KALRAY
THE POWER OF MORE

SOFTWARE PIPELINING FOR KERNEL CODE GENERATION

MLIR (Un)School 2025

Benoît Dupont de Dinechin,
Chief Technology Officer

www.kalrayinc.com

AGENDA

SOFTWARE PIPELINING MOTIVATION

Compiler optimizations to approach the effects of OoO superscalar execution in inner loops

- reorganize instructions across iterations,
- rename registers, preload registers,
- predicate or speculate instructions.

SUPERSCALAR OUT-OF-ORDER (OOO)	SUPERSCALAR IN-ORDER (ARM A53)	VLIW (FISHER STYLE, NOT EPIC)
Dynamic instruction scheduling	Multiple instruction issue in one cycle if no resource or register conflicts	Multiple instruction issue in one cycle if bundled by compiler
Architectural register renaming	Optional architectural register renaming	No architectural register renaming
Speculative execution past predicted branches	No speculative execution past branches	Control speculative execution directed by compiler
Hit-under-miss L1 cache	Hit-under-miss L1 cache	Exploit hit-under-miss or cache-bypass loads
Hardware prefetching into L2 cache	Hardware prefetching into L2 cache	

SOFTWARE PIPELINING EXAMPLE ON ARM A53

‘SAXPY’ loop from BLAS

Need to use [restrict] to inform compiler there are no data dependences between the memory accesses

```
#include "math.h"

void
saxpy(int n, double a, double x[restrict],
      double y[restrict], double z[restrict])
{
  for (int i = 0; i < n; i++) {
    z[i] = a * x[i] + y[i];
  }
}
```

Compilation with GCC –O2 on a ARM workstation (cortex A54 cores)

Loop: load x[i], load y[i], FMA, store z[i], update loop counter, loop branch back

```
        .type    saxpy, %function
saxpy:
        cmp     w0, 0
        ble     .L1
        mov     x4, 0
        .p2align 3
.L3:
        ldr     d1, [x1, x4, lsl 3]
        ldr     d2, [x2, x4, lsl 3]
        fmadd    d1, d1, d0, d2
        str     d1, [x3, x4, lsl 3]
        add     x4, x4, 1
        cmp     w0, w4
        bgt     .L3
.L1:
        ret
        .size    saxpy, .-saxpy
```

SOFTWARE PIPELINING USING MODULO SCHEDULING

Loop body schedule

One iteration takes 11 clock cycles

Cycle	LSU	FPU	Other
0	v1=[x4*0x8+x1]		
1	v2=[x4+0x8+x2]		
2			
3		v1={v1*v0+v2}	
4			
5			
6			
7			
8	[x4*0x8+x3]=v1		x4=x4+0x1
9			cc=cmp(x0,x4)
10			pc={(cc>0)?Loop:pc}

Overlapped execution of successive loop body

One iteration started every λ = 3 clock cycles

Cycle	LSU	FPU	Other
0	v1=[x4*0x8+x1]		
1	v2=[x4+0x8+x2]		
2			
3	v1'=[x4'*0x8+x1]	v1={v1*v0+v2}	
4	v2'=[x4'+0x8+x2]		
5			
6	v1''=[x4''*0x8+x1]	v1'={v1'*v0'+v2'}	
7	v2''=[x4''+0x8+x2]		
3*N+5	[x4*0x8+x3]=v1		x4=x4+0x1
3*N+6	v1'''=[x4'''*0x8+x1]	v1''={v1''*v0''+v2''}	cc=cmp(x0,x4)
3*N+7	v2'''=[x4'''+0x8+x2]		pc={(cc>0)?Loop:pc}
…	[x4'*0x8+x3]=v1'		x4'=x4'+0x1
…			cc=cmp(x0,x4')
…			pc={(cc>0)?Loop:pc}
…			x4''=x4''+0x1
…			cc=cmp(x0,x4'')
…			pc={(cc>0)?Loop:pc}

pipelined loop prolog

pipelined loop kernel

pipelined loop epilog

In order to software pipeline, some loop temporary variables must be 'modulo expanded'

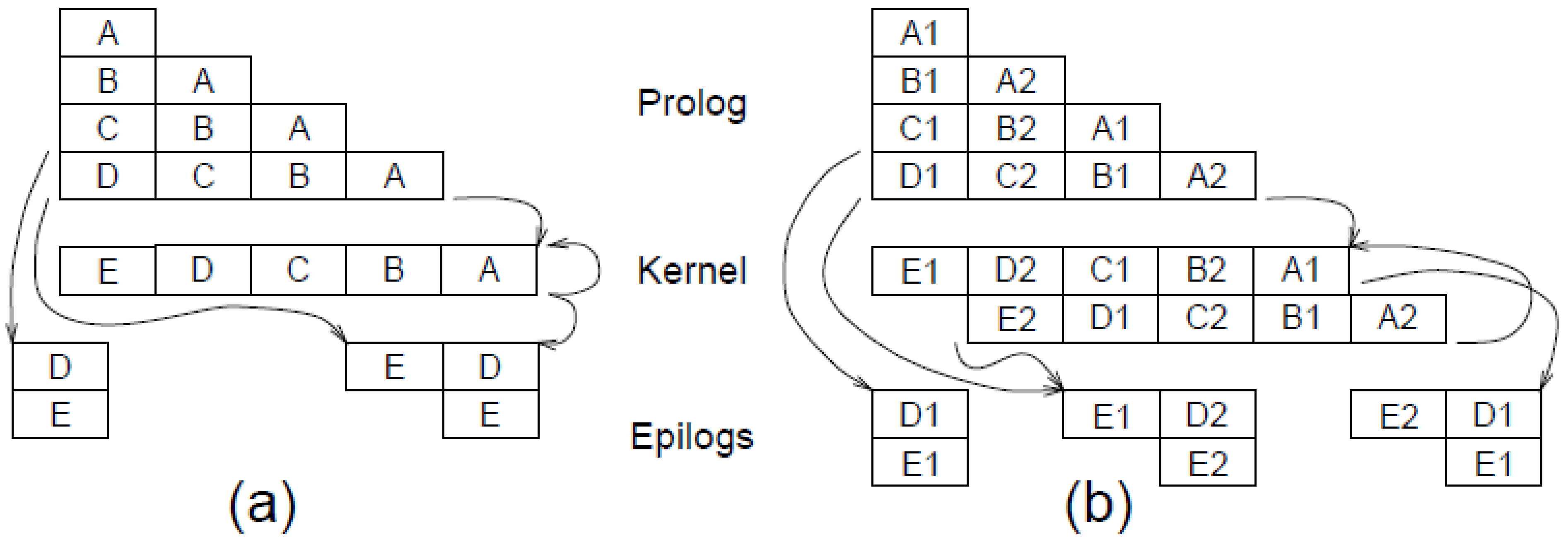
Modulo variable expansion (MVE) consumes registers and requires loop kernel unrolling

MVE can be omitted on OoO CPU implementations with architectural register renaming

SOFTWARE PIPELINE CONSTRUCTION FROM KERNEL

Software pipelined reconstruction from the 1-periodic cyclic schedule at period λ

- Software pipeline stages A, B, C, D, E: blocks of instructions that span λ cycles
- Modulo expansion: variables that live more than λ cycles are assigned different registers
- Kernel unrolling: enable the cyclic register renaming of modulo expanded variables
- Alternative to modulo expansion: hardware or software register move of temporaries



SUCCESSFUL VLIW ACCELERATORS

Main applications in image processing, signal/telecom, AI

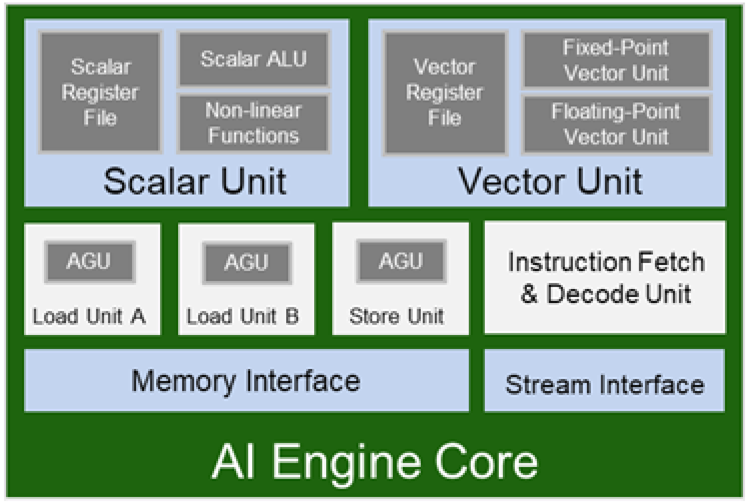
Synopsys, CEVA, Intel/Habana, AMD / Xilinx

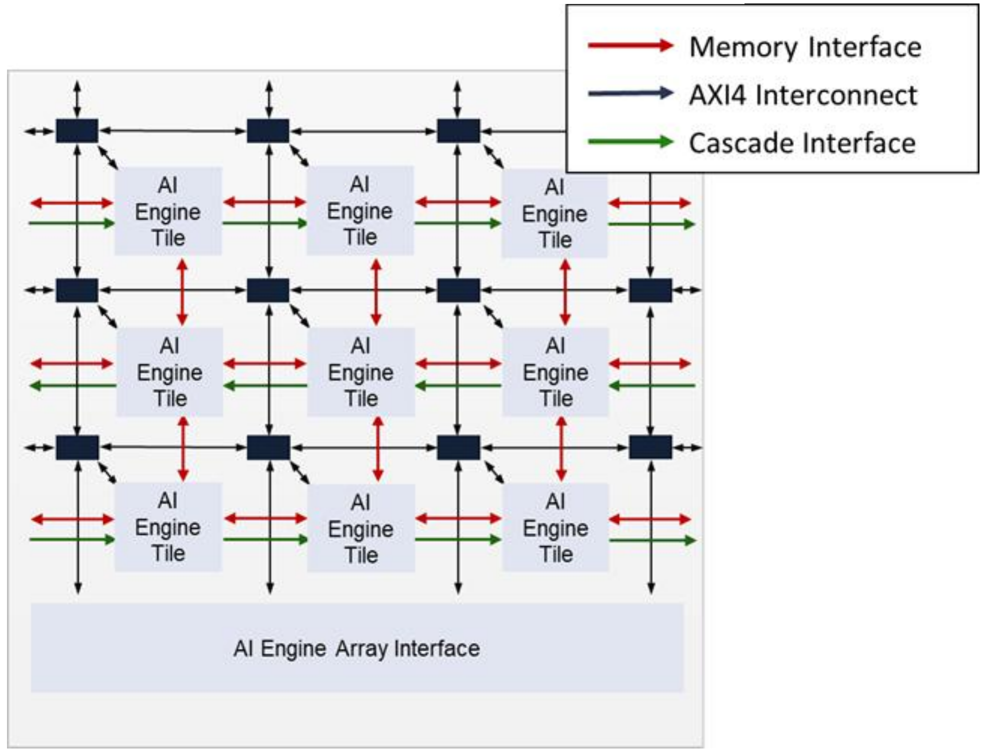
INTEL GAUDI AI ACCELERATORS

- Gaudi2 integrates Habana's fourth generation Tensor Processor Core.
- The TPC is a general purpose VLIW processor which is 256B SIMD wide and supports FP32, BF16, FP16 & FP8, in addition to INT32, INT16 & INT8 data types.
- the TPC exposes a DMA-free programming model which significantly eases SW development.

XILINX VERSAL / AMD XDNA AI ENGINES

- Each AI Engine tile consists of a very long instruction word (VLIW), single instruction multiple data (SIMD) vector processor optimized for machine learning and advanced signal processing applications.
- AMD XDNA is a spatial dataflow NPU architecture consisting of a tiled array of AI Engine processors.





