GCC for Embedded VLIW Processors: Why Not?

Benoît Dupont de Dinechin
Research & Development Responsible
STS Compilation Expertise Center
STMicroelectronics Grenoble (France)
benoit.dupont-de-dinechin@st.com

Presentation Outline

- VLIW Code Generation Requirements
- Programmer-Supplied Information
- Machine-Level SSA Form
- Predicated Code Support
- Near-Optimal Software Pipelining
- Summary and Conclusions

VLIW Code Generation Requirements

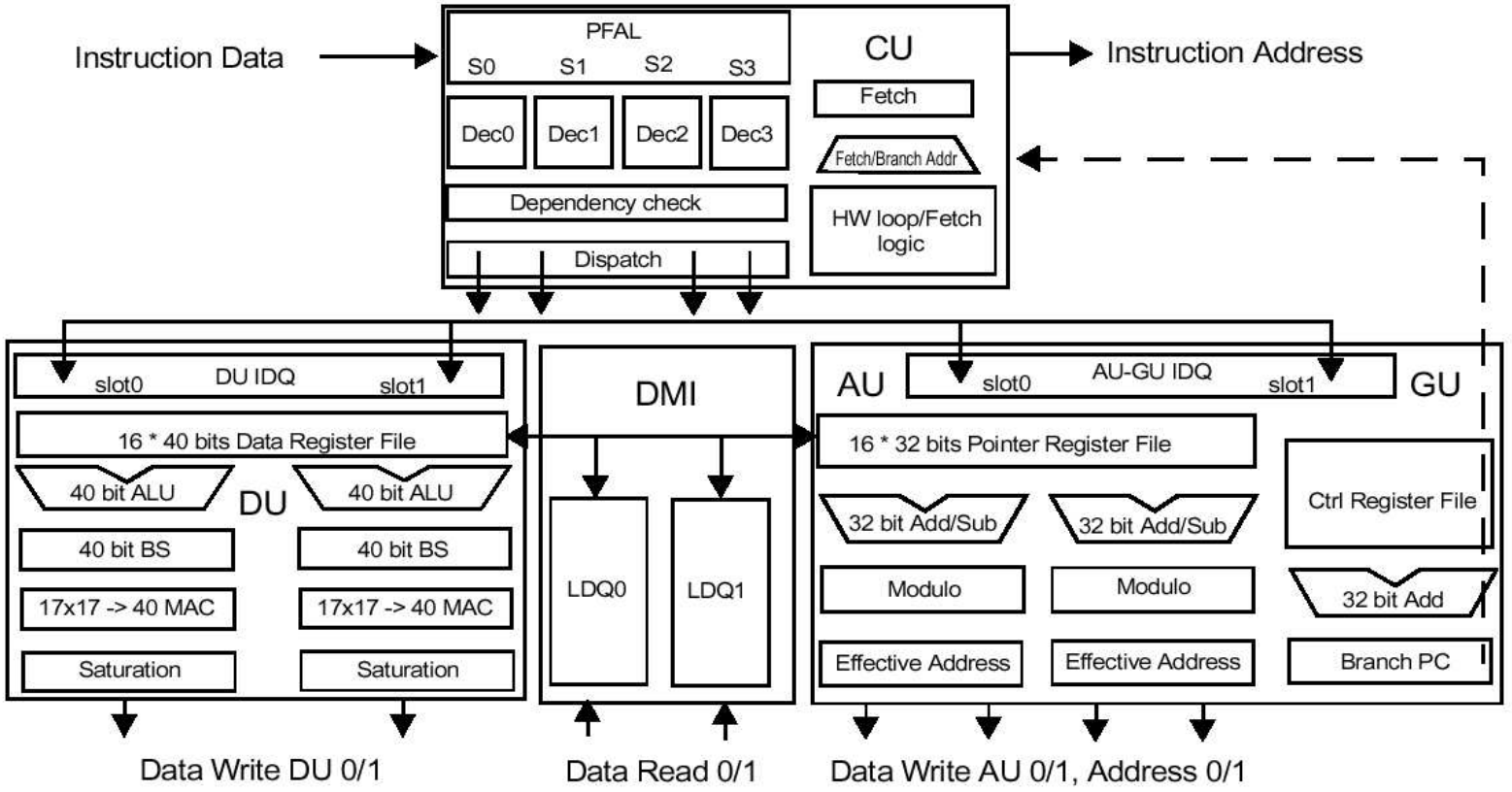
Processors and Compilation Technologies at STMicroelectronics

Processor	Type	Compilers
ARM 926 (v5)	Embedded RISC	GCC
ST40 / SH4	Embedded RISC	GCC
MMDSP V80	16/24bit DSP	ACE CoSy + EliXir
ST100 Family	VLIW DSP	GHS + LAO v1, PGI
ST200 Family	VLIW Media	GCCFE + Open64 + LAO v2
STxP70 Family	Hardware Controller	GCCFE + Open64
ARM v6–v7	RISC Media	GCCFE + Open64 + LAO v2

