wedge \sigma_j - \sigma_i \geq \rho_i^j - (k_i^j + 1)\lambda \quad \text{with} \quad k_i^j \stackrel{\text{def}}{=} \lfloor \dfrac{\sigma_i - \sigma_j}{\lambda} \rfloor
\end{array}$$

This result is similar to the inequalities (9) given by Feautrier [30] for the exclusive constraints of D-periodic scheduling, with the refinement that we express the value of k_i^j given σ_i and σ_j.

3.3 The Modulo Insertion Scheduling Theorems

Insertion scheduling [18] is a modulo scheduling heuristic that builds a series of partial modulo schedules by starting from a resource-free modulo schedule and by adding dependences or increasing λ to resolve the modulo resource conflicts until a complete modulo schedule is obtained. A *partial modulo schedule* is a modulo schedule where all the dependences are satisfied, while some of the resource conflicts are ignored. Insertion scheduling relies on conditions for transforming a partial modulo schedule $\{\sigma_i\}_{1\leq i\leq n}$ at period λ, into another partial modulo schedule $\{\sigma'_i\}_{1\leq i\leq n}$ at period λ'. To present these conditions, we first define $\{\phi_i, \tau_i, \phi'_i, \tau'_i, \delta_i\}_{1\leq i\leq n}$ such that:

$$\forall i \in [0, n+1] : \quad \left\{ \begin{array}{l} \sigma_i = \phi_i\lambda + \tau_i \wedge 0 \leq \tau_i < \lambda \\ \sigma'_i = \phi'_i\lambda' + \tau'_i \wedge 0 \leq \tau'_i < \lambda' \\ \delta_i = \tau'_i - \tau_i \end{array} \right. \tag{19}$$

We are interested in modulo schedule transformations where $\forall i \in [1, n] : \phi'_i = \phi_i$, $\tau'_i = \tau_i + \delta_i$ and $\delta_i \leq \Delta$ with $\Delta \stackrel{\text{def}}{=} \lambda' - \lambda$. Let $O_i \rightsquigarrow O_j$ denote the fact that O_i precedes O_j in the transitive closure of the loop-independent dependences ($\omega_i^j = 0$) of the dependence graph[7]. The following

[7]This relation can be safely approximated by taking the lexical order of the operations in the program text.

result states the conditions that must be met by the δ_i in order to preserve all the dependence constraints:

Theorem 1 *Let $\{\sigma_i\}_{1\leq i\leq n}$ be a dependence-feasible modulo schedule of a modulo scheduling problem P at period λ. Let $\{\sigma'_i\}_{1\leq i\leq n}$ be n integers such that:*

$$\forall i,j \in [0,n+1]: \quad \begin{cases} \phi_i = \phi'_i \\ 0 \leq \delta_i \leq \Delta \\ \tau_i < \tau_j \Longrightarrow \delta_i \leq \delta_j \\ \tau_i = \tau_j \wedge \phi_i > \phi_j \Longrightarrow \delta_i \leq \delta_j \\ \tau_i = \tau_j \wedge \phi_i = \phi_j \wedge O_i \leadsto O_j \Longrightarrow \delta_i \leq \delta_j \end{cases} \tag{20}$$

Then $\{\sigma'_i\}_{1\leq i\leq n}$ is a dependence-feasible modulo schedule of P at period $\lambda + \Delta$.

Proof: Let $O_i \overset{\theta_i^j,\omega_i^j}{\longrightarrow} O_j$ be a dependence of P. From the definition of a dependence constraint, $\sigma_j - \sigma_i \geq \theta_i^j - \omega_i^j\lambda$, or equivalently $\phi_j\lambda + \tau_j - \phi_i\lambda - \tau_i \geq \theta_i^j - \omega_i^j\lambda$. Given the hypothesis (20), let us show this dependence holds for $\{\sigma'_i\}_{1\leq i\leq n}$, that is, $\phi_j\lambda' + \tau'_j - \phi_i\lambda' - \tau'_i \geq \theta_i^j - \omega_i^j\lambda'$.

Dividing the dependence inequality by λ and taking the floor yields $\phi_j - \phi_i + \lfloor\frac{\tau_j - \tau_i}{\lambda}\rfloor \geq -\omega_i^j$, since all θ_i^j values are non-negative. We have $0 \leq \tau_i < \lambda$, $0 \leq \tau_j < \lambda$; hence, $0 \leq |\tau_j - \tau_i| < \lambda$ and the value of $\lfloor\frac{\tau_j-\tau_i}{\lambda}\rfloor$ is -1 or 0. Therefore, $\phi_j - \phi_i \geq -\omega_i^j$ and we consider its sub-cases:

$\phi_j - \phi_i = -\omega_i^j$: Let us show that $\tau'_j - \tau'_i \geq \theta_i^j$.

Since $\theta_i^j \geq 0$, we have $\tau_j \geq \tau_i$. Several sub-cases need to be distinguished:

$\tau_i < \tau_j$: From (20) we have $\delta_j \geq \delta_i \Leftrightarrow \tau'_j - \tau_j \geq \tau'_i - \tau_i \Leftrightarrow \tau'_j - \tau'_i \geq \tau_j - \tau_i \geq \theta_i^j$.

$\tau_i = \tau_j \wedge \phi_i \neq \phi_j$: Either $\phi_i > \phi_j$ or $\phi_i < \phi_j$. The latter is impossible, for $\omega_i^j = \phi_i - \phi_j$ and since all ω_i^j are non-negative. From (20), $\tau_i = \tau_j \wedge \phi_i > \phi_j$ yields $\delta_j \geq \delta_i$, so the conclusion is the same as above.

$\tau_i = \tau_j \wedge \phi_i = \phi_j$: Since $\omega_i^j = \phi_i - \phi_j = 0$, there are no dependence constraints unless $O_i \leadsto O_j$. In this case, taking $\delta_j \geq \delta_i$ works like in the cases above.

$\phi_j - \phi_i > -\omega_i^j$: Let us show that $(\phi_j - \phi_i + \omega_i^j)\lambda' + \tau'_j - \tau'_i - \theta_i^j \geq 0$.

