Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors

Technical Report A/392/CRI

Benoît Dupont de Dinechin
benoit.dupont-de-dinechin@st.com

STMicroelectronics STS/CEC
12, rue Jules Horowitz - BP 217. F-38019 Grenoble

Abstract. We present a postpass instruction scheduling technique suitable for Just-In-Time (JIT) compilers targeted to VLIW processors. Its key features are: reduced compilation time and memory requirements; satisfaction of scheduling constraints along all program paths; and the ability to preserve existing prepass schedules, including software pipelines. This is achieved by combining two ideas: instruction scheduling similar to the dynamic scheduler of an out-of-order superscalar processor; the satisfaction of inter-block scheduling constraints by propagating them across the control-flow graph until fixed-point. We implemented this technique in a Common Language Infrastructure JIT compiler for the ST200 VLIW processors and the ARM processors.

1 Introduction

Just-In-Time (JIT) compilation of programs distributed as Java or .NET Common Language Infrastructure (CLI) byte-codes is becoming increasingly relevant for consumer electronics applications. A typical case is a game installed and played by the end-user on a Java-enabled mobile phone. In this case, the JIT compilation produces native code for the host processor of the system-on-chip.

However, systems-on-chip for consumer electronics also contain powerful media processors that could execute software installed by the end-user. Media processing software is usually developed in C or C++ and exposes instruction-level parallelism. Such media processing software can be compiled to CLI byte-codes thanks to the Microsoft Visual Studio .NET compilers or the `gcc/st/cli` compiler branch contributed by STMicroelectronics [4]. This motivates Just-In-Time compilation for embedded processors like the Texas Instruments C6000 VLIW-DSP family and the STMicroelectronics ST200 VLIW-media family[1].

In the setting of JIT compilation of Java programs, instruction scheduling is already expensive. For instance, the IBM Testarossa JIT team reports that combined prepass and postpass instruction scheduling costs up to 30% of the

[1] The ST200 VLIW architecture is based on the Lx technology [7] jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

compilation time [14] for the IBM zSeries 990 and the POWER4 processors. To lower these costs, the IBM Testarossa JIT compiler relies on profiling to identify the program regions where instruction scheduling is enabled. In addition, the register allocator tracks its changes to the prepass instruction schedules in order to decide where postpass instruction scheduling might be useful.

In the case of JIT compilation of media processing applications for VLIW processors, more ambitious instruction scheduling techniques are required. First, software pipelining may be applied in spite of higher compilation costs, as these applications spend most of their time in inner loops where instruction-level parallelism is available. However, software pipelines implement cyclic schedules that may be destroyed when the code is postpass scheduled using an acyclic scheduler. Second, JIT instruction scheduling techniques should accommodate VLIW processors without interlocking hardware [9, 2], such as the TI C6000 VLIW-DSP family or the STMicroelectronics ST210 / Lx [7]. This means that JIT compilation must ensure that no execution path presents scheduling hazards.

To address these issues specific to JIT compilation on VLIW processors, we propose a new postpass instruction scheduling whose main features are:

– Efficiency (code quality) and speed (compilation time). This is possible thanks to *Scoreboard Scheduling*, that is, instruction scheduling by emulating the hardware scheduler of an out-of-order superscalar processor.
– Satisfaction of resource and dependence constraints along all program paths, as required by processors without interlocking hardware. We formulate and solve this *Inter-Block Scheduling* problem by propagating constraints until reaching a fixed-point, in a way reminiscent of forward data-flow analysis.

In addition, we prove our technique preserves the instruction schedules created by prepass scheduling and by software pipelining, provided register allocation and basic block alignment only introduced redundant scheduling constraints.

The presentation is as follows. In Section 2, we review local instruction scheduling heuristics and we propose Scoreboard Scheduling. We then describe an optimized implementation of this technique. In Section 3, we discuss inter-region instruction scheduling and we introduce Inter-Block Scoreboard Scheduling. This technique relies on iterative constraint propagation and we characterize its fixed-points. In Section 4, we provide an experimental evaluation of our contributions, which are implemented in the STMicroelectronics CLI-JIT compiler that targets the ST200 VLIW and the ARM processors.

2 Local Instruction Scheduling

2.1 Acyclic Instruction Scheduling

Acyclic instruction scheduling is the problem of ordering the execution of a set of *operations* on a target processor microarchitecture, so as to minimize the latest completion time. Executions of operations are partially ordered to ensure correct results. Precisely, effects on registers must be ordered in the following cases: Read

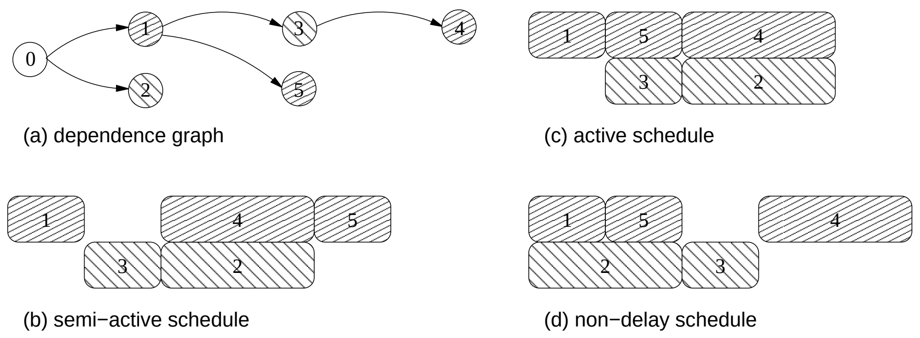


Fig. 1. Sample schedules for a two-resource scheduling problem (horizontal time).