EliXir STMicroelectronics proprietary code generator [LCTES'04]

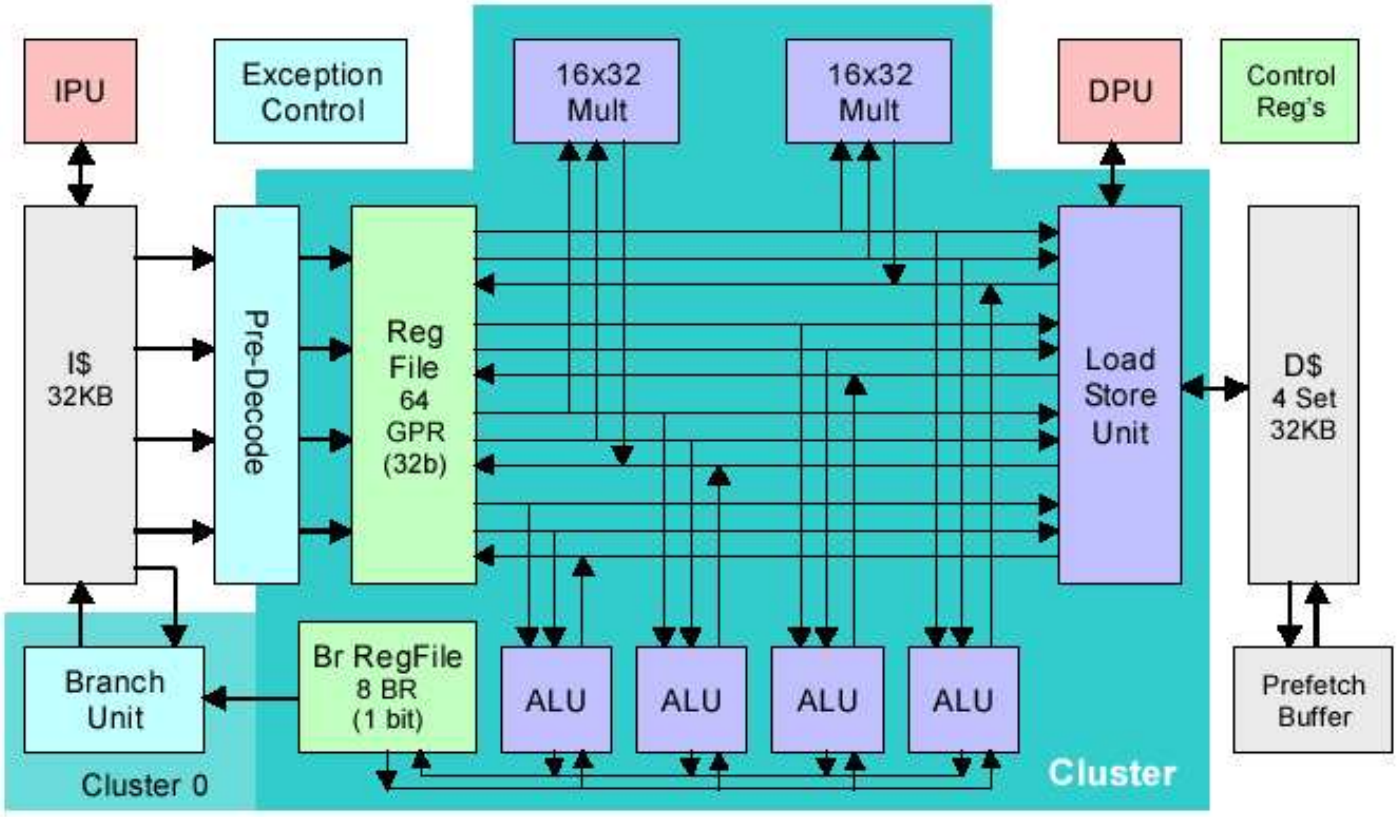
LAO was "Linear Assembly Optimizer" for ST100 [CASES 2000], is now an open source superblock scheduler / software pipeliner

The ST120 VLIW-DSP Processor Core



- 4-issue clustered VLIW (2 address operations, 2 data operations)
- fully predicated, 1-target ”Normal PDI” (IMPACT terminology)

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)

The STxP70 Hardware Controller

- ISA loosely based on the ST100 (DSP arithmetic, hardware loops)
- controller for SIMD (512 bit) application-defined co-processors
- co-processor registers seen as 256 bit pairs or 64 bit quadruples
- fully predicated, 2-target "Unconditional PDI":

CMPLT Gy, Rn, Rp

$$\begin{cases} G_y & \leftarrow & (R_n < R_p) \\ G_{y+4} & \leftarrow & \neg(R_n < R_p) \end{cases}$$

Gx& CMPLT Gy, Rn, Rp

$$\begin{cases} G_y & \leftarrow & G_x \wedge (R_n < R_p) \\ G_{y+4} & \leftarrow & G_x \wedge \neg(R_n < R_p) \end{cases}$$