We have $\phi_j - \phi_i + \omega_i^j \geq 1$, so $(\phi_j - \phi_i + \omega_i^j)\lambda' \geq (\phi_j - \phi_i + \omega_i^j)\lambda + \Delta$. By (20), we also have $\tau_i \leq \tau'_i \leq \tau_i + \Delta$ and $\tau_j \leq \tau'_j \leq \tau_j + \Delta$, so $\tau'_j - \tau'_i \geq \tau_j - \tau_i - \Delta$. Hence, $(\phi_j - \phi_i + \omega_i^j)\lambda' + \tau'_j - \tau'_i - \theta_i^j \geq (\phi_j - \phi_i + \omega_i^j)\lambda + \Delta + \tau_j - \tau_i - \Delta - \theta_i^j = (\phi_j - \phi_i + \omega_i^j)\lambda + \tau_j - \tau_i - \theta_i^j \geq 0$. □

Corollary 1 *In a dependence-feasible modulo schedule $\{\sigma_i\}_{1\leq i\leq n}$, for any dependence $O_i \overset{\theta_i^j,\omega_i^j}{\longrightarrow} O_j$, then:*

$$\phi_i = \phi_j + \omega_i^j \Longrightarrow \tau_i + \theta_i^j \leq \tau_j \quad \textit{and} \quad \tau_i + \theta_i^j > \tau_j \Longrightarrow \phi_i < \phi_j + \omega_i^j$$

A result similar to Theorem 1 holds for the modulo resource constraints of a modulo schedule, assuming these constraints only involve regular reservation tables and exclusive resources.

Theorem 2 *Let $\{\sigma_i\}_{1 \le i \le n}$ be schedule dates satisfying the modulo resource constraints at period λ, assuming regular reservation tables and exclusive resources. Let $\{\sigma'_i\}_{1 \le i \le n}$ be such that:*

$$\forall i, j \in [1, n] : \quad \begin{cases} \phi_i = \phi'_i \\ 0 \le \delta_i \le \Delta \\ \tau_i < \tau_j \Longrightarrow \delta_i \le \delta_j \end{cases} \tag{21}$$

Then $\{\sigma'_i\}_{1 \le i \le n}$ also satisfies the modulo resource constraints at period $\lambda + \Delta$.

Proof: With the regular reservation tables, the modulo resource constraints at λ for the schedule dates σ_i, σ_j of two operations O_i, O_j that need the exclusive resource are equivalent to (see §3.2):

$$\sigma_i - \sigma_j \ge \rho^i_j + k^j_i \lambda \quad \wedge \quad \sigma_j - \sigma_i \ge \rho^j_i - (k^j_i + 1)\lambda \quad \text{with} \quad k^j_i \stackrel{\text{def}}{=} \lfloor \frac{\sigma_i - \sigma_j}{\lambda} \rfloor \tag{22}$$

These constraints look exactly like ordinary dependence constraints, save the fact that the ω values are now of arbitrary sign. Since the sign of the ω values is only used in the proof of Theorem 1 for the cases where $\tau_i = \tau_j$, which need not be considered here because they imply no resource conflicts between σ_i and σ_j, we deduce from the proof of Theorem 1 that the $\{\sigma'_k\}_{1 \le k \le n}$ satisfy the modulo resource constraints at period λ'. □

Corollary 2 *Let $\{\sigma_i\}_{1 \le i \le n}$ be schedule dates satisfying the modulo resource constraints at period λ, assuming cumulative resources. Then, under the conditions (21), $\{\sigma'_i\}_{1 \le i \le n}$ also satisfies the modulo resource constraints at period $\lambda' \stackrel{\text{def}}{=} \lambda + \Delta$.*

Proof: In the proof of Theorem 2, whenever (22) holds for $\{\sigma_i\}_{1 \le i \le n}$, then it holds under the conditions (21) for $\{\sigma'_i\}_{1 \le i \le n}$, for any assumed values of ρ^j_i and ρ^i_j. This ensures that the simultaneous use of the cumulative resources does not increase in $\{\sigma'_i\}_{1 \le i \le n}$. □

By Theorem 1 and Corollary 2, any transformation of a partial modulo schedule $\{\sigma_i\}_{1 \le i \le n}$ at period λ into $\{\sigma'_i\}_{1 \le i \le n}$ at period $\lambda' \stackrel{\text{def}}{=} \lambda + \Delta$, under the conditions (20), yields a partial modulo schedule.

3.4 Insertion Scheduling in the ST200 LAO Instruction Scheduler

Insertion scheduling [18] is a modulo instruction scheduling heuristic that *issues* the operations in a given priority order, where the issuing of an operation is materialized in the dependence graphs by adding dependence arcs to constrain the feasible schedule dates of this operation to be exactly its issue date. In addition, insertion scheduling applies the insertion theorems of §3.3 to resolve the resource conflicts between the previously issued operations and the current operation to issue by increasing the period λ to make room for this operation. Under its simplest form, insertion scheduling relies on single-source longest path computations like most scheduling heuristics.

The ST200 LAO instruction scheduler implementation, however, uses MCRL minimization to achieve lifetime-sensitive instruction scheduling (§ 3.1). To solve the MCRL problems, this implementation relies on a network simplex algorithm to optimize the maximum cost flow problems [3] and on a network simplex longest path algorithm to find the initial solutions [35]. Besides being efficient in practice [19], the network simplex algorithms maintain the explicit values of the primal (edge flows) and the dual (node potentials) variables. The latter are precisely the schedule dates of the operations. The ST200 LAO implementation of insertion scheduling is as follows:

Step 0 The resource-free MCRL modulo scheduling problem, called P_0, is built and solved. This problem is represented by the forward dependence graph, the backward dependence graph and the MCRL augmented graph. We solve P_0 with a network simplex longest path algorithm, followed by a dual network simplex algorithm. The network simplex longest path algorithm starts with λ the period set to the lower bound λ_{res} and solves the optimum cost to time ratio problem by increasing the period to λ_{rec} if necessary for P_0 feasibility.

Step j This step transforms the modulo scheduling problem P_{j-1} into P_j. Let $\{S_{i_k}^{j-1}\}$ denote the issue dates assigned to the operations $\{O_{i_k}\}_{1\leq k\leq j-1}$ in the previous steps, that is, $P_{j-1} \stackrel{\text{def}}{=} P_0 \wedge \{\sigma_{i_1} = S_{i_1}^{j-1}\} \wedge \ldots \wedge \{\sigma_{i_{j-1}} = S_{i_{j-1}}^{j-1}\}$. The not yet issued operations are ranked in priority order and the one with the highest priority, denoted O_{i_j}, is issued as follows:

1. Run a primal network simplex algorithm on the three dependence graphs to compute the early start date e_{i_j}, the late start date l_{i_j} and the MCRL date m_{i_j}, of O_{i_j} in P_{j-1}. Then, compute $a_{i_j} \stackrel{\text{def}}{=} \min(l_{i_j} - m_{i_j}, \lambda - 1)$ and $b_{i_j} \stackrel{\text{def}}{=} \min(m_{i_j} - e_{i_j}, \lambda - 1 - a_{i_j})$. The possible issue dates are scanned first in $[m_{i_j}, m_{i_j} + a_{i_j}]$ in increasing order and second in $[m_{i_j} - b_{i_j}, m_{i_j} - 1]$ in decreasing order. This ensures that no more than λ feasible dates are scanned and that the issue dates next to the MCRL date m_{i_j} are scanned first.
2. Select $S_{i_j}^{j-1}$ as the first scanned date with the lowest modulo resource conflicts between O_{i_j} and $\{O_{i_k}\}_{1\leq k\leq j-1}$. This is measured by the minimum number of cycles Δ the period λ_{j-1} needs to increase to remove the modulo resource conflicts at period $\lambda_j \stackrel{\text{def}}{=} \lambda_{j-1} + \Delta$. If there are no conflicts between O_{i_j} and $\{O_{i_k}\}_{1\leq k\leq j-1}$ for $\sigma_{i_j} = S_{i_j}^{j-1}$ then $\Delta = 0$.
3. Let $\{S_{i_k}^{j-1}\}_{1\leq k\leq j}$ be the issue dates in P_{j-1}, and $\{S_{i_k}^{j}\}_{1\leq k\leq j}$ be the issue dates in P_j. The issue dates in P_j are computed from the issue dates in P_{j-1} by applying Theorem 1, with the $\{\delta_{i_k}\}_{1\leq k\leq j}$ chosen to eliminate the modulo resource constraints between O_{i_j} and $\{O_{i_k}\}_{1\leq k\leq j-1}$. See [18] for the details of the δ_{i_k} computations.
4. Build the modulo scheduling problem $P_j \stackrel{\text{def}}{=} P_0 \wedge \{\sigma_{i_1} = S_{i_1}^{j}\} \wedge \ldots \wedge \{\sigma_{i_j} = S_{i_j}^{j}\}$ from $P_{j-1} \wedge \{\sigma_{i_j} = S_{i_j}^{j-1}\}$ and the $\{\delta_{i_k}\}_{1\leq k\leq j}$ and update the modulo resource table.

At the end of Step j, all the dependence constraints are satisfied, as are the modulo resource constraints for the operations $\{O_{i_k}\}_{1\leq k\leq j}$. At the end of Step n, a modulo schedule is available.

In the ST200 LAO instruction scheduler, insertion scheduling operates both in pre-pass and post-pass instruction scheduling and the instruction scheduling regions are super blocks. During pre-pass scheduling, the lifetime-sensitive scheduling is enabled and inner loops are either modulo scheduled or loop scheduled. In post-pass scheduling, the lifetime-sensitive scheduling is disabled and all inner loops, including the software pipelines created during pre-pass scheduling, are loop scheduled. Loop scheduling is a restricted form of modulo scheduling, where the period is exactly the local schedule length. This is achieved by adding a dependence $O_{n+1} \to O_0$ with latency 1 and distance 1 to the P_0 problem. As a result, loop scheduling accounts for the effects of the loop-carried dependences, while preventing code motion between the loop iterations.

3.5 The STMicroelectronics ST200 Production Compiler

The STMicroelectronics ST200 production compiler `st200cc` is based on the GCC compiler front-end components, the Open64 compiler technology [58], the ST200 LAO global instruction scheduler adapted from the ST100 LAO code generator [22], an instruction cache optimizer [5] and the GNU back-end tools. The Open64 compiler has been re-targeted from the Intel IA64 to the

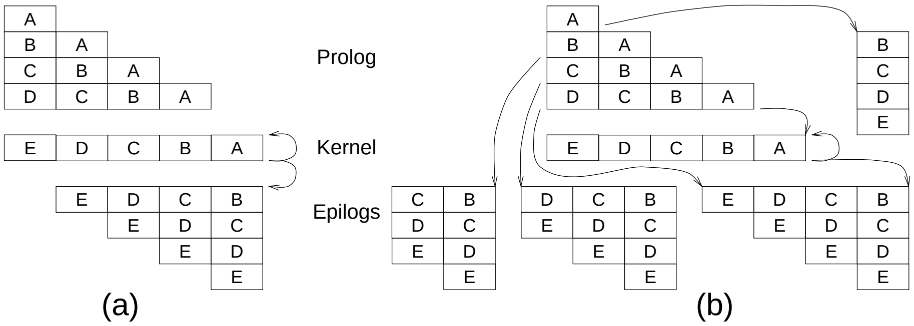


Figure 9: Counted loop software pipeline construction with and without pre-conditioning.

STMicroelectronics ST200 architecture, except for the instruction predication, the global code motion (GCM) instruction scheduler and the software pipeliner (SWP). Indeed, these components appeared to be too dependent on the IA64 architecture predicated execution model and on its support of modulo scheduling through rotating register files.

In the `st200cc` compiler, the Open64 GCM and SWP components are replaced by the ST200 LAO instruction scheduler, which is enabled at optimization level `-O3`. At optimization level `-O2` and below, only the Open64 basic block instruction scheduler is active. The ST200 LAO instruction scheduling and software pipelining relies on the following techniques:

- A transactional interface between the Open64 code generator intermediate representation (CGIR) and the LAO intermediate representation (LIR).
- Parsing of the control flow graph into a loop nesting forest and super block formation. The super block formation does not include tail duplication [40], as this is performed in the Open64 code generator by the ST200-specific instruction predication.
- Prior to pre-pass and post-pass scheduling, the algorithm of Leung, Palem and Pnueli [49] is iterated over each resource of the ST220 model (Figure 2), starting with the more heavily used resources. This iteration starts with the period set to the minimum value λ_{res} such that the renewable resources $\vec{B}$ are not over-subscribed; then, the period is increased until the relaxation is feasible.
- Pre-pass instruction scheduling based on insertion scheduling, which performs lifetime-sensitive super block scheduling, loop scheduling and modulo scheduling. The scheduling order is the lower MCRL dates first and the scheduling priories are the modified deadlines computed by the scheduling algorithm of Leung, Palem and Pnueli [49].
- Post-pass instruction scheduling based on insertion scheduling, which performs block scheduling and loop scheduling. The scheduling order is the lower consistent release dates first and the scheduling priories are the modified deadlines computed by the scheduling algorithm by Leung, Palem and Pnueli [49].