After Write (RAW), Write After Read (WAR), and Write After Write (WAW). Other dependences arise from the partial ordering of memory accesses and from control-flow effects. We assume that the resource requirements of each operation are represented by a *reservation table* [12], where rows correspond to scheduled resources and columns to time steps relative to the operation start date.

Classic instruction scheduling heuristics fall in two main categories [3]:

Cycle Scheduling Scan time slots in non-decreasing order. For each time slot, order the dependence-ready operations in non-increasing priority and try to schedule each operation in turn, subject to resource availability. Dependence-ready means that execution of the operation predecessors has completed early enough to satisfy the dependences. This is Graham list scheduling.

Operation Scheduling Consider each operation in non-increasing priority order. Schedule each operation at the earliest time slot where it is dependence-ready and its required resources are available. In order to prevent deadlock, the priority list order must be a topological sort of the dependence graph.

Cycle Scheduling is a time-tested instruction scheduling heuristic that produces high quality code on simple instruction pipelines, given a suitable priority of operations [9]. One such priority is the "critical path" length from any operation to the dependence graph sink node. A refinement is the "backward scheduling" priority that ensures optimal schedules on homogeneous pipelines [8] and on typed pipelines [6] for special classes of dependence graphs.

For the proofs of §2.2, we assume *monotonic reservation tables*, that is, reservation tables whose entries in any row are monotonically non-increasing. Single-column reservation tables, which are virtually always found on modern VLIW processors, are obviously monotonic. Monotonicity enables leverage of classic results from Resource Constrained Project Scheduling Problems (RCPSP) [13]:

Semi-Active Schedule as in Figure 1 b. No operation can be completed earlier without changing some execution sequence. Equivalently, in a semi-active schedule any operation has at least one dependence or resource constraint that is tight, preventing the operation from being locally left shifted.

Active Schedule as in Figure 1 c. No operation can be completed earlier without delaying another operation. The schedule of Figure 1 b is not active, because operation 5 can be globally left-shifted to time slot 1, without delaying other operations. Operation Scheduling generates active schedules.

Non-Delay Schedule as in Figure 1 d. No execution resources are left idle if there is an operation that could start executing. The schedule of Figure 1 c is not non-delay, because operation 2 could start executing at time slot 0. Cycle Scheduling generates non-delay schedules.

The non-delay schedules are a subset of the active schedules, which are a subset of the semi-active schedules [13]. Active schedules and non-delay schedules are the same in case of operations that require resources for a single time unit.

2.2 Scoreboard Scheduling Principles

The main drawback of the classic scheduling heuristics is their computational cost. In the case of Cycle Scheduling, the time complexity contributions are:

1. constructing the dependence graph is $O(n^2)$ with n the number of operations, but can be lowered to $O(n)$ with conservative memory dependences [15];
2. computing the operation priorities is $O(n + e)$ with e the number of dependences for the critical path, and is high as $O(n^2 \log n + ne)$ in [8, 6];
3. issuing the operations in priority order is $O(n^2)$ according to [9], as each time step has a complexity proportional to m (where m is the number of dependence-ready operations), and m can be $O(n)$.

The complexity of operation issuing results from: sorting the dependence-ready operations in priority order; and matching the resource availabilities of the current cycle with the resource requirements of the dependence-ready operations. The latter motivates the finite-state automata approach of Proebsting & Fraser [11], later generalized to Operation Scheduling by Bala & Rubin [3].

To reduce instruction scheduling costs, we rely on the following principles:

– Verbrugge [15] replaces the dependence graph by an Array of Dependency Lists (ALD), with one list per *dependence record* (see §2.3). We show how Operating Scheduling can avoid the explicit construction of such lists.
– In the setting of JIT postpass scheduling, either basic blocks are prepass scheduled because their performance impact is significant, or their operations are in original program order. In either case, the order operations are presented to the postpass scheduler encodes a priority that is suitable for Operation Scheduling, since it is a topological ordering of the dependences. So (re-)computing the operation priorities is not necessary.
– We limit the number of resource availability checks by restricting the number of time slots considered for issuing the current operation.