- the core predication model applies to co-processor operations

Classic Code Generation [Aho 1986]

- instruction selection and calling conventions lowering
- control-flow (dominators, loop nesting) analyzes
- data-flow (liveness, reaching definitions) analyzes
- register allocation and stack frame building
- peephole and branch optimizations

Modern Code Generation [Muchnick 1997]

- loop unrolling and basic block replication
- extended block optimizations with instruction re-selection
- instruction scheduling and software pipelining
- basic block alignment and procedure placement

Code Generation for Embedded VLIW Processors

- matching code idioms such as DSP arithmetic by target processor instructions
- if-conversion based on conditional MOVEs, SELECTs, or fully predicated instructions
- taking advantage of specialized addressing modes and of hardware looping capabilities
- rewriting loops in order to exploit SIMD instructions
- management of register tuples and of register aliasing
- complex software pipelining in case of clustered architectures
- tricks to reduce code size or enhance code compressibility, including instruction mode switching

VLIW Code Generation Experience at STMicroelectronics

- programmer-supplied information is critical for high performances and reduced code sizes
- SSA form is beneficial in a code generator: range propagation, hardware looping, auto-modified addressing
- predicated code support must be considered early in code generator design
- with register tuples and register aliasing, register allocation is still challenging
- solving integer linear programming formulations of software pipelining is practical

The ST200 production compiler significantly outperforms the HP Lx compiler, a descendant of the Multiflow Trace Scheduling compiler.

Programmer-Supplied Information

Intrinsic Functions

- better optimized than ASM statements

```
#if __c64x
# define DIG_REV(i, m, j) ((j) = (_shfl(_rotl(_bitr(_deal(i)), 16)) >> (m)))
#else
# define DIG_REV(i, m, j)                                                    \
    do {                                                                     \
        unsigned _ = (i);                                                    \
        _ = ((_ & 0x33333333) <<  2) | ((_ & ~0x33333333) >>  2);            \
        _ = ((_ & 0x0F0F0F0F) <<  4) | ((_ & ~0x0F0F0F0F) >>  4);            \
        _ = ((_ & 0x00FF00FF) <<  8) | ((_ & ~0x00FF00FF) >>  8);            \
        _ = ((_ & 0x0000FFFF) << 16) | ((_ & ~0x0000FFFF) >> 16);            \
        (j) = _ >> (m);                                                      \
    } while (0)
#endif
```

- target-specific intrinsics functions
- application-level intrinsics functions

`restrict` Pointers

- introduced by Cray Research to bring FORTRAN non-aliasing of function parameters to C [Homer "Restricted Pointers in C"]

```
void f10(int n, float *restrict a, float *b, float *c) {
   int i;
   for ( i=0; i<n; i++ )
       a[i] = b[i] + c[i];
}
```

- non-aliasing of memory references asserted by `restrict` is only valid in the pointer lexical scope
- most compilers forget lexical scopes in the code generator
  - GCC and Open64 miss this dependency between `*p` and `*q`

```
{ int *restrict p = r; *p++; }
{ int *restrict q = s; *q = 0; }
```

`restrict` Compiler Implementations

Cray Research PVP only consider `restrict` on parameters of non-inlined functions (correct)

Multiflow only consider `restrict` on functions parameters, in case of inlining 'incarnation numbers' must match (correct)

Open64 `restrict` pointer dereferences do not alias except with pointers derived from self (incorrect)

GCC `restrict` pointer dereferences do not alias with any other `restrict` pointer dereferences (incorrect)

- S. Freudenberger proposes to combine the 'incarnation number' of inlining with the DFS number of scope nesting: will be implemented in STMicroelectronics Open64-based compilers
- M. Mock "Why Programmer-specified Aliasing is a Bad Idea"

`#pragma ivdep` in High-Performance Compilers

```
#pragma ivdep
for (i = 0; i < N; i++) {
    a[i] = a[i+k] + 1;
}
```

Cray Research PVP "the compiler ignores vector dependencies, including explicit dependencies, in any attempt to vectorize the loop"

MIPSPRO & Open64 "IVDEP informs the compiler that no loop carried dependencies should be assumed"

Intel ICC "none of the conservatively assumed data dependences that prohibit vectorization of the loop actually occur"

Multiflow interprets IVDEP as "no memory dependences" after register variable promotion

`#pragma ivdep` Interpretations

vector ignore lexically upward dependences (Cray PVP, intel ICC)

parallel ignore loop-carried dependences (MIPSPRO, Open64)

liberal ignore loop-variant dependences (Multiflow)

