From Machine Scheduling to VLIW Instruction Scheduling

Benoît Dupont de Dinechin

bd3@cri.ensmp.fr

Abstract

This report discusses the similarities and the differences between machine scheduling problems, and instruction scheduling problems on modern VLIW processors 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 theoretical contributions to the field of instruction scheduling that are applied in the STMicroelectronics ST200 production compiler, and we introduce a new time-indexed formulation for the register constrained instruction scheduling problems.

Introduction

Instruction scheduling is a performance critical optimization when compiling applications on 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 transform this problem into 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 relationships with the field of machine scheduling are 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 instance, these results assume simple precedence constraints on the order of operations in a schedule, instead of dependences with a delay like in pipelined processors, and only consider machine models where each operation uses one of m identical machines 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 processors such as the STMicroelectronics ST200 series[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 expose our contributions to VLIW instruction scheduling that have been implemented in several production compilers, in particular the STMicroelectronics ST200 production compiler. We also present a new time-indexed formulation for the register-constrained instruction scheduling problems (RCISP), whose efficient resolution is a topic of our research.

[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, there are m identical machines available to process n given operations. In order to complete, each operation O_i requires one machine for a *processing time* of p_i cycles (time units), subject to *resource constraints*: no more than m machines are simultaneously processing operations at any time. Our focus is the *non-preemptive* machine scheduling problems, where any operation is processed from its *schedule date* σ_i for p_i consecutive cycles on the same machine without any interruption. In *preemptive* machine scheduling problems, the processing of any operation can be preempted, to be resumed later possibly on another machine.

In addition to the resource constraints, operations in parallel machine scheduling problems must satisfy *temporal constraints* on their schedule dates σ_i. A *release date* $r_i \geq 0$ enforces the constraint $r_i \leq \sigma_i$. A *deadline date* d_i enforces the constraint $\sigma_i + p_i \leq d_i$. Let C_i denote the *completion date* of an operation O_i. In non-preemptive scheduling, the completion date is normally defined as $C_i \overset{\text{def}}{=} \sigma_i + p_i$. When *processing tails* $q_i \geq 0$ are given instead of deadlines, the completion date is defined as $C_i \overset{\text{def}}{=} \sigma_i + p_i + q_i$. Temporal constraints may also include *precedence constraints*, specified by a partial order between the operations. A precedence between two operations O_i and O_j constrains O_i to complete before O_j starts, that is, $\sigma_i + p_i \leq \sigma_j$.

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

Machine scheduling problems are denoted by a triplet notation $\alpha|\beta|\gamma$, where α describes the processing environment, β specifies the operation properties, and γ defines the optimization criterion [7]. For the parallel machine scheduling problems, the common values of α, β, γ are:

α : 1 for single machine, P for parallel machines, Pm for the given m parallel machines.

β : $prec$ for precedence constraints, r_i for release dates, q_i for processing tails, d_i for deadlines.

γ : either the *makespan* C_{max}, or the *maximum lateness* $L_{max} \overset{\text{def}}{=} \max_{i=1}^{n} L_i : L_i \overset{\text{def}}{=} \sigma_i + p_i - d_i$.

The main complexity results of parallel machine scheduling [8, 69, 47] are summarized in table 1 for the single machine problems, in table 2 for the dual parallel machine problems, and in table 3 for the general parallel machine problems. In these tables, **PTS** stands for polynomial-time solvable, and **NPH** for *NP*-hard. The meaning of the various β fields is as follows:

$p_i = 1$ All the processing times are 1; this property is referred to as *unit execution time* (UET).

$p_i = p$ All the processing times have the same constant value p.

$prec(l_i^j)$ Precedence constraints with time-lags l_i^j: $O_i \rightarrow O_j$ implies $\sigma_i + p_i + l_i^j \leq \sigma_j$.

$prec(l_i^j = l)$ All the l_i^j have the same constant value l; this applies to pipelined processing.

$intree$ The precedence constraints are structured as an in-tree.

$outtree$ The precedence constraints are structured as an out-tree.

$chains$ The precedence constraints are structured as independent chains.

$intorder(mono\ l_i^j)$ The precedence constraints are structured as monotone interval order (see §1.4).

Problem	Result	Complexity
$1\|prec;r_i\|C_{max}$	Lawler 1973	PTS
$1\|p_i = p;prec;r_i\|L_{max}$	Simons 1978	PTS
$1\|p_i = p;prec(l_i^j = 1);r_i\|L_{max}$	Bruno et al. 1980	PTS
$1\|p_i = 1;prec\|L_{max}$	Garey et al. 1981	PTS
$1\|prec(l_i^j = 1)\|C_{max}$	Finta & Liu 1996	PTS
$1\|prec(l_i^j \in \{0,1\})\|C_{max}$	Leung et al. 2001	PTS
$1\|p_i = 1;prec(l_i^j \in \{0,1\});r_i\|L_{max}$	Leung et al. 2001	PTS
$1\|r_i\|L_{max}$	Lenstra et al. 1977	NPH
$1\|p_i = 1;prec(l_i^j)\|C_{max}$	Hennessy & Gross 1983	NPH
$1\|p_i = 1;prec(l_i^j = l)\|C_{max}$	Leung et al. 1984	NPH
$1\|p_i = 1;intree(l_i^j = l);r_i\|C_{max}$	Brucker & Knust 1998	NPH
$1\|p_i = 1;outtree(l_i^j = l)\|L_{max}$	Brucker & Knust 1998	NPH

Table 1: Complexity results for the single machine scheduling problems.

Problem	Result	Complexity
$P2\|p_i = 1;prec\|C_{max}$	Coffman & Graham 1972	PTS
$P2\|p_i = 1;prec\|L_{max}$	Garey & Johnson 1976	PTS
$P2\|p_i = 1;prec;r_i\|L_{max}$	Garey & Johnson 1977	PTS
$P2\|p_i = 2;chains;c_k\|C_{max}$	Timkovski 2000	PTS
$P2\|p_i = 1;prec(l_i^j \in \{-1,0\});r_i\|L_{max}$	Leung et al. 2001	PTS
$P2\|\|C_{max}$	Karp 1972	NPH
$P2\|p_i \in \{1,2\};prec\|C_{max}$	Ullman 1975	NPH
$P2\|chains\|C_{max}$	Du et al. 1991	NPH

Table 2: Complexity results for the dual parallel machine scheduling problems.

In these complexity results, all the PTS parallel machine scheduling problems that include precedence constraints, either have a constant processing time ($p_i = p$), or are UET ($p_i = 1$). The UET property is especially important in practice, as it allows the results of preemptive scheduling to apply to non-preemptive scheduling. In particular, the **Preemption Chaining Theorem** of Timkovski [69], which describes the effects of transforming each operation O_i into a chain of p_i UET operations, implies that preemption is redundant on UET problems with precedence constraints for many formulations of γ, including C_{max} and L_{max}.

1.2 Extensions and Relaxations of Machine Scheduling Problems

A first class of extensions of the machine scheduling problems results from associating delays d_i^j with the precedence constraints, yielding *general temporal constraints* [7], also called *dependence constraints*. Precisely, if O_i and O_j are two dependent operations, then in any feasible schedule $\sigma_i + d_i^j \leq \sigma_j$. That is, the minimum delay between the start of O_i and O_j is at least d_i^j, a value that is not related to p_i or p_j. By introducing the dummy operations O_0 and O_{n+1}, release dates r_i and processing tails q_i can be expressed as dependences between O_0, O_i, and O_{n+1} by using the delays $d_0^i = r_i$ and $d_i^{n+1} = p_i + q_i$. The deadlines d_i can also be expressed as dependences between O_i and

Problem	Result	Complexity
$P\|p_i = p; tree\|C_{max}$	Hu 1961	PTS
$P\|p_i = p; outtree; r_i\|C_{max}$	Brucker et al. 1977	PTS
$P\|p_i = p; intree\|L_{max}$	Brucker et al. 1977	PTS
$P\|p_i = 1; intree(l_i^j = l)\|L_{max}$	Bruno et al. 1980	PTS
$P\|p_i = 1; outtree(l_i^j = l); r_i\|C_{max}$	Bruno et al. 1980	PTS
$P\|p_i = p; r_i\|L_{max}$	Simons 1983	PTS
$P\|p_i = 1; intorder(mono\ l_i^j)\|L_{max}$	Palem & Simons 1993	PTS
$P\|p_i = 1; intorder(mono\ l_i^j); r_i\|L_{max}$	Leung et al. 2001	PTS
$P\|p_i = 1; chains; r_i\|L_{max}$	Baptiste et al. 2002	PTS
$P\|p_i = 1; prec\|C_{max}$	Ullman 1975	NPH
$P\|p_i = 1; intree; r_i\|C_{max}$	Brucker et al. 1977	NPH
$P\|p_i = 1; outtree\|L_{max}$	Brucker et al. 1977	NPH
$P\|\|C_{max}$	Garey & Johnson 1978	NPH

Table 3: Complexity results for the general parallel machine scheduling problems.