Precisely, we define *Scoreboard Scheduling* to be a scheduling algorithm that operates like Operation Scheduling, with the following additional restrictions:

```
issue=0     ldb $r23 = 9[$r16]
start=0   | +0 +1 +2 +3 +4 +5 +6 +7
----------|------------------------
ISSUE     |  1
MEM       |  1
CTL       |
----------|------------------------
Control   | a
$r16      | a
          |
$r23      | aw aw  w  w  w  w
Memory    | a  a
```

```
issue=0     add $r18 = $r18, -12
start=0   | +0 +1 +2 +3 +4 +5 +6 +7
----------|------------------------
ISSUE     |  2
MEM       |  1
CTL       |
----------|------------------------
Control   | a
$r16      | a
$r18      | aw aw  w  w
$r23      | aw aw  w  w  w  w
Memory    | a  a
```

Fig. 2. Scoreboard Scheduling within the time window (*window_size* = 4).

```
issue=4     shl $r24 = $r24, 24
start=0   | +0 +1 +2 +3 +4 +5 +6 +7
----------|------------------------
ISSUE     |  3  2  1  3  1
MEM       |  1  1  1  1
CTL       |
----------|------------------------
Control   | a  a  a  a  a
          |
$r16      | aw aw aw aw aw  w  w
$r18      | aw aw  w  w
$r23      | aw aw aw aw aw  w  w
$r24      | aw aw aw aw aw aw  w  w
Memory    | a  a
```

```
issue=5     add $r15 = $r15, $r24
start=1   | +0 +1 +2 +3 +4 +5 +6 +7
----------|------------------------
ISSUE     |  2  1  3  1  1
MEM       |  1  1  1
CTL       |
----------|------------------------
Control   | a  a  a  a  a
$r15      | aw aw aw aw aw aw  w  w
$r16      | aw aw aw aw  w  w
$r18      | aw  w  w
$r23      | aw aw aw aw  w  w
$r24      | aw aw aw aw aw  w  w
Memory    | a
```

Fig. 3. Scoreboard Scheduling and moving the time window (*window_size* = 4).

– any operation is scheduled within a time window of constant *window_size*,
– the *window_start* cannot decrease and is lazily increased while scheduling.

That is, given an operation to schedule, the earliest date considered is *window_start*. Moreover, if the earliest feasible schedule date *issue_date* of operation is greater than *window_start* + *window_size*, then the Scoreboard Scheduling *window_start* value is adjusted to *issue_date* − *window_size*.

Theorem 1 *Scoreboard Scheduling an active schedule yields the same schedule.*

Proof. By contradiction. Scheduling proceeds in non-decreasing time, as the priority list is a schedule. If the current operation can be scheduled earlier than it was, this is a global left shift so the priority list is not an active schedule.

Corollary 1. *Schedules produced by Operation Scheduling or Cycle Scheduling are invariant under Scoreboard Scheduling and Operation Scheduling.*

2.3 Scoreboard Scheduling Implementation

A *dependence record* is the atomic unit of state that needs to be considered for accurate register dependence tracking. Usually these are whole registers, except in cases of register aliasing. If so, registers are partitioned into sub-registers, some of which are shared between registers, and there is one dependence record per sub-register. Three technical records named *Volatile*, *Memory*, *Control* are also included in order to track the corresponding dependences.

Let $\mathit{read\_stage}[\,][\,]$, $\mathit{write\_stage}[\,][\,]$ be processor-specific arrays indexed by operation and by dependence record that tabulate the operand access pipeline stages. Let $\mathit{RAW}[\,]$, $\mathit{WAR}[\,]$, $\mathit{WAW}[\,]$ be latency tuning parameters indexed by dependence record. For any dependence record r and operations i and j, we generalize the formula of [16] and specify any dependence $\mathit{latency}_{i\rightarrow j}$ on r as follows:

RAW Dependence	$\mathit{latency}_{i\rightarrow j} \geq \mathit{write\_stage}[i][r] - \mathit{read\_stage}[j][r] + \mathit{RAW}[r]$	(a)
	$\mathit{latency}_{i\rightarrow j} \geq \mathit{RAW}[r]$	(b)
WAW Dependence	$\mathit{latency}_{i\rightarrow j} \geq \mathit{write\_stage}[i][r] - \mathit{write\_stage}[j][r] + \mathit{WAW}[r]$	(c)
	$\mathit{latency}_{i\rightarrow j} \geq \mathit{WAW}[r]$	(d)
WAR Dependence	$\mathit{latency}_{i\rightarrow j} \geq \mathit{read\_stage}[i][r] - \mathit{write\_stage}[j][r] + \mathit{WAR}[r]$	(e)
	$\mathit{latency}_{i\rightarrow j} \geq \mathit{WAR}[r]$	(f)

Assuming that $\mathit{write\_stage}[i][r] \geq \mathit{read\_stage}[j][r]\ \forall i, j, r$, that is, operand write is no earlier than operand read in the instruction pipeline for any given r, the dependence inequalities (b) and (e) are redundant. This enables dependence latencies to be tracked by maintaining only two entries per dependence record r: the latest access date and the latest write date. We call *access_actions* and *write_actions* the arrays with those entries indexed by dependence record.

