

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

Benoît Dupont de Dinechin

Research & Development Responsible

STS Compilation Expertise Center

STMicroelectronics Grenoble (France)

`benoit.dupont-de-dinechin@st.com`



## Presentation Outline

- JIT for VLIW Motivations
- Scheduling Background
- Scoreboard Scheduler
- Inter-Block Scheduling
- Experimental Results
- Summary and Conclusions

## JIT for VLIW Motivations

### Systems-On-Chip at STMicroelectronics

- STMicroelectronics SoC(s) primary markets:
  - mobile devices (phones)
  - consumer electronics(set-top boxes, car entertainment)
- STMicroelectronics SoC(s) typically comprise:
  - Host processors: ARM family, ST40/SH4 processors
  - Application processors: DSPs, VLIW-Media (ST200 family)
  - Dedicated or reconfigurable hardware blocks
- Problem: how to expose application processors to third-party application developers beyond firmware API(s)?

STMicroelectronics is investigating the CLI program representation

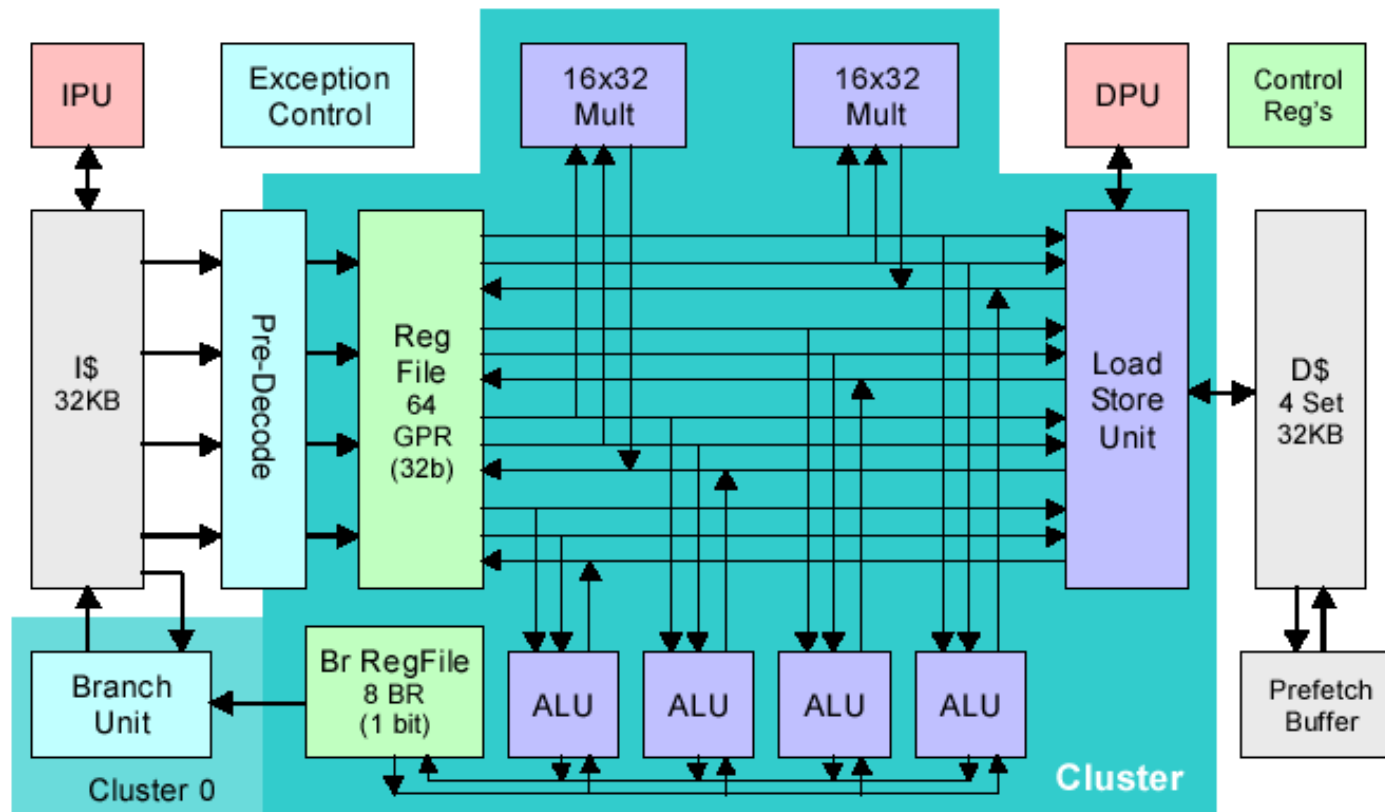
### Microsoft .NET Common Language Infrastructure (CLI)