`#pragma ivdep` in STMicroelectronics Compilers

- a command-line option selects among these three interpretations
- `#pragma loopdep` `VECTOR` or `PARALLEL` or `LIBERAL`
- only dependences that involve at least one loop-variant memory reference are considered for removal
- a loop variant is: a scalar induction: a dereference of a loop variant; a function with a loop variant argument

needed: C equivalent of the FORALL construct of Fortran 95 / HPF

Programmer Assumptions Example

- TI DSP Library 32-bit fixed-point FFT for C64x processors:

```
#ifndef NOASSUME
 _nassert((int)(w)%8 == 0);
 _nassert((int)(x)%8 == 0);
 _nassert(h2 %8 == 0);
 _nassert(l1 %8 == 0);
 _nassert(l2 %8 == 0);
 _nassert(npoints >= 16);
 #pragma MUST_ITERATE(4, , 1)
 #endif

 for (i = 0; i < npoints; i += 4)
 {
     /*----------------------------------------------------------*/
     /*  Read the first three twiddle factor values. This loop co- */
     /*  mputes one radix 4 butterfly at a time.                  */
     /*----------------------------------------------------------*/
     co10 = w[j+1];            si10 = w[j+0];
     co20 = w[j+3];            si20 = w[j+2];
     co30 = w[j+5];            si30 = w[j+4];
```

Programmer Assumptions Motivations

- loop iterates at least n, at most m, loop counter is $ak + b, k \in \mathbb{N}$
  - reduce loop unrolling overhead, enable modulo expansion / kernel unrolling in case of counted hardware loops
- data pointer is aligned or mis-aligned w.r.t. wider data type
  - enable memory access packing, reduce SIMDization overhead

Programmer Assumptions Exploitation

- a simple built-in function `__builtin_assume(`*boolean expression*`)` is enough to feed the compiler
- propagate facts collected from conditionals, data declarations, and `__builtin_assume()`, by data-flow analysis
  - build on Wegman & Zadeck "Constant Propagation with Conditional Branches" [TOPLAS 13(2) 1991] algorithm

Machine-Level SSA Form

SSA Construction Issues: Non-Kill Definitions

- all SSA variable definitions are kills, while conditional definitions and partial register writes are not
- work-around: enforce ordering of non-kill definitions

SSA Destruction Issues: Operand Constraints

- some SSA variables must be renamed into dedicated registers, or same/different virtual registers: architectural limitations, procedure-calling conventions
- work-around: insert COPY operations to isolate the constrained operands, then rely on the register allocator biased coloring

detrimental effects especially with pre-pass instruction scheduling

Review of the SSA the Destruction Techniques

- Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph” [TOPLAS 13(4) 1991]
  - insert COPY for the arguments of Φ-functions in the predecessors, then rely on the register allocator coalescing
- Briggs et al. “Practical Improvements to the Construction and Destruction of Static Single Assignment Form” [SPE 28(8) 1998]
  - identifies ’Lost Copy’ and ’Swap’ problems
  - fix incorrect behavior of Cytron et al. [TOPLAS 13(4) 1991] when critical edges are not split

- Budimlić et al. “Fast Copy Coalescing and Live-Range Identification” [PLDI’02]
  - lightweight SSA destruction motivated by JIT compilation
  - use the SSA form dominance of definitions over uses to avoid explicit interference graph
  - construct SSA-webs with early pruning of interfering variables, then partition into non-interfering classes
  - introduce the “dominance forest” data-structure to avoid quadratic number of interference tests
  - critical edge splitting is required

- Sreedhar et al. “Translating Out of Static Single Assignment Form” [SAS’99] (US patent 6182284):
  - Method 1 inserts COPY for the arguments of Φ-functions in the predecessors *and* for the Φ-functions targets in the current block, then applies a new SSA-based coalescing algorithm
  - Method 3 maintains liveness and interference graph to insert COPY that will not be removed by the new SSA-based coalescing algorithm
  - the new SSA-based coalescing algorithm is more effective than register allocation coalescing