The state of scheduled resources is tracked by a *resource_table*, which serves as the scheduler reservation table. This table has one row per resource and $\mathit{window\_size} + \mathit{columns\_max}$ columns, where *columns_max* is the maximum number of columns across the reservation tables of all operations. The first column of the *resource_table* corresponds to the *window_start*. This is just enough state for checking resource conflicts in $[\mathit{window\_start}, \mathit{window\_start} + \mathit{window\_size}]$.

Scoreboard Scheduling is performed by picking each operation i according to the priority order and by calling $\mathit{add\_schedule}(i, \mathit{try\_schedule}(i))$, defined by:

try_schedule Given an operation i, return the earliest dependence- and resource-feasible *issue_date* such that $\mathit{issue\_date} \geq \mathit{window\_start}$. For each dependence record r, collect the following constraints on *issue_date*:

Effect	Constraints
Read$[r]$	$\mathit{issue\_date} \geq \mathit{write\_actions}[r] - \mathit{read\_stage}[i][r] + \mathit{RAW}[r]$
Write$[r]$	$\mathit{issue\_date} \geq \mathit{write\_actions}[r] - \mathit{write\_stage}[i][r] + \mathit{WAW}[r]$
	$\mathit{issue\_date} \geq \mathit{access\_actions}[r]$

The resulting *issue_date* is then incremented while there exists scheduled resource conflicts with the current contents of the *resource_table*.

add_schedule Schedule an operation i at a dependence- and resource-feasible *issue_date* previously returned by *try_schedule*. For each dependence record r, update the action arrays as follows:

Effect	Updates
Read$[r]$	$\mathit{access\_actions}[r] \leftarrow \max(\mathit{access\_actions}[r], \mathit{issue\_date} + \mathit{WAR}[r])$
Write$[r]$	$\mathit{access\_actions}[r] \leftarrow \max(\mathit{access\_actions}[r], \mathit{issue\_date} + \mathit{WAW}[r])$
	$\mathit{write\_actions}[r] \leftarrow \mathit{issue\_date} + \mathit{write\_stage}[i][r]$

In case $\mathit{issue\_date} > \mathit{window\_start} + \mathit{window\_size}$, the *window_start* is set to $\mathit{issue\_date} - \mathit{window\_size}$ and the *resource_table* is shifted accordingly. The operation reservation table is then added into the *resource_table*.

In Figure 2, we illustrate Scoreboard Scheduling of two ST200 VLIW operations, starting from an empty scoreboard. There are three scheduled resources: `ISSUE`, 4 units; `MEM`, one unit; `CTL`, one unit. The *window_start* is zero and the two operations are scheduled at *issue_date* zero. We display *access_actions*$[r]$ and *write_actions*$[r]$ as strings of `a` and `w` from *window_start* to *actions*$[r]$. In Figure 3, several other operations have been scheduled since Figure 2, the latest being `shl $r24` at *issue_date* 4. Then operation `add $r15` is scheduled at *issue_date* 5, due to the RAW dependence on `$r24`. Because *window_size* is 4, the *window_start* is set to 1 and the *resource_table* rows `ISSUE`, `MEM`, `CTL` are shifted.

Theorem 2 *Scoreboard Scheduling correctly enforces the dependence latencies.*

Proof. Calling *add_schedule*$(i,$*issue_date*$_i)$ followed by *try_schedule*$(j,$*issue_date*$_i+$ *latency*$_{i\to j})$ implies that *latency*$_{i\to j}$ satisfies the inequalities $(a), (c), (d), (f)$.

3 Global Instruction Scheduling

3.1 Postpass Inter-Region Scheduling

We define the *inter-region scheduling problem* as scheduling the operations of each scheduling region such that the resource and dependence constraints inherited from the scheduling regions (transitive) predecessors, possibly including self, are satisfied. When the scheduling regions are reduced to basic blocks, we call this problem the *inter-block scheduling problem.* Only inter-region scheduling is allowed to move operations between basic blocks (of the same region).

The basic technique for solving the inter-region scheduling problem is to schedule each region in isolation, then correct the resource and latency constraint violations that may occur along control-flow transfers from one scheduling region to the other by inserting NOP operations. Such *NOP padding* may occur after region entries, before region exits, or both, and this technique is applied after postpass scheduling on state-of-the-art compilers such as the Open64.

Meld Scheduling is a prepass inter-region scheduling technique proposed by Abraham, Kathail, Deitrich [2] that minimizes the amount of NOP padding required after scheduling. This technique is demonstrated using superblocks, which are scheduled from the most frequently executed to the least frequently executed, however it applies to any program partition into acyclic regions.

Consider a dependence whose source operation is inside a scheduling region and whose target operation is outside the scheduling region. Its *latency dangle* is the minimum number of time units required between the exit from the scheduling region and the execution of the target operation to satisfy the dependence. For a dependence whose source operation is outside the scheduling region and whose target operation is inside, its latency dangle is defined in a symmetric way [2].

Meld Scheduling only considers dependence latency dangles, however resource dangles can be similarly defined. Latency dangle constraints originate from predecessor regions or from successor regions, depending on the order the regions are scheduled. Difficulties arise with cycles in the control-flow graph, and also with latency dangles that pass through scheduling regions. These are addressed with conservative assumptions on the dangles.