O_0, by using the (negative) delays $d_i^0 = p_i - d_i$.

Depending on authors and context, the dependence delays d_i^j can be replaced by the finish to start *time-lags* l_i^j, defined such that $p_i + l_i^j = d_i^j$. In all cases, the general temporal constraints can be represented by an weighted graph named *activity-on-node graph*, *potential-task graph*, or *dependence graph*, where each node represents an operation, where there is an arc for each dependence between O_i and O_j, and where each arc weight is the corresponding dependence delay d_i^j. In this text we use the dependence graph terminology.

A second class of extensions of the machine scheduling problems is known as the *resource-constrained project scheduling problems* (RCPSP) [7]. In a RCPSP, the m parallel machines are replaced by a set of *renewable resources* or *cumulative resources*, which are given by a vector $\vec{B}$ of integral resource availabilities. Each operation O_i is also associated with an integral vector $\vec{b}_i$ of resource requirements. The resource constraints become: $\forall t, \sum_{i \in I_t} \vec{b}_i \leq \vec{B}$, with $I_t \overset{\text{def}}{=} \{1 \leq i \leq n : \sigma_i \leq t < \sigma_i + p_i\}$. That is, the total requirements on the cumulative resources by all the operations executing at any given time must not be greater than the resource availabilities $\vec{B}$.

A *relaxation* is a simplified version of a given scheduling problem, such that if the relaxation is infeasible (has no feasible schedules), then the original problem is infeasible too. Relaxations are typically obtained by allowing preemption, by assuming a particular structure of the precedence constraints, by ignoring some resources, or by dropping the integrality 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.

The most aggressive yet useful relaxation of machine scheduling problems is to ignore all 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 [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.

Problem	Result	Performance Ratio
$P\|prec\|C_{max}$	Graham 1966	$2 - \frac{1}{m}$
$P\|p_i = 1; prec(l_i^j = l)\|C_{max}$	Palem & Simons 1990	$2 - \frac{1}{m(l+1)}$
$1\|prec(l_i^j)\|C_{max}$	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 $\rho \overset{\text{def}}{=} \frac{\max l_j^k}{\min p_i}$.

Table 4: Performance guarantees of the GLSA with arbitrary priority.

The *cumulative scheduling problems* (CuSP) are relaxations of the RCPSP obtained by focusing on a single resource at a time, by assuming an upper bound D on the schedule length, and by summarizing the temporal constraints into release dates r_i and processing tails q_i [5]. This summary is computed from the dependence graph by taking the earliest start dates for the r_i, and by adding p_i to the latest start dates for the q_i. The result is a scheduling problem $P|b_i, r_i, q_i|C_{max}$ that is still *NP*-hard, but which can itself be relaxed in several effective ways [10].

1.3 List Scheduling Algorithms and List Scheduling Priorities

The *list scheduling algorithms* (LSA) are the workhorses of machine scheduling [44], 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 directed cycles[3]. Precisely, a LSA is a greedy scheduling heuristic where the operations are ranked by priority order in a list. Greedy scheduling means that the machines or resources are kept busy as long as there are available operations to schedule. Two variants of list scheduling must be distinguished [54]:

Graham List Scheduling (parallel schedule generation scheme [44]) Scheduling is performed by scanning the time slots in increasing order. For each time slot, if a machine is idle, schedule the highest priority operation available at this time.

Job-Based List Scheduling (serial schedule generation scheme [44]) 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 on the quality of the schedules they build [43]:

- The job-based LSA generates *active schedules*, where none of the activities can be started earlier without delaying some other activity.
- The Graham LSA constructs *non-delay schedules*, where, even if activity preemption is allowed, none of the activities can be started earlier without delaying some other activity.
- The set of non-delay schedules is a subset of the set of active schedules. It has the drawback that it might not contain and optimal schedule with a regular performance measure (like C_{max} or L_{max}).

[3]For cyclic dependence graphs, no polynomial time algorithm is guaranteed to build a schedule unless *P* = *NP*.

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

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

Table 5: Performance guarantees of the GLSA with a specific priority.

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 [54], 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 4. 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 §2.1). 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 5. The main observation here is 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 of operation O_i is a date d'_i such that $\sigma_i + p_i \leq d'_i$ in any feasible schedule. For instance, in case of Hu's algorithm, since the precedences are a in-tree, and thanks to the UET operations, the highest in-tree level first is also a modified deadlines priority.

1.4 The Scheduling Algorithm of Leung, Palem, and Pnueli

The idea of using the modified deadlines as the priorities of a GLSA culminates in recent work by Leung, Palem, and Pnueli [47]. In order to compute the modified deadlines, Leung et al. apply a technique of *backward scheduling* [58], where a series of relaxations are optimally solved by list-scheduling in order to find, for each operation, its latest schedule date such that the relaxations are feasible. Precisely, the implementation of backward scheduling is as follows:

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

- For each scheduling sub-problem S_i, 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 such p, define the modified deadline of O_i as $d'_i \overset{\text{def}}{=} p + 1$. 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 delays 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 a simpler scheduling problem $P|p_i = 1; r_i|L_{max}$.

- Optimally solve this $P|p_i = 1; r_i|L_{max}$ relaxation using a GLSA with the earliest deadlines first priority (Jackson's rule). This gives the feasibility status of the relaxation.

Thanks to a fast list scheduling technique based on union-find data-structures, solving a $P|p_i = 1; r_i|L_{max}$ relaxation takes $O(n\alpha(n))$ time[4]. 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 delay computation. So the total time complexity of backward scheduling is $O(n^2 \log n\alpha(n) + ne)$.

Once the backward scheduling procedure has computed the modified deadlines d'_i, running a GLSA with the earliest modified deadlines first as priorities builds the final schedule. Because the 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 by Leung, Palem, and Pnueli, are the proofs that the feasible schedules computed this way are in fact optimal for all the cases reported in table 5, and the unification of many earlier modified deadlines techniques under a single framework [47]. On the implementation side, besides introducing the technique of fast list scheduling in $O(n\alpha(n))$ time for solving the $P|p_i = 1; r_i|L_{max}$ relaxations, they prove that iterating once over the operations in backward topological order of the precedence constraints yields the same set of modified deadlines as the brute-force approach of iterating until stabilization.

Among the scheduling problems that are optimally solved by the algorithm of Leung, Palem, and Pnueli, $P|p_i = 1; intorder(mono\ l_i^j); r_i|L_{max}$ introduces precedence constraints structured as monotone interval order. An *interval order* is a partial order $(V, \prec)$, such that for every $v \in V$, there is a corresponding interval $I(v)$ on the real line such that $v_1 \prec v_2 \Leftrightarrow \forall x \in I(v_1), \forall y \in I(v_2) : x < y$. Interval orders have the property that, given any $v_1, v_2 \in V$, either the predecessors of v_1 are included in the predecessors of v_2, or the predecessors of v_2 are included in the predecessors of v_1 [58]. A *monotone interval order* is a weighted graph whose arcs define an interval order, and whose arc weights are such that if (u, v_1) and (u, v_2) are arcs, then $w(u, v_1) \leq w(u, v_2)$ whenever the predecessors of v_1 are included in the predecessors of v_2.