VLIW ARCHITECTURE PRINCIPLES

Promote "horizontal microcode" to bundles of RISC-like instructions

Co-design architecture, microarchitecture and compiler optimizations

Classic VLIW architecture (J. A. Fisher)

- SELECT operation on Boolean value
- Conditional load/store/FPU operations
- Dismissible loads (non-trapping)
- Multi-way conditional branches

Compiler techniques

- Trace scheduling (global instruction scheduling)
- Partial predication (S. Freudenberger algorithm)

Main examples

- Multiflow TRACE (1987)
- Philips Trimedia (1998)
- HP Labs Lx / ST200 family (2000)

EPIC VLIW architecture (B. R. Rau)

- Fully predicated ISA + predicate define instructions
- Speculative loads (control speculation)
- Advanced loads (data speculation)
- Rotating registers

Compiler techniques

- Modulo scheduling (software pipelining)
- Full predication (R-K algorithm, J. Fang algorithm)

Main examples

- Cydrome Cydra-5 (1987)
- TI C6X DSPs (1998)
- HP-intel IA64 (2001)

MULTIFLOW TRACE 7 SERIES

Designed as a target for a trace scheduling compiler

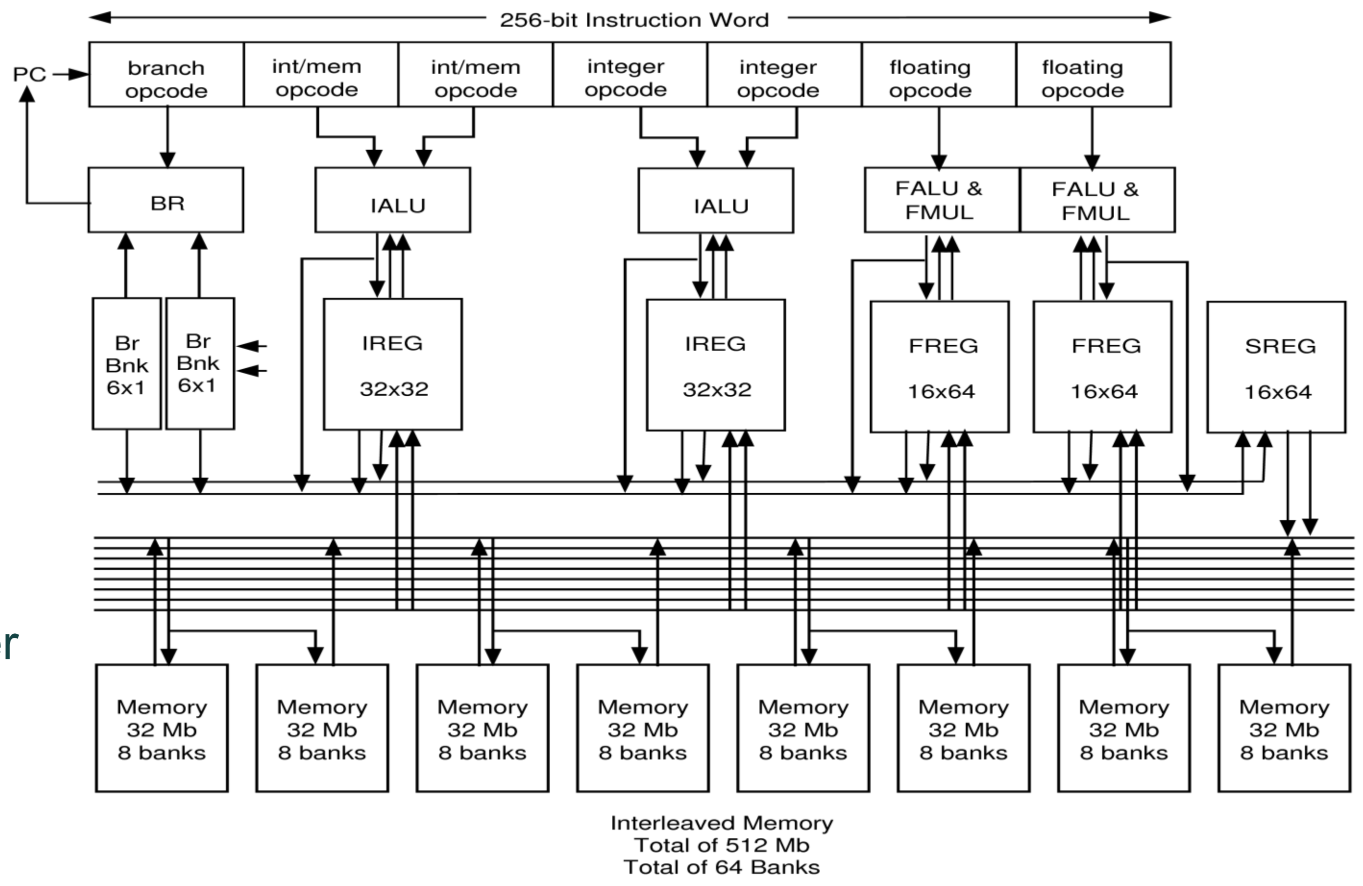
The most suitable VLIW should exhibit four basic features [Colwell et al. 1998 IEEE TC].

- One central controller issues a single long instruction word per cycle.
- Each long instruction simultaneously initiates many small independent operations.
- Each operation requires a small, statically predictable number of cycles to execute.
- Each operation can be pipelined.

In the same spirit as MIPS and the IBM 801, the microarchitecture is exposed to the compiler so that the compiler can make better decisions about resource usage

- the architecture is load/store,
- there is no microcode.

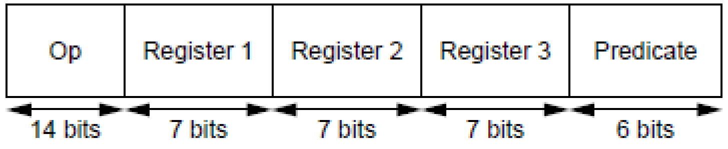
Multiflow TRACE 7/300 is the entry model with one 'cluster' of execution units



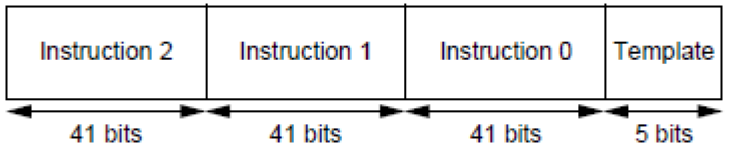
ITANIUM ARCHITECTURE (IA64)

“Evolution of the VLIW architecture” through the Cydrome Cydra-5 then the HPL Play Doh

Encoding instructions requires 41 bits and instructions are fully predicated



Bundles of 3 instructions are encoded in 128 bits, with template bits that specify the parallel groups

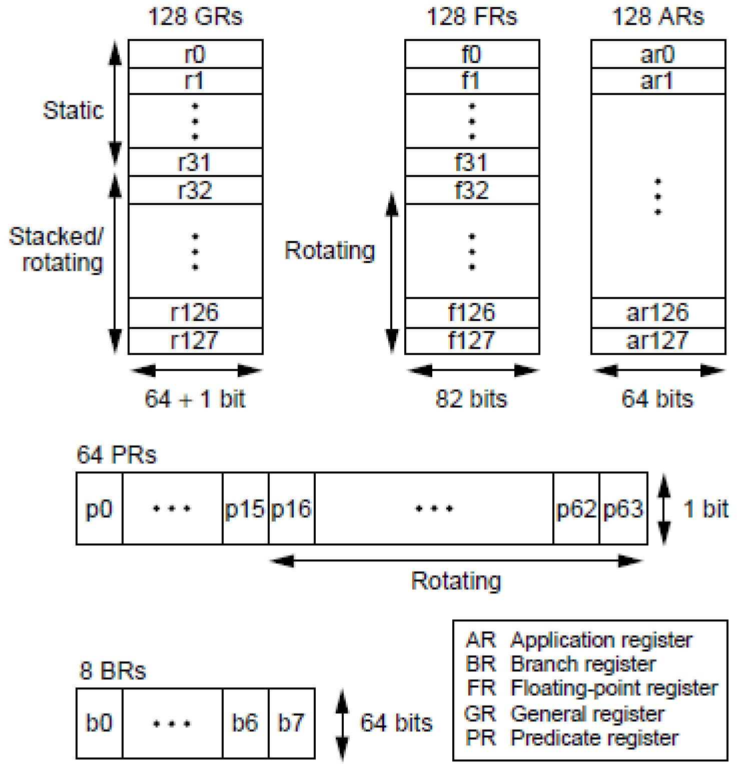


The GR registers have 64+1 bits for “Not a Thing”

Loads can be control-speculative, in case of traps the “Not a Thing” bit is set and checked later

Loads can be advanced before possibly interfering stores, with interference checked by the “Advanced Load Address Table” (ALAT)

Rotating register are available on GR, FR (floating-point) and PR (predicate)



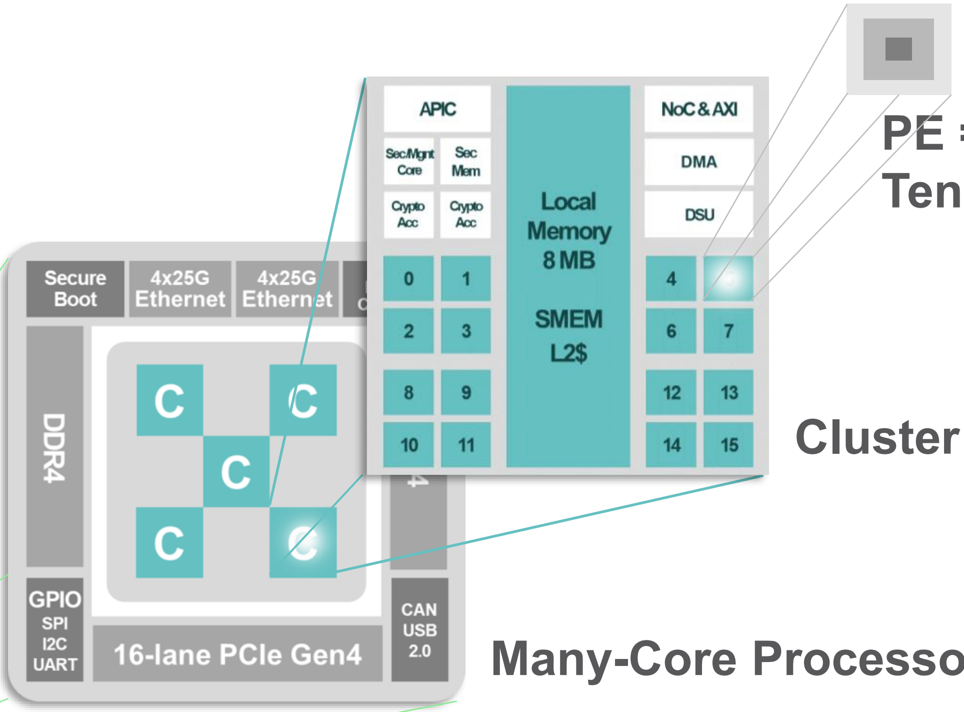
KALRAY MPPA® SCALABLE MANY-CORE ACCELERATORS

3rd-gen MPPA® processor manufactured in TSMC 16nm, running at up to 1.2 GHz

4× MPPA3 v2 processors with 80 PEs per processor:

- 4× 49 TOPS INT8.32
- 4× 24.5 TFLOPS FP16.32
- 4× 1.5 TFLOPS FP32

Multiple Processors per Card



PE = VLIW Core + Tensor Coprocessor

Cluster of 16 PEs

Many-Core Processor

MPPA3 COOLIDGE V2 64-BIT KV3 CORE

VLIW architecture co-designed for compilers to appear as an in-order superscalar core