- The Microsoft Common Language Infrastructure (CLI), is the open ECMA standard that supports .NET
- Unlike Java, the CLI representation supports a wide variety of languages including C
- MS Visual Studio 2003 generates CLI from C, however MS Visual Studio 2005 generates CLI from C++ (C no longer supported)
- Active open-source community with GNU Portable.NET (pnet) focused on C and Mono focused on C#
- Both projects provide a VEE with interpreter, various JITs, compilers and utilities, however they lack a C compiler

Media-processing code is written in C/C++, can be compiled to CLI

### CLI Just-In-Time Compilation at STMicroelectronics

- STMicroelectronics developed a GCC4-based C to CLI compiler (`gcc/st/cli`)  $\rightsquigarrow$  warm reception from GCC4 researchers (INRIA) and from the Mono project, who need a C compiler
- Between 2005 and 2007, STMicroelectronics developed a series of JIT for the ST231 VLIW and the ARM  $\rightsquigarrow$  about half the geometric performance of best static compilation on media processing
- CLI executables are processor-neutral, so binding to a processor can be delayed and a single binary is flashed
- Same C to CLI compilation chain for all processors in SoC  $\rightsquigarrow$  enable third-party developers to program the application processors (ST240 VLIW), not just the host processors (ARM)
- Expect excellent fit with Software Component frameworks: CLI software components could be optimized after being loaded

The ST200 VLIW Media Family (ST210, ST220, ST231, ST240)

- 4-issue VLIW from the Lx architecture Faraboschi et al. [ISCA'00]
- partially predicated with SELECT operations (Fisher style VLIW)

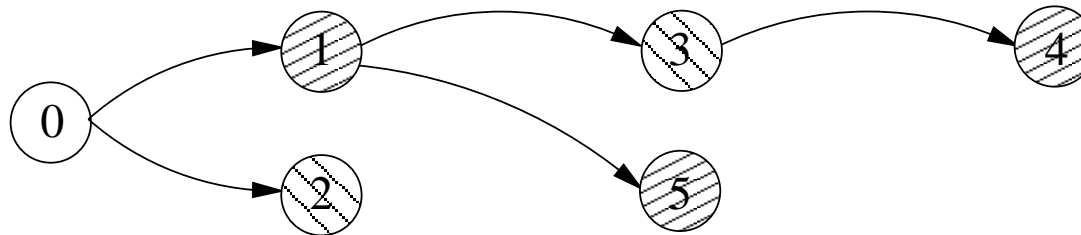
### JIT for VLIW Challenges Addressed in this Work

- Instruction Scheduling is a key optimization of VLIW code generation, however it is time- and memory-expensive
- Java-centric JIT compilation either omits prepass scheduling, or restricts it to a few code paths (IBM Testarossa for the IBM zSeries 990 and the POWER4 processors)
- JIT compilation of CLI media processing programs exposes more instruction-level parallelism than Java JIT compilation
- VLIW processors that lack interlocking hardware require hazard-free postpass scheduling across all program paths
- Expensive prepass schedules including software pipelines should not be destroyed by postpass scheduling