1.5 The Time-Indexed RCPSP Formulation and its Min-Cut Relaxation

Due to its ability to fully describe the RCPSP with general temporal constraints and other extensions, the *non-preemptive time-indexed formulation* introduced by Pritsker, Watters and Wolfe in 1969 [60] 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 $x_i^t \overset{\text{def}}{=} 1$ if $\sigma_i = t$, else $x_i^t \overset{\text{def}}{=} 0$. In particular $\sigma_i = \sum_{t=1}^{T-1} t x_i^t$. In case of C_{max} minimization, this formulation is as follows:

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

[4]Here $\alpha(n)$ is the inverse Ackermann function, which is in practice lower than a small constant.

$$\begin{aligned}
\sum_{t=0}^{T-1} x_i^t &= 1 \quad \forall i \in [1, n+1] && (2)\\
\sum_{s=t}^{T-1} x_i^s + \sum_{s=0}^{t+d_i^j-1} x_j^s &\leq 1 \quad \forall t \in [0, T-1], \forall (i,j) \in E_{dep} && (3)\\
\sum_{i=1}^{n} \sum_{s=t-p_i+1}^{t} x_i^s \, \vec{b}_i &\leq \vec{B} \quad \forall t \in [0, T-1] && (4)\\
x_i^t &\geq 0 \quad \forall i \in [1, n+1], \forall t \in [0, T-1] && (5)\\
x_i^t &\in \mathbb{Z} \quad \forall i \in [1, n+1], \forall t \in [0, T-1] && (6)
\end{aligned}$$

The equations (2) constrain any operation to be scheduled once. The inequalities (3) enforce the dependence delay constraints with T inequalities per dependence, as proposed by Christofides et al. [12]. While the original formulation [60] introduced only one inequality per dependence $\sum_{t=0}^{T-1} t(x_j^t - x_i^t) \geq d_i^j \Leftrightarrow \sigma_j - \sigma_i \geq d_i^j$, it was reported to be significantly more difficult to solve in practice. Last, the inequalities (4) enforce the cumulative resource constraints for $p_i \geq 1$. Extension of the RCPSP where the resource availabilities and requirements are time-dependent can also be described in this formulation by generalizing (4) into (7):

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

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. [68] 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].

In order to solve the large-scale RCPSP instances, Möhring et al. [53] work from the Lagrangian relaxation of the time-indexed RCPSP formulation where the resource constraints are dualized, as proposed by Christofides et al. [12]. That is, the inequalities (4) are removed from the constraints, and the corresponding equalities are added into the objective, weighted by the Lagrangian multipliers $\lambda_k^t \geq 0$. Each set of Lagrangian multipliers yields a Lagrangian sub-problem (2), (3), (5), (6), with the objective $\sum_{j=0}^{n+1} \sum_{t=0}^{T-1} w_i^t x_i^t$, where the w_i^t are obtained from the λ_k^t and the $\vec{b}_i$.

The insight of Möhring et al. [53] is that the Lagrangian sub-problems are integer programs solvable by minimum-cut over a graph of $O(nT)$ nodes and $O(eT)$ arcs in $O(neT^2 \log(n^2T/e))$ time with a network flow algorithm. Using a standard sub-gradient method that solves a series of Lagrangian sub-problems, near-optimal Lagrangian multiplier values λ_k^t are obtained. Because the polytope (2), (3), (5), is integral, this Lagrangian relaxation is not stronger than the linear programming relaxation (1), (2), (3), (4), (5). However this Lagrangian relaxation is much easier to solve in practice, even when solving each Lagrangian sub-problem from scratch [53].

Every solution of a Lagrangian sub-problem is a partial schedule $\{\sigma_i'\}_{1 \leq i \leq n}$ that may violate the resource constraints of the original problem. This solution is converted into a feasible schedule $\{\sigma_i\}_{1 \leq i \leq n}$ by list scheduling using the α-completion times $C_i'(\alpha)$ as priorities, a technique also used in [54]. An α-completion time is defined as $C_i'(\alpha) \overset{\text{def}}{=} \sigma_i' + \alpha p_i$ for $\alpha \in [0, 1]$. Different values of α can be tried, but in case of UET operations they all result in the same LSA schedules.

Renewable Resource	**Width**	**IMX**	**MUL**	**MEM**	**CTL**	**ODD**
Availability	4	2	2	1	1	2

Issue 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 1: The ST220 cumulative resource availabilities and issue class requirements.

2 VLIW Instruction Scheduling Problems

2.1 The Instruction Scheduling Problem and its Specializations

We define an *instruction scheduling problem* as a RCPSP with dependence constraints, where the dependence latencies $\alpha_i^j \overset{\text{def}}{=} p_i + l_i^j$ (called precedence delays d_i^j in §1.2) are non-negative integers. In case of UET operations, we call it a *VLIW instruction scheduling problem*. Indeed in modern processors all the functional units are pipelined, meaning there is an apparent processing time of one cycle, and a fixed non-negative latency 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.

As an illustration, let us consider the ST220 processor produced by STMicroelectronics, a single-cluster implementation of the Lx core [28]. 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 the Branch Register file (8×1-bit). The multiply instructions are restricted to 16-bit $\times$ 32-bit variants, and there is no divide instruction. In order to reduce the use of conditional branches, the ST200 architecture also provides conditional selection instructions. 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 can be modeled with cumulative resources. In figure 1, **Width** represents the issue width, **IMX** represents the large 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 *issue classes*, which in case of the ST220 processor are { **ALU**, **ALUX**, **MUL**, **MULX**, **MEM**, **MEMX**, **CTL** }. The issue classes **ALU**, **MUL**, **MEM**, and **CTL**, correspond respectively to the single-cycle, multiply, memory, and control, operations. The issue classes **ALUX**, **MULX**, and **MEMX**, similarly abstract the operations with a long immediate operand.

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 [23]. Code selection translates the computations from the upper stage of the compiler into efficient target processor operation sequences. The control optimizations reduce the run-time penalty of branches by predicating code [52], and by reordering the basic blocks [59]. The register allocator maps the operation operands to the processor registers or stack memory locations. The instruction scheduler reorders the operations to reduce the run-time processor stalls.

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, that can only be entered at the first operation, and which 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 different types. A first parameter is the shape of the program regions that compose each instruction scheduling problem, or *scheduling regions*. The simplest scheduling region is a basic block, but in order to increase the instruction scheduling problem scope, *super blocks* are generally used [38]. 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*).

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* (II) [63].

In order to illustrate these two specializations of instruction scheduling, let us consider the sample of C source code in figure 2, and the translation of the inner loop body in ST220 operations input to the pre-pass instruction scheduling. When considering the inner loop body of figure 2 as a BISP, our ST200 LAO instruction scheduler (see §3.4) generates the minimum length instruction schedule of 7 cycles displayed in figure 3, which starts at cycle `[0]`, and stops at cycle `[6]`. In the right part of figure 3, we display the *issue table*, a central data structure of the instruction scheduler that tracks the resource uses of each scheduled operation (here represented as issue classes).

When considering the inner loop of figure 2 as a PISP, the result is a 8 cycle *local schedule* displayed in figure 4. However, this particular local schedule allows the execution of the loop operations to be overlapped , or *software pipelined*, with a constant II of 2 cycles, as illustrated by the software pipeline issue table in figure 4. The particular software pipelining technique used by the ST200 LAO to solve the PISP is a variant of the *modulo scheduling* technique (§2.3).

2.2 Cyclic, Periodic, and Pipeline Scheduling Problems

A *cyclic machine scheduling problem* [36] 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 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

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

```
int                                         L?__0_8:
prod(int n, short a[], short b) {             LDH_1      g96 = 0, G92
  int s = 0, i;                               MULL_2     g97 = G91, g96
  for (i = 0; i < n ; i++) {                  ADD_3      G94 = G94, g97
    s += a[i]*b;                              ADD_4      G93 = G93, 1
  }                                           ADD_5      G92 = G92, 2
  return s;                                   CMPNE_6    b100 = G83, G93
}                                             BR_7       b100, L?__0_8
```

Figure 2: Source code and the inner loop body code generator representation.

```
[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 3: The block scheduled loop body and the block schedule issue table.

the dependences between the operation instances are respected. The common model of generic dependences in cyclic scheduling problem are the *uniform dependences* [36]:

$$O_i \xrightarrow{\alpha_i^j,\beta_i^j} O_j \Longleftrightarrow \sigma_i^k + \alpha_i^j \leq \sigma_j^{k+\beta_i^j} \quad \forall k > 0$$

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

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 *initiation interval* $\lambda \in \mathbb{Q}$:

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

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 K-periodic schedules contain an optimal solution, but this K is not guaranteed to be 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.

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 [36], where in addition to the original uniform dependences, the operations are constrained as non reentrant ($\sigma_i^k + p_i \leq \sigma_i^{k+1}$) through additional uniform dependences. In both cases [37]:

$$\lambda \geq \lambda_{rec} \stackrel{\text{def}}{=} \max_C \frac{\sum_C \alpha_i^j}{\sum_C \beta_i^j} : \quad C \text{ directed cycle of the dependence graph}$$

```
[0] LDH_1       g96 = 0, G92
[0] ADD_5       G92 = G92, 2
[2] ADD_4       G93 = G93, 1
[3] MULL_2      g97 = G91, g96
[3] CMPNE_6     b100 = G83, G93
[6] ADD_3       G94 = G94, g97
[7] BRF_7       b100, .LAO001
```

```
[0] ALU_5.1 MEM_1.1
[1]
[2] ALU_4.1 ALU_5.2 MEM_1.2
[3] ALU_6.1 MUL_2.1
[4] ALU_4.2 ALU_5.2 MEM_1.3
[5] ALU_6.2 MUL_2.2
[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 4: The software pipeline local schedule and the software pipeline issue table.