Meld Scheduling is a prepass technique, so register allocation or basic block alignment may introduce extra code or non-redundant WAR and RAW register dependences. Also with JIT compilation, prepass scheduling is likely to be omitted on cold code regions. On processors without interlocking hardware, compilers must ensure that no execution path presents hazards. In the Open64 compiler, hazards are detected and corrected by a dedicated "instruction bundler".

When focusing on postpass scheduling, the latency and resource dangles of Meld Scheduling are implied by the scoreboard scheduler states at region boundaries. Moreover, we assume that the performance benefits of global code motion are not significant during the postpass scheduling of prepass scheduled regions, so we focus on inter-block scheduling. Last, we would like to avoid duplicate work between an "instruction bundler" and postpass scheduling.

Based on these observations, we propose the *Inter-Block Scoreboard Scheduling* technique to iteratively propagate the dependence and resource constraints of local scheduling across the control-flow graph until fixed-point. As we shall prove, it is possible to ensure this technique converges quickly and preserves prepass schedules, including software pipelines, that are still valid.

3.2 Inter-Block Scoreboard Scheduling

We propagate the scoreboard scheduler states at the start and the end of each basic block for all program basic blocks by using a worklist algorithm, like in forward data-flow analysis [10]. This state comprises *window_start*, the action array entries and the *resource_table*. Each basic block extracted from the worklist is processed by Scoreboard Scheduling its operations in non-decreasing order of their previous *issue_date*s (in program order the first time). This updates the operation *issue_date*s and the state at the end of the basic block.

Following this basic block update, the start scoreboard scheduler states of its successor basic blocks are combined through a meet function (described below) with the end scoreboard scheduler state just obtained. If any start scoreboard scheduler state is changed by the meet function, this means new inter-block scheduling constraints need to be propagated so the corresponding basic block is put on the worklist. Initially, all basic blocks are in the worklist and the constraint propagation is iterated until the worklist is empty.

In order to achieve quick convergence of this constraint propagation, we enforce a *non-decrease rule*: **the operation *issue_date*s do not decrease when rescheduling a basic block.** That is, when scheduling an operation, its release date is the *issue_date* computed the last time the basic block was scheduled. This is implemented in $\mathit{try\_schedule}(i)$ by initializing the search for a feasible $\mathit{issue\_date}_i$ to the maximum of the previous $\mathit{issue\_date}_i$ and the *window_start*.

The meet function propagates the scheduling constraints between two basic blocks connected in the control-flow graph. Each control-flow edge is annotated with a *delay* that accounts for the time elapsed along that edge. Delay is zero for fall-through edges and is the minimum branch latency for other edges. Then:

– Advance the scoreboard scheduler state at the end of the origin basic block by elapsing time so *window_start* reaches the *issue_date* of the last operation plus one (zero if the basic block is empty), plus the *delay* of the connecting control-flow edge (zero if fall-through edge, else the taken branch latency).
– Translate the time of this scoreboard scheduler state so that *window_start* becomes zero. With our implementation, this amounts to subtracting *window_start* from the action array entries and moving the *resource_table*.
– Merge the two scoreboard scheduler states by taking the maximum of the entries of the *resource_table* and of the action arrays.

Theorem 3 *Inter-Block Scoreboard Scheduling converges in bounded time.*

Proof. The latest *issue_date* of a basic block never exceeds the number of operations plus one, times the maximum dependence latency or the maximum span of a reservation table (whichever is larger). The *issue_date*s are also non-decreasing by the non-decrease rule, so they reach a fixed-point in bounded time. The fixed-point of the scoreboard scheduler states follows.

3.3 Characterization of Fixed-Points

Theorem 4 *Any locally scheduled program that satisfies the inter-block scheduling constraints is a fixed-point of Inter-Block Scoreboard Scheduling.*

Proof. By hypothesis, all operations have valid *issue_date*s with respect to basic block instruction scheduling. Also, the inter-block scheduling constraints are satisfied. By the non-decrease rule, each operation previous *issue_date* is the first date tried by Scoreboard Scheduling, and this succeeds.

A first consequence is that any prepass region schedule which satisfies the inter-block scheduling constraints at its boundary basic blocks will be unchanged by Inter-Block Scoreboard Scheduling, provided no non-redundant instruction scheduling constraints are inserted in the region by later compilation steps. Interestingly, this holds for any prepass region scheduling algorithm: superblock scheduling; trace scheduling; wavefront scheduling; and software pipelining.

A second consequence is that Inter-Block Scoreboard Scheduling of a program with enough NOP padding to satisfy the inter-block scheduling constraints will converge with only one Scoreboard Scheduling pass on each basic block. In practice, such explicit NOP padding should be reserved for situations where a high-frequency execution path may suffer from the effects of latency and resource dangles at a control-flow merge with a low-frequency execution path, such as entry to an inner loop header from a loop pre-header.