Vector-scalar ISA

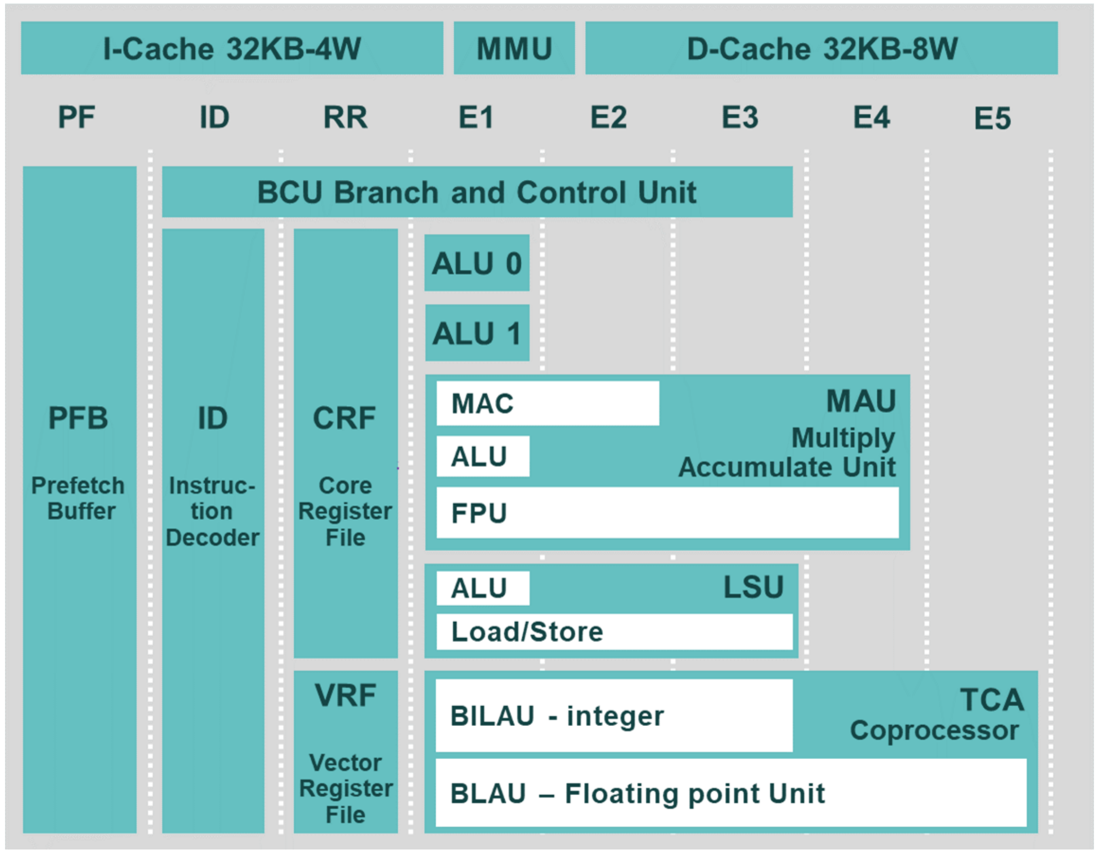
- 64x 64-bit general-purpose registers
- SIMD operands can be single registers (64-bit), register pairs (128-bit) or register quadruples (256-bit)
- 128-bit SIMD instructions by dual-issuing 64-bit on the two ALUS or by using the 128-bit FPU datapath
- FPU capable of 4x FP32 FDP2A operations / cycle
  The FDP2A operator computes $a \pm b \times c \pm d \times e$
- 256-bit load/store unit with byte masking

DSP capabilities

- Counted or while hardware loops with early exits
- Non-temporal loads (L1 cache bypass / preload)

CPU capabilities

- 4 privilege levels (rings), MMU (runs Linux kernel)
- Recursive ISA virtualization (Popek & Goldberg)



VLIW CORE PIPELINE

MOTIVATIONS FOR DIRECT CODE GENERATION FROM MLIR

Kalray MPPA kernels for AI inference toolchain are manually written

- C/C++ with GCC asm statements
- Assembler code with CPP macros

Software pipelining in C/C++ GCC and LLVM Kalray compilers achieved mixed results

- LLVM software pipeliner retargeting terminated after unrolling created multi basic block loops
- GCC software pipeliner improved with MVE and presented at the GNU Tools Cauldron 2024
- Both software pipeliners require single basic block counted loops and rely on Swing Modulo Scheduling
- Passing accurate memory dependence information from C/C++ is an unsolved problem

« A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions »

- Architectural registers represented as MLIR types with register number attribute
- SIMD and stream instruction selection, loop unroll-and-jam, spill-free register allocation
- Implemented using the xDSL compiler framework

AGENDA

PARALLEL MACHINE SCHEDULING

Deterministic scheduling of an operation set $\{O_i\}_{1 \le i \le n}$ to execute on m identical processors

A valid schedule is a set of schedule dates $\{\sigma_i\}_{1 \le i \le n}$ (for $\{O_i\}_{1 \le i \le n}$) that satisfies:

- Execution of O_i consumes p_i units of time on an arbitrary processor
- For each operation O_i, $r_i \le \sigma_i$ (release date) and $C_i = \sigma_i + p_i \le d_i$ (deadline or due date)
- For each precedence constraint between O_i and O_j: $\sigma_i + p_i + l_i^j \le \sigma_j$ (dependence latency is $p_i + l_i^j$)

Most parallel machine scheduling problems studied assume $l_i^j = 0$ (precedences, not dependences)

The α|β|γ scheduling problem notation:

- α environment: 1 (one processor), P2 (two processors), Pm (m processors), P (> 2 processors), …
- β assumptions: p_i processing times, r_i release dates, d_i due dates, prec(l_ij) precedence with time lag l_i^j, …
- γ objective: _ (feasibility), C_{max} (max C_i : C_i = completion time O_i), L_{max} (max L_i : L_i = lateness $O_i = C_i - d_i$), …

Most polynomial-time solvable scheduling problems assume $p_i = 1$ (Unit Execution Time or UET)

Limit case of no resource constraints (unbounded number of parallel processors)

- Scheduling problem can be solved by a single-source longest path algorithm on the dependence graph

PARALLEL MACHINE SCHEDULING COMPLEXITY

Polynomial Time

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

NP Hard

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

PARALLEL MACHINE SCHEDULING EXTENSIONS

Extensions of the resource constraints to model real-world digital processors

Multiprocessor tasks (denoted size$_i$ in the β field):

- Executing operation O$_i$ require size$_i$ processors for p$_i$ units of time
- Polynomial time: $P2|r_i; p_i = 1; size_i|L_{max}$, $P2|prec; p_i = p; size_i|C_{max}$
- NP hard: $P|p_i = 1; size_i|C_{max}$

Typed tasks (denoted Σ^k in the α field):

- Processors are partitioned into k types and each operation Oi may only execute on processor of type τi
- Polynomial time: $\Sigma^k P|intOrder; p_i = 1|C_{max}$, $\Sigma^k P|intOrder(mono\ l_i^j); r_i; d_i; p_i = 1|-$
- NP hard: $\Sigma^2 1|chains; p_i = 1|C_{max}$, $\Sigma^2 1|forest; p_i = 1|C_{max}$

Cumulative resource constraints (generalize typed tasks and multiprocessor tasks):

- Processors are replaced by a set of resources, whose availabilities are given by an integral vector B.
- Each operation O$_i$ is also associated with an integral vector b$_i$ of resource requirements
- The sum of b$_j$ for all operations O$_j$ executing at a given time does not exceed B

LIST SCHEDULING ALGORITHMS

Greedy scheduling of operations driven by their position in a pre-built priority list

Scheduling with resource constraints always succeeds if the dependence graph is acyclic

Graham List Scheduling (cycle scheduling in compilers)

- Scheduling is performed by scanning the time slots in increasing order
- For each time slot, if a processor is idle, schedule the highest priority operation available
- Performance bounds: $P|prec(l_i^j)|C_{max}$ [Munier 1998], $\Sigma^k P|prec(l_i^j \leq z)|C_{max}$ [Chou 1992]

$$2 - \frac{1}{m(1+\rho)} \qquad \rho \overset{\text{def}}{=} \frac{\max l_j^k}{\min p_i} \qquad 1 + k - \frac{1}{(z+1)\max_i m_i}$$

Job-Based List Scheduling (operation scheduling in compilers)

- 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
- May deadlock if the priority list is not a topological order of the dependences

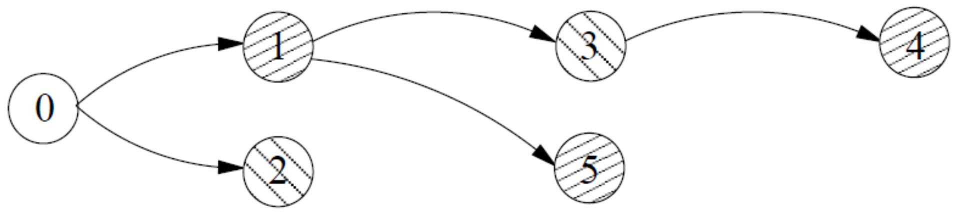
List scheduling priority functions

- Critical path: longest path in the dependence graph from operation to end of execution
- Deadlines: earliest d_i (Jackson's rule), earliest d_i' (d_i modified with 'backward scheduling')

LIST SCHEDULING EXAMPLES

Typed tasks with two processors, one of each type

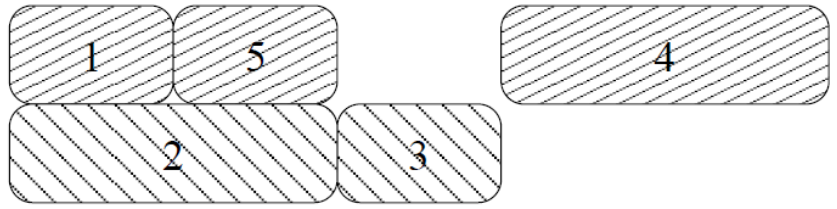
Precedence graph with dummy operation O_0



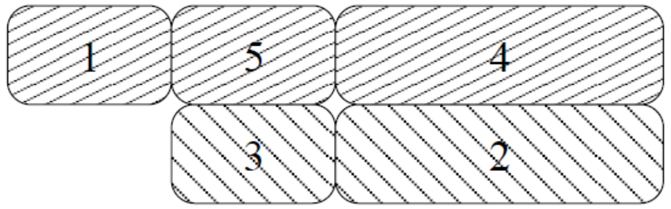
Critical path list scheduling priority

Operation	O_1	O_2	O_3	O_4	O_5
Execution Time	1	2	1	2	1
Priority	4	2	3	2	1

Graham List Scheduling (cycle scheduling)



Job-Based List Scheduling (operation scheduling)



Non-Delay Schedule:

- No execution resources are left idle if there is an operation that could start executing
- May exclude C_{max} or L_{max} optimal schedules in case of non UET operations

Active Schedule:

- No operation can be completed earlier without changing some execution sequence
- Includes Non-Delay schedules and some of the optimal C_{max} or L_{max} schedules

SCHEDULING ALGORITHM COMPONENTS

MinDist(O_i, O_j): minimum number of cycles (possibly negative) by which O_i must precede O_j in any schedule

- O(V^3) with Floyd-Warshall, where V is the number of nodes (operations) in the dependence graph

EStart(O_i) = MinDist(Source, O_i)

- O(V * E) with Bellman-Ford, where E is the number or arcs in the dependence graph

LStart(O_i) = LStart(Sink) – MinDist(O_i, Sink) with LStart(Sink) ≥ Estart(Sink)

- O(V * E) with Bellman-Ford on the reverse dependence graph once LStart(Sink) is chosen

Slack(O_i) = LStart(O_i) – Estart(O_i)

Updating EStart(O_j) and LStart(O_j) after O_i is scheduled at date τ_i

- EStart(O_j) = max (EStart(O_j), τ_i + MinDist(O_i, O_j))
- LStart(O_j) = min (LStart(O_j), τ_i - MinDist(O_i, O_j))

AGENDA

FROM MACHINE SCHEDULING TO INSTRUCTION SCHEDULING

Dependence graphs

- Dependence latency no longer related to processing times: 0 for WAR, 1 for WAW on registers
- Scheduling before register allocation (‘prepass’) does not see spill code, register moves and most dependences related to register WAR / WAW
- Scheduling after register allocation (‘postpass’) is constrained by the register WAR / WAW dependences
- Dependence cost for memory loads instructions must statically assume whether L1 or L2 cache hits

Scheduling objectives

- Scheduling beyond basic blocks: chain or tree of basic blocks entered at the top basic block
- Scheduling inner loops: software pipelining optimizes for minimum period (initiation interval)
- In prepass instruction scheduling, delay ‘floaters’ instructions to decrease register pressure
- On microarchitectures with ‘macro-op fusion’, schedule the macro-ops in sequence

Resource constraints

- Issue slots, execution units, register file accesses

VLIW INSTRUCTION SCHEDULING AS DETERMINISTIC SCHEDULING

ST/HP ST2OO VLIW core, implements the Lx architecture [Faraboschi et al. ISCA 2000]

Source code and machine code

- Before register allocation
- Minimize makespan (C_{max})

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

```
L?__0_8:
  LDH_1      g131 = 0, G127
  MULL_2     g132 = G126, g131
  ADD_3      G129 = G129, g132
  ADD_4      G128 = G128, 1
  ADD_5      G127 = G127, 2
  CMPNE_6    b135 = G118, G128
  BRF_7      b135, L?__0_8
```