Once a modulo schedule is available, it must be transformed into a software pipeline. In Figure 9, we illustrate the classic software pipeline construction for the local schedule of a counted loop that spans five stages named A, B, C, D, E. In the basic case (a), loop pre-conditioning [67] is required to ensure the software pipeline will execute enough iterations to run the prologue and

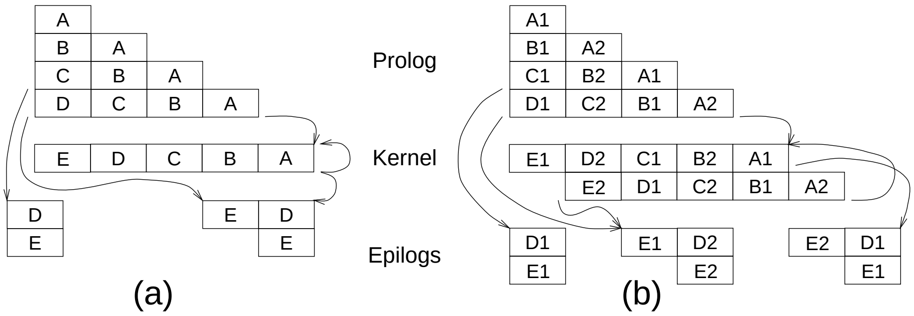


Figure 10: While-loop software pipeline construction without and with modulo expansion.

the epilogues. Without pre-conditioning, case (b) software pipeline construction is required. In the embedded computing setting, these classic software pipeline construction schemes are not effective because they only apply to counted loops and involve significant code size expansion.

For the ST200 modulo scheduling, we construct all software pipelines like while loops, as illustrated in case (a) of Figure 10. This while-loop construction scheme speculatively starts executing the next iterations before knowing if the current iteration is the last one and relies on the dismissible loads of the ST200 architecture. Another refinement implemented in the ST200 LAO is induction variable relaxation [20] and the restriction of modulo expansion to variables that are not live on loop exit. As a result, epilogues are unnecessary for virtually all the software pipelines.

Modulo expansion is a technique introduced by Lam [47], where the non-RAW register dependences on variables are omitted from the dependence graph. This exposes more parallelism to the modulo scheduler, but, when the software pipeline is constructed, each such variable may require more than one register for allocation. Modulo renaming is supported by architectures like the IA64 through *rotating registers* [66]. On architectures without rotating registers like the ST200, modulo expansion is implemented by *kernel unrolling* [47, 20, 52], as illustrated in case (b) of Figure 10.

To compare the effects of the instruction scheduling techniques available for the ST220 processor, we use the benchmark suite collected by the Hewlett Packard (HP) Laboratories for embedded computing and we also include the performances of the HP/Lx Multiflow-based compiler. This compiler implements Trace Scheduling [53] and targets the ST200 architecture including the ST220 processor. In Figure 11, we compare the performance in cycles and without the cache miss effects of the Multiflow-based compiler at optimization level `-O3` (MF), the `st200cc` compiler at `-O3` with software pipelining disabled (O3-nopipe), and the `st200cc` at `-O3` that enables software pipelining by default (O3). All the performances in Figure 11 are normalized by those of `st200cc` compiler at `-O3` with the ST200 LAO disabled, where basic block scheduling is done in pre-pass and post-pass by the Open64.

When enabling the ST200 LAO block scheduling and loop scheduling on super blocks, the geometric mean improvement is 5.5%, compared to the reference. When enabling the ST200 LAO software pipelining, the geometric mean improvement is 9.5%. This progression illustrates the benefits of the LAO super block scheduling over the Open64 basic block scheduling and the significant performance benefits brought by software pipelining. Thanks to the while-loop software pipeline construction scheme and to the elimination of software pipeline epilogues, there are no significant performance degradations related to software pipelining. This explains why software pipelining is activated by default in the `st200cc` compiler at optimization level `-O3`.

With the ST200 LAO disabled, the performances of the `st200cc` compiler at `-O3` and those

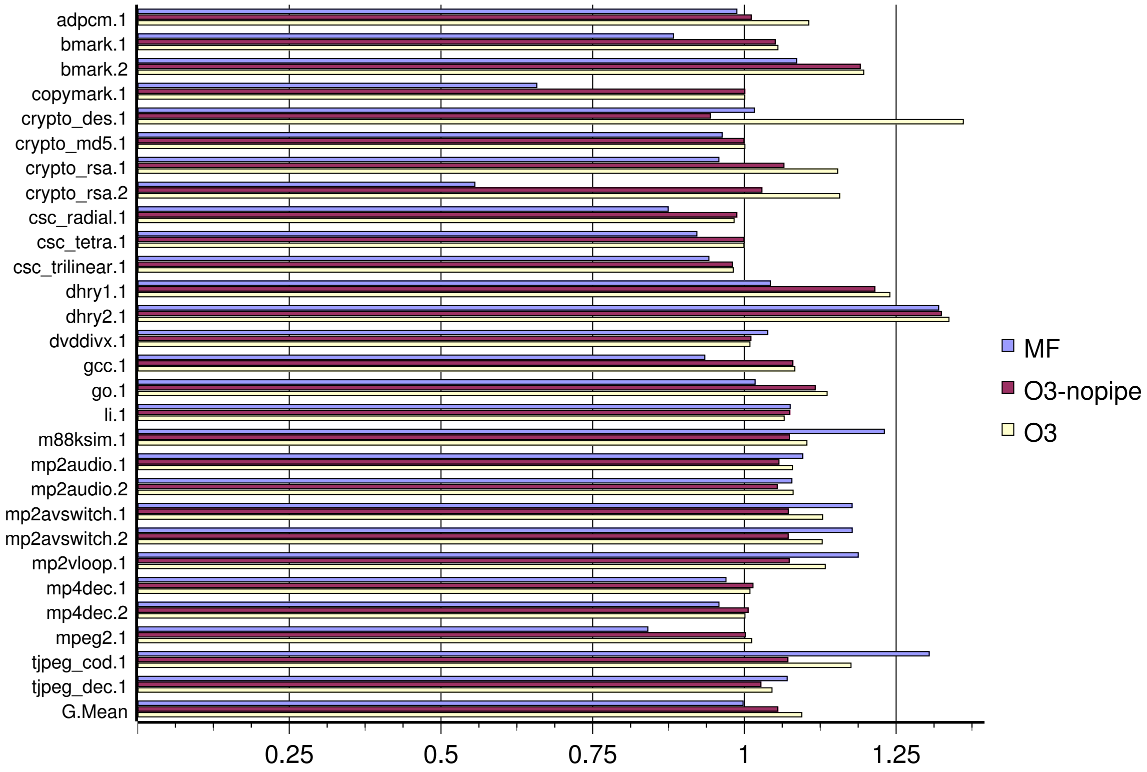


Figure 11: The `st200cc R3.2` compiler performances on the HP benchmarks.

of the Multiflow-based compiler at `-O3` are equivalent, as the geometric mean entry for MF is almost 1 (last bars in Figure 11). However, on benchmarks like the tjpeg_coder, the Multiflow-based compiler significantly outperforms the `st200cc` compiler, even with software pipelining. Investigations revealed that virtually all such `st200cc` performance problems are triggered by high register pressure on large basic blocks, where the pre-pass scheduling / register allocation / post-pass scheduling approach of Open64 is unable to compete with the integrated register allocation and instruction scheduling of the Multiflow-based compiler [53].