4 Experimental Results

4.1 Comparing Scoreboard Scheduling to Cycle Scheduling

In the setting of the STMicroelectronics CLI-JIT compiler, we implemented Scoreboard Scheduling as described in Section 2.3 and also a Cycle Scheduling

Origin	Size	IPC	RCost	RPerf.	RQuery
mergesort	12	0.92	2.35	1.00	0.60
maxindex	12	2.00	2.52	1.00	0.67
fft32x32s	20	4.00	2.57	1.00	0.50
autcor	21	1.50	3.34	1.00	1.08
d6arith	27	0.87	2.78	1.00	0.60
sfcfilter	29	2.90	3.00	1.00	0.62
strwc	32	3.56	3.17	1.00	0.70
bitonic	34	3.78	3.55	1.00	1.00
floydall	52	1.41	3.62	1.00	0.67
pframe	59	1.59	3.82	1.00	0.63
polysyn	79	2.55	5.95	1.19	1.29
huffdec2	81	0.80	4.23	1.00	0.56
fft32x32s	83	3.61	5.21	1.09	1.00
dbuffer	108	3.18	5.67	1.03	1.00
polysyn	137	3.51	7.29	1.03	1.50
transfo	230	3.59	9.00	1.16	1.04
qplsf5	231	2.96	8.91	1.13	0.11
polysyn	256	1.63	8.79	1.00	0.57
polysyn	297	3.23	9.95	1.04	0.76
radial33	554	3.26	18.78	1.21	1.95

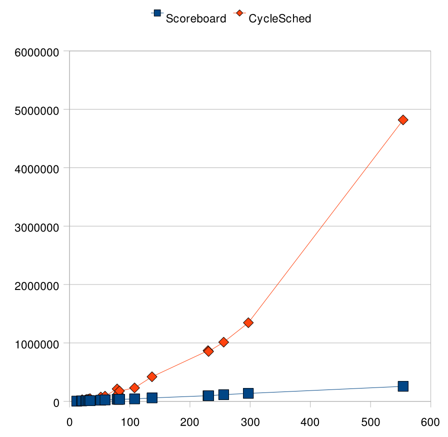


Fig. 4. Benchmark basic blocks and instruction scheduling results.

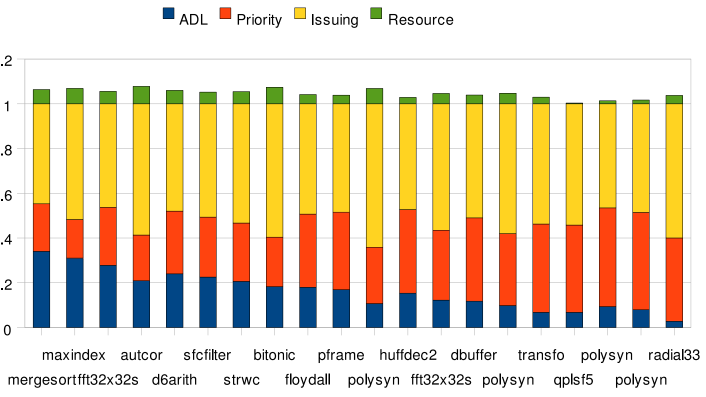
algorithm that closely follows the description of Abraham [1], including reference counting for detecting operations whose predecessors have all been scheduled.

We optimized this Cycle Scheduling implementation for speed. In particular, we replaced the dependence graph by a variant of the Array of Dependence Lists (ADL) of Verbrugge [15] to ensure a $O(n)$ time complexity of the dependence graph construction. This implies conservative memory dependences, however we assume such a restriction is acceptable for postpass scheduling. We also merged the *ReadyList* and *CCReadyList* of [1] into a single radix-4 priority heap lexicographically ordered by lower available date and higher critical path priority.

We selected a series of basic blocks from STMicroelectronics media processing application codes and performance kernels compiled at the highest optimization level by the Open64-based production compiler for the ST200 VLIW family [5]. The proposed CLI-JIT postpass scheduler was connected to this compiler.

These benchmarks are listed in the left side of Figure 4. Columns **Size** and **IPC** respectively give the number of instructions and of instructions per cycle after Cycle Scheduling. Column **RCost** is the ratio of compilation time between the cycle scheduler and the scoreboard scheduler at $window\_size = 15$. Column **RPerf** is the relative performance of the two schedulers, as measured by inverse of schedule length. Column **RQuery** is the ratio of compilation time for resource checking between the cycle scheduler and the scoreboard scheduler. In the right side of Figure 4, we plot the compilation time as function of basic block size. Unlike Cycle Scheduling, Scoreboard Scheduling clearly operates in linear-time.