Inter-Block Scoreboard Scheduling for postpass VLIW scheduling

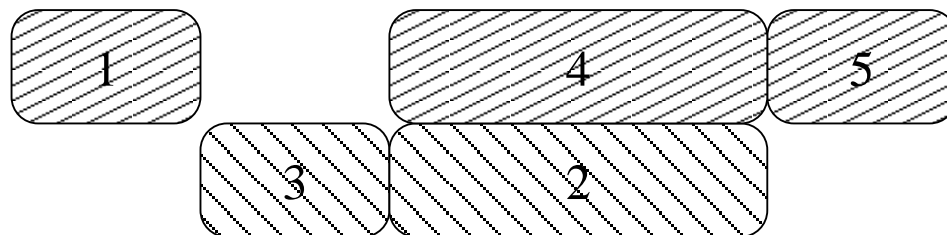
# Scheduling Background

Dependence Graph



Semi-Active Schedule

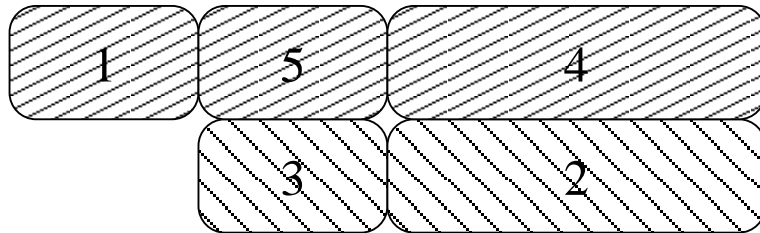
To complete any task earlier, must change an execution sequence:



Produced by any 'greedy scheduler' (no unenforced idling)

Active Schedule

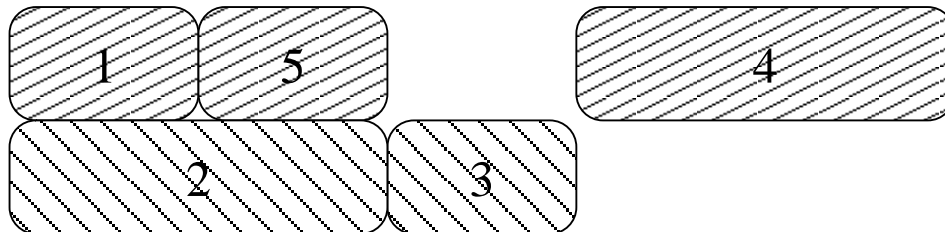
To complete any task earlier, need to delay another task:



Produced by any 'operation scheduler' (serial SGS)

Non-Delay Schedule

No execution resource is left idle if a task can start executing:



Produced by any 'cycle scheduler' (parallel SGS)

### List Scheduling Properties

**Cycle Scheduler** Scan time slots in increasing order

- If two tasks may schedule at same slot, priority breaks ties

**Operation Scheduler** Schedule tasks by priority order

- The priority order must be a topological sort of dependences

### Cost Breakdown of List Scheduling

- Build the dependence graph:  $O(n^2)$
- Maintain and prioritize the dependence-ready tasks:  $O(n \log n)$
- Check resource conflicts and schedule tasks:  $O(nmp)$

With  $n$  number of tasks,  $m$  number of execution resources,  $p$  maximum processing time of tasks

### Applications to Instruction Scheduling

- Notions of Semi-Active, Active, Non-Delay apply to instruction scheduling provided reservation tables are *monotonic*:
  - the reservation row entries are non-increasing with time
- It is known that  $\text{Non-Delay} \subset \text{Active} \subset \text{Semi-Active}$  and Non-Delay may exclude all optimal solutions (unlike Active)
- In case of single-cycle reservation tables, Active schedules are Non-Delay schedules and reservation tables are monotonic
- Classic instruction schedulers [Muchnich 1997] are cycle-based  $\rightsquigarrow$  Non-Delay schedules
- Classic modulo schedulers [Rau 1995, Ruttenberg 1996] are operation-based (with backtracking)  $\rightsquigarrow$  Active schedules

Prepass instruction schedules and modulo schedules are Active

## Scoreboard Scheduler

### Scoreboard Scheduler Principles

Emulate a OOO superscalar processor without register renaming:

- An operation is always scheduled withing a moving time window [*current\_date*, *current\_date*+*window\_size*]
- The execution resources are tracked by a *resource table*:
  - a sliding window into the schedule reservation table
- The dependences are maintained by using two *action arrays*:
  - encode RAW, WAR, WAW latencies with only two arrays
  - *access\_actions* and *write\_actions* record last time a dependence resource was accessed or written
  - a dependence resource is a (sub)register or PC or MEM

Scoreboard Scheduler Example 1 (*window\_size* = 5)

issue=0	ldb \$r23 = 9[\$r16]							
[0]	+0	+1	+2	+3	+4	+5	+6	+7
-----								
ISSUE	1							
MEM	1							
CTL								
ODD								
EVEN								
-----								
PC	a							
GR16	a							
GR23	aw	aw	w	w	w	w		
MEM	a	a						

issue=0	add \$r18 = \$r18, -12							
[0]	+0	+1	+2	+3	+4	+5	+6	+7
-----								
ISSUE	2							
MEM	1							
CTL								
ODD								
EVEN								
-----								
PC	a							
GR16	a							
GR18	aw	aw	w	w				
GR23	aw	aw	w	w	w	w		
MEM	a	a						