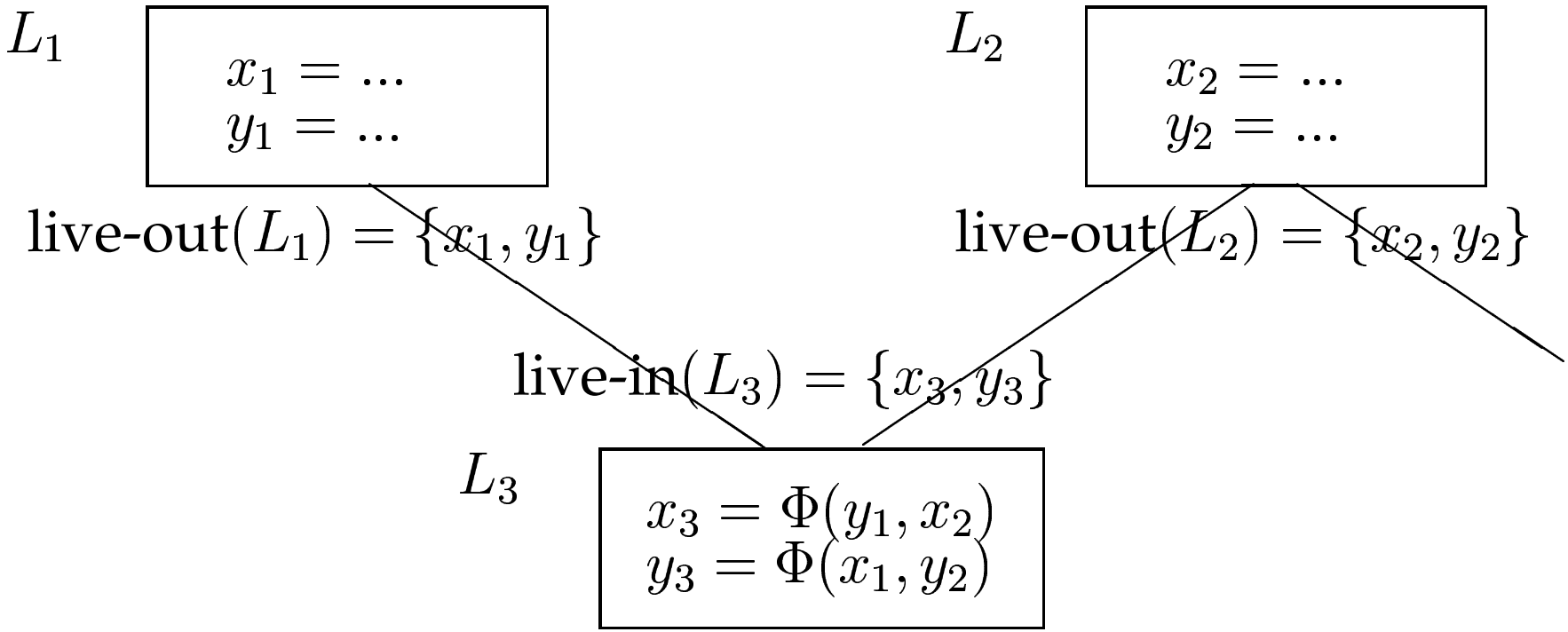
- Leung & George "Static Single Assignment Form for Machine Code" [PLDI'99]
  - handles the operand constraints of machine-level SSA form
  - builds on the algorithm by Briggs et al. [SPE 28(8) 1998]
- Rastello et al. "Optimizing Translation Out of SSA using Renaming Constraints" [CGO'04] (STMicroelectronics)
  - fix bugs and generalize Leung & George [PLDI'99]
  - generalize Sreedhar et al. [SAS'99] (and avoid patent)

STMicroelectronics Compilers SSA Destruction

STMicroelectronics compilers use Sreedhar et al. [SAS'99] extensions

- work on extensions for operand constraints and predicated code

Sreedhar et al. [SAS 1999] Liveness and Congruence Example



- variables $x_1, x_3, x_3, y_1, y_2, y_3$ are in the same congruence class
- in this example, several interferences inside the congruence class

Insights of Sreedhar et al. [SAS 1999]

- a Φ-congruence class is the closure of the Φ-connected relation
- liveness under SSA form: Φ arguments are live-out of predecessor blocks and Φ targets are live-in of Φ block
- SSA form is *conventional* if no two members of a Φ-congruence class interfere under this liveness
- correct SSA destruction is the removal of Φ-functions from a conventional SSA form
- after SSA construction (without COPY propagation), the SSA form is conventional
- Methods 1 – 3 restore a conventional SSA form
- the new SSA-based coalescing is able to coalesce interfering variables, as long as the SSA form remains conventional

Predicated Code Support

Fully Predicated Code in a Code Generator

- fully predicated code results from intrinsic function expansion and from explicit if-conversion
- it is necessary to contain the live-range of variables with conditional definitions
  - flow analysis to insert 'pseudo-KILL' operations or set a 'kill property' on conditional definitions
  - as shown by Gillies et al. [MICRO 1996], this is critical for a graph-coloring register allocation to converge
- SSA form not applicable to variables with conditional definitions

If-Conversion Outside SSA Form

- Park & Schlansker "On Predicated Execution" [HPL-91-58 1991]
  - 'R-K algorithm' generalizes the Cydrome if-conversion
  - operates on the control dependence graph
- Fang "Compiler Algorithms on If-Conversion, Speculative Predicates Assignment and Predicated Code Optimizations" [LCPC 1996]
  - simple and effective if-conversion using (post) dominance
  - operates on acyclic SEME code regions
- Chuang et al. "Phi-Predication for Light-Weight If-Conversion" [CGO 2003]
  - generates SELECT operations (not SSA form if-conversion)

If-Conversion Under SSA Form (STMicroelectronics)

- Stoutchinin & Gao “If-Conversion in SSA Form” [Euro-Par 2004]
  - prove it is correct to replace Φ-functions by Ψ-functions in conventional SSA form
  - apply to Fang [LCPC 1996] for if-conversion under SSA form
  - implement in Open64 for IA64
- Bruel “If-Conversion SSA Framework for Partially Predicated VLIW Architectures” [ODES-4 2006]
  - rework Muliflow-like if-conversion algorithm for SSA form
  - locally generate Ψ-functions for predicated LOAD & STORE
  - production use in ST200 compilers (ST220, ST231, ST240)

