Automatic Compiler Backend Generation from Structural Processor Models

Compilation and Embedded Computing Systems Group
Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon

INRIA

Florian Brandner

This work was supported in part by OnDemand Microelectronics and the
Christian Doppler Forschungsgesellschaft (Vienna, Austria).

Processor Description Languages

- Description of processor features
  - Programmable hardware device
  - Including RISC-, CISC-, VLIW-style architectures

Processor Description Languages

- Description of processor features
  - Programmable hardware device
  - Including RISC-, CISC-, VLIW-style architectures
- Instruction set architecture
  - Assembly syntax, binary encoding
  - Abstract behavior
  - *High-level view*

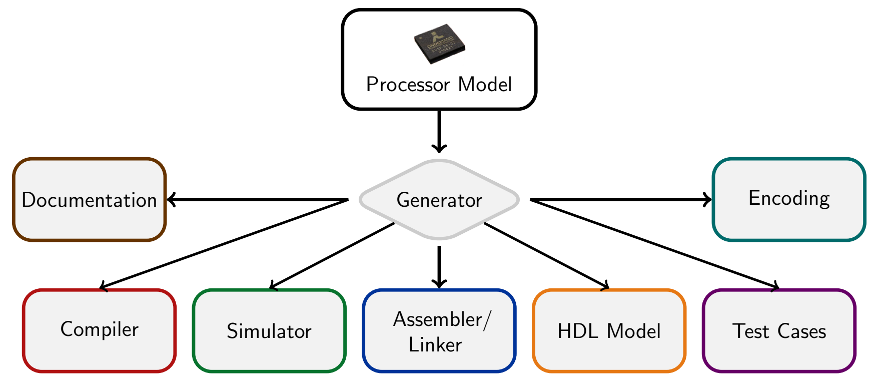
Processor Description Languages

- Description of processor features
  - Programmable hardware device
  - Including RISC-, CISC-, VLIW-style architectures
- Instruction set architecture
  - Assembly syntax, binary encoding
  - Abstract behavior
  - *High-level view*
- Hardware organization
  - Capabilities and number of computational resources
  - Storage elements such as register files, caches, and memories
  - Wires, buses, and interconnect
  - *Low-level view*

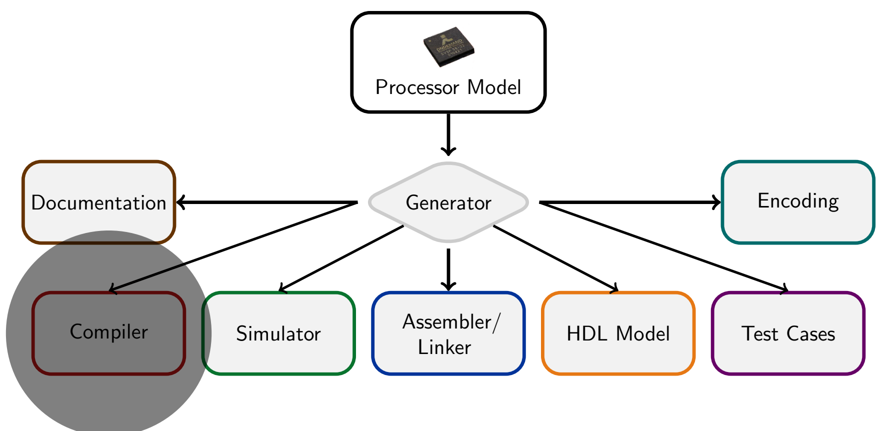
Processor Description Languages

- Description of processor features
  - Programmable hardware device
  - Including RISC-, CISC-, VLIW-style architectures
- Instruction set architecture
  - Assembly syntax, binary encoding
  - Abstract behavior
  - *High-level view*
- Hardware organization
  - Capabilities and number of computational resources
  - Storage elements such as register files, caches, and memories
  - Wires, buses, and interconnect
  - *Low-level view*
- Language classification:
  - Behavioral (focusing on the high-level view)
  - Structural (focusing on the low-level view)
  - Mixed (covering both views)