The maximum ratio λ_{rec} is reached in particular on simple directed cycles, 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 [15]. Solving the BCSP for any $\lambda \geq \lambda_{rec}$ then reduces to a simple longest path computation [32].

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 $\lambda = \lambda_{res}$ [63]. This technique was generalized by Eisenbeis & Windheiser to non-UET generic operations [27].

In the pipeline scheduling techniques, a local schedule $\{\sigma_i^1\}_{1\leq i\leq n}$ is created first, then K copies of it are overlapped in a way to achieve the maximum average throughput, thus creating a K-periodic schedule of constrained structure. These *pipeline scheduling problems* originate from hardware pipeline optimization [42], 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 §2.3.

One heuristic to solve 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 [2]. In order to converge, this method requires that the span of any local schedule be bounded, a condition that is enforced by adding span-limiting dependence from O_{n+1} to O_0 [6]. These techniques suffer either from slow convergence (large values of K), or from the sequentializing effect of the span-limiting dependence. On the other hand, they do not require the loop body to be a single basic block or super block.

The techniques that optimally solve K-periodic instruction problems originate in work by Feautrier [29], where D-periodic schedules are defined with $A, \{B_i\}_{1\leq i\leq n}, D \in \mathbb{Z}$, and:

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

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

For uniform dependences, and exclusive resources (of unit availability), Feautrier shows [29]:

$$O_i \xrightarrow{\alpha_i^j, \beta_i^j} O_j \iff B_i + D\alpha_i^j - A\beta_i^j \leq B_j \quad (8)$$

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

Here q_i^j is an integer variable. By guessing the value of D from the structure of the dependence cycles, 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 [30], then to find the optimal values of A and D without having to enumerate them [31].

2.3 Modulo Instruction Scheduling Problems and Techniques

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 [16]. 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 [46], and further memory access bandwidth optimizations are enabled [22].

Modulo scheduling is a framework for building periodic schedules, where only the generic operations and the generic dependences are considered. The original modulo scheduling technique was discovered by Rau & Glaeser in 1981 [63], popularized under this name by Lam in 1988 [45], and later refined under the name *iterative modulo scheduling* by Rau [66]. When considering the scheduling constraints on the generic operations, the problem is transformed as follows:

- Uniform dependence constraints $O_i \xrightarrow{\alpha_i^j, \beta_i^j} O_j$ are expressed as $\sigma_i + \alpha_i^j - \lambda\beta_i^j \leq \sigma_j$.
- The resource constraints become *modulo resource constraints*: each operation O_i now requires $\vec{b}_i$ cumulative resources for all the time intervals $[\sigma_i + k\lambda, \sigma_i + k\lambda + p_i - 1], k \in \mathbb{Z}$.

A *modulo instruction scheduling problem* (MISP) is an instruction scheduling problem with modulo resource constraints, and uniform dependences whose latencies α_i^j and distances β_i^j are non-negative. When $\beta_i^j = 0$, the dependence is *loop-independent*, else it is *loop-carried*. Modulo scheduling problems are parametric with λ the II, and their objective is to build a feasible periodic schedule with minimum integer λ. Given two modulo schedules of equal λ, the best is of minimum C_{max}. Indeed the number of cycles required for executing l iterations equals $C_{max} + (l-1)\lambda$.

In the classic modulo scheduling technique [66], a particular value of $\lambda \geq \max(\lambda_{rec}, \lambda_{res})$ is assumed. Then an acyclic scheduling algorithm extended to manage the modulo resource constraints is applied to the scheduling problem defined by the generic operation set, and by the dependence graph where the uniform dependences $O_i \xrightarrow{\alpha_i^j, \beta_i^j} O_j$ have a latency $\alpha_i^j - \lambda\beta_i^j$. As this dependence graph may contain directed cycles, or dependence with negative latencies, heuristics may fail to build feasible schedules for some feasible modulo scheduling problems.

Failure conditions arise during modulo scheduling when an operation cannot be scheduled within its current earliest and latest start dates without modulo resource conflicts. The practice to address this issue is to implement limited backtracking in the modulo scheduling algorithms, with excellent reported results [66, 67]. If failure persists, modulo scheduling is restarted at a higher λ. Another solution implemented in *insertion scheduling* [19] is to keep the partial modulo schedule obtained before failure, to increase the λ enough to remove the problematic modulo resource conflicts, then to resume modulo scheduling.

A different approach that approximately solves modulo scheduling problems was pioneered by Gasperoni & Schwiegelshohn in 1991 [32]. Their main idea is to decompose the schedule dates into a quotient and remainder when divided 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 decomposed as follows [32]:

- build the resource-free schedule $\{\sigma_i^*\}_{1\leq i\leq n}$ with II λ^*, and 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 $\tau_i^* + \alpha_i^j > \tau_j^*$, 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 allows to transform the modulo scheduling problem with UET operations into a minimum-length scheduling problem involving only the column numbers and normal resource constraints. This works because an apparent dependence violation on the row numbers implies that the difference between the column numbers is large enough: $\sigma_i + \alpha_i^j - \lambda\beta_i^j \leq \sigma_j \wedge \tau_i + \alpha_i^j > \tau_j \Rightarrow \phi_i < \beta_i^j + \phi_j$ (see Corollary 1 in §3.2). 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 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]:

$$\sum_C \left\lceil \frac{\alpha_i^j + \tau_i - \tau_j}{\lambda} \right\rceil - \beta_i^j \quad \leq 0 \quad \forall C \text{ directed cycle of the dependence graph}$$

Another generalization that still computes the column numbers first in resource-free context, then the row numbers with normal acyclic scheduling, is the loop shifting and compaction of Darte & Huard [14]. We also developed the idea of rescheduling while keeping the column numbers invariant in insertion scheduling [19], a modulo scheduling technique that neither assumes UET operations, nor requires a decomposition of the modulo scheduling problem (see §3.2 and §3.4).

2.4 The Register Instruction Scheduling Problems

In a compiler 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 to either the architectural registers, or stack memory locations. When a virtual register is mapped to a stack 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 stack 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 MOVE operations, and conversely may have to *split* a register live range by inserting a MOVE 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 [57], global register allocation between the basic blocks is performed first, followed by local register allocation.

The mainstream results and techniques available for minimizing the penalty of register allocation work from 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 complementary, as we aim 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 as 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 [34, 11]. 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 *register lifetimes instruction scheduling problems* (RLISP) 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 particular 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.

All these problems exist in the block and the modulo instruction scheduling variants.

The simplest attempt to solve a RLISP is to perform pre-pass instruction scheduling backward. This works well in practice because many expression DAGs 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, and 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 [40], hypernode reduction modulo scheduling [48], minimum cumulative register lifetime modulo scheduling [18], and swing modulo scheduling [49]. 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 II. 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 [34], 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 [41]. For the modulo RPISP, several heuristic approaches have been proposed, based on the *stage scheduling* of Eichenberger & Davidson [25]. 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 [56] for the resource-free modulo scheduling problems, Govindarajan et al. [35] 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 next in §2.5, was given in 1995 by Eichenberger, Davidson, and Abraham [24].

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 [39], whose minimum node coloring problem is *NP*-hard. As observed by Hendren et al. [39], splitting some of the register live ranges (by inserting MOVE 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 [64].

An heuristic for the block RCISP was proposed by Natarajan & Schlansker in 1995 [55]. 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-DAGs in a resource-free schedule.

For the periodic RCISP, Fimmel et al. [30] introduce time-indexed variables ("binary decomposition"), with register pressure equations that are similar to those proposed by Eichenberger et al. [24]. For solving the modulo RCISP, we propose a new time-indexed formulation in §3.5, 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. In case of the software pipelined loops, it has been solved in the MIPS-PRO production compiler [67]. For the acyclic scheduling regions, the only successful industrial solution in use is implemented in the Multiflow Trace Scheduling compiler [51].

2.5 Time-Indexed Formulation for the Modulo RPISP

The modern formulation of the modulo RPISP is from Eichenberger et al. [24, 26]. 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: $\{\tau_i \overset{\text{def}}{=} \sum_{t=0}^{\lambda-1} t x_i^t\}_{1\le i\le n}$. On the other hand, the column numbers $\{\phi_i\}_{1\le i\le n}$ are directly used in the formulation. The objective is to minimize the register pressure integer variable P:

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

Although this formulation was elaborated without apparent knowledge of the time-indexed RCPSP formulation of Pritsker et al. [60], 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. [24, 26] is the development of the inequalities (14) that compute the contributions v_i^t to the register pressure at date t.