To understand how compilation time is spent, we display in Figure 5 the stacked contributions of the different scheduling phases normalized by the total instruction scheduling time, so their sum is one. We also single out the cumulative time spent in resource checking, yielding the bars above one. For Cycle Scheduling (left side), the cost of computing the dependences **ADL** becomes relatively smaller, as it is linear with basic block size. The **Priority** computing



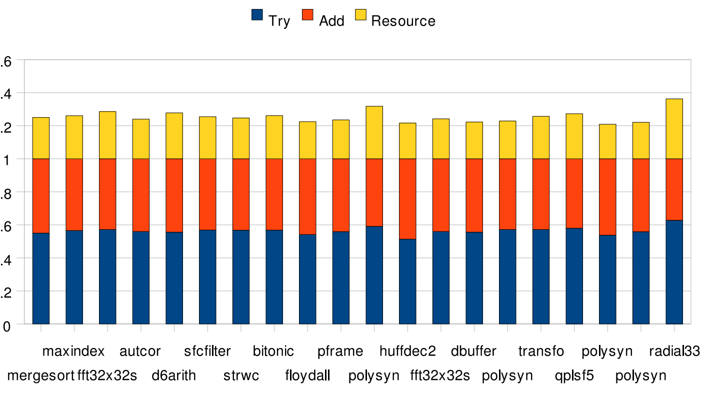


Fig. 5. Time breakdown for Cycle Scheduling and Scoreboard Scheduling.

phase is of comparable time complexity yet smaller than the operation **Issuing** phase. For Scoreboard Scheduling (right side), the **Try** schedule phase is consistently slightly more expensive than the **Add** schedule phase.

It also appears from Figure 5 that using finite state automata as proposed by Bala & Rubin [3] for speeding up resource checking is not always justified, in particular for processors whose reservation tables are single-cycle. For more complex processors, it would be straightforward to replace the *resource_table* of a Scoreboard Scheduling implementation by such finite state automata.

4.2 Experiments with the STMicroelectronics CLI-JIT

STMicroelectronics is developing a JIT compiler for the CLI program representation [4], whose targets are the STMicroelectronics ST200 VLIW family and the ARM processors models 926E and 1176. Media processing applications are developed in C or C++ for the sake of high performances, so this CLI-JIT compiler does not need to support managed data or other virtual machine features that are required for the execution of C# programs. The STMicroelectronics CLI-JIT compiler generates native code by running the following phases:

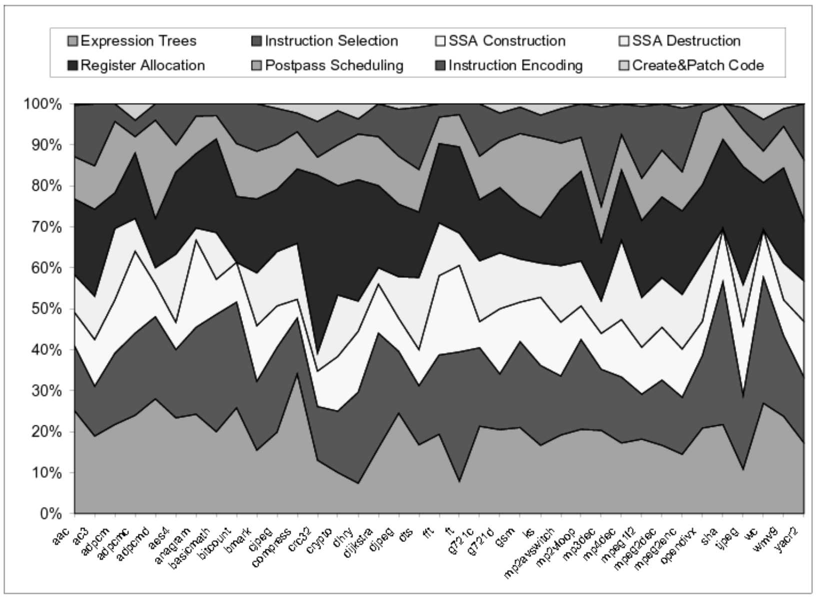


Fig. 6. STMicroelectronics CLI-JIT compilation time breakdown.

Expression Trees The CLI expressions of the evaluation stack are typed and converted to a tree form.

Instruction Selection Machine-level instructions are generated and the calling conventions are implemented.

SSA Construction, SSA Destruction Coalesce register copies, satisfy the ISA and calling conventions register constraints.

Register Allocation Linear-scan register allocation of [17].

Postpass Scheduling Inter-Block Scoreboard Scheduling as discussed.

Instruction Encoding Encode instructions, match bundle templates, encode instruction groups into bundles.

Create & Patch Code Implant the native code in memory, including trampolines and relay jumps.

We applied the STMicroelectronics CLI-JIT compiler to a mix of audio, cryptographic, video, signal processing, and media-bench programs, whose names appear in Figure 6. These benchmarks were compiled from C to CLI executables using the `gcc/st/cli` compiler [4] and the Mono `ilasm` tool. These experiments show that Postpass Scheduling consumes on geometric average 10% of JIT compilation time, closer to the 7% average of Instruction Encoding than to the 18% average of the linear-scan Register Allocation. Instruction Encoding is more expensive on the ST200 VLIW processors than on scalar processors because it entails an additional bundle template matching step, based on the parallel instructions group that result from scheduling and the current PC alignment.