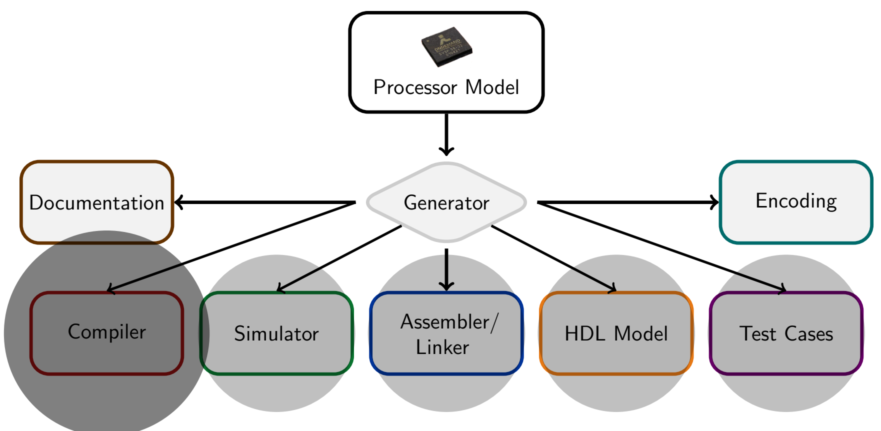
Application of Processor Description Language



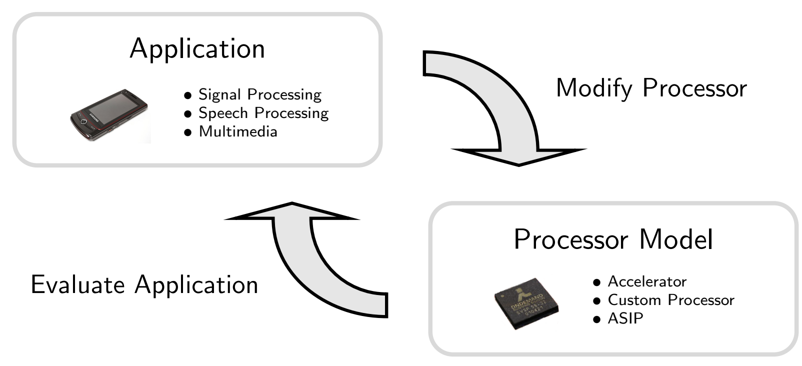
Application of Processor Description Language



Application of Processor Description Language



Design Space Exploration



The ultimate goal is to support seamless Design Space Exploration using automatically generated development tools, simulation tools, automatic testing and verification, as well as hardware generation.

xADL Processor Description Language

- Structural description of processor features
    - Components interconnected by *links*
        - Functional units, caches, memories, registers
        - Based on extensible types
        - Support for generics
        - Abstractions (bypassing, hazard resolution, pipelining, ...)

xADL Processor Description Language

- Structural description of processor features
  - Components interconnected by *links*
    - Functional units, caches, memories, registers
    - Based on extensible types
    - Support for generics
    - Abstractions (bypassing, hazard resolution, pipelining, ...)
  - Binary encoding, assembly syntax, programming conventions

xADL Processor Description Language

- Structural description of processor features
  - Components interconnected by *links*
    - Functional units, caches, memories, registers
    - Based on extensible types
    - Support for generics
    - Abstractions (bypassing, hazard resolution, pipelining, ...)
  - Binary encoding, assembly syntax, programming conventions
  - Instruction set architecture
    - Automatically extracted from the structural model
    - Along *instruction paths*

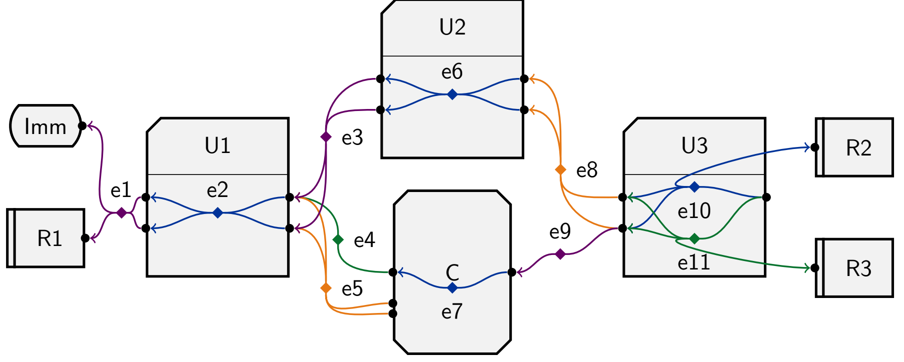
xADL Processor Description Language

- Structural description of processor features
  - Components interconnected by *links*
    - Functional units, caches, memories, registers
    - Based on extensible types
    - Support for generics
    - Abstractions (bypassing, hazard resolution, pipelining, ...)
  - Binary encoding, assembly syntax, programming conventions
  - Instruction set architecture
    - Automatically extracted from the structural model
    - Along *instruction paths*