Dependence graph

- Acyclic (basic block scheduling)
- With zero latency arcs (lij ≥ –pi)
- Dummy source O_0 and sink O_8

$$\begin{tikzcd}
 & 4 \arrow[r, bend left, "1"] & 6 \arrow[drr, "0"] & & & \\
0 \arrow[ur, "0"] \arrow[dr, "0"'] & & 5 \arrow[rr, "0"] & & 7 \arrow[r, "1"] & 8 \\
 & 1 \arrow[ur, bend left, "0"] \arrow[dr, bend right, "3"'] & & & & \\
 & & 2 \arrow[r, "3"'] & 3 \arrow[uur, bend right, "0"'] & &
\end{tikzcd}$$

Resource constraints

- Unit execution time (UET)
- Cumulative resources

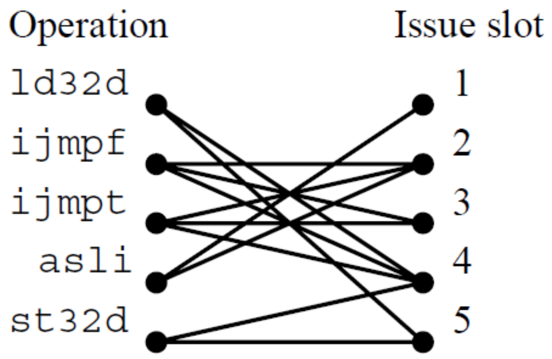
Resource	Available	O_1	O_2	O_3	O_4	O_5	O_6	O_7
Issue	4	1	1	1	1	1	1	1
Memory	1	1	0	0	0	0	0	0
Control	1	0	0	0	0	0	0	1
Align	2	0	1	0	0	0	0	1

INSTRUCTION SCHEDULING RESOURCE MODELS

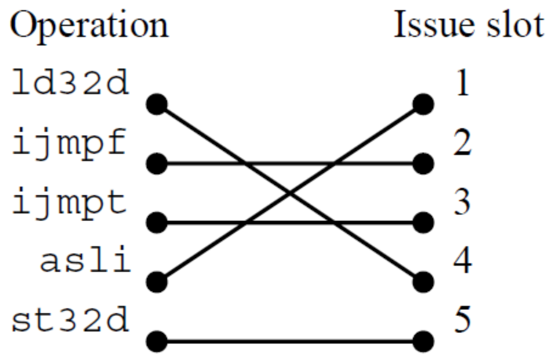
Cumulative resource model may not model accurately the target microarchitecture

Bipartite graph of operations and issue slots

on the Trimedia [Hoogerbrugge 1999]



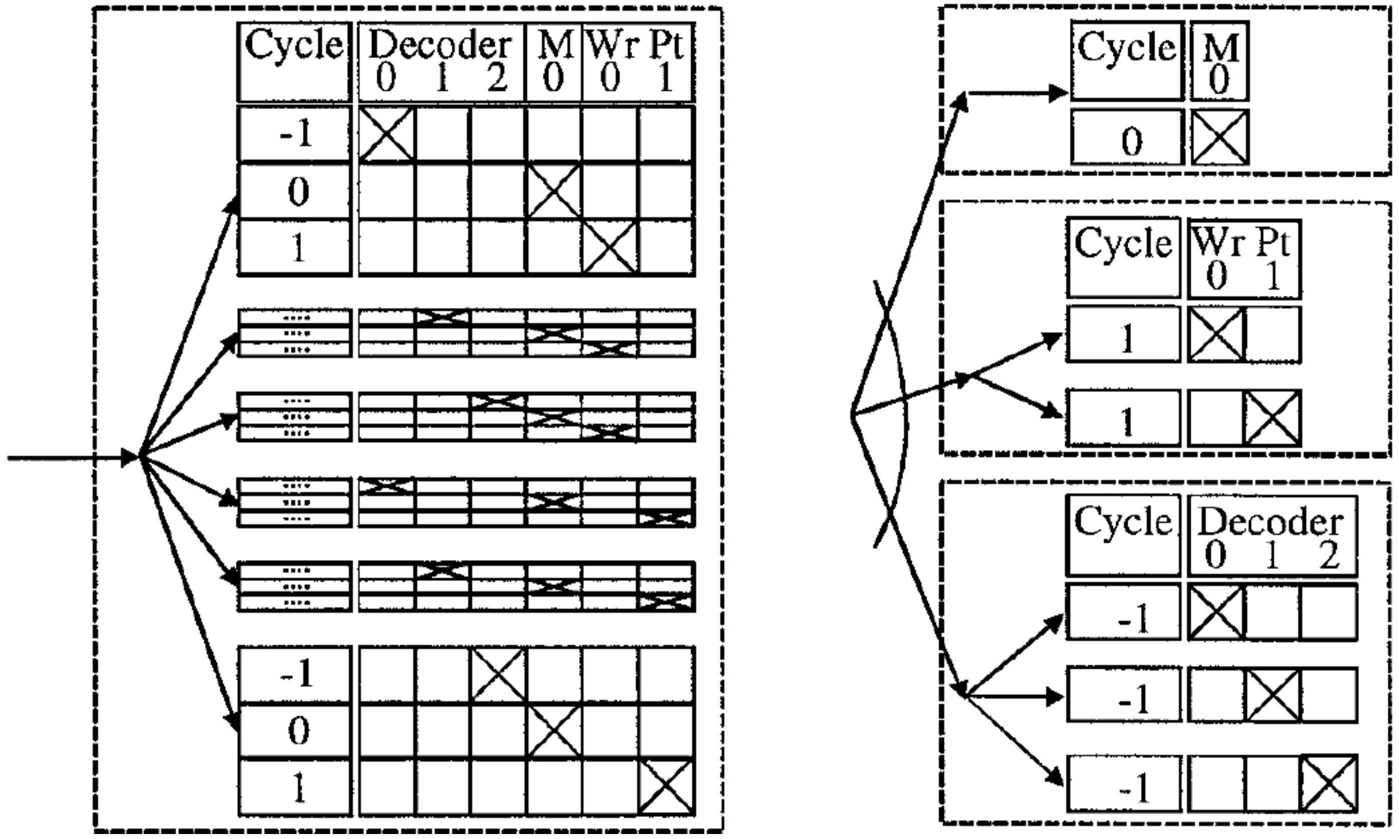
(a) A matching problem



(b) A solution

Operations with alternative reservation tables

Represented as AND-OR tree [Gyllenhaal 1998]



RESOURCE CONSTRAINTS WITH FINITE-STATE AUTOMATA

Finite-state automaton (FSA) transitions rather than reservation tables [Bala & Rubin 1995]

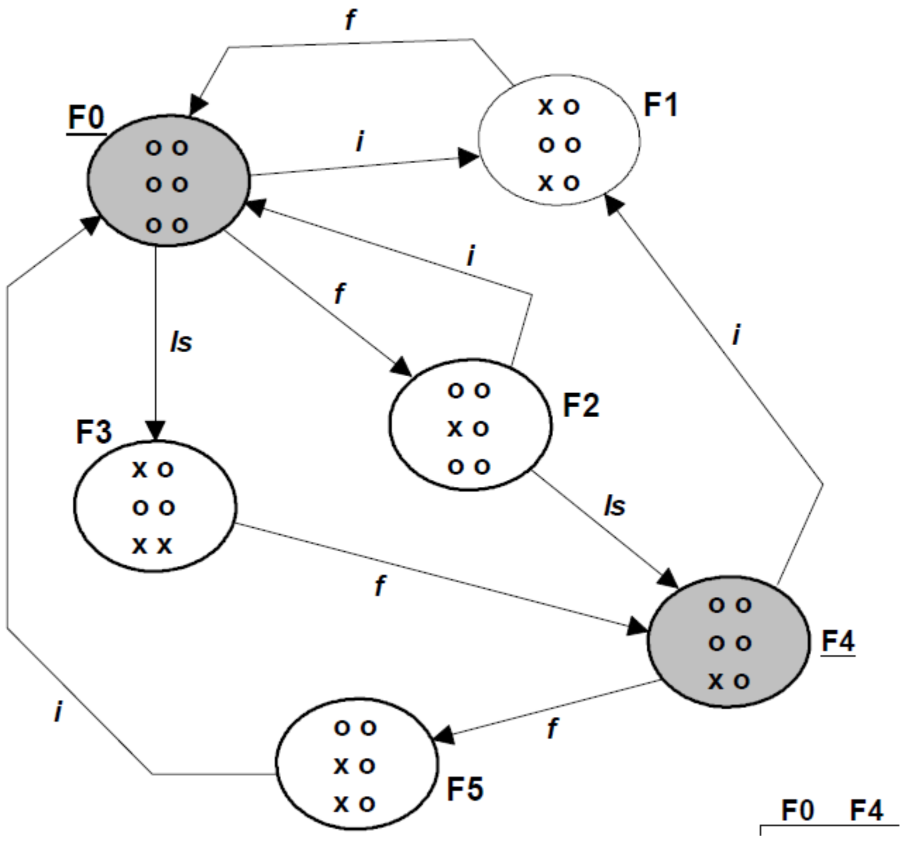
FSA approach supported in GCC and LLVM (resource regular expressions / itineraries)

Predecessors of advancing states (grey) cannot accept more instructions, meaning that the clock cycle must increase

Basic interface (GCC):

- int state_size(void);
- void state_reset(state_t);
- int state_transition(state_t, insn *);

The return value of state_transition() is the minimum delay in cycles to issue insn, if ≤ 0 insn can issue at current clock cycle



Collision Matrices

	0	1
i	x	o
f	o	o
ls	x	o

int / inop (i class)

	0	1
i	o	o
f	x	o
ls	o	o

fp / fnop (f class)

	0	1
i	x	o
f	o	o
ls	x	x

ld / st (ls class)

Directly applicable to 'cycle scheduling', and in principle to 'operation scheduling' (cumbersome in practice, see operation scheduling [Hagog & Zaks 2004] in GCC)

REGISTER ALLOCATION CONSTRAINTS

Maximum register pressure

- Number or registers live at a given program point or schedule date
- Prepass instruction scheduling tries to reduce register pressure, or a proxy such as register lifetimes
- In SSA form and without register class aliasing, can allocate with the MaxReg number of registers

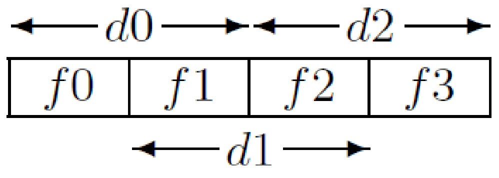
Register class alisasing (including register tuples)

- « Aliased Register Allocation for Straight-line Programs is NP-Complete » [Lee et al. 2007]
- « A Generalized Algorithm for Graph-Coloring Register Allocation » [Smith et al. 2004]

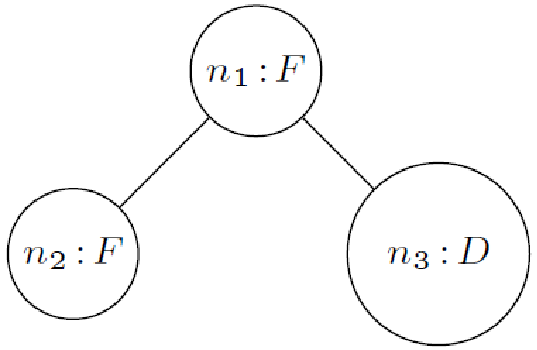
Architectural definition:

$F = \{f0, f1, f2, f3\} \qquad D = \{d0, d1, d2\}$

$$\begin{aligned}
alias(f0) &= \{f0, d0\} & alias(d0) &= \{d0, d1, f0, f1\} \\
alias(f1) &= \{f1, d0, d1\} & alias(d1) &= \{d0, d1, d2, f1, f2\} \\
alias(f2) &= \{f2, d1, d2\} & alias(d2) &= \{d1, d2, f2, f3\} \\
alias(f3) &= \{f3, d2\} & &
\end{aligned}$$



Example interference graph:



AGENDA

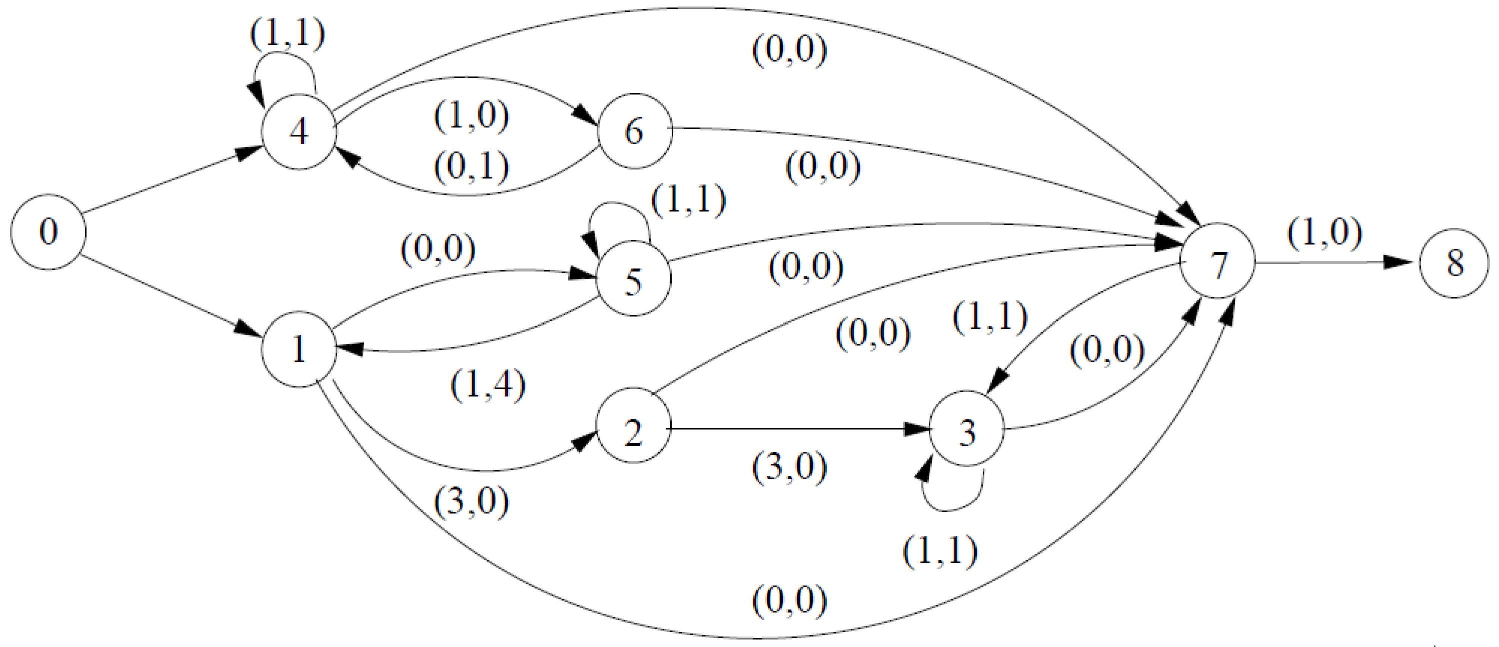
CYCLIC SCHEDULING WITH UNIFORM DEPENDENCES

Repetitive execution of an operation set $\{O_i\}_{1 \leq i \leq n}$ with the objective of maximizing throughput

- Dependence arcs are bi-valued (latency, omega) with constant values
- Omega values are the dependence iteration distances between source and destination operations
- Dominant cyclic schedules are K-periodic: they repeat a schedule of K instances of $\{O_i\}_{1 \leq i \leq n}$ every λ cycles
- Modulo scheduling techniques focus on 1-periodic cyclic schedules with pruning of the register WAR/ WAW dependences on temporary variables

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

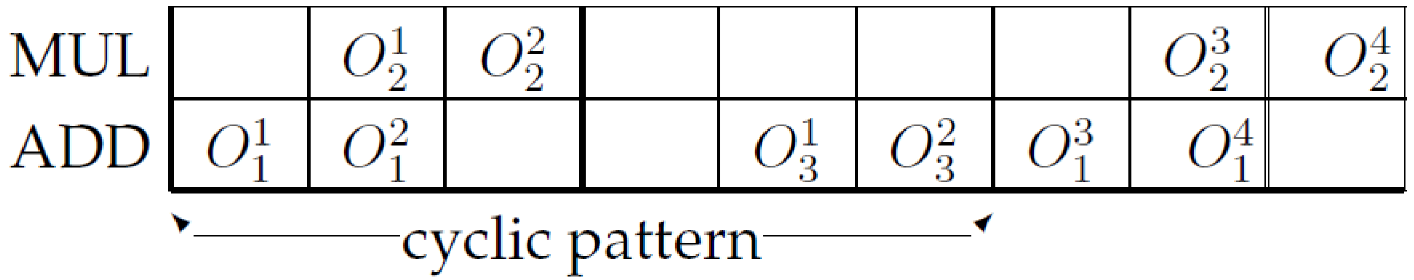
```
L?__0_8:
  LDH_1     g131 = 0, G127
  MULL_2    g132 = G126, g131
  ADD_3     G129 = G129, g132
  ADD_4     G128 = G128, 1
  ADD_5     G127 = G127, 2
  CMPNE_6   b135 = G118, G128
  BRF_7     b135, L?__0_8
```



CYCLIC SCHEDULING TECHNIQUES

Non-preemptive resource-constrained scheduling with uniform dependences

- Cyclic scheduling on parallel processors [Hanen & Munier 1994]
- Unroll and compact to find K-periodic schedules [Bodin & Charot 1990, Aiken et al. 1996] (requires a span-limiting constraint in order to converge)



MUL		O_2^1	O_2^2					O_2^3	O_2^4
ADD	O_1^1	O_1^2			O_3^1	O_3^2	O_1^3	O_1^4	

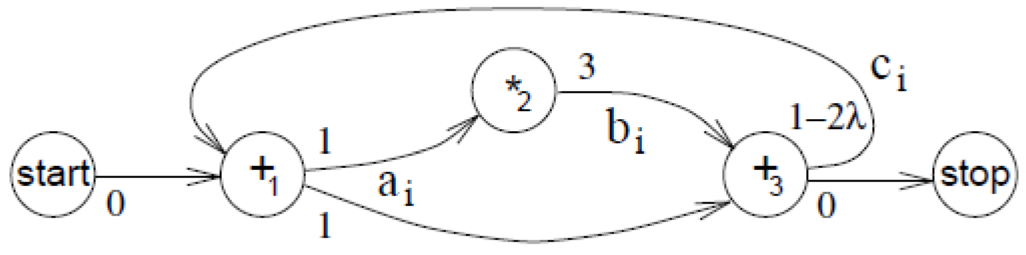
- Optimal K-periodic scheduling [Feautrier 1994, Fimmel & Müller 2000]
- Classic modulo scheduling [Rau 1981, Touzeau 1984, Lam 1988]
- Decomposed modulo scheduling [Gasperoni & Schwiegelshohn 1991, Wang & Eisenbeis 1994], Insertion Scheduling [Dinechin 1995]
- Optimal modulo scheduling using time-indexed integer linear programming formulations [Govindarajan et al. 1994, Eichenberger et al. 1995, Dinechin 2003]
- Converging to periodic schedules for cyclic scheduling problems with resources and deadlines [Dinechin & Munier-Kordon 2014]

CLASSIC MODULO SCHEDULING FRAMEWORK

Compute lower bounds on the period λ (called II): ResMII from resources, RecMII from dependence circuits

Thanks 1-periodic cyclic scheduling with period λ, the bi-valued dependence arcs become single-valued:

- Arc $O_3 \rightarrow O_2$ with values (latency=1, omega=2) now has a possibly negative value (length = 1 - 2λ)



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

Schedule $\{O_i\}_{1 \leq i \leq n}$ under modulo resource constraints: any resource busy at cycle t is also busy at cycle t+kλ

- Illustrated here with UET operations $\{O_1, O_2, O_3\}$, two resources MUL, ADD, and λ = 3 clock cycles

MUL		O_2			$\bullet_2$			$\bullet_2$	
ADD	O_1			$\bullet_1$	O_3		$\bullet_1$	$\bullet_3$	

$\longleftarrow \lambda \longrightarrow$

Construct the software pipeline code including prologue, (unrolled) kernel, epilogues

Fixups for the pruned register WAR/WAW dependences: regmoves or modulo variable expansion (MVE)

MODULO SCHEDULING IN PRODUCTION COMPILERS (1)

« A Fortran Compiler for the FPS-164 Scientific Computer » [Touzeau 1984]

- Cycle scheduling with earliest deadline first priority and simultanesous register allocation
- When the scheduler cannot register allocate a candidate, it inserts spill code and resumes scheduling
- Optimizes uniform loop-carried memory dependences into register moves (e.g. x[i] = a*x[i-2] + y[i])

« Compiling for the Cydra 5» [Dehnert & Towle 1993]

- Operation scheduling with backtracking: de-schedules resource or dependence conflicting operations
- Priority is lowest Slack first, increased if operation belongs to an inner recurrence circuit
- Register allocation after scheduling, reschedule after spill code insertion

« Lifetime-Sensitive Modulo Scheduling » [Huff 1993]

- Operation scheduling with backtracking whose Slack priority is corrected by the criticality of resources
- Scheduling by increasing from EStart or decreading from LStart depending on lifetime stretching effects
- Compute RecMII in O(V E log V) time by finding a circuit with the minimum cost-to-time ratio, where a dependence arc is viewed as having a “cost” of -latency, and a “time” of omega

MODULO SCHEDULING IN PRODUCTION COMPILERS (2)

« Software Pipelining Showdown: Optimal vs. Heuristic Methods … » [Ruttenberg et al. 1996]

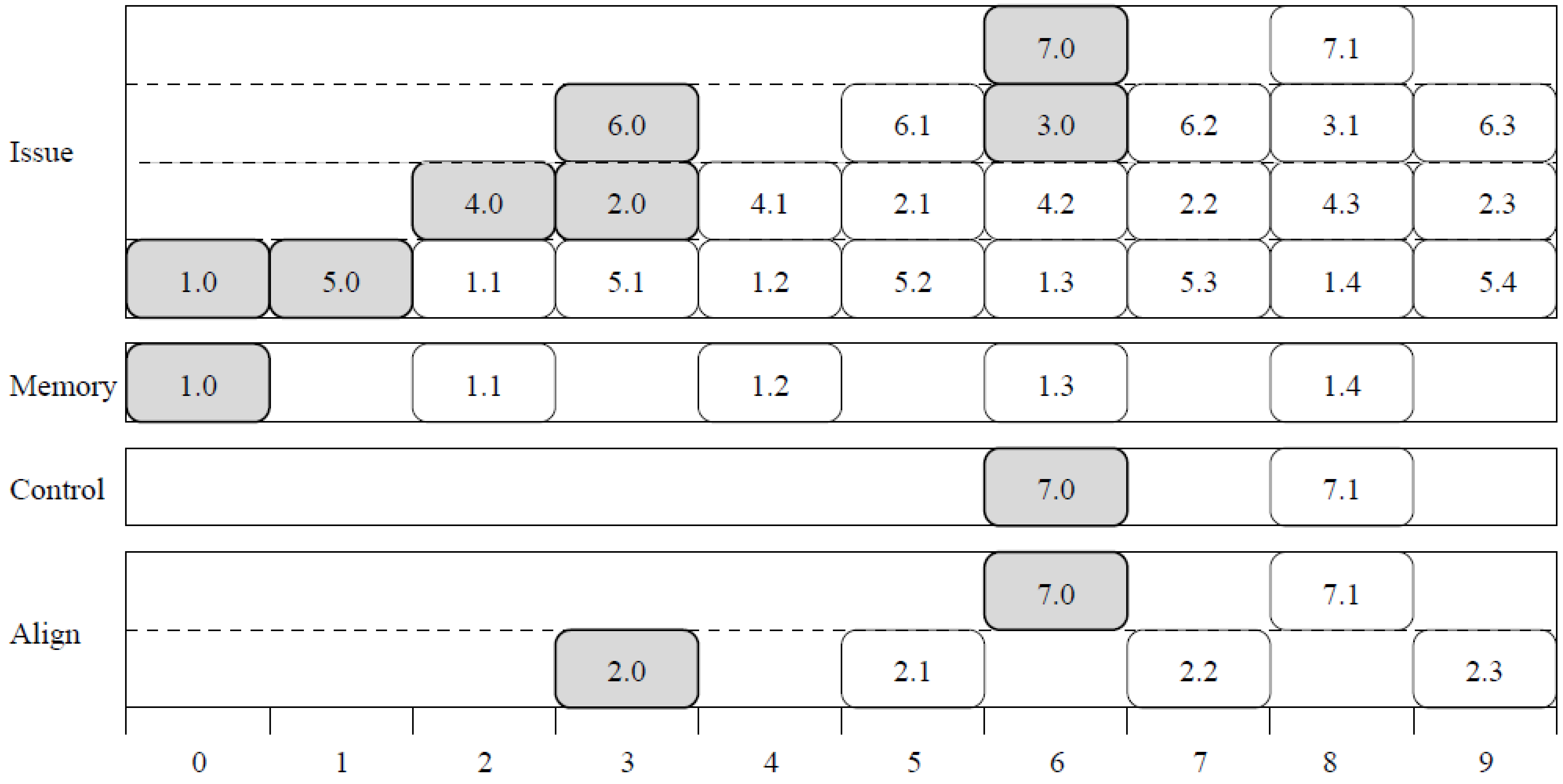
- Operation scheduling with linear backtracking, driven by the order in a priority list
- When scheduling O_i fails, backtrack to 'catch point' O_j in the prioriy list to and reschedule O_j at other cycle
- Four different priority list are tried before incrementing the II, as none of them dominates the others
- Chaitin-Briggs register coloring on the constructed software pipeline, if fail insert spill code and try again
- Code quality almost identical to optimal modulo scheduling by solving integer programming formulation