5 Conclusions

We propose a postpass instruction scheduling technique motivated by Just-In-Time (JIT) compilation for VLIW processors. This technique combines two ideas: Scoreboard Scheduling, a restriction of classic Operation Scheduling that considers only the time slots inside a window that moves forward in time; and Inter-Block Scheduling, an iterative propagation of the scheduling constraints across the control-flow graph, subject to the non-decrease of the schedule dates. This Inter-Block Scoreboard Scheduling technique offers three benefits:

– reducing the instruction scheduling compilation time, compared to classic Cycle Scheduling and Operation Scheduling;
– ensuring that all program paths do not present scheduling hazards, as required by processors without interlocking hardware;
– preserving prepass region schedules that are still valid when postpass scheduling runs, in particular software pipelines without spill code.

Experiments with the STMicroelectronics ST200 VLIW production compiler and the STMicroelectronics CLI-JIT compiler confirm the interest of our approach.

Our results further indicate that compiler instruction schedules produced by Cycle Scheduling and Operation Scheduling are essentially unchanged by the hardware operation scheduler of out-of-order superscalar processors. Indeed active schedules are invariant under Scoreboard Scheduling. Finally, the proposed non-decrease rule provides a simple way of protecting cyclic schedules such as software pipelines from the effects of rescheduling with an acyclic scheduler.

6 Acknowledgments

Special thanks to Alain Darte for fruitful discussions that improved this paper. Thanks to Giuseppe Desoli, Stefan Freudenberger, Mon-Ping Wang and Marco Garatti, my esteemed colleagues as STMicroelectronics, for the sharing of their extensive knowledge of VLIW compilation techniques.

References

1. S. G. Abraham. Efficient Backtracking Instruction Schedulers. Technical Report HPL- 2000-56, Hewlett-Packard Laboratories, May 2000.
2. S. G. Abraham, V. Kathail, and B. L. Deitrich. Meld Scheduling: Relaxing Scheduling Constraints across Region Boundaries. In *MICRO 29: Proc. of the 29th annual ACM/IEEE int. symp. on Microarchitecture*, pages 308–321, 1996.
3. V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *MICRO 28: Proc. of the 28th annual int. symp. on Microarchitecture*, pages 46–56, 1995.
4. Marco Cornero, Roberto Costa, Ricardo Fernandez Pascual, Andrea Ornstein, and Erven Rohou. An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems. In *2008 International Conference on High Performance Embedded Architectures and Compilers*, 2008.

5. B. Dupont de Dinechin. From Machine Scheduling to VLIW Instruction Scheduling. *ST Journal of Research*, 1(2), 2004.
6. B. Dupont de Dinechin. Scheduling Monotone Interval Orders on Typed Task Systems. In *26th Workshop of the UK Planning And Scheduling Special Interest Group (PlanSIG)*, Prague, Czech Republic, 2007.
7. P. Faraboschi, G Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a Technology Platform for Customizable VLIW Embedded Processing. In *ISCA'00: Proc. of the 27th annual Int. Symposium on Computer Architecture*, pages 203–213, 2000.
8. Allen Leung, Krishna V. Palem, and Amir Pnueli. Scheduling Time-Constrained Instructions on Pipelined Processors. *ACM Trans. Program. Lang. Syst.*, 23(1):73–103, 2001.
9. Steven S. Muchnick and Phillip B. Gibbons. Best of PLDI 1979 – 1999: Efficient Instruction Scheduling for a Pipelined Architecture. *SIGPLAN Notices*, 39(4):167–174, 2004.
10. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
11. Todd A. Proebsting and Christopher W. Fraser. Detecting Pipeline Structural Hazards Quickly. In *POPL'94: Proceedings of the 21st symposium on Principles of Programming Languages*, pages 280–286, New York, NY, USA, 1994. ACM.
12. B. Ramakrishna Rau. Iterative Modulo Scheduling. *International Journal of Parallel Processing*, 24(1):3–64, Feb 1996.
13. A. Sprecher, R. Kolisch, and A. Drexl. Semi-Active, Active, and Non-Delay Schedules for the Resource-Constrained Project Scheduling Problem. *European Journal of Operational Research*, 80:94–102, 1995.
14. V. Tang, J. Siu, A. Vasilevskiy, and M. Mitran. A Framework for Reducing Instruction Scheduling Overhead in Dynamic Compilers. In *CASCON'06: Proc. of the 2006 Conf. of the Center for Advanced Studies on Collaborative Research*, page 5, 2006.
15. C. Verbrugge. Fast Local List Scheduling. Technical Report SABLE-TR-2002-5, School of Computer Science, McGill University, March 2002.
16. Oliver Wahlen, Manuel Hohenauer, Gunnar Braun, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Xiaoning Nie. Extraction of Efficient Instruction Schedulers from Cycle-True Processor Models. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003)*, volume 2826 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2003.
17. C. Wimmer and H. Mössenböck. Optimized Interval Splitting in a Linear Scan Register Allocator. In *VEE'05: Proc. of the 1st ACM/USENIX int. conf. on Virtual Execution Environments*, pages 132–141, 2005.