When accounting for the cache miss effects, the R3.2 `st200cc` compiler at optimization level `-O3` outperforms the Multiflow-based compiler at optimization level `-O3` by 15% in cycles in geometric mean on the HP benchmarks, with the best case at 18% higher performance and the worst case at 5% lower performance. This is explained by the 12% geometric mean code size reduction of the `st200cc` compiler and its effective instruction cache optimizations [5].

3.6 Time-Indexed Formulation of the Modulo RCISP

Based on the time-indexed formulation of the RCPSP of Pritsker et al. [62] in §1.5, the register-pressure equations of the modulo RPISP formulation of Eichenberger et al. [25, 27] in §2.6 and the definition of the augmented dependence graph in §3.1, we introduce a new time-indexed formulation for the modulo RCISP. This formulation is implemented in the ST200 LAO instruction scheduler as an experimental feature, as it requires linking with the commercial CPLEX package in order to solve the resulting integer linear programs.

The first extension of the RCPSP formulation needed for the modulo RCISP formulation is to

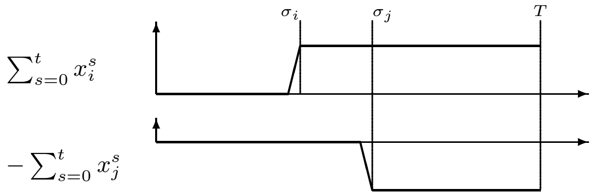


Figure 12: Definition of the contributions v_i^t to the register pressure.

account for the modulo resource constraints. In a modulo schedule, an operation O_i requires $\vec{b}_i$ resources for the dates in $[\sigma_i + k\lambda, \sigma_i + k\lambda + p_i - 1], \forall k \in \mathbb{Z}$ (§ 2.4). The resource requirement function $\vec{b}_i(t)$ of O_i is written $\sum_{k\in\mathbf{Z}}(t \in [k\lambda, k\lambda + p_i - 1])\vec{b}_i$, so the inequalities (7) become:

$$
\begin{array}{rll}
 & \sum_{i=1}^{n}\sum_{k\in\mathbf{Z}}\sum_{s=0}^{t} x_i^s(t - s \in [k\lambda, k\lambda + p_i - 1])\vec{b}_i & \leq \vec{B} \quad \forall t \in [0, T-1] \\
\Longrightarrow & \sum_{i=1}^{n}\sum_{k\in\mathbf{Z}}\sum_{s=0}^{t} x_i^s(s \in [t + k\lambda - p_i + 1, t + k\lambda])\vec{b}_i & \leq \vec{B} \quad \forall t \in [0, T-1] \\
\Longrightarrow & \sum_{i=1}^{n}\sum_{k\in\mathbf{Z}}\sum_{s=t+k\lambda-p_i+1}^{t+k\lambda} x_i^s\vec{b}_i & \leq \vec{B} \quad \forall t \in [0, T-1] \\
\Longrightarrow & \sum_{i=1}^{n}\sum_{k=0}^{\lfloor\frac{T-1}{\lambda}\rfloor}\sum_{s=t+k\lambda-p_i+1}^{t+k\lambda} x_i^s\vec{b}_i & \leq \vec{B} \quad \forall t \in [0, \lambda-1]
\end{array}
$$

For the register pressure constraints, we consider the dependence graph augmented with the register lifetime nodes as defined in §3.1. Let E_{life} denote the (i, j) pairs such that O_i is a producer node and O_j is its lifetime node. Let us also take advantage of the fact that most processors (in particular in the ST200 series) enable to use and redefine a register at the same cycle and have a non-zero minimum register RAW dependence latency. In such cases, it is correct to assume that a register lifetime extends from its definition date to its last use date minus one.

In case $\omega_i^j = 0$, the contribution v_i^t to the register pressure at date t of the register produced by O_i and consumed by O_j is exactly the difference of two terms as illustrated in Figure 12. When $\omega_i^j > 0$, the contribution to register pressure extends from date σ_i to date $\sigma_j + \lambda\omega_i^j$. This yields the definition of v_i^t by the equalities (23). In (24), we express the constraint that the register pressure at any date in the modulo schedule is not greater than the register availability R.

$$
\sum_{s=0}^{t} x_i^s - \sum_{s=0}^{t} x_j^{s-\lambda\omega_i^j} = v_i^t \quad \forall t \in [0, T-1], \forall (i, j) \in E_{life} \tag{23}
$$

$$
\sum_{i=1}^{n}\sum_{k=0}^{\lfloor\frac{T-1}{\lambda}\rfloor} v_i^{t+k\lambda} \leq R \quad \forall t \in [0, \lambda-1] \tag{24}
$$

$$
v_i^t \in \{0, 1\} \quad \forall i \in [1, n], \forall t \in [0, T-1] \tag{25}
$$

To complete the modulo RCISP formulation, we minimize the schedule date of operation O_{n+1}, we adapt the dependence inequalities (3) to the dependence latencies of $\theta_i^j - \lambda\omega_i^j$ and we eliminate the register pressure variables v_i^t from (24), thanks to the equalities (23). This yields:

$$
\begin{aligned}
\text{minimize } \sum_{t=1}^{T-1} t\, x_{n+1}^t \quad &: && && (26)\\
\sum_{t=0}^{T-1} x_i^t \quad &= \quad 1 && \forall i \in [1, n+1] && (27)\\
\sum_{s=t}^{T-1} x_i^s + \sum_{s=0}^{t+\theta_i^j-\lambda\omega_i^j-1} x_j^s \quad &\leq \quad 1 && \forall t \in [0, T-1], \forall (i,j) \in E_{dep} && (28)\\
\sum_{i=1}^{n} \sum_{k=0}^{\lfloor \frac{T-1}{\lambda} \rfloor} \sum_{s=t+k\lambda-p_i+1}^{t+k\lambda} x_i^s\, \vec{b}_i \quad &\leq \quad \vec{B} && \forall t \in [0, \lambda-1] && (29)\\
\sum_{(i,j)\in E_{life}} \sum_{k=0}^{\lfloor \frac{T-1}{\lambda} \rfloor} \left(\sum_{s=0}^{t+k\lambda} x_i^s - \sum_{s=0}^{t+k\lambda-\lambda\omega_i^j} x_j^s \right) \quad &\leq \quad R && \forall t \in [0, \lambda-1] && (30)\\
x_i^t \quad &\in \quad \{0,1\} && \forall i \in [1, n+1], \forall t \in [0, T-1] && (31)
\end{aligned}
$$