As explained in [24], these equations derive from the observation that the function $D_i(t) \overset{\text{def}}{=} \sum_{s=0}^{t} x_i^s$ transitions from 0 to 1 for $t = \tau_i$. Similarly, the function $U_j(t) \overset{\text{def}}{=} \sum_{s=0}^{t-1} x_j^s$ transitions from 0 to 1 for $t = \tau_j + 1$. Assuming that $\tau_i \le \tau_j$, 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 $\phi_j - \phi_i + \beta_i^j$, or $\phi_j - \phi_i + \beta_i^j - 1$, 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).

	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 5: A reservation table, a regular reservation table, and a reservation vector.

Like the other classic modulo scheduling techniques [45, 66, 67], this time-indexed formulation takes λ as a constant given in the scheduling problem description. So 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. In order to offer this time-indexed formulation as an option in production compilers, efficient relaxations of the modulo scheduling problem are required, first to find an accurate lower bound on λ, and second to reduce the number of variables by pre-processing [17].

3 Instruction Scheduling for the ST200

3.1 Instruction Scheduling Problems Resource Modeling

In general, the resource constraints for instruction scheduling on a VLIW processor are specified by a set of admissible *bundles*, that is, combinations of operations that may issue without resource conflicts. Precisely, a bundle is specified as a sequence of issue classes, where each issue class is an abstraction for the resource requirements of the operations in the instruction scheduling problem. Going from the bundle specifications for a processor, to an accurate model with cumulative resources, is not guaranteed to succeed. Techniques to build the cumulative resource model from a given processor description are described in [61, 62]. Their idea is to introduce artificial resources, based on the identification of critical sets of issue classes.

Even with the introduction of artificial resources like **ODD** for the ST220 in figure 1, a processor cumulative resource model can still be an approximation, because the cumulative constraints can only represent convex regions of issue classes. To illustrate this point, consider a processor with two issue classes **A** and **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 $3a+b \le c \wedge a+3b \le c \wedge 2a+2b > c$, 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 $\sigma_i + p_i \le \sigma_j$, or $\sigma_j + p_j \le \sigma_i$. 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 $\sigma_j + \rho_j^i \le \sigma_i$, where ρ_i^j and ρ_j^i both depend on O_i and O_j. These constraints arise when the resource requirements are represented by *regular reservation tables*.

A *regular reservation table* is the traditional pipeline reservation table of an operation [42, 66], 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 5 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 most modern processors. For instance, when describing the DEC Alpha 21064 processor, we found [19] that the regular reservation tables were slightly inaccurate only for the integer multiplications, and the floating-point divisions.

A main advantage of using 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 II 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 $\Sigma_i^j(\lambda) \stackrel{\text{def}}{=} \cup_{k \in \mathbb{Z}}[\sigma_i + k\lambda, \sigma_i + k\lambda + \rho_i^j - 1]$. In the modulo scheduling problem, O_i and O_j do not conflict iff $\Sigma_i^j(\lambda) \cap \Sigma_j^i(\lambda) = \emptyset$, that is:

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

This result is similar to the inequalities (9) given by Feautrier [29] 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.

3.2 The Modulo Insertion Scheduling Theorems

Insertion scheduling [19] is a modulo scheduling technique 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 modulo resource conflicts, until a feasible modulo schedule is obtained. A *partial modulo schedule* as 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 II λ, into another partial modulo schedule $\{\sigma'_i\}_{1 \leq i \leq n}$ at II 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:

$$\forall i \in [0, n+1] : \begin{cases} \sigma_i = \phi_i \lambda + \tau_i \wedge 0 \leq \tau_i < \lambda \\ \sigma'_i = \phi'_i \lambda' + \tau'_i \wedge 0 \leq \tau'_i < \lambda' \\ \delta_i = \tau'_i - \tau_i \end{cases}$$

Let $\sigma_i \leadsto \sigma_j$ denote the fact that σ_i precedes σ_j in the transitive closure of the loop-independent dependences (whose $\beta_i^j = 0$) of the dependence graph[6]. The following 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 initiation interval λ. Let $\{\sigma'_i\}_{1 \leq i \leq n}$ be n integers such that:*

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

$$\forall i, j \in [0, n+1] : \quad \begin{cases} \phi_i = \phi'_i \\ 0 \leq \delta_i \leq \Delta \\ \tau_i < \tau_j \Longrightarrow \delta_i \leq \delta_j \\ \tau_i = \tau_j \wedge \phi_i > \phi_j \Longrightarrow \delta_i \leq \delta_j \\ \tau_i = \tau_j \wedge \phi_i = \phi_j \wedge \sigma_i \rightsquigarrow \sigma_j \Longrightarrow \delta_i \leq \delta_j \end{cases} \tag{19}$$

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

Proof: Let $O_i \xrightarrow{\alpha_i^j, \beta_i^j} O_j$ be a dependence of P. From the definition of a dependence constraint, $\sigma_j - \sigma_i \geq \alpha_i^j - \beta_i^j \lambda$, or equivalently $\phi_j \lambda + \tau_j - \phi_i \lambda - \tau_i \geq \alpha_i^j - \beta_i^j \lambda$. Given the hypothesis (19), let us show this dependence holds for $\{\sigma'_i\}_{1 \leq i \leq n}$, that is, $\phi_j \lambda' + \tau'_j - \phi_i \lambda' - \tau'_i \geq \alpha_i^j - \beta_i^j \lambda'$.

Dividing the dependence inequality by λ and taking the floor yields $\phi_j - \phi_i + \lfloor \frac{\tau_j - \tau_i}{\lambda} \rfloor \geq -\beta_i^j$, since all α_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 $\phi_j - \phi_i \geq -\beta_i^j$, and we consider its sub-cases:

$\phi_j - \phi_i = -\beta_i^j$: The dependence inequality reduces to $\tau'_j - \tau'_i \geq \alpha_i^j$. Since $\alpha_i^j \geq 0$, we have $\tau_j \geq \tau_i$. Several sub-cases need to be distinguished:

- **$\tau_i < \tau_j$:** From (19) we have $\delta_j \geq \delta_i \Leftrightarrow \tau'_j - \tau_j \geq \tau'_i - \tau_i \Leftrightarrow \tau'_j - \tau'_i \geq \tau_j - \tau_i \geq \alpha_i^j$.
- **$\tau_i = \tau_j \wedge \phi_i \neq \phi_j$:** Either $\phi_i > \phi_j$, or $\phi_i < \phi_j$. The latter is impossible, for $\beta_i^j = \phi_i - \phi_j$, and since all β_i^j are non-negative. From (19), $\tau_i = \tau_j \wedge \phi_i > \phi_j$ yields $\delta_j \geq \delta_i$, so the conclusion is the same as above.
- **$\tau_i = \tau_j \wedge \phi_i = \phi_j$:** Since $\beta_i^j = \phi_i - \phi_j = 0$, there are no dependence constraints unless $\sigma_i \rightsquigarrow \sigma_j$. In this case taking $\delta_j \geq \delta_i$, works like in the cases above.

$\phi_j - \phi_i > -\beta_i^j$: Let us show that $(\phi_j - \phi_i + \beta_i^j)\lambda' + \tau'_j - \tau'_i - \alpha_i^j \geq 0$. We have $\phi_j - \phi_i + \beta_i^j \geq 1$, so $(\phi_j - \phi_i + \beta_i^j)\lambda' \geq (\phi_j - \phi_i + \beta_i^j)\lambda + \Delta$. By (19) 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 $(\phi_j - \phi_i + \beta_i^j)\lambda' + \tau'_j - \tau'_i - \alpha_i^j \geq (\phi_j - \phi_i + \beta_i^j)\lambda + \Delta + \tau_j - \tau_i - \Delta - \alpha_i^j = (\phi_j - \phi_i + \beta_i^j)\lambda + \tau_j - \tau_i - \alpha_i^j \geq 0$. □

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

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

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

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

Then $\{\sigma'_i\}_{1 \leq i \leq n}$ also satisfies the modulo resource constraints at initiation interval $\lambda' \overset{def}{=} \lambda + \Delta$.