- Available generator tools:

  - Compiler backend
  - Instruction set simulator
  - GNU binutils (in progress)
  - Instruction decoder
  - Prototype: VHDL model
  - ...

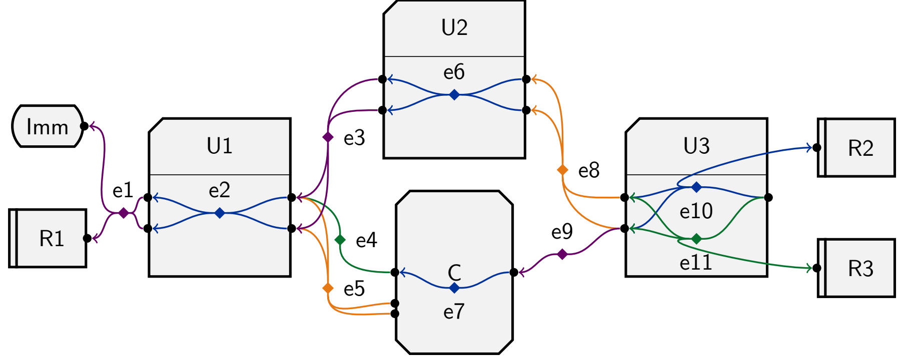
Hypergraph Representation



Reduced, abstract model of the processor's data path

- Captures flow of instructions through the pipeline
- Legal port assignments expressed by *hyperedges*
- Ignore internal details, bypasses, hazard resolution logic, ...

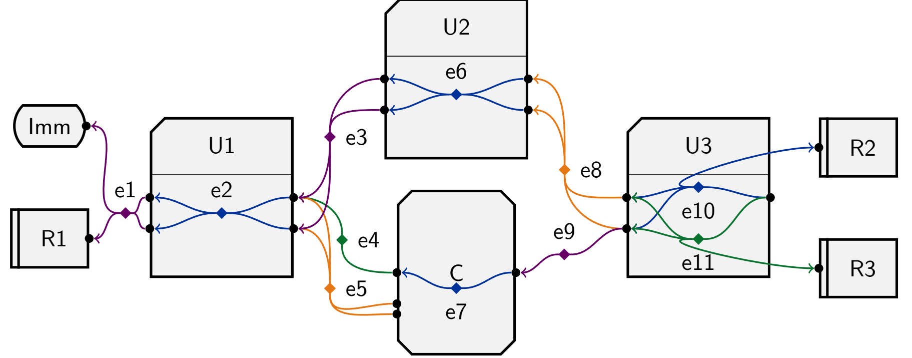
Hypergraph Representation



Discover paths using a backward traversal

- Starting at *endpoints*
- Collect hyperedges until the start of the pipeline is reached

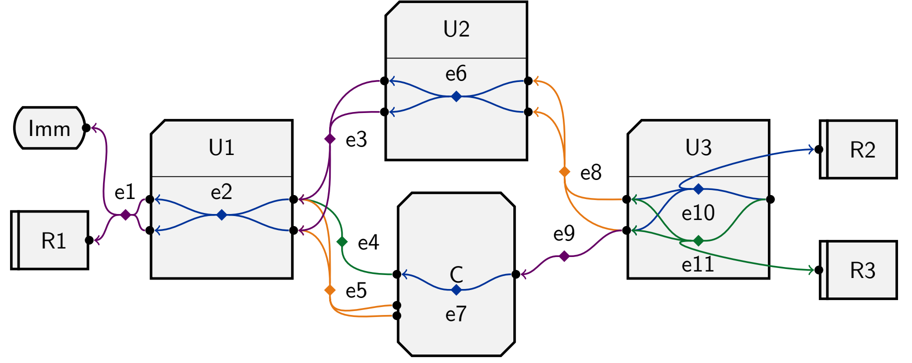
Hypergraph Representation



Discover paths using a backward traversal

- Starting at *endpoints* ($e5$, $e10$, $e11$)
- Collect hyperedges until the start of the pipeline is reached