Summary and Conclusions

The first part of this report surveys the results and techniques of deterministic machine scheduling that are relevant to instruction scheduling, including:

- The three-field notation of machine scheduling problems, the parallel machine scheduling problems and the resource-constrained project scheduling problems (RCPSP), along with their main complexity results.
- The differences between the Graham list scheduling algorithm (GLSA) and the job-based list scheduling algorithm, the performances guarantees of the GLSA and the conditions for the GLSA to build minimum length schedules.
- The polynomial-time scheduling algorithm of Leung, Palem and Pnueli [49] for the unit execution time (UET) parallel machine scheduling problems with release dates and deadlines, which is relevant to instruction scheduling.
- The time-indexed formulation of the RCPSP, which enables their exact resolution by means of integer linear programming.

The second part of this report discusses the extensions and the specializations of VLIW instruction scheduling problems compared to the RCPSP, in particular:

- The variants of the register instruction scheduling problems: minimize the register lifetimes, minimize the register pressure, constrain the register pressure and combine the pre-pass instruction scheduling with register allocation.
- The VLIW instruction scheduling problem is defined as a RCPSP with UET operations, cumulative resources and non-negative dependence latencies. We introduce the block instruction scheduling problem and the pipeline instruction scheduling problem.

- The main results and techniques of cyclic scheduling, periodic scheduling and modulo scheduling. For modulo scheduling, we contrast the classic modulo scheduling framework [68] with the decomposed software pipelining techniques [71].
- The time-indexed formulation of the minimum register pressure modulo scheduling problem introduced by Eichenberger, Davidson and Abraham [25, 27].

The third part of this report presents our contributions to the field of instruction scheduling, in particular modulo scheduling:

- The equations that enable cumulative register lifetimes minimization and an efficient way of solving them as a maximum-cost network flow problem.
- Some of the resource modeling issues that arise when defining instruction scheduling problems for a particular processor.
- The modulo insertion scheduling theorems, which allow for the transformation of a partial modulo schedule by increasing its period without violating the dependence constraints or the modulo resource constraints of the already scheduled operations.
- The application of these results to the LAO instruction scheduler / software pipeliner we developed for the Cray T3E computers, improved for the STMicroelectronics ST100 [22] and which is now in use in the STMicroelectronics ST200 production compiler.
- A new time-indexed formulation for the modulo register-constrained instruction scheduling problem. The motivation for this formulation is to enable the application of the modern machine scheduling techniques for the resolution of the large-scale RCPSP [16].

To conclude, we anticipate that the techniques of machine scheduling will become increasingly relevant to VLIW instruction scheduling, in particular for the embedded computing applications. In such applications, high-performances of the compiled applications enable the reduction of the system costs, whereas the compilation time is not really constrained. Embedded computing also yields instruction scheduling problems of significant size, especially for media processing.

Embedded computing thus motivates the development of advanced instruction scheduling techniques that deliver results close to optimality, rather than accepting the observed limitation of the traditional list-based instruction scheduling [12, 72]. An equally promising area of improvement of VLIW instruction scheduling is its full integration with the register allocation. In this area, the Multiflow Trace Scheduling technology [53] is unchallenged.

Fortunately, modern embedded media processors, in particular VLIW processors such as the STMicroelectronics ST200 family, present a micro-architecture that allows for an accurate description using resources like in cumulative scheduling problems. This situation is all the more promising in cases of processors with clean pipelines like the ST200 family, as the resulting instruction scheduling problems only involve UET operations. Scheduling problems with cumulative resources and UET operations benefit from stronger machine scheduling relaxations, easier resource management in modulo scheduling and simpler time-indexed formulations.

With regards to the already compiler-friendly STMicroelectronics ST220 processor, the VLIW instruction scheduling problems it involves could be better solved if all its operation requirements on the cumulative resources were 0 or 1. This is mostly the case, except for the operations with immediate extensions, which require two issue slots (see Figure 2). Besides increasing the peak performances, removing this restriction enables the relaxation of the VLIW instruction scheduling problems into parallel machine scheduling problems with UET, problems which are optimally solved in polynomial time in several cases by the algorithm of Leung, Palem and Pnueli [49].

References

[1] S. G. Abraham, V. Kathail, B. L. Dietrich: *Meld Scheduling: A Technique for Relaxing Scheduling Constraints*. International Journal on Parallel Programming, 26, 4, 1998.

[2] A. Aiken, A. Nicolau, S. Novack: *Resource-Constrained Software Pipelining*. IEEE Transactions on Parallel and Distributed Systems, 6, 12, 1996.

[3] R. K. Ahuja, T. L. Magnanti, J. B. Orlin: *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

[4] J. M. van den Akker, C. A. J. Hurkens, M. W. P. Savelsbergh: *Time-Indexed Formulations for Single-machine Scheduling problems: Column Generation*. INFORMS Journal on Computing, 12, 2, 2000.

[5] T. Bidault, C. Guillon, F. Bouchez, F. Rastello: *Procedure Placement Using Temporal Ordering Information: Dealing with Code Size Expansion*. 2004 International Conference on Compilers, Architectures and Synthesis of Embedded Systems – CASES'04, Washinton, USA, 2004.

[6] F. Bodin, F. Charot: *Loop Optimization for Horizontal Microcoded Machines*. Proceedings of the 1990 International Conference on Supercomputing, Amsterdam 1990.

[7] P. Brucker, A. Drexl, R. Möhring, K. Neumann, E. Pesch: *Resource-Constrained Project Scheduling: Notation, Classification, Models, and Methods*. European Journal of Operational Research 112, 1999.

[8] P. Brucker: *Scheduling Algorithms -4th ed.*. Springer Verlag, 2004.

[9] D. Callahan, S. Carr, K. Kennedy: *Improving Register Allocation for Subscripted Variables*. Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation – PLDI'90, June 1990.