« Swing Modulo Scheduling: A Lifetime-Sensitive Approach » [Llosa et al. 1996]

- Introduces Swing Modulo Scheduling (SMS), an operation scheduling algorithm without backtracking
- Two-level algorithm to build operation ordering: order the recurrence sets, then order within each set
- Aim to minimize the loop variables lifetime by scheduling operations either from EStart or from LStart
- SMS is implemented in GCC and LLVM, but restricted to single basic block counted loops
- [Llosa et al. 2001] *“It is a heuristic technique that has a low computational cost while producing schedules very close to those generated by optimal approaches based on exhaustive search”* [?]

MODULO SCHEDULING IN PRODUCTION COMPILERS (3)

ST200 production compiler on sample loop finds modulo schedule at $\lambda = 2$ clock cycles



AGENDA

GCC MODULO SCHEDULING FOR THE KALRAY VLIW CORE

Kalral VLIW core features:

- MADD (int.), FMA (f.p.), CMOVE instructions have 4 operands encoded with 3 registers specifiers
- Load opcodes have a modifier to enable bypassing L1 cache, exposing the L2 cache latency
- SIMD instructions operate on 64-bit register pairs (128 bits) or quadruples (256 bits)

GCC modulo scheduling [Hagog & Zaks 2004]:

- Only applies to single basic block counted loops, that must be mapped to hardware loops (‘do loop’)
- Modulo scheduling in prepass, disables re-scheduling of software pipeline code in postpass
- Implements Swing Modulo Scheduling (SMS) without modulo variable expansion (MVE)
- Generate register moves to split live ranges > II and try to insert them in the modulo schedule
- GCC does not preserve accurate memory dependences, e.g. vectorization discards #pragma gcc ivdep

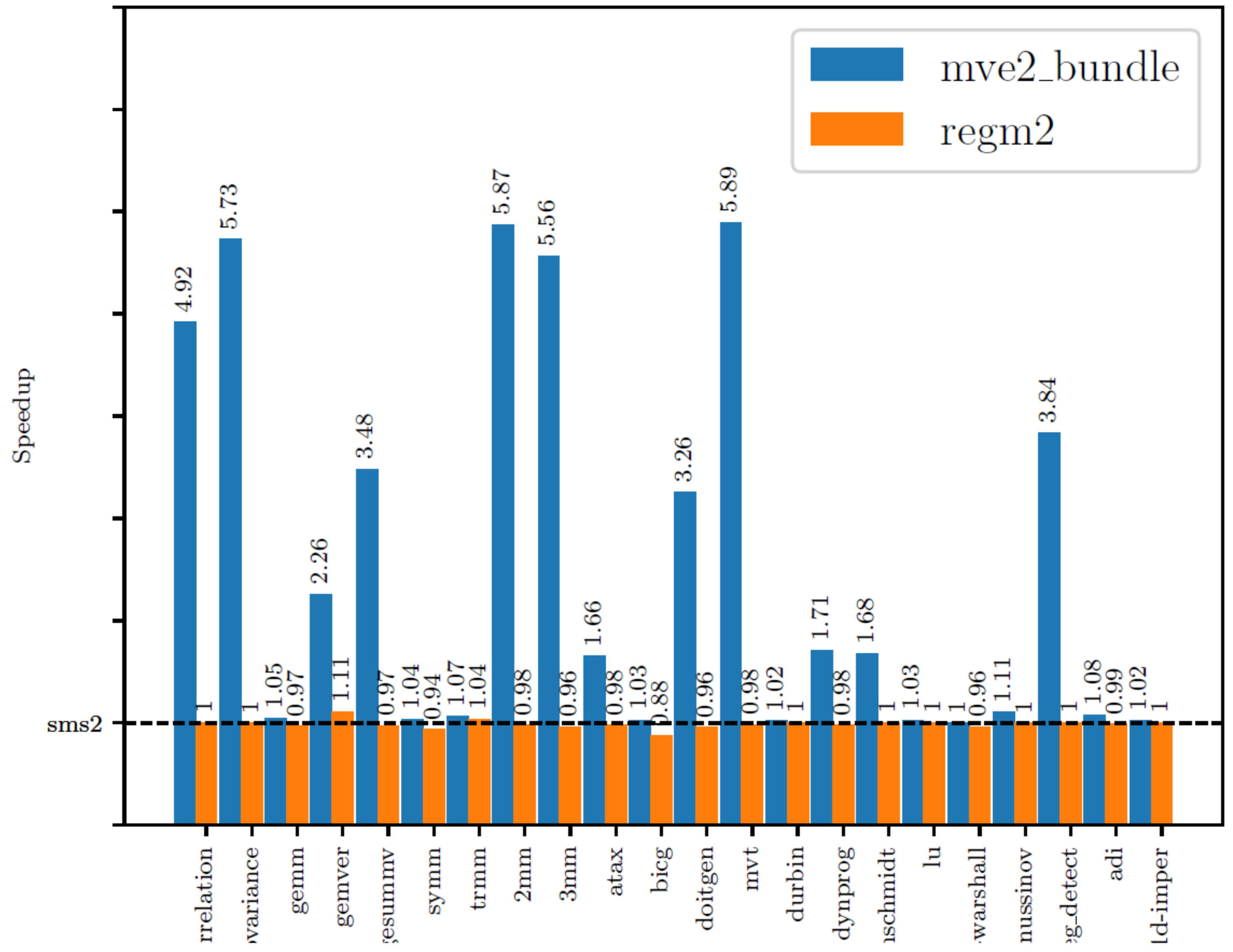
Improvements to GCC SMS [GNU Tools Cauldron 2024]:

- Implement MVE and presolve the register naming constraints by inserting regmoves before scheduling

MODULO VARIABLE EXPANSION VS REGISTER MOVES

Explicit register moves (regmoves) are not the same as classic MVE [Lam 1988]:

- With register moves, there is no need for kernel unrolling and multiple epilogs
- Moving the destination register of a long latency operation stalls in-order microarchitectures
- Illustrated by executing PolyBench on the Kalray VLIW core with array loads bypassing the L1 cache
- Code compiled with GCC and its improved SMS, with MVE (mve2) instread of regmoves (regm2)



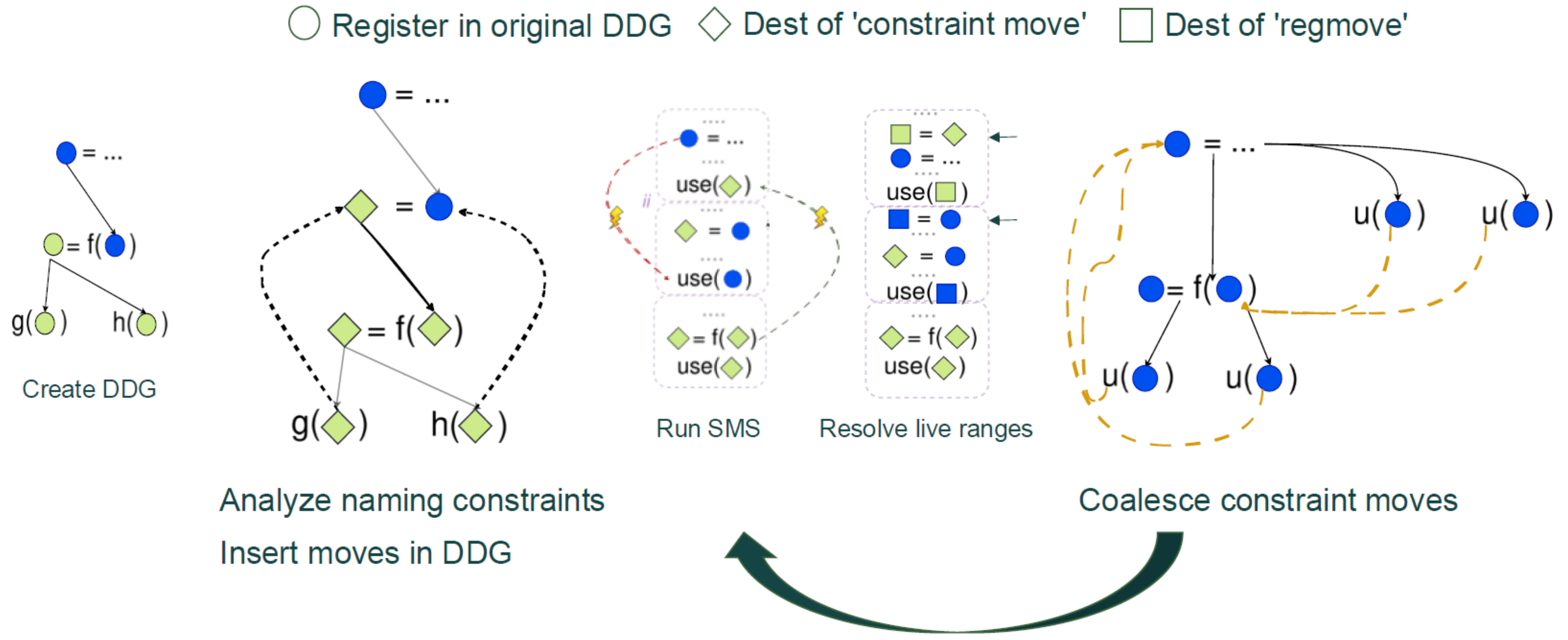
PRESOLVING THE REGISTER NAMING CONSTRAINTS

Register naming constraints come from ISA [and from calling conventions]

Solved by GCC local register allocation (LRA) by splitting then coalescing live ranges

Try to prevent LRA from inserting register copies after SMS, by iterating:

- Run the SMS scheduler on the DDG extended with constraint moves
- Coalesce the constraint moves if non-interfering and the resulting lifetime ≤ II



ANOTHER APPROACH TO MODULO SCHEDULING

Motivated by issues with GCC [LLVM] modulo scheduling approaches

- Require single basic block loops, however if-conversion occurs after modulo scheduling
- Require that the loop be counted, thus excluding all while loops
- Require that the target ISA provides counted hardware loops
- Provides no visibility on register moves or spill code that may be later inserted by register allocation
- The [acyclic] postpass scheduler is unaware of the loop-carried dependence and resource dangles

Proposed approach: ‘convergent modulo scheduling’ (CMS) on register-allocated machine code