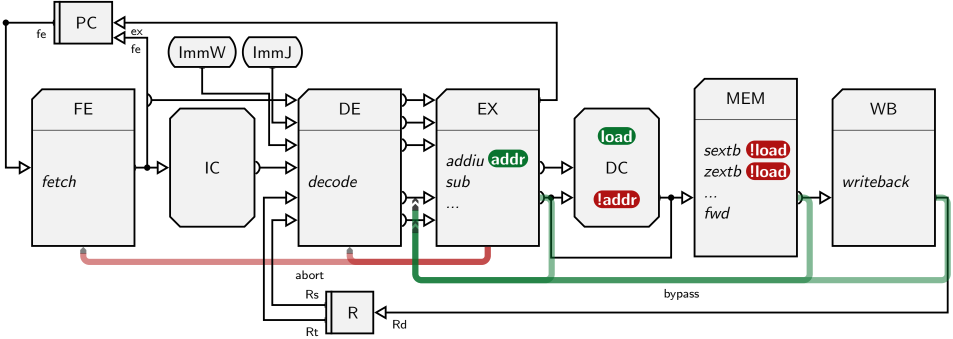
Hypergraph Representation



Discovered paths:

- $\{e1, e2, e3, e6, e8, \mathbf{e10}\}$
- $\{e1, e2, e4, e7, e9, \mathbf{e10}\}$
- $\{e1, e2, e3, e6, e8, \mathbf{e11}\}$
- $\{e1, e2, e4, e7, e9, \mathbf{e11}\}$
- $\{e1, e2, \mathbf{e5}\}$

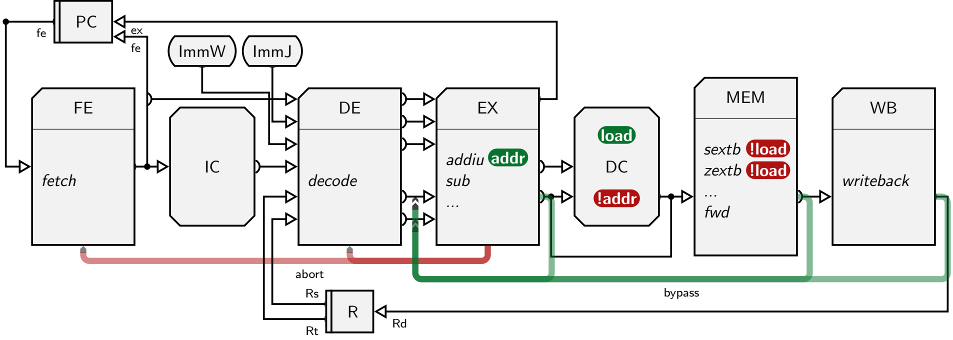
Example: MIPS Model



Instruction paths:

- FE:IC:DE:EX:DC:MEM::WB
- FE:IC:DE:EX:MEM::WB
- FE:IC:DE:EX
- ...

Example: MIPS Model



Instructions:

- FE:fetch IC DE:decode EX:ori MEM:fwd WB:writeback
- FE:fetch IC DE:decode EX:addiu MEM:fwd WB:writeback
- FE:fetch IC DE:decode EX:addiu DC MEM:zextb WB:writeback
- ...