[10] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, W-M. W. Hwu: *The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors*. IEEE Transactions on Computers, 44, 3, March 1995,

[11] N. Christofides, R. Alvarez-Valdés, J. M. Tamarit: *Project Scheduling with Resource Constraints: A Branch and Bound Approach*. European Journal of Operational Research 29, 1987.

[12] K.D. Cooper, P.J. Schielke, D. Subramanian: *An Experimental Evaluation of List Scheduling*. TR98-326, Dept. of Computer Science, Rice University, September 1998.

[13] A. Darte, G. Huard: *Loop Shifting for Loop Compaction*. International Journal of Parallel Programming, v.28 n.5, p.499-534, Oct. 2000.

[14] A. Dasdan, S. S. Irani, R. K. Gupta: *Efficient Algorithms for Optimum Cycle Mean and Optimum Cost to Time Ratio Problems*. ACM IEEE Design Automation Conference – DAC'1999, 1999.

[15] J. W. DAVIDSON, S. JINTURKAR: *Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation* . Proceedings of the 28th Annual international Symposium on Microarchitecture – MICRO-28, December 1995.

[16] S. DEMASSEY, C. ARTIGUES, P. MICHELON: *Comparing Lower Bounds for the RCPSP Under a Same Hybrid Constraint-linear Programming Approach*. International Conference on Constraint Programming – CP'01, Proceedings of the Workshop on Cooperative Solvers in Constraint Programming – CoSolv'01, 2001.

[17] B. DUPONT DE DINECHIN: *An Introduction to Simplex Scheduling*. 1994 International Conference on Parallel Architecture and Compiler Techniques – PACT'94, 1994.

[18] B. DUPONT DE DINECHIN: *Insertion Scheduling: An Alternative to List Scheduling for Modulo Schedulers*. 8th International Workshop on Languages and Compilers for Parallel Computing – LCPC'95, LNCS #1033, Colombus, Ohio, Aug. 1995.

[19] B. DUPONT DE DINECHIN: *Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates*. 9th International Workshop on Languages and Compilers for Parallel Computing – LCPC'96, LNCS #1239, San Jose, California, Aug. 1996.

[20] B. DUPONT DE DINECHIN: *A Unified Software Pipeline Construction Scheme for Modulo Scheduled Loops*. PaCT'97 – 4th International Conference on Parallel Computing Technologies, LNCS #1277, Sep. 1997.

[21] B. DUPONT DE DINECHIN: *Extending Modulo Scheduling with Memory Reference Merging*. CC'99 – 8th International Conference on Compiler Construction, LNCS #1575, March 1999.

[22] B. DUPONT DE DINECHIN, F. DE FERRIÈRE, C. GUILLON, A. STOUTCHININ: *Code Generator Optimizations for the ST120 DSP-MCU Core*. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems – CASES, Nov. 2000.

[23] B. DUPONT DE DINECHIN: *Modulo Scheduling with Regular Unwinding*. 2nd International Workshop on Discrete Optimization Methods in Production and Logistics – DOM-2004, 2004. `http://www.cri.ensmp.fr/classement/2004.html`

[24] K. ECKER: *Scheduling Theory.*, 2003.

[25] A. E. EICHENBERGER, E. S. DAVIDSON, S. G. ABRAHAM: *Optimum Modulo Schedules for Minimum Register Requirements*. International Conference on Supercomputing – ICS, 1995.

[26] A. E. EICHENBERGER, E. S. DAVIDSON: *Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule*. Proceedings of the 28th Annual Symposium on Microarchitecture – MICRO-28, 1995.

[27] A. E. EICHENBERGER, E. S. DAVIDSON: *Efficient Formulation for Optimal Modulo Schedulers*. SIGPLAN Conference on Programming Language Design and Implementation – PLDI'97, June 1997.

[28] C. EISENBEIS, D. WINDHEISER:, *Optimal Software Pipelining in Presence of Resource Constraints*. Proceedings of the International Conference on Parallel Computing Technologies – PaCT'93, 1993.

[29] P. FARABOSCHI, G. BROWN, J. A. FISHER, G. DESOLI, F. HOMEWOOD: *Lx: a Technology Platform for Customizable VLIW Embedded Processing*. 27th Annual International Symposium on Computer Architecture – ISCA'00, June 2000.

[30] P. FEAUTRIER: *Fine-Grain Scheduling under Resource Constraints*. Languages and Compilers for Parallel Computing – LCPC'94, 1994.

[31] D. FIMMEL, J. MÜLLER: *Optimal Software Pipelining Under Register Constraints*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA, volume V Las Vegas, June 2000.

[32] D. FIMMEL, J. MÜLLER: *Optimal Software Pipelining with Rational Initiation Interval*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA, volume II, Las Vegas, June 2002.

[33] G. FINKE, V. GORDON, J.-M. PROTH: *Scheduling with Due Dates (Annotated Bibliography of Complexity and Algorithms)*. Les cahiers du laboratoire Leibniz, Jan. 2002. `http://www-leibniz.imag.fr/ LEIBNIZ/LesCahiers/2002/Cahier42/`

[34] F. GASPERONI, U. SCHWIEGELSHOHN: *Generating Close to Optimum Loop Schedules on Parallel Processors*. Parallel Processing Letters 4: 391-403, 1994.

[35] D. GOLDFARB, J. HAO, S-R. KAI: *Shortest Path Algorithms Using Dynamic Breadth-First Search*. Networks, vol. 21, pp. 29–50, 1991.

[36] J. R. GOODMAN, W-C. HSU: *Code Scheduling and Register Allocation in Large Basic Blocks*. Proceedings of the International Conference on Supercomputing – ICS'88, 1988.

[37] R. GOVINDARAJAN, E. R. ALTMAN, G. R. GAO: *Minimizing Register Requirements Under Resource-Constrained Rate-Optimal Software Pipelining*. Proceedings of the 27th Annual International Symposium on Microarchitecture – MICRO-27, December 1994.

[38] C. HANEN, A. MUNIER: *Cyclic Scheduling on Parallel Processors: an Overview*. Scheduling Theory and its Applications, Wiley & Sons, 1994.

[39] C. HANEN, A. MUNIER: *A Study of the Cyclic Scheduling Problem on Parallel Processors*. DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science, 57, 1995.

[40] R. HANK, S. MAHLKE, R. BRINGMANN, J. GYLLENHAAL, W. HWU: *Superblock Formation Using Static Program Analysis*. 26th International Symposium on Micro-architecture – MICRO-26, 1993.

[41] L. J. HENDREN, G. R. GAO, E. R. ALTMAN, C. MUKERJI: *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*. The Journal of Programming Languages, 1(3):155–185, Sept. 1993.