Ψ-SSA Form for Predicated Code

Stoutchinin & Ferriere "Efficient Static Single Assignment Form for Predication" [MICRO 2001] motivated by the ST120 LAO

```
if(p)
   a = op1;          p?    a = op1;
else
   b = op2;          p̄?    b = op2;
x = Φ(a, b)               x = Ψ(a, b)
```

- Ψ arguments are ordered from left to right in dominance order
- conditional definitions are seen as unconditional definitions and condition argument are treated like other arguments
- classic SSA form optimizations just work on Ψ-SSA form
- Ψ-SSA form does not require a predicate query system

Ψ-SSA Form Construction and Optimizations

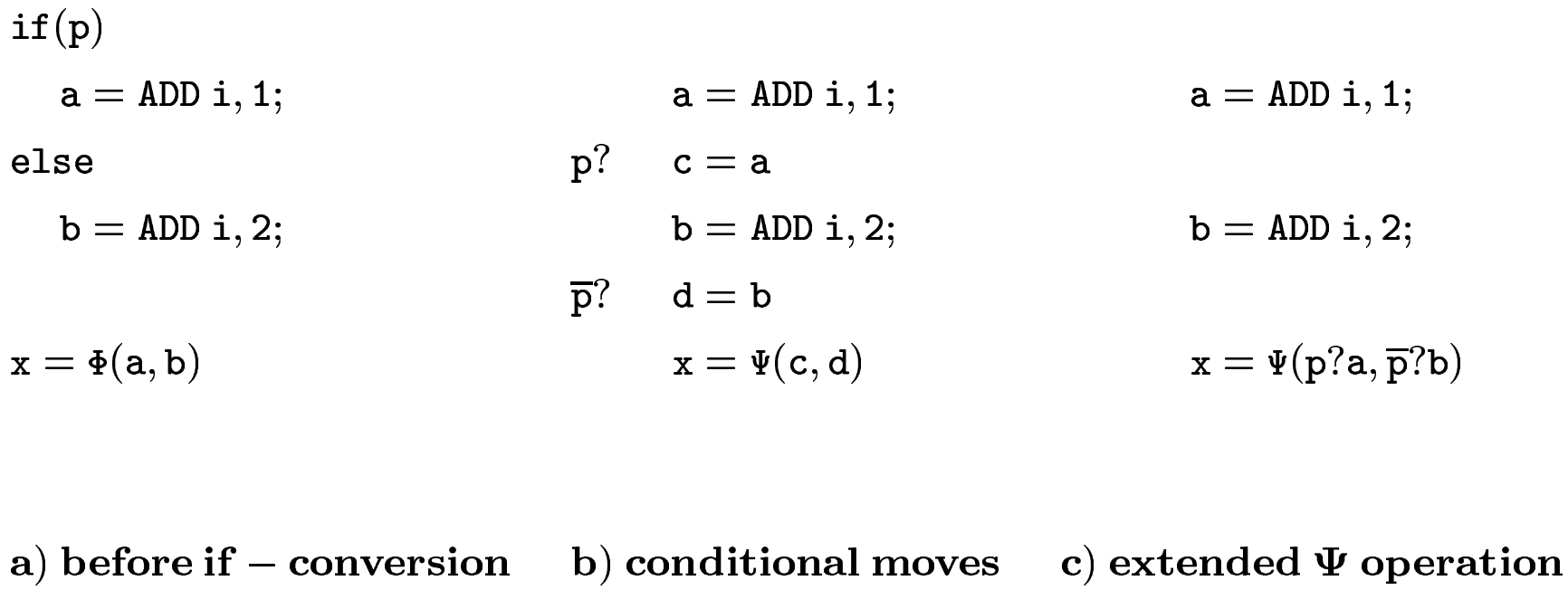
- Ψ insertion is a simple extension of the classic SSA construction variable renaming process
- Ψ-inlining recursively replaces a Ψ argument defined by another Ψ operation by the arguments of this second Ψ operation

```
if(p)
   a = 1;          p?   a = 1;
else
   b = -1;         p̄?   b = -1;
x = Φ(a, b)             x = Ψ(a, b)
if(q)
   c = 0;          q?   c = 0;
y = Φ(x, c)             y = Ψ(a, b, c)
```

- Ψ-reduction removes the Ψ arguments overridden on their right
- Ψ-projection clones and specialize Ψ for different uses predicates

Ψ-SSA Form for Partial Predication

Ferriere "Improvements to the Psi-SSA Representation" [SCOPES 2007] implemented in the STMicroelectronics ST200 compiler



a) before if − conversion **b) conditional moves** **c) extended Ψ operation**

- general methods to manipulate SELECT and CMOV operations
- extends the Ψ-functions with predicates on arguments

The Ψ-SSA Form Destruction [Ferriere SCOPES'06]