Instruction Set Representation

Instructions are derived from paths:

- Instruction model tightly coupled with structural view

Instruction Set Representation

Instructions are derived from paths:

- Instruction model tightly coupled with structural view
- Instructions are characterized by
  - The source instruction path
  - *Operations* attached to functional units along the path

Instruction Set Representation

Instructions are derived from paths:

- Instruction model tightly coupled with structural view
- Instructions are characterized by
  - The source instruction path
  - *Operations* attached to functional units along the path
- Serialize operations
  - Operations in turn consist of *micro-operations*
  - Micro-operations have well-defined semantics
  - Limited control flow in behavioral model

Instruction Set Representation

Instructions are derived from paths:

- Instruction model tightly coupled with structural view
- Instructions are characterized by
  - The source instruction path
  - *Operations* attached to functional units along the path
- Serialize operations
  - Operations in turn consist of *micro-operations*
  - Micro-operations have well-defined semantics
  - Limited control flow in behavioral model
- Enrich the instruction model
  - Annotate with timing information
  - Information on data hazards, stalls, and bypasses
  - Analyze branching and memory access patterns

Example: *or immediate* instruction

```
FE::pc_i = move(pc::p_fe)                 [st: 0, op: fe]
FE::pc_o = add(FE::pc_i, const_4)         [st: 0, op: fe]
pc::p_fe = move(FE::pc_o)                 [st: 0, op: fe]
ICache::@read = move(FE::pc_o)            [st: 0]
ICache::read = read(ICache::@read)        [st: 0]
------------------------------------------------------------
DE::ImmW_i = move(ImmW)                   [st: 1, op: de]
DE::Rs_i = move(R::Rs[0,31])              [st: 1, op: de]
DE::IW_i = move(ICache::read)             [st: 1, op: de]
decode(IW_i)                              [st: 1, op: de]
DE::Rs_o = move(DE::Rs_i)                 [st: 1, op: de]
DE::ImmWu_o = zext(DE::ImmW_i)            [st: 1, op: de]
------------------------------------------------------------
EX::ImmWu_i = move(DE::ImmWu_o)           [st: 2, op: ori]
EX::Rs_i = move(DE::Rs_o)                 [st: 2, op: ori]
EX::Rd_o = or(EX::Rs_i, EX::ImmWu_i)      [st: 2, op: ori]
------------------------------------------------------------
MEM::Rd_i = move(EX::Rd_o)                [st: 3, op: fwd]
MEM::Rd_o = move(MEM::Rd_i)               [st: 3, op: fwd]
------------------------------------------------------------
WB::Rd_i = move(MEM::Rd_o)                [st: 4, op: wb]
WB::Rd_o = move(WB::Rd_i)                 [st: 4, op: wb]
R::Rd[0,31] = move(WB::Rd_o)              [st: 4, op: wb]
```

Compiler Backend Generator

Derive compiler backend from processor models

- Retarget LLVM compiler infrastructure (version 2.4)
- Retarget proprietary compiler backend *acc*

Compiler Backend Generator

Derive compiler backend from processor models

- Retarget LLVM compiler infrastructure (version 2.4)
- Retarget proprietary compiler backend *acc*
- Instruction selector
  - Derive tree patterns from instruction set model
  - Extend coverage
  - Verify completeness

Compiler Backend Generator

Derive compiler backend from processor models

- Retarget LLVM compiler infrastructure (version 2.4)
- Retarget proprietary compiler backend *acc*
- Instruction selector
  - Derive tree patterns from instruction set model
  - Extend coverage
  - Verify completeness
- Instruction scheduler
  - Instruction paths resemble possible execution flow
  - Correspond to resource tables for scheduling in LLVM
  - Additional information required for *Operation Tables* in *acc*

Compiler Backend Generator

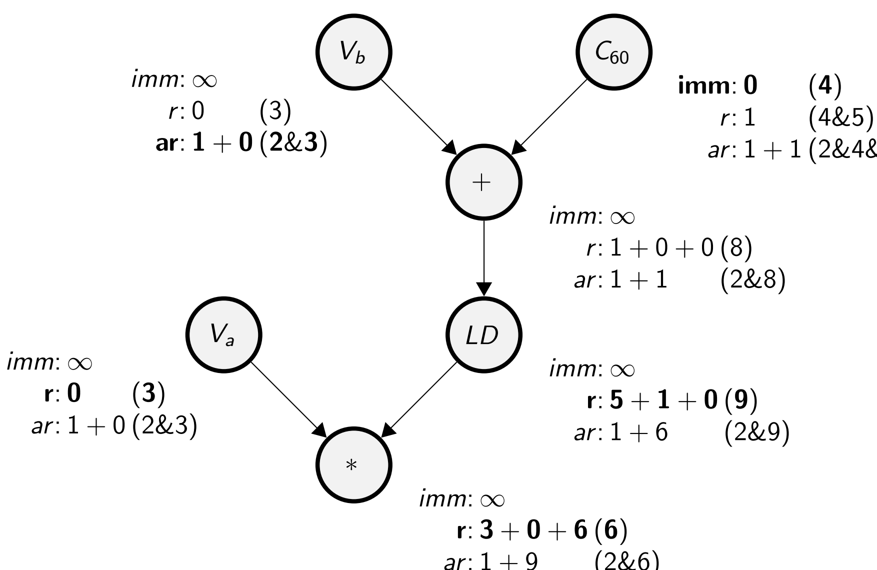
Derive compiler backend from processor models