[42] R. A. HUFF: *Lifetime-Sensitive Modulo Scheduling*. SIGPLAN Conference on Programming Language Design and Implementation – PLDI'93, June 1993.

[43] C.W. KESSLER: *Scheduling Expression DAG for Minimal Register Need*. Computer Languages, vol. 24(1), 1998.

[44] P. M. KOGGE: *The Architecture of Pipelined Computers*. New York, N.Y.: McGraw-Hill, 1981.

[45] R. KOLISCH: *Serial and Parallel Resource-Constrained Project Scheduling Methods Revisited: Theory and Computation*. European Journal of Operational Research, 90 (2): 320-333, 1996.

[46] R. KOLISCH, S HARTMANN: *Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*. Project Scheduling: Recent Models, Algorithms and Applications, Kluwer, 1999.

[47] M. LAM: *Software Pipelining: an Effective Scheduling Technique for VLIW Machines*. SIGPLAN Conference on Programming Language design and Implementation – PLDI'88, June 1988.

[48] D. M. LAVERY, W-M. W. HWU: *Unrolling-Based Optimizations for Modulo Scheduling*. Proceedings of the 28th Annual Symposium on Microarchitecture – MICRO-28, 1995.

[49] A. LEUNG, K. V. PALEM, A. PNUELI: *Scheduling Time-Constrained Instructions on Pipelined Processors*. ACM TOPLAS 23, 1, Jan. 2001.

[50] J. LLOSA, M. VALERO, E. AYGUADÉ AND A. GONZÁLEZ: *Hypernode Reduction Modulo Scheduling*. 28th International Symposium on Microarchitecture – MICRO-28, December 1995.

[51] J. LLOSA, A. GONZÁLEZ, E. AYGUADÉ, M. VALERO: *Swing Modulo Scheduling: a Lifetime-Sensitive Approach*. Conference on Parallel Architectures and Compilation Techniques – PACT'96, October 1996.

[52] J. LLOSA, S. M. FREUDENBERGER: *Reduced Code Size Modulo Scheduling in the Absence of Hardware Support*. Proceedings of the 35th annual ACM/IEEE international Symposium on Microarchitecture – MICRO-35, 2002.

[53] P. G. LOWNEY, S. M. FREUDENBERGER, T. J. KARZES, W. D. LICHTENSTEIN, R. P. NIX, J. S. O'DONNELL, J. C. RUTTENBERG: *The Multiflow Trace Scheduling Compiler*. The Journal of Supercomputing, 7, 1-2, 1993.

[54] S. A. MAHLKE, R. E. HANK, J. E. MCCORMICK, D. I. AUGUST, W-M. W. HWU: *A Comparison of Full and Partial Predicated Execution Support for ILP Processors*. Proceedings of the 22nd International Symposium on Computer Architecture – ISCA'95, June 1995

[55] A. MUNIER, M. QUEYRANNE, A. S. SCHULZ: *Approximation Bounds for a General Class of Precedence Constrained Parallel Machine Scheduling Problems*. 6th International Integer Programming and Combinatorial Optimization – IPCO, 1998.

[56] B. NATARAJAN, M. S. SCHLANSKER: *Spill-Free Parallel Scheduling of Basic Blocks*. Proceedings of the 28th Annual International Symposium on Microarchitecture – MICRO-28, 1995.

[57] Q. NING, G. R. GAO: *A Novel Framework of Register Allocation for Software Pipelining*. SIGPLAN Symposium on Principles of Programming Languages – PLDI'93, Jan. 1993.

[58] *Open64 Compiler Tools*. `http://open64.sourceforge.net/`

[59] K. V. PALEM, B. SIMONS: *Scheduling Time-Critical Instructions on RISC Machines*. ACM Transactions on Programming Languages and Systems (TOPLAS) 15, 4, pages 632–658, Sept. 1993.

[60] C. PAPADIMITRIOU, M. YANNAKAKIS: *Scheduling Interval-Ordered Tasks*. SIAM Journal of Computing, 8, 3,pages 405–409, 1979.

[61] K. PETTIS, R. C. HANSEN: *Profile Guided Code Positioning*. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation – PLDI'90, June 1990.

[62] A. A. B. PRITSKER, L. J. WATTERS, P. M. WOLFE: *Multi-Project Scheduling with Limited Resources: A Zero-One Programming Approach*. Management Science 16, 1969.

[63] S. RAJAGOPALAN, M. VACHHARAJANI, S. MALIK: *Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints*. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems – CASES, November 2000.

[64] S. RAJAGOPALAN, S. P. RAJAN, S. MALIK, S. RIGO, G. ARAUJO: *A Re-targetable VLIW Compiler Framework for DSPs with Instruction Level Parallelism*. IEEE Transactions on Computer-Aided Design, 20, 11, November 2001.

[65] B. R. RAU, C. D. GLAESER: *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing*. 14th Annual Workshop on Microprogramming – MICRO-14, Dec. 1981.

[66] B. RAU, M. LEE, P. TIRUMALAI, P. SCHLANSKER: *Register Allocation for Software Pipelined Loops*. *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation – PLDI'92.*, June 1992.

[67] B. R. RAU, M. S. SCHLANSKER, P. P. TIRUMALAI: *Code generation Schemas for Modulo Scheduled Loops*. Proceedings of the 25th Annual International Symposium on Microarchitecture – MICRO-25, December 1992.

[68] B. R. RAU: *Iterative Modulo Scheduling*. The International Journal of Parallel Processing, 24, 1, Feb 1996.

[69] J. RUTTENBERG, G. R. GAO, A. STOUTCHININ, W. LICHTENSTEIN: *Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler*. SIGPLAN Conference on Programming Language Design and Implementation – PLDI'96, May 1996.

[70] M. W. P. SAVELSBERGH, R. N. UMA, AND J. WEIN: *An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems*. Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 1998.

[71] J. WANG, C. EISENBEIS, M. JOURDAN, B. SU: *Decomposed Software Pipelining*. International Journal of Parallel Programming, 22(3), 1994.

[72] K. WILKEN, J. LIU, M. HEFFERNAN: *Optimal Instruction Scheduling Using Integer Programming*. Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation – PLDI'00, 2000.

[73] J. ZALAMEA, J. LLOSA, E. AYGUADÉ, M. VALERO: *MIRS: Modulo Scheduling with Integrated Register Spilling*. Proceedings of the 14th Annual Workshop on Languages and Compilers for Parallel Computing – LCPC'01, August 2001.