Generalize the Sreedhar et al. [SAS'99] algorithm:

- a Ψ-congruence class is the transitive closure of the Ψ-connected and Φ-connected relations
- algorithm builds a "Ψ conventional SSA form":

  Ψ-normalize ensure that all Ψ functions are well-behaved w.r.t. Ψ argument ordering and predicate association

  Ψ-congruence grow the Ψ-congruence classes from the Ψ functions, introducing repair code to prevent interferences

  Φ-congruence extend the Ψ-congruence classes with Φ operations, like in the Sreedhar et al. [SAS'99] algorithm

- eliminate the Ψ and Φ functions

Near-Optimal Software Pipelining

Integer Linear Programming Instruction Scheduling

- Wilken "Optimal Instruction Scheduling Using Integer Programming" [PLDI 2000]
  - basic block scheduling on superscalar processors
  - use time-indexed variables, range reductions, integer cuts
  - schedules up to 1000 instructions with CPLEX
- Kästner & Winkel "ILP-based Instruction Scheduling for IA-64" [LCTES 2001]
  - basic block scheduling with bundling constraints
  - use time-indexed variables and the dependence equations of Chaudhuri et al. [IEEEtoVLSI 1994]

- Streeter "An Integer Programming Approach to Instruction Scheduling" [15-745 Project 2006]
  - superblock scheduling with cluster assignment and cross-path management of the TI C6x processors
  - use time-indexed variables and classic dependence equation
  - improve heuristic solution with variable neighborhood search
- Dinechin "Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem" [MISTA 2007]
  - superblock scheduling and software pipelining for the ST200
  - new time-indexed formulation that differs from Eichenberger & Davidson [PLDI'97] modulo scheduling formulation
  - improve heuristic solution with variable neighborhood search

Time-Indexed Project Scheduling Formulation [Pritsker et al. 1969]

- a set of operations $\{O_i\}_{1 \leq i \leq n}$ with schedule dates $\{\sigma_i\}_{1 \leq i \leq n}$
- T denotes the time horizon and $\{x_i^t\}_{1 \leq i \leq n}^{0 \leq t < T}$ are $\{0,1\}$ variables such that $x_i^t \stackrel{\text{def}}{=} 1$ if $\sigma_i = t$, else $x_i^t \stackrel{\text{def}}{=} 0$
- in particular we have $\sigma_i = \sum_{t=1}^{T-1} t x_i^t$ and $\sum_{t=0}^{T-1} x_i^t = 1$
- classic dependence equation for $O_i \rightarrow O_j$ of latency l_i^j:

$$\sigma_i + l_i^j \leq \sigma_j \quad \Rightarrow \quad \sum_{t=1}^{T-1} t x_i^t + l_i^j \leq \sum_{t=1}^{T-1} t x_j^t$$

- Christofides et al. [1987] use the following equations instead:

$$\sum_{s=t}^{T-1} x_i^s \leq \sum_{s=t+l_i^j}^{T-1} x_j^s \quad t \in [0, T-1]$$

indeed, $x_i^t = 1$ implies some $x_j^s = 1$ with $s \in [t + l_i^j, T-1]$

- assume m resources with availabilities $B_r > 0, \forall r \in [1, m]$
- operation O_i requires b_i^r units of resource r for all dates in $[\sigma_i, \sigma_i + p_i - 1]$

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

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

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

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

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

Modulo Scheduling Principles

- modulo scheduling is cyclic scheduling where schedules $\{\sigma_i^k\}_{1\leq i\leq n}^{k\in\mathbb{N}}$ must be 1-periodic of integral period λ:

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

- uniform dependences between the generic operation schedule dates $\{\sigma_i = \sigma_i^0\}_{1\leq i\leq n}$:

$$O_i \overset{\theta_i^j,\omega_i^j}{\rightarrow} O_j \Longrightarrow \sigma_i^k + \theta_i^j \leq \sigma_j^{k+\omega_i^j} \Longrightarrow \sigma_i + \theta_i^j - \lambda\omega_i^j \leq \sigma_j$$

- modulo resource constraints: each generic operation O_i requires $\vec{b_i} \geq \vec{0}$ resources for all the time intervals $[\sigma_i + k\lambda, \sigma_i + k\lambda + p_i - 1], k \in \mathbb{Z}$
- The primary objective of modulo scheduling is to decrease the cyclic scheduling period λ, called the *initiation interval*