- Retarget LLVM compiler infrastructure (version 2.4)
- Retarget proprietary compiler backend *acc*
- Instruction selector
  - Derive tree patterns from instruction set model
  - Extend coverage
  - Verify completeness
- Instruction scheduler
  - Instruction paths resemble possible execution flow
  - Correspond to resource tables for scheduling in LLVM
  - Additional information required for *Operation Tables* in *acc*
- Register allocator
  - Derive register classes from register files/ports

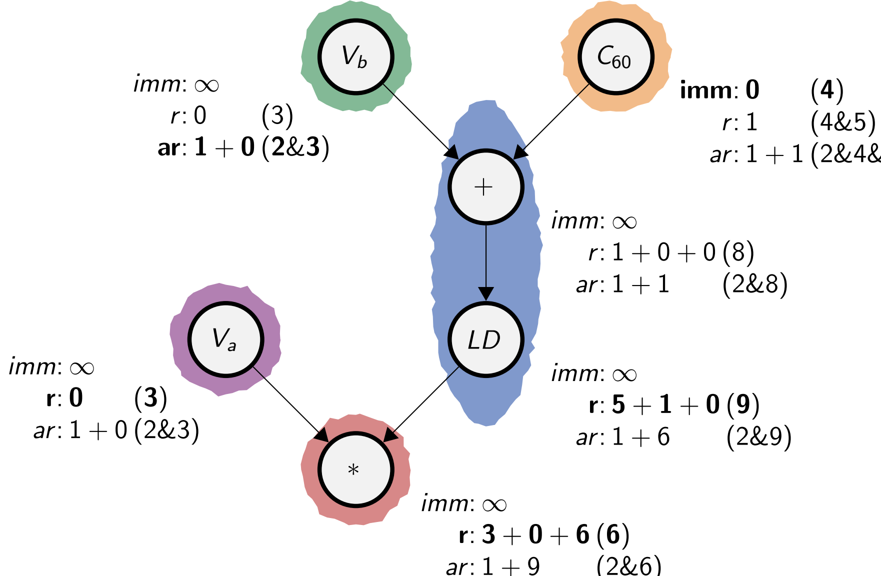
Instruction Selection using Tree Pattern Matching

Pattern				Cost	Emit
(1)	r	$\rightarrow$	ar	1	`mov r = ar`
(2)	ar	$\rightarrow$	r	1	`mov ar = r`
(3)	r	$\rightarrow$	$\mathbf{V}$	0	`V`
(4)	imm	$\rightarrow$	$\mathbf{C}$	0	`C`
(5)	r	$\rightarrow$	imm	1	`ldi r = imm`
(6)	r	$\rightarrow$	$*(r_1, r_2)$	3	$\texttt{mul r} = \texttt{r}_1 * \texttt{r}_2$
(7)	r	$\rightarrow$	$+(r_1, r_2)$	1	$\texttt{add r} = \texttt{r}_1 + \texttt{r}_2$
(8)	r	$\rightarrow$	$+(r_1, imm)$	1	$\texttt{add r} = \texttt{r}_1 + \texttt{imm}$
(9)	r	$\rightarrow$	$\mathbf{LD}(+(ar_1, imm))$	5	$\texttt{ld r} = [\texttt{ar}_1 + \texttt{imm}]$