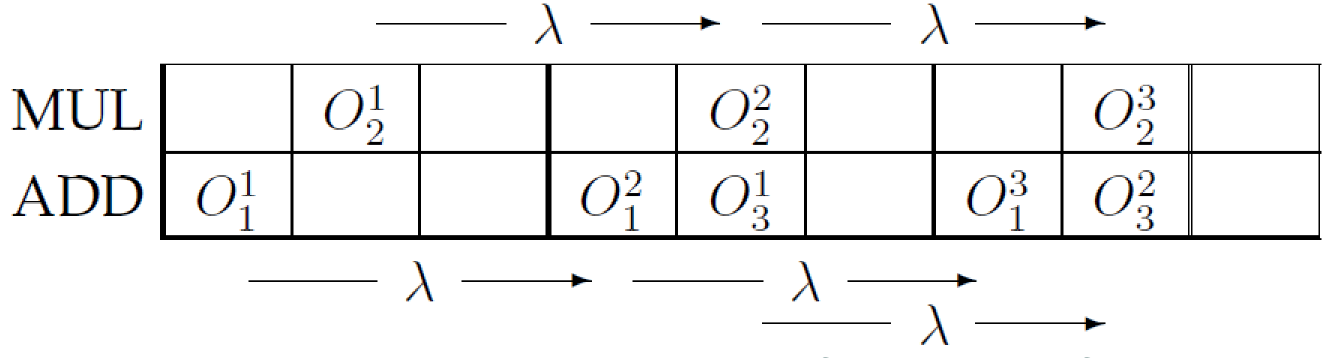
- Software pipelining of superblock counted and while loops after register allocation and if-conversion
- Deallocation of registers at the granularity of live ranges then reallocation and possibly MVE
- Modulo scheduling while allowing control speculation of side-effect free operations, including loads
- Better non-backtracking modulo scheduling engine than SMS, by not managing register lifetimes
- Currently prototyped in GCC

CONVERGING TO 1-PERIODIC SCHEDULES

Applies to cyclic scheduling problems with resources, uniform precedences, release dates & deadlines

[Dinechin & Munier-Kordon 2014] heuristic or optimal acyclic scheduling algorithm can be used to build 1-periodic cyclic schedules → direct application to modulo scheduling

- Assume a value of the period λ, unwind the cyclic scheduling problem $\{O_i\}_{1\leq i\leq n}$ u times
- Regularize the unwinded scheduling problem $\{O_i^j\}_{1\leq i\leq n}^{1\leq j\leq u}$ by adding a dependence arc of latency λ between any two successive instances O_i^k, O_i^{k+1} of O_i for all $1\leq i\leq n$
- Try acyclic scheduling of the regularized unwinded scheduling problem, if it succeeds any two successive instances of an operation O_i^k, O_i^{k+1} are scheduled no less than λ cycles apart
- Eventually, this schedule becomes λ-stationary for Δ iterations, and this yields a 1-periodic schedule at period λ

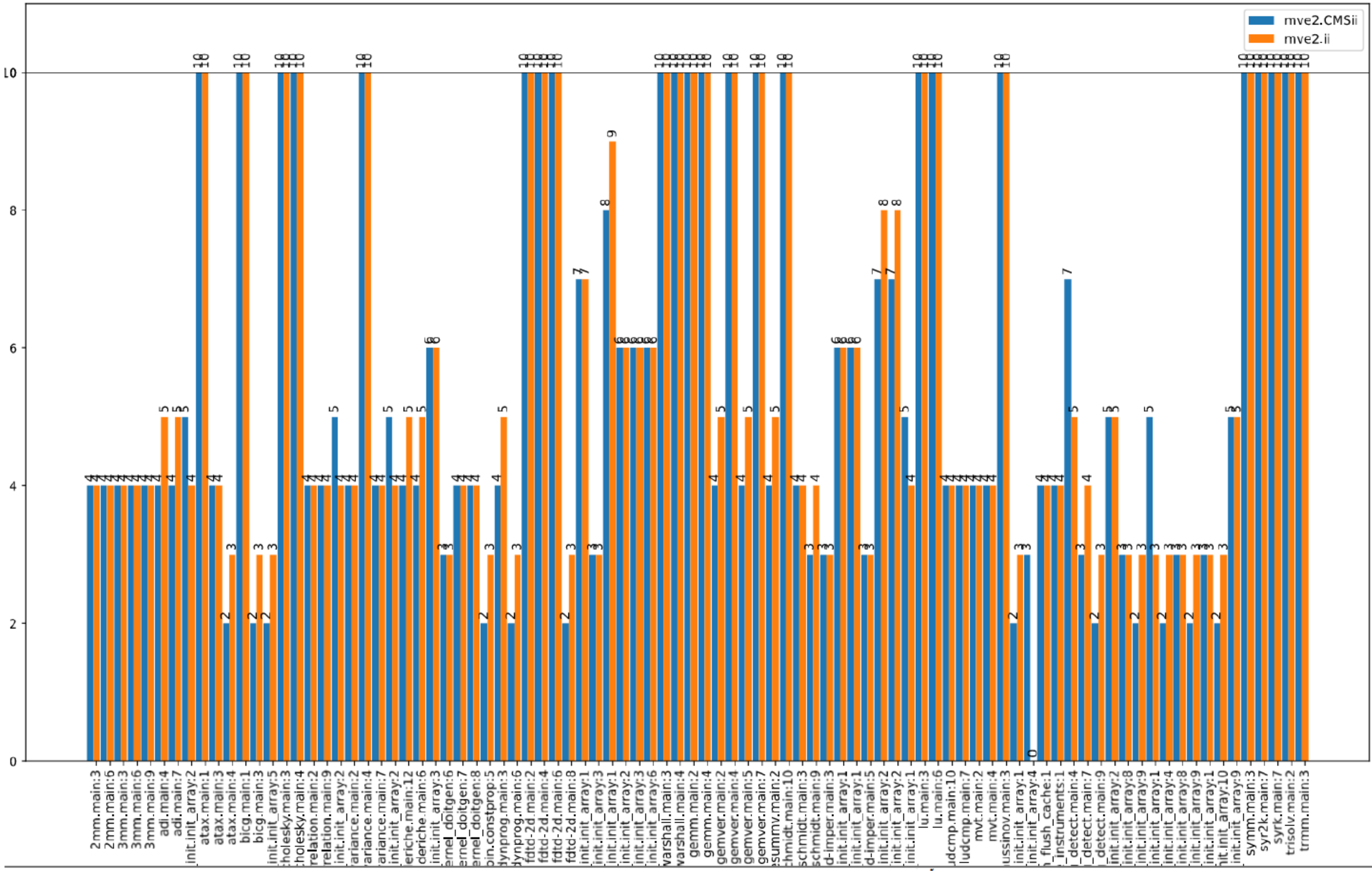


- [Dinechin & Munier-Kordon 2014] provide two lower bounds for Δ to satisfy respectively the uniform precedence relations and the resource constraints of the modulo schedule

COMPARING THE CMS TO SMS IMPLEMENTATIONS IN GCC

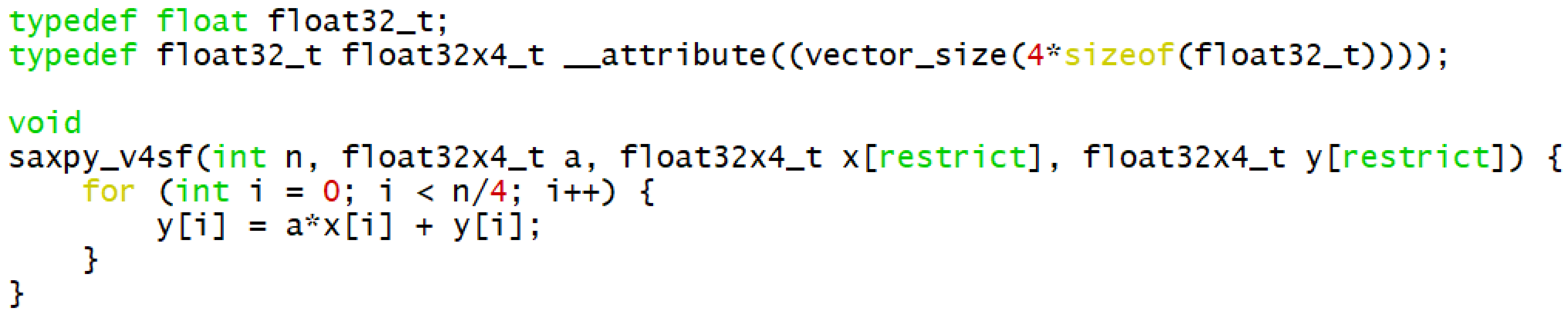
CMS implemented in GCC for the Kalray VLIW core:

- Use a cycle scheduler, with operation Height as priority
- Cycle scheduling enables a straigtforward use of the resource FSA (called DFA)
- Implementation emulates the unbounded unwiding of the loop body with only two instances of its operations
- Even with the simple Height priority, CMS finds lower II than SMS in GCC
- [Codina et al. 2002] compare SMS to Iterative Modulo Scheduling, Slack Scheduling, Integrated Register-Sensitive Iterative Software pipelining (IRIS)



CMS ON REGISTER ALLOCATED MACHINE CODE IN GCC (1)