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 is equivalent to (see §3.1):

$$\sigma_i - \sigma_j \geq \rho_j^i + k_i^j\lambda \quad \wedge \quad \sigma_j - \sigma_i \geq \rho_i^j - (k_i^j + 1)\lambda \quad \text{with} \quad k_i^j \overset{\text{def}}{=} \lfloor \frac{\sigma_i - \sigma_j}{\lambda} \rfloor \tag{21}$$

These constraints look exactly like ordinary dependence constraints, save the fact that the β values are now of arbitrary sign. Since the sign of the β 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 collisions between σ_i and σ_j, we deduce from the proof of Theorem 1 that the $\{\sigma'_k\}_{1\leq k\leq n}$ satisfy the modulo resource constraints at II λ'. □

Corollary 2 *Let $\{\sigma_i\}_{1\leq i\leq n}$ be schedule dates satisfying the modulo resource constraints at initiation interval λ, assuming cumulative resources. Then under the conditions (20), $\{\sigma'_i\}_{1\leq i\leq n}$ also satisfies the modulo resource constraints at initiation interval $\lambda' \overset{\text{def}}{=} \lambda + \Delta$.*

Proof: In the proof of Theorem 2, whenever (21) holds for $\{\sigma_i\}_{1\leq i\leq n}$, then it holds under the conditions (20) for $\{\sigma'_i\}_{1\leq i\leq n}$, for any assumed values of ρ_i^j and ρ_j^i. This ensures that the simultaneous use of the cumulative resources does not increase in $\{\sigma'_i\}_{1\leq i\leq n}$. □

By Theorem 1 and Corollary 2, any transformation of a partial modulo schedule $\{\sigma_i\}_{1\leq i\leq n}$ at II λ into $\{\sigma'_i\}_{1\leq i\leq n}$ at II $\lambda' \overset{\text{def}}{=} \lambda + \Delta$, under the conditions (19), yields a partial modulo schedule. Although motivated by the implementation of insertion scheduling discussed in §3.4, this result has wider applications. In particular, any relaxation of a modulo scheduling problem that builds a partial modulo schedule, like Lagrangian relaxations of the time-indexed formulation of modulo scheduling given in §3.5, can be converted into a feasible modulo schedule thanks to this result.

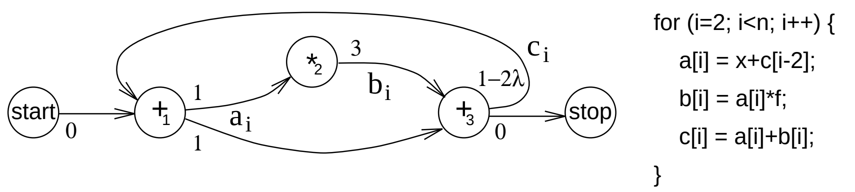
3.3 Minimum Cumulative Register Lifetime Modulo Scheduling

Since controlling the register pressure during instruction scheduling is a difficult problem, *lifetime-sensitive* modulo schedulers have been proposed [40], 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, following the conversion of 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 [18]. Then we have the following system of inequalities:

$$\begin{pmatrix} \textit{dependence} \\ \textit{inequalities} \end{pmatrix} \begin{cases} \sigma_2 - \sigma_1 & \geq & 1 \\ \sigma_3 - \sigma_1 & \geq & 1 \\ \sigma_3 - \sigma_2 & \geq & 3 \\ \sigma_1 - \sigma_3 & \geq & 1 - 2\lambda \end{cases} \qquad \begin{pmatrix} \textit{lifetime} \\ \textit{inequalities} \end{pmatrix} \begin{cases} l_1 & \geq & \sigma_2 - \sigma_1 \\ l_1 & \geq & \sigma_3 - \sigma_1 \\ l_2 & \geq & \sigma_3 - \sigma_2 \\ l_3 & \geq & \sigma_1 + 2\lambda - \sigma_3 \end{cases}$$



```
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.

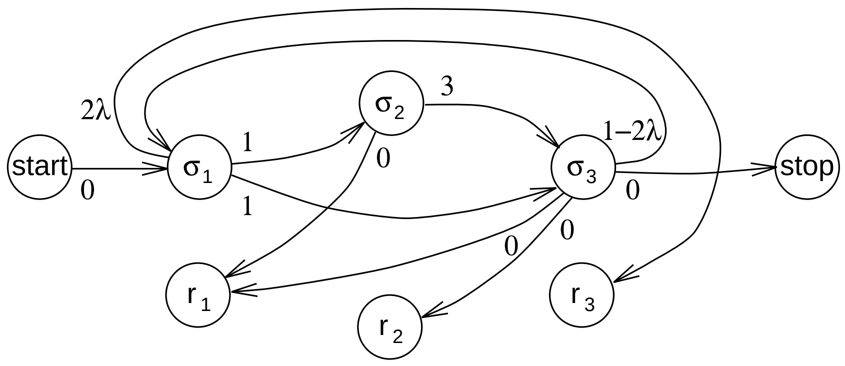


Figure 7: Augmented dependence graph.

This illustrates the equations we introduced in [18], and the Minimum Cumulative Register Lifetimes (MCRL) problem is defined with the objective to minimize $\sum_i l_i$. In [20], 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 $r_i \overset{\text{def}}{=} l_i + \sigma_i$, and eliminate the variables $\vec{l}$:

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

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 $l_i + \sigma_i \overset{\text{def}}{=} r_i \ \forall i$, 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 [56]. In the minimum buffers problem, the so-called buffer variables $\vec{b}$ are defined, and the objective to minimize is $\sum_i b_i$:

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

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 [56] make the change of variables $\lambda b_i \overset{\text{def}}{=} b'_i \ \forall i$, 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 [20]. 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 linear programming relationships can be written as:

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

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 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.

To solve the MCRL problems in our implementations [20], we use a network simplex algorithm to optimize the maximum cost flow problems [3], and a network simplex shortest path algorithm to find the initial solutions [33]. Besides being efficient in practice, 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.

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 to introduce "soft constraints" into scheduling problems. In the STMicroelectronics ST100 LAO code generator [23], increasing some of the flow requirements addressed the non-standard instruction scheduling features of the ST120 decoupled implementation. The augmented dependence graph has been re-discovered by Touati & Eisenbeis [70], and more generally, this graph is important for the integer linear programming formulations of the register instruction scheduling problems including ours (see §3.5).

3.4 Instruction Scheduling in the ST200 Production Compiler

The STMicroelectronics ST200 production compiler `st200cc` is based on the GCC compiler front-end components, the Open64 compiler technology [57], a global instruction scheduler adapted from the ST100 LAO code generator [23], an instruction cache optimizer, and the GNU assembler, linker, and binary tools. The Open64 compiler has been re-targeted from the Intel IA64 to the STMicroelectronics ST200, 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 / software pipeliner, which is only activated at optimization level `-O3`. At optimization level `-O2` and below, the Open64 basic block instruction scheduler is active. The ST200 LAO instruction scheduler / software pipeliner relies on the following techniques:

- A transactional interface to translate between the Open64 code generator intermediate representation (CGIR), and the LAO intermediate representation (LIR).

- Parsing of the control flow into a loop nesting forest, and super block formation. The super block formation does not include tail duplication, as this is performed in the Open64 code generator by the ST200-specific instruction predication.

- Pre-pass instruction scheduling based on the scheduling algorithm by Leung, Palem, and Pnueli [47], and software pipelining using MCRL insertion scheduling [19, 20].

- Post-pass instruction scheduling based on the scheduling algorithm by Leung, Palem, and Pnueli [47].

The algorithm of Leung et al. is iterated over each cumulative resource of our ST220 model (figure 1), both on the forward and the backward dependence graph, starting with the more heavily used resources. This yields strengthened release dates and deadline dates that are either used for list scheduling the operations, or propagated into the insertion scheduling algorithm.

Insertion scheduling [19] is a modulo instruction scheduling heuristic that *issues* the operations in priority order, where 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 assigned issue date. In addition, insertion scheduling applies the insertion theorems of §3.2 to resolve the resource conflicts between the previously issued operations and the current operation to issue, by increasing the II to make room for this operation. Thus, insertion scheduling does not need to backtrack, works with any priority order, and always builds a modulo schedule.

In the ST200 LAO modulo scheduler, insertion scheduling is implemented as follows[7]:

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. When solving P_0 with the network simplex algorithms, the minimum value λ_{rec} of the II such that P_0 is feasible is computed. Also compute a lower bound λ_{res} set by the resource constraints on the II, and take $\lambda_0 \overset{\text{def}}{=} \max(\lambda_{rec}, \lambda_{res})$.

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, $P_{j-1} \overset{\text{def}}{=} P_0 \wedge \{\sigma_{i_1} = S_{i_1}^{j-1}\} \wedge \ldots \wedge \{\sigma_{i_{j-1}} = S_{i_{j-1}}^{j-1}\}$. The not yet issued operations are ranked by a priority order, and the one with the highest rank, denoted O_{i_j}, is issued as follows:

1. Run the network simplex 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 two numbers $a_{i_j} \overset{\text{def}}{=} \min(l_{i_j} - m_{i_j}, \lambda - 1)$, and $b_{i_j} \overset{\text{def}}{=} \min(m_{i_j} - e_{i_j}, \lambda - 1 - a_{i_j})$. 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 II λ_{j-1} needs to increase to remove the modulo resource conflicts at II $\lambda_j \overset{\text{def}}{=} \lambda_{j-1} + \Delta$. If there are no conflicts between O_{i_j} and $\{O_{i_k}\}_{1\leq k\leq j-1}$ for $\sigma_{i_j} = S_{i_j}^{j}$, then $\Delta = 0$.

[7]This implementation uses network simplex algorithms to optimize the MCRL network flows. Insertion scheduling without the MCRL only requires single-source longest path computations like most scheduling heuristics.

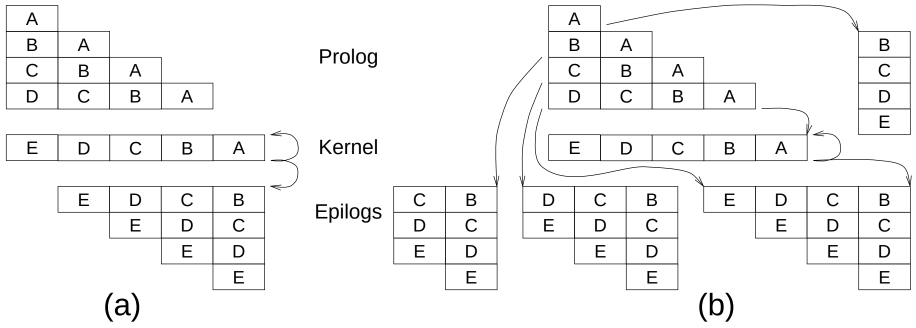


Figure 8: Software pipeline construction with and without pre-conditioning.

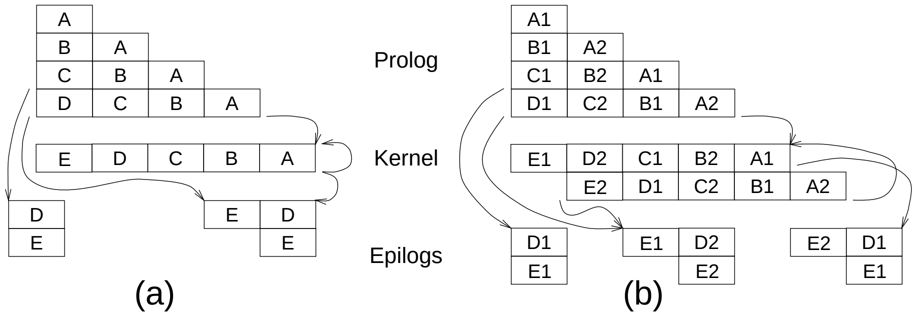


Figure 9: Software pipeline construction without and with modulo expansion.

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 [19] for the details of the δ_{i_k} computations.
4. Build the modulo scheduling problem $P_j \overset{\text{def}}{=} P_0 \wedge \{\sigma_{i_1} = S_{i_1}^{j}\} \wedge \ldots \wedge \{\sigma_{i_j} = S_{i_j}^{j}\}$ 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 issue table.

At the end of Step j, all the dependence constraints are satisfied, as well as 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.

Although insertion scheduling was introduced to solve the modulo scheduling problems [19], it is also used for the super block instruction scheduling in the ST200 production compiler. This is implemented by adding a dependence $O_{n+1} \rightarrow O_0$ with latency 1 and distance 1. The effect of this dependence is to constrain λ exactly like the minimum schedule length. As a result, the benefits of the MCRL scheduling carry to the pre-pass super block instruction scheduling.

Once a modulo schedule is available, it must be transformed into a software pipeline. Software pipeline construction is rather involved, as it manages the loop exits, and because corrective actions must be taken for the register dependences that are omitted from the modulo scheduling problem. In figure 8, we illustrate software pipeline construction for the local schedule of a counted loop that spans five stages named A, B, C, D, E. Compared to the textbook case (a), case (b) illustrates the construction that is required when loop pre-conditioning [65] does not apply.

Prior to modulo scheduling, the non-RAW register dependences on temporary variables of the loop body may be omitted in order to expose more parallelism [66]. As a result, in the local schedule temporary variables may live more than λ cycles, so their allocation in registers requires *modulo renaming*. Modulo renaming can be supported at the processor architecture level through *rotating registers* [64], or implemented entirely by software through *modulo expansion* and *kernel unrolling* [45, 21, 50], illustrated in figure 9. In [21], we also handle the cases where some dependences on the loop induction variables are omitted from the modulo scheduling problem.

3.5 Time-Indexed Formulations for the Modulo RCISP

Based on the time-indexed formulation of the modulo RPISP of Eichenberger et al. [24, 26] in §2.5, the time-indexed formulation of the RCPSP of Pritsker et al. [60] in §1.5, and the definition of the augmented dependence graph in §3.3, we now present two new time-indexed formulations for the modulo RCISP, whose ideas also carry to the modulo RPISP time-indexed formulations.

Our first formulation of the modulo RPISP keeps the λ-decomposed view of the modulo scheduling problems introduced by Eichenberger et al. [24], but eliminates the register pressure variables v_i^t. In our second formulation of the modulo RPISP, we get rid of the λ-decomposition, resulting in a time indexed formulation where the recent RCPSP solving techniques may apply.

The main idea for these RCISP formulations is to consider the dependence graph augmented with the register lifetime nodes as defined in §3.3. Let E_{life} denote the (i, j) pairs such that O_i is a producer node, and O_j is its lifetime node. Each v_i^t is then exactly defined by one equality (22), instead of several inequalities (14). Let us also take advantage of the fact that most processors (in particular the ST200) allow 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 cycle, to its last use cycle minus one. Now, by replacing the register pressure variable P by a constant R, equations (14), (15), (17), become:

$$\begin{aligned}
\sum_{s=0}^{t} x_i^s - \sum_{s=0}^{t} x_j^s + \phi_j - \phi_i - v_i^t &= -\beta_i^j \quad \forall t \in [0, \lambda-1], \forall (i,j) \in E_{life} && (22)\\
\sum_{i=1}^{n} v_i^t &\leq R \quad \forall t \in [0, \lambda-1] && (23)\\
v_i^t &\in \mathbb{N} \quad \forall i \in [1,n], \forall t \in [0, \lambda-1] && (24)
\end{aligned}$$

Thanks to the equalities (22), we eliminate the register pressure variables v_i^t from (23). We complete our first modulo RCISP formulation by minimizing the schedule date of operation O_{n+1}:

$$\begin{aligned}
\text{minimize} \sum_{t=1}^{\lambda-1} t\, x_{n+1}^t + \lambda\phi_{n+1} \quad &: && (25)\\