Scoreboard Scheduler Example 2 (*window\_size = 5*)

issue=4	shl \$r24 = \$r24, 24							
[0]	+0	+1	+2	+3	+4	+5	+6	+7
ISSUE	3	2	1	3	1			
MEM	1	1	1	1				
CTL								
ODD								
EVEN								
PC	a	a	a	a	a			
BR3	aw	aw	aw	w	w			
GR16	aw	aw	aw	aw	aw	w	w	
GR18	aw	aw	w	w				
GR19	aw	aw	w	w				
GR20	aw	aw	aw	aw	w	w	w	w
GR23	aw	aw	aw	aw	aw	w	w	
GR24	aw	aw	aw	aw	aw	aw	w	w
MEM	a	a	a	a	a			

issue=5	add \$r15 = \$r15, \$r24							
[1]	+0	+1	+2	+3	+4	+5	+6	+7
ISSUE	2	1	3	1	1			
MEM	1	1	1					
CTL								
ODD								
EVEN								
PC	a	a	a	a	a			
BR3	aw	aw	w	w				
GR15	aw	aw	aw	aw	aw	aw	w	w
GR16	aw	aw	aw	aw	w	w		
GR18	aw	w	w					
GR19	aw	w	w					
GR20	aw	aw	aw	w	w	w	w	
GR23	aw	aw	aw	aw	w	w		
GR24	aw	aw	aw	aw	aw	w	w	
MEM	a	a	a	a				

## Scoreboard Scheduling Properties

Efficient implementation compared to list schedulers:

- $O(n)$  maintenance of the action arrays instead of  $O(n^2)$  dependence graph construction
- number of resource availability checks bounded by  $window\_size + columns\_count\_max$
- no priority management saves the  $O(n \log n)$  contribution

**Proposition 1** *Scoreboard scheduling produces semi-active schedules.*

**Definition 1** *A fixed-point schedule is a schedule that is invariant when used as a priority list in an operation scheduler.*

**Theorem 1** *Scoreboard scheduling a fixed-point schedule yields the same.*

**Corollary 1** *Scoreboard scheduling an active schedule yields the same.*

## Inter-Block Scheduling

### Inter-Region Scheduling

**Definition 2** *The inter-region scheduling problem is scheduling each scheduling region such that the resource and dependence constraints inherited from the scheduling regions (transitive) predecessors, possibly including self, are satisfied.*

- Inter-region scheduling does not move code between regions
- Basic technique is NOP padding at region boundaries (Open64)
- Meld Scheduling [Abraham 1996] is a prepass inter-region scheduling technique demonstrated on superblocks
- When the scheduling regions are reduced to basic blocks, we call it *inter-block scheduling problem*

We solve postpass inter-block scheduling as a data-flow problem

### Inter-Block Scheduling Data-Flow Analysis

A forward data-flow problem solved by a work-list algorithm:

**Propagated Facts** The scoreboard scheduler states at the start and at the end of each basic block **and** the operations *issue\_dates* relative to the start of their basic block

**Transfer Function** Scoreboard schedule the operations according to the order of their previous *issue\_date* **moreover** the *issue\_date* of any given operation cannot decrease

**Meet Function** Translate time to nullify the scoreboard scheduler *current\_dates* at the end of predecessor basic blocks, then take the maximum of the resource table and the action array entries

**Theorem 2** *The proposed data-flow analysis formulation for inter-block scheduling is monotone.*

### Data-Flow Convergence and Fix-Points

**Theorem 3** *The inter-block scoreboard scheduling data-flow analysis converges in bounded time.*

Thanks to the Semi-Active property of scoreboard scheduling □

**Theorem 4** *Any active schedule that obeys the inter-block scheduling constraints is a fixed-point of the inter-block scoreboard scheduling data-flow analysis.*

### Use of NOP Padding to Improve Fix-Points

- Loop headers schedules may suffer damaging effects from long pending dependence latencies originating in loop pre-headers
- This is addressed by pre-padding the low-frequency paths that merge into a high-frequency path, as identified by the *mutual most likely* rule of Trace Scheduling