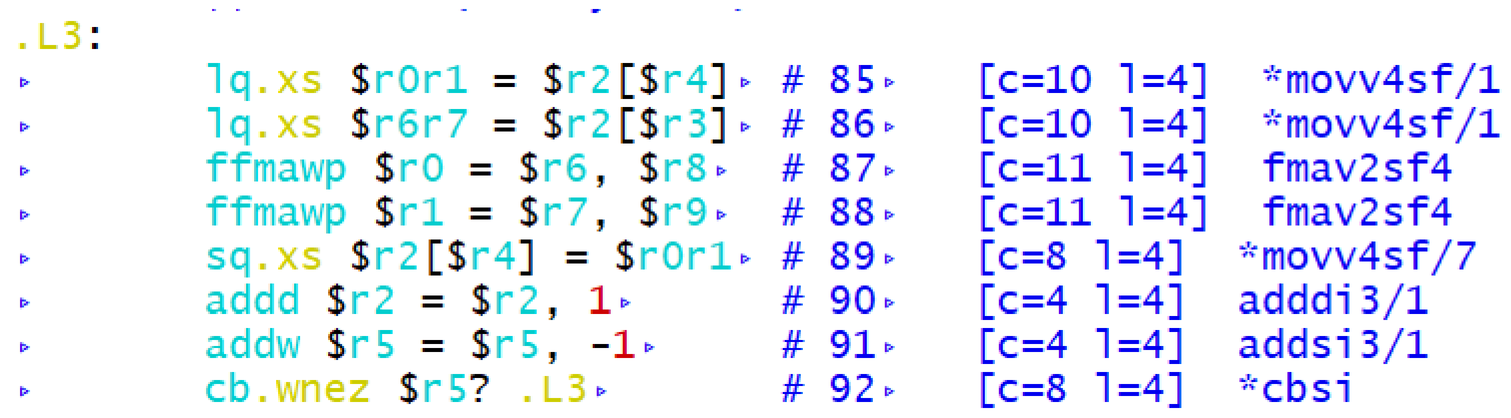
Manual vectorization of SAXPY with GCC/LLVM vector type extensions



```
typedef float float32_t;
typedef float32_t float32x4_t __attribute((vector_size(4*sizeof(float32_t))));

void
saxpy_v4sf(int n, float32x4_t a, float32x4_t x[restrict], float32x4_t y[restrict]) {
    for (int i = 0; i < n/4; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

Inner loop code after register allocation



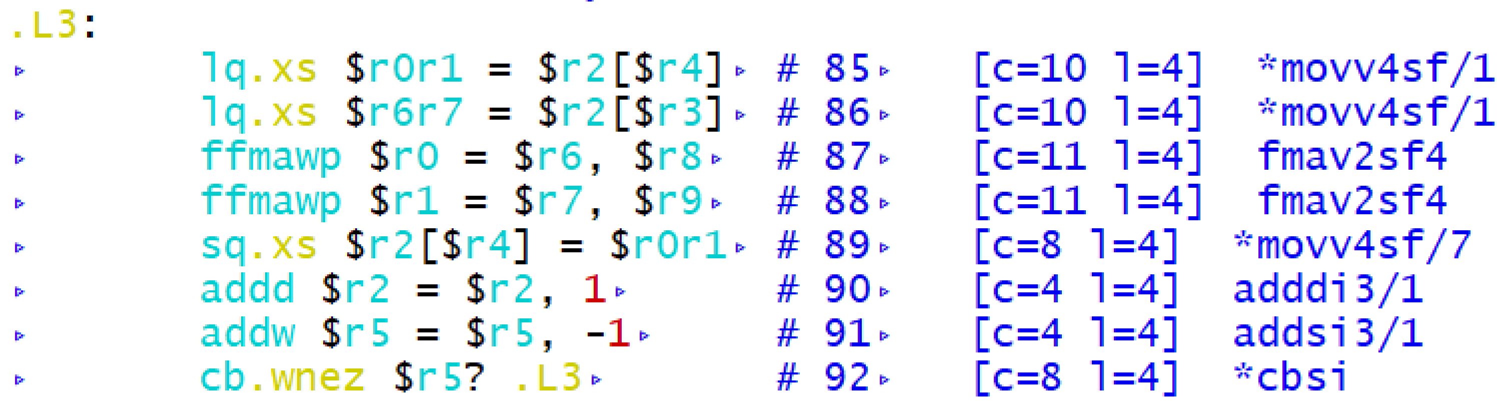
```
.L3:
▸       lq.xs $r0r1 = $r2[$r4]▸ # 85▸   [c=10 l=4]  *movv4sf/1
▸       lq.xs $r6r7 = $r2[$r3]▸ # 86▸   [c=10 l=4]  *movv4sf/1
▸       ffmawp $r0 = $r6, $r8▸  # 87▸   [c=11 l=4]  fmav2sf4
▸       ffmawp $r1 = $r7, $r9▸  # 88▸   [c=11 l=4]  fmav2sf4
▸       sq.xs $r2[$r4] = $r0r1▸ # 89▸   [c=8 l=4]  *movv4sf/7
▸       addd $r2 = $r2, 1▸      # 90▸   [c=4 l=4]  adddi3/1
▸       addw $r5 = $r5, -1▸     # 91▸   [c=4 l=4]  addsi3/1
▸       cb.wnez $r5? .L3▸       # 92▸   [c=8 l=4]  *cbsi
```

} 128-bit loads to register pairs

} FMA lowered to 64-bit operations

CMS ON REGISTER ALLOCATED MACHINE CODE IN GCC (2)

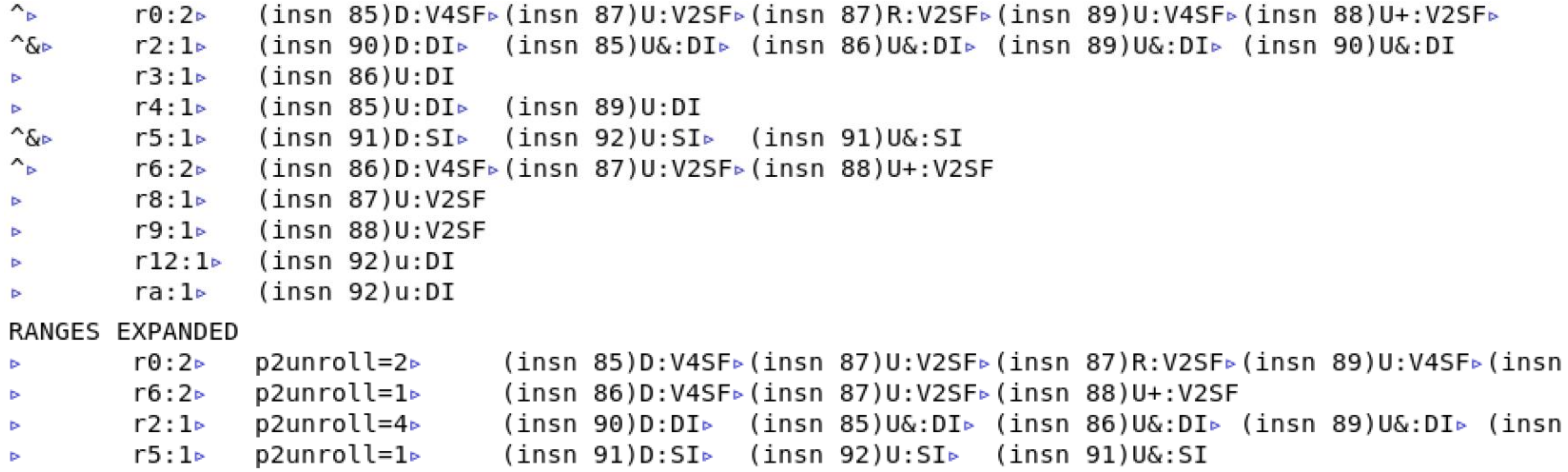
Inner loop code after register allocation



```
.L3:
        lq.xs $r0r1 = $r2[$r4]   # 85    [c=10 l=4]  *movv4sf/1
        lq.xs $r6r7 = $r2[$r3]   # 86    [c=10 l=4]  *movv4sf/1
        ffmawp $r0 = $r6, $r8    # 87    [c=11 l=4]  fmav2sf4
        ffmawp $r1 = $r7, $r9    # 88    [c=11 l=4]  fmav2sf4
        sq.xs $r2[$r4] = $r0r1   # 89    [c=8 l=4]  *movv4sf/7
        addd $r2 = $r2, 1        # 90    [c=4 l=4]  adddi3/1
        addw $r5 = $r5, -1       # 91    [c=4 l=4]  addsi3/1
        cb.wnez $r5? .L3         # 92    [c=8 l=4]  *cbsi
```

} FFMA register naming constraints

Register live ranges



```
^       r0:2    (insn 85)D:V4SF (insn 87)U:V2SF (insn 87)R:V2SF (insn 89)U:V4SF (insn 88)U+:V2SF
^&      r2:1    (insn 90)D:DI   (insn 85)U&:DI  (insn 86)U&:DI  (insn 89)U&:DI  (insn 90)U&:DI
        r3:1    (insn 86)U:DI
        r4:1    (insn 85)U:DI   (insn 89)U:DI
^&      r5:1    (insn 91)D:SI   (insn 92)U:SI   (insn 91)U&:SI
^       r6:2    (insn 86)D:V4SF (insn 87)U:V2SF (insn 88)U+:V2SF
        r8:1    (insn 87)U:V2SF
        r9:1    (insn 88)U:V2SF
        r12:1   (insn 92)u:DI
        ra:1    (insn 92)u:DI

RANGES EXPANDED
        r0:2    p2unroll=2      (insn 85)D:V4SF (insn 87)U:V2SF (insn 87)R:V2SF (insn 89)U:V4SF (insn
        r6:2    p2unroll=1      (insn 86)D:V4SF (insn 87)U:V2SF (insn 88)U+:V2SF
        r2:1    p2unroll=4      (insn 90)D:DI   (insn 85)U&:DI  (insn 86)U&:DI  (insn 89)U&:DI  (insn
        r5:1    p2unroll=1      (insn 91)D:SI   (insn 92)U:SI   (insn 91)U&:SI
```

PIECES OF TARGET-SPECIFIC DESCRIPTION

Standard pattern name for FMA operating on 2x FP32 vectors (V2SF)

```
;; ../../gcc/config/kvx/vector.md: 6605
(define_insn ("fmav2sf4")
     [
        (set (match_operand:V2SF 0 ("register_operand") ("=r"))
            (fma:V2SF (match_operand:V2SF 1 ("register_operand") ("r"))
                (match_operand:V2SF 2 ("register_operand") ("r"))
                (match_operand:V2SF 3 ("register_operand") ("0"))))
     ] ("") ("ffmawp %0 = %1, %2")
     [
        (set (attr ("type"))
            (if_then_else (match_operand 1 ("float16_inner_mode") (""))
                (const_string ("madd_fp3"))
                (const_string ("madd_fp4"))))
    ])
```

← FMA operand 3 must use same register as operand 0

Further machine code lowering done after register allocation

```
;; ../../gcc/config/kvx/vector.md: 7165
(define_split [
        (set (match_operand:V4SF 0 ("register_operand") (""))
            (fma:V4SF (match_operand:V4SF 1 ("register_operand") (""))
                (match_operand:V4SF 2 ("register_operand") (""))
                (match_operand:V4SF 3 ("register_operand") (""))))
     ] ("!HAVE_KVX_FMA_V4SF_V4SF_V4SF && reload_completed")
     [
        (set (subreg:V2SF (match_dup 0) 0)
            (fma:V2SF (subreg:V2SF (match_dup 1) 0)
                (subreg:V2SF (match_dup 2) 0)
                (subreg:V2SF (match_dup 3) 0)))
        (set (subreg:V2SF (match_dup 0) 8)
            (fma:V2SF (subreg:V2SF (match_dup 1) 8)
                (subreg:V2SF (match_dup 2) 8)
                (subreg:V2SF (match_dup 3) 8)))
    ] (""))
```

← FMA on vectors of 4x FP32

← FMA on vectors of 2x FP32 at offset 0 bytes in 128-bit register

← FMA on vectors of 2x FP32 at offset 8 bytes in 128-bit register

AGENDA

DIRECTIONS FOR KERNEL CODE SOFTWARE PIPELINING

Issues with kernel code generation through C/C++ compiler

- Restricted applicability of sofware pipelining
- Conservative memory dependence information
- Register allocator challenged by register class aliasing

MLIR lowering to register allocated machine code

- Vectorization and instruction selection
- Simple hierarchical register allocation
- Expansion of operations on register tuples
- If-conversion in case of conditional execution

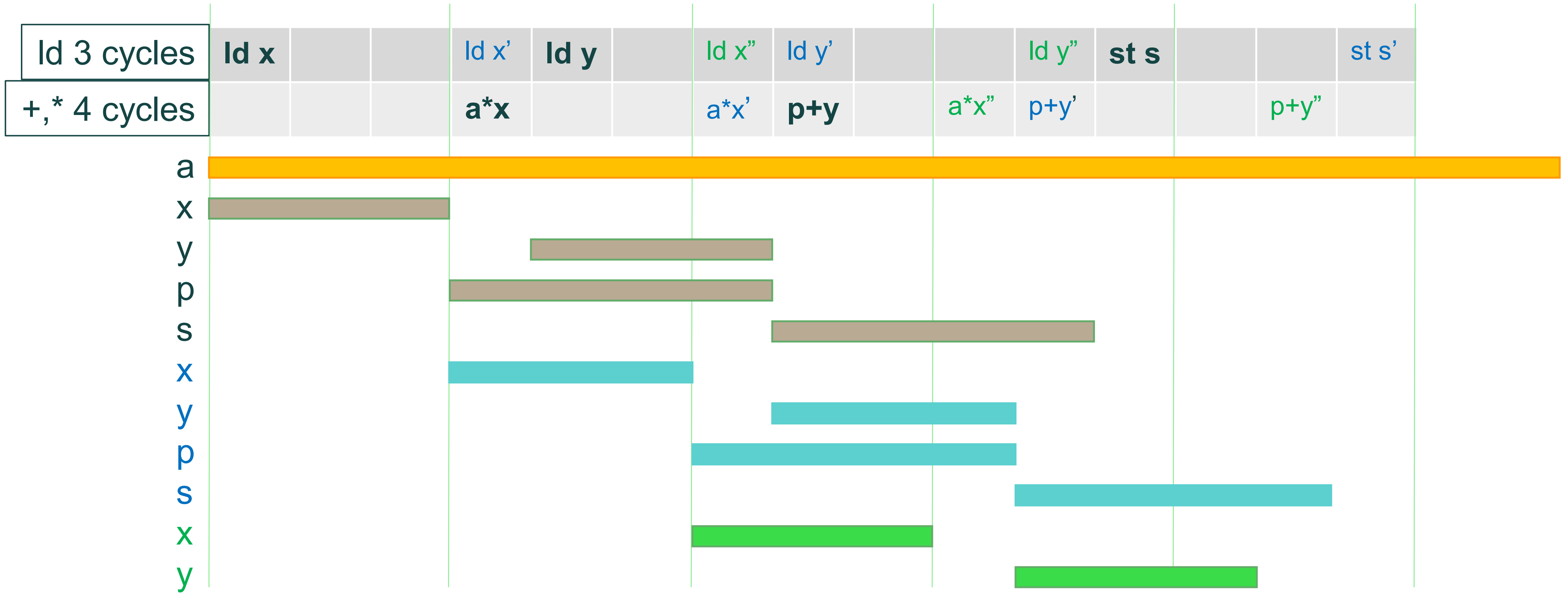
CMS on register allocated machine code

- Register lifetimes to ensure naming constraints (ISA, sub-registers and conditional execution)
- Using cycle scheduling enables direct reuse of DFA approach to scheduled resource management

REGISTER PRESSURE WHILE MODULO SCHEDULING

Example of SAXPY loop scheduled at $\lambda = 3$ clock cycles

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```



THANK YOU

KALRAY

THE POWER OF MORE

www.kalrayinc.com