\sum_{t=0}^{\lambda-1} x_i^t &= 1 \quad \forall i \in [1, n+1] && (26)\\
\sum_{s=t}^{\lambda-1} x_i^s + \sum_{s=0}^{(t+\alpha_i^j-1) \bmod \lambda} x_j^s + \phi_i - \phi_j &\leq \beta_i^j - \lfloor \tfrac{t+\alpha_i^j-1}{\lambda} \rfloor + 1 \quad \forall t \in [0, \lambda-1], \forall (i,j) \in E_{dep} && (27)\\
\sum_{i=1}^{n} \sum_{r=0}^{p_i-1} x_i^{(t-r) \bmod \lambda}\, \vec{b}_i &\leq \vec{B} \quad \forall t \in [0, \lambda-1] && (28)
\end{aligned}$$

$$\begin{array}{rcll}
\sum_{(i,j)\in E_{life}} \sum_{s=0}^{t} x_i^s - \sum_{s=0}^{t} x_j^s + \phi_j - \phi_i & \leq & R - \sum_{(i,j)\in E_{life}} \beta_i^j & \forall t \in [0, \lambda-1] \quad (29) \\
x_i^t & \in & \{0,1\} & \forall i \in [1, n+1], \forall t \in [0, \lambda-1] \quad (30) \\
\phi_i & \in & \mathbb{N} & \forall i \in [1, n+1] \quad (31)
\end{array}$$

Although the λ-decomposed view of the modulo scheduling problems has the desirable feature to reduce the time horizon to $[0, \lambda-1]$, it might not be the best choice for a time-indexed formulation, especially with regards to constraint propagation [17, 72], and Lagrangian relaxation [53]. Indeed, the dependence inequalities (27) involve the $\{\phi_i\}_{1\leq i\leq n+1}$ variables, so even after dualizing the resource inequalities (28) and the register pressure inequalities (29), the resulting Lagrangian relaxation cannot be solved by minimum-cut as in [53]. This motivates our second time-indexed formulation of the modulo RCISP, without the λ-decomposition of the schedule dates:

$$\begin{array}{rcll}
\text{minimize} \sum_{t=1}^{T-1} t\, x_{n+1}^t & : & & \quad (32) \\
\sum_{t=0}^{T-1} x_i^t & = & 1 & \forall i \in [1, n+1] \quad (33) \\
\sum_{s=t}^{T-1} x_i^s + \sum_{s=0}^{t+\alpha_i^j-\lambda\beta_i^j-1} x_j^s & \leq & 1 & \forall t \in [0, T-1], \forall (i,j) \in E_{dep} \quad (34) \\
\sum_{i=1}^{n} \sum_{k=0}^{k=\lceil\frac{T-1}{\lambda}\rceil} \sum_{s=t+k\lambda-p_i+1}^{t+k\lambda} x_i^s\, \vec{b}_i & \leq & \vec{B} & \forall t \in [0, \lambda-1] \quad (35) \\
\sum_{(i,j)\in E_{life}} \sum_{k=0}^{k=\lceil\frac{T-1}{\lambda}\rceil+\beta_i^j} \sum_{s=0}^{t+k\lambda} x_i^s - \sum_{s=0}^{t+k\lambda-\lambda\beta_i^j} x_j^s & \leq & R & \forall t \in [0, \lambda-1] \quad (36) \\
x_i^t & \in & \{0,1\} & \forall i \in [1, n+1], \forall t \in [0, T-1] \quad (37)
\end{array}$$

This formulation is best understood with regards to the RCPSP formulation discussed in §1.5, and to our first time-indexed formulation of the modulo RCISP. In (34), we adapt the dependence inequalities (3) to the dependence latencies of $\alpha_i^j - \lambda\beta_i^j$. In (35), we extend the resource inequalities (4) with the modulo resource constraints. For (36), consider the following equations:

$$\begin{array}{rcll}
\sum_{s=0}^{t} x_i^s - \sum_{s=0}^{t} x_j^{s-\lambda\beta_i^j} - v_i^t & = & 0 & \forall t \in [0, T-1+\lambda\beta_i^j], \forall (i,j) \in E_{life} \quad (38) \\
\sum_{i=1}^{n} \sum_{k=0}^{k=\lceil\frac{T-1}{\lambda}\rceil+\max\beta_i^j} v_i^{t+k\lambda} & \leq & R & \forall t \in [0, \lambda-1] \quad (39) \\
v_i^t & \in & \{0,1\} & \forall i \in [1, n], \forall t \in [0, T-1] \quad (40)
\end{array}$$

In (38), we express that the register lifetime of an arc $O_i \xrightarrow{\alpha_i^j, \beta_i^j} O_j$ contributes to register pressure from date σ_i to date $\sigma_j + \lambda\beta_i^j - 1$. As the register lifetime wrap-around is accounted for in (39), the v_i^t are $\{0,1\}$ variables. Eliminating these variables from (39) with (38) yields (36).

Summary and Conclusions

This report discusses the similarities and the differences between machine scheduling problems, and instruction scheduling problems on modern VLIW processors 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 theoretical contributions to the field of instruction scheduling that are applied in the STMicroelectronics ST200 production compiler, and we introduce a new time-indexed formulation for the register constrained instruction scheduling problems.

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 cumulative scheduling problems (CuSP), 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 [47], 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, and the Lagrangian relaxation proposed by Möhring, Schulz, Stork, and Uetz [53].

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

- The VLIW instruction scheduling problem is defined as a RCPSP with UET operations, cumulative resources, and non-negative dependence delays. 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 [66] with the decomposed software pipelining techniques [71].
- 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 time-indexed formulation of the minimum register pressure modulo scheduling problem introduced by Eichenberger, Davidson, and Abraham [24, 26].

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

- Some of the resource modeling issues that arise when defining instruction scheduling problems for a particular processor.

- The modulo insertion scheduling theorems, which allow to transform a partial modulo schedule by increasing its II without violating the dependence constraints or the modulo resource constraints of the already scheduled operations.

- The equations that enable cumulative register lifetimes minimization, and an efficient way of solving them as a maximum-cost network flow problem.

- The application of these results to the instruction scheduler / software pipeliner we developed for the Cray T3E computers, improved for the STMicroelectronics ST100 [23], 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 [17, 53].

Future work for the STMicroelectronics ST200 production compiler includes the resolution of this time-indexed formulation for the modulo register constrained instruction scheduling problem, either optimally, or through a Lagrangian heuristic for the large-scale problems. These resolution techniques, and and their integration with the register allocation, are researched under the framework the STMicroelectronics collaboration with the IMAG-ID laboratory.

To conclude, we anticipate that the techniques of machine scheduling will become increasingly relevant to VLIW instruction scheduling, in particular for embedded computing applications. In such applications, high-performances of the compiled application enable to reduce the system costs, while 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 [13, 72]. An equally promising area of improvement of VLIW instruction scheduling is its full integration with the register allocation. In this area, the solution implemented in the Multiflow Trace Scheduling compiler [51] still rules.

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 case 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 1). Besides increasing the peak performances, removing this restriction enables the relaxation of the VLIW instruction scheduling problems into parallel machine scheduling problems with UET, that are optimally solved in polynomial time in several cases by the algorithm of Leung, Palem, and Pnueli [47].

References

[1] S. G. ABRAHAM, V. KATHAIL, B. L. DIETRICH: *Meld Scheduling: A Technique for Relaxing Scheduling Constraints* International Journal on Parallel Programming, Vol. 26, No. 4, 1998.

[2] A. AIKEN, A. NICOLAU: *Perfect Pipelining: A New Loop Parallelization Technique* European Symposium on Programming, 1998.

[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, Vol. 12, No. 2 spring 2000.

[5] P. BAPTISTE, C. LE PAPE, W. NUIJTEN: *Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems* Annals of Operations Research 92, 1999.

[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, S. KNUST: *Complexity Results for Scheduling Problems* http://www.mathematik.uni-osnabrueck.de/research/OR/class/

[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] J. CARLIER, E. NERON: *On Linear Lower Bounds for the Resource Constrained Project Scheduling Problem* European Journal of Operational Research 149, 2003.

[11] 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, Vol. 44, No. 3, March 1995,

[12] 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.

[13] K.D. COOPER, P.J. SCHIELKE, D. SUBRAMANIAN: *An Experimental Evaluation of List Scheduling* TR98-326, Dept. of Computer Science, Rice University, September 1998.

[14] A. DARTE, G. HUARD: *Loop Shifting for Loop Compaction* 12th International Workshop on Languages and Compilers for Parallel Computing – LCPC'99, 1999.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] F. GASPERONI, U. SCHWIEGELSHOHN: *Efficient Algorithms for Cyclic Scheduling* IBM Technical Report TR1991-571, 1991.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

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

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

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

[44] 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.

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

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

[47] A. LEUNG, K. V. PALEM, A. PNUELI: *Scheduling Time-Constrained Instructions on Pipelined Processors* ACM TOPLAS Vol. 23, No. 1, Jan. 2001.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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

[53] R. H. MÖHRING, A. S. SCHULZ, F. STORK, AND M. UETZ: *Solving Project Scheduling Problems by Minimum Cut Computations* Management Science 49(3), March 2003.

[54] 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.

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

[56] 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.

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

[58] K. V. PALEM, B. SIMONS: *Scheduling Time-critical Instructions on RISC Machines* ACM Symposium on Principles of Programming Languages – POPL'90, 1990.

[59] 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.

[60] 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.

[61] 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.

[62] 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, Volume 20, Number 11, November 2001.

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

[64] 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.

[65] 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 – MCRO-25, December 1992.

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

[67] 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.

[68] M. W. P. SAVELSBERGH, R. N. UMA, AND J. WEIN: *An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems* Submitted to INFORMS J. on Computing, 2002.

[69] V. G. TIMKOVSKY: *Identical Parallel Machines vs Unit-Time Shops and Preemptions vs Chains in Scheduling Complexity* http://citeseer.nj.nec.com/timkovsky00identical.html, 2000.

[70] S-A-A. TOUATI, C. EISENBEIS: *Early Control of Register Pressure for Software Pipelined Loops* International Conference on Compiler Construction – CC'03, April, 2003.

[71] J. WANG, C. EISENBEIS: *Decomposed Software Pipelining: A New Approach to Exploit Instruction Level Parallelism for Loop Programs* Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, 1993.

[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.