## Experimental Results

### The STMicroelectronics CLI-JIT

**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 ABI conventions are implemented.

**SSA Construction, SSA Destruction** Coalesce register copies and ensure ISA register operand constraints.

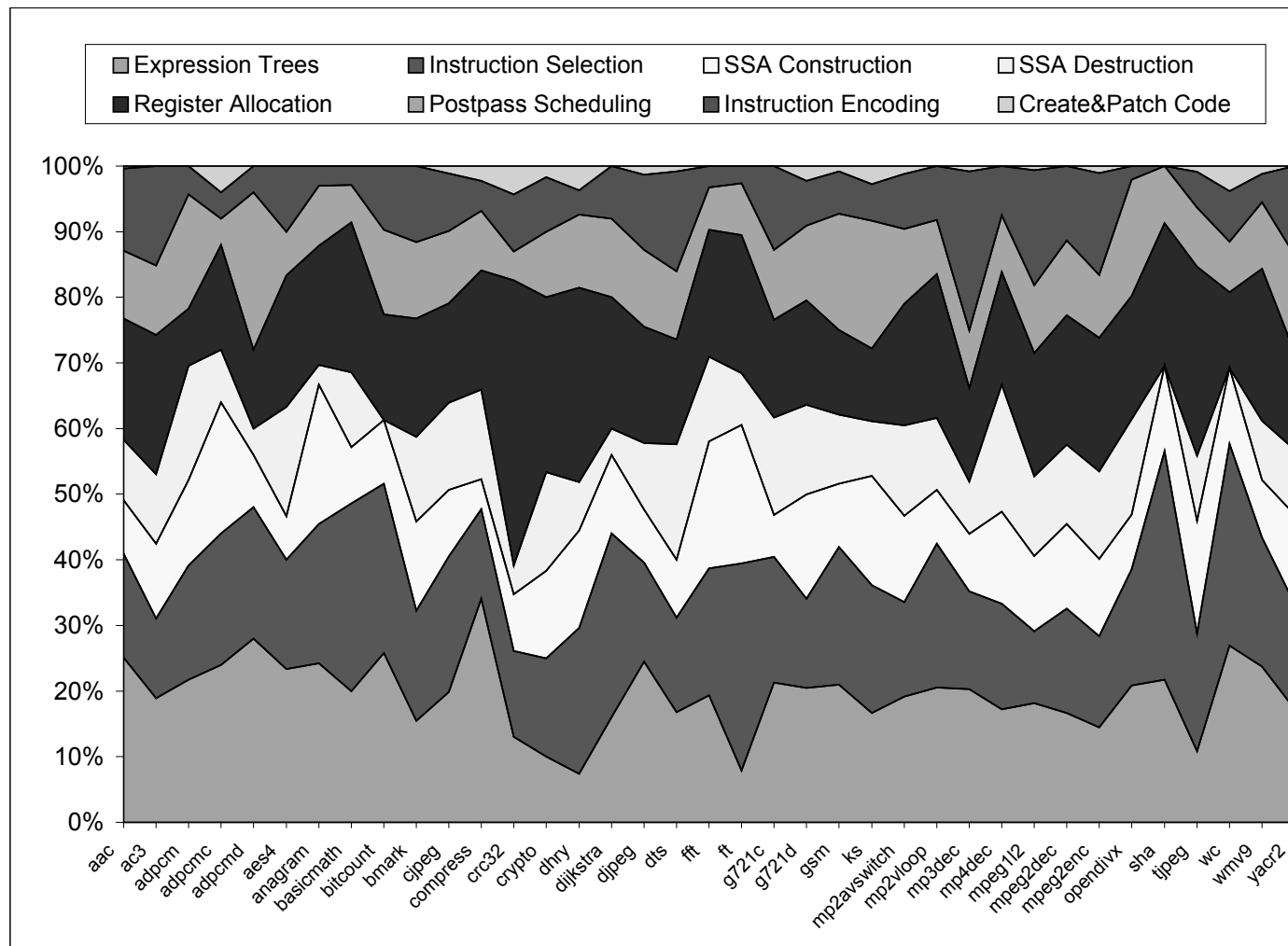
**Register Allocation** Linear-scan register allocation [Wimmer 2005]