Time-Indexed Modulo Scheduling Formulation [Dinechin STJSR 2004]

$$\sum_{t=1}^{T-1} t\, x_{n+1}^t : \quad \text{minimize} \tag{6}$$

$$\sum_{t=0}^{T-1} x_i^t = 1 \qquad i \in [1, n+1] \tag{7}$$

$$\sum_{s=t}^{T-1} x_i^s + \sum_{s=0}^{t+\theta_i^j-\lambda\omega_i^j-1} x_j^s \leq 1 \qquad t \in [0, T-1], (i,j) \in E_{dep} \tag{8}$$

$$\sum_{i=1}^{n} \sum_{k=0}^{\lfloor \frac{T-1}{\lambda} \rfloor} \sum_{s=t+k\lambda-p_i+1}^{t+k\lambda} x_i^s\, \vec{b}_i \leq \vec{B} \qquad t \in [0, \lambda-1] \tag{9}$$

$$x_i^t \in \{0,1\} \qquad i \in [1, n+1], t \in [0, T-1] \tag{10}$$

Solving the Time-Indexed Formulations

- time-indexed scheduling formulations become quickly intractable in practice: beyond a few thousand variables and constraints, few instances can be solved in reasonable time
- the dependence equations of Christofides et al. [1987] (Chaudhuri et al. [IEEEtoVLSI 1994]) are easier to solve
- to reduce the number of variables and constraints, need to reduce the earliest $\{e_i\}_{1 \leq i \leq n+1}$ and latest $\{l_i\}_{1 \leq i \leq n+1}$ assumed schedule dates (margins)
- with our modulo scheduling formulation, the number of variables is $\sum_{i=1}^{n+1} l_i - e_i + 1$
- with our modulo scheduling formulation, the dependence equations (8) are redundant whenever $e_j - \theta_i^j + \lambda \omega_i^j \geq l_i$

Large Neighborhood Search for Time-Indexed Formulations

- explore a large number of solutions in the neighborhood of an incumbent solution using implicit enumeration of a MIP solver
- the neighborhood of an incumbent solution $\{\sigma_i^*\}_{1 \leq i \leq n}$ is obtained by choosing margins $[e_i, l_i] \ni \sigma_i^*$ that are made dependence-consistent with a forward and a backward label-correcting algorithm
- at each LNS step, variables $x_i^t : t \notin [e_i, l_i]$ are fixed to zero and the redundant dependence equations are removed from the MIP
- our LNS for modulo scheduling alternates between two phases:
  - try to reduce the makespan M for a given period λ
  - try to find a feasible solution at period $\lambda - 1$, given an incumbent solution at period λ

- implemented both our time-indexed modulo scheduling formulation and Eichenberger & Davidson [PLDI'97] formulation in the ST200 VLIW production compiler (LAO v2)
- with LNS and CPLEX, our new modulo scheduling formulation could be solved for the largest instances and with $\lambda = \lambda_{min}$

		Heuristic	LNS GLPK 30s		LNS CPLEX 30s	
Loop	#O,#D	λ, M	λ, M	#V,#C	λ, M	#V,#C
q_plsf_5.0_215	231,313	81,97	78,79	698,677	75,78	1873,2042
q_plsf_5.0_227	121,163	42,92	41,82	413,440	39,46	1378,1685
q_plsf_5.0_201	124,168	42,92	42,84	689,790	40,65	1197,1421
q_plsf_5.2_11	233,317	82,100	78,79	1178,1129	75,79	1897,2045
subbands.0_196	130,234	44,65	41,47	598,666	35,48	1008,1248
transfo.IMDCT_L	232,370	71,109	58,58	1127,881	58,58	1985,1961

  - #O and #D are the number of operations and dependences
  - λ and M are the period (initiation interval) and the makespan
  - #V and #C are the number of variables and constraints

Summary and Conclusions

Embedded VLIW Code Generation Successes

- programmer-supplied information for performance & code size
  - memory disambiguation with `restrict` and `#pragma ivdep`, inspired from Fortran high-performance features
  - intrinsic functions and `__builtin_assume()`
- SSA form on machine code with operand constraints
  - better instruction scheduling and simpler register allocation
- Ψ-SSA form for predicated code and if-conversion
  - Ψ-SSA form with operand constraints under development
- time-indexed formulations for instruction scheduling extensions
  - clusterization, spill code control, memory access grouping
  - from our experience and Streeter's, GLPK is quite capable

GCC Improvements for Embedded VLIW Processors

- manual memory disambiguation will not go away
  - `restrict` should work better
  - `#pragma ivdep` is a necessary evil
- other programmer assumptions should be standardized
  - `__builtin_assume(`*boolean expression*`)`
- simple predication support at GIMPLE level?
  - SELECT operator does not require Ψ-SSA
- other issues not discussed in this presentation
  - LOAD control speculation without architectural support
  - register tuples in SSA form and register allocation
  - data placement, memory hierarchy optimizations

Alternative to GCC Code Generator Re-Engineering

- compile for the Common Language Infrastructure (CLI) program representation
  - STMicroelectronics `st/cli` GCC branch for C to CLI
  - LLVM 2.0 MSIL back-end (transforms LLVM IR into CIL)
- complete compilation with a CLI to native code generator
  - Mono is able to execute CLI produced by GCC `st/cli`
  - STMicroelectronics has prototyped a CLI to ST200 and ARM JIT code generator, based on the LAO v2 technology
- how to carry programmer-supplied information through the CLI program representation?

STMicroelectronics is looking for collaborations on CLI compilation