**Postpass Scheduling** Inter-block scoreboard scheduling

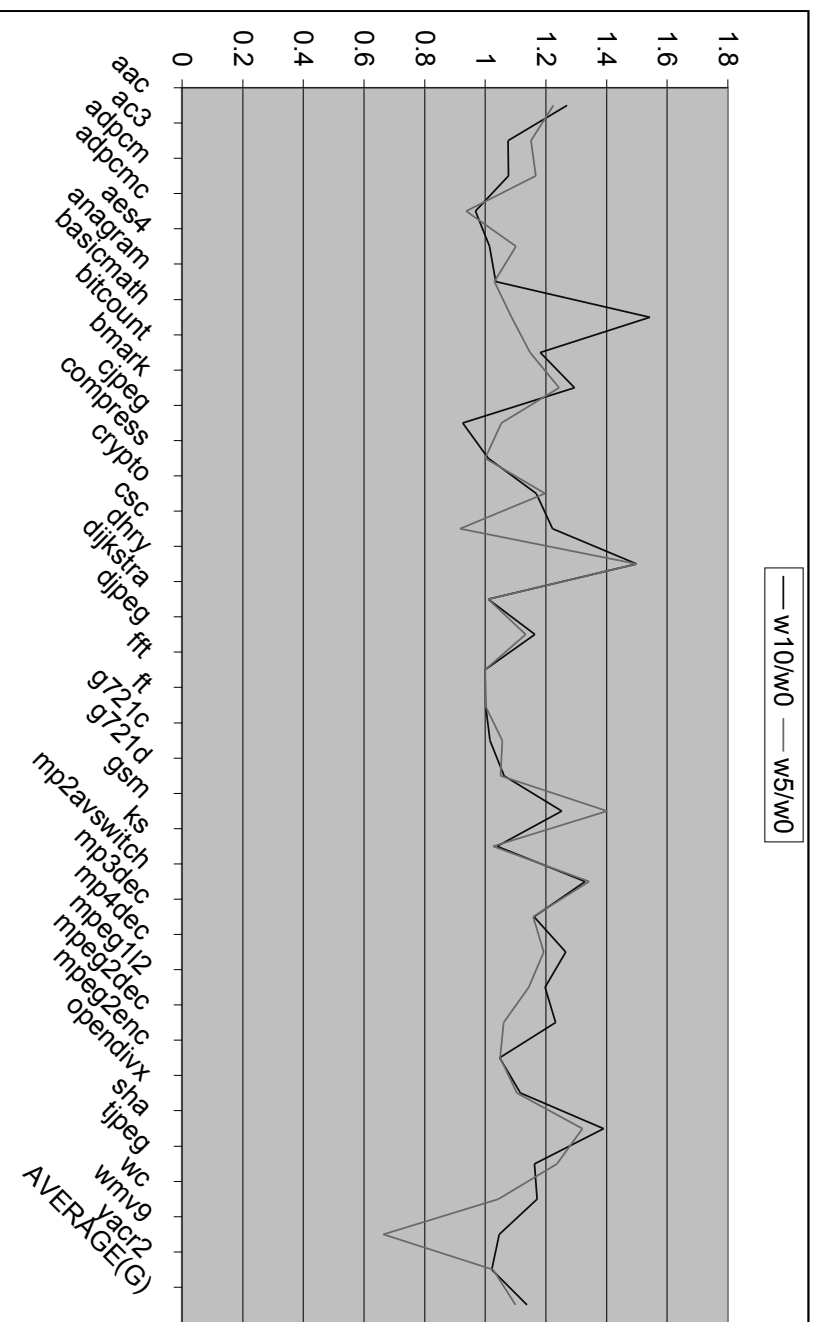
**Instruction Encoding** Encode instructions, match bundle templates, encode instruction groups into bundles.

**Create & Patch Code** Emit code, trampolines and relay jumps.

STMicroelectronics CLI-JIT compilation time breakdown



Scoreboard scheduling performance improvements



## Summary and Conclusions

New postpass instruction scheduling addresses JIT compilation for VLIW processors:

- Low postpass scheduling time compared to list scheduling, thanks to the emulation of a OOO-like dynamic scheduler
- Produces correct schedules across all program paths  $\rightsquigarrow$  a requirement for non-interlocked VLIW processors
- Does not change active schedules  $\rightsquigarrow$  does not change prepass schedules and software pipelines that are active
- Fully implemented in the STMicroelectronics CLI JIT compiler for ST200 VLIW processors and ARM processors (V5e, V6)

Future work: compute scheduling priorities for non-postpass scheduled code by reverse scoreboard scheduling.