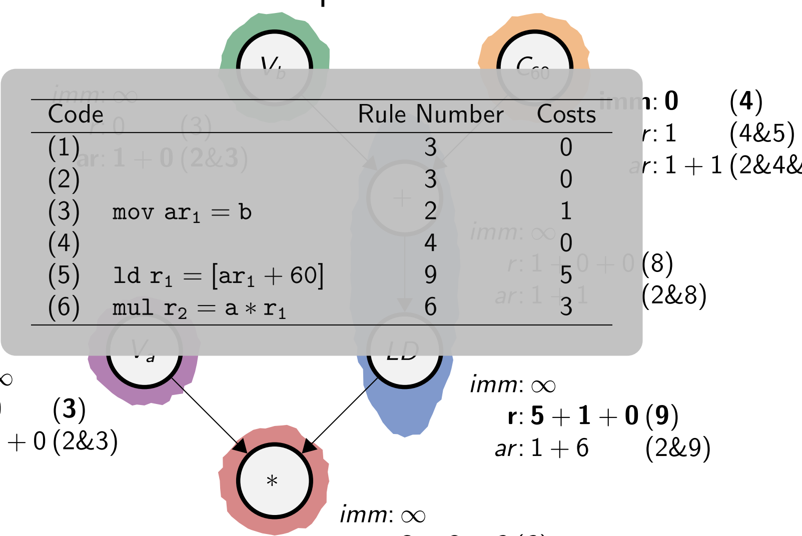
Covering the Intermediate Representation



Covering the Intermediate Representation



Covering the Intermediate Representation



Code		Rule Number	Costs
(1)		3	0
(2)		3	0
(3)	`mov ar₁ = b`	2	1
(4)		4	0
(5)	`ld r₁ = [ar₁ + 60]`	9	5
(6)	`mul r₂ = a * r₁`	6	3

Cost Functions and Dynamic Checks

Dynamic cost functions

- The cost of covering a tree fragment can be computed dynamically at compile time
- May inspect compiler options, the compilation context, or the covered tree fragment
- Usually, specified using code, and is thus hardly analyzable

Cost Functions and Dynamic Checks

Dynamic cost functions

- The cost of covering a tree fragment can be computed dynamically at compile time
- May inspect compiler options, the compilation context, or the covered tree fragment
- Usually, specified using code, and is thus hardly analyzable
- What happens when a cost function returns infinity?

Cost Functions and Dynamic Checks

Dynamic cost functions

- The cost of covering a tree fragment can be computed dynamically at compile time
- May inspect compiler options, the compilation context, or the covered tree fragment
- Usually, specified using code, and is thus hardly analyzable
- What happens when a cost function returns infinity?
  - The rule is effectively disabled
  - We call such cost functions dynamic checks

Completeness

An instruction selector is said to be complete if:

> For every possible input program, accepted by the compiler frontend, a cover of the intermediate representation can be found by the instruction selector.

Completeness

An instruction selector is said to be complete if:

> For every possible input program, accepted by the compiler frontend, a cover of the intermediate representation can be found by the instruction selector.

Can we prove completeness?

- Using recognizable tree languages/finite tree automata

Completeness

An instruction selector is said to be complete if:

> For every possible input program, accepted by the compiler frontend, a cover of the intermediate representation can be found by the instruction selector.

Can we prove completeness?

- Using recognizable tree languages/finite tree automata
- Specify all possible input programs (IR) and the instruction selector (IS) using tree grammars G_{ir} and G_{is}:
    - Check emptiness of $L_{ir} \cap \overline{L_{is}}$

Completeness

An instruction selector is said to be complete if:

> For every possible input program, accepted by the compiler frontend, a cover of the intermediate representation can be found by the instruction selector.

Can we prove completeness?

- Using recognizable tree languages/finite tree automata
- Specify all possible input programs (IR) and the instruction selector (IS) using tree grammars G_{ir} and G_{is}:
  - Check emptiness of $L_{ir} \cap \overline{L_{is}}$
- **Problem: Dynamic checks cannot be modeled!**

Representing Dynamic Checks

We need a formalization of conditions:

- Extended tree grammars with conditions
- Associate tree terms with properties
- Conditions modeled as conjunction of simple tests
- Simple tests are modeled as subsets over property domains
- Formally:
  - Domain: $D_i, \ldots, D_n$
  - Properties: $p = (p_i, \ldots, p_n) \in D_i \times \ldots \times D_n$
  - Conditions: $c = (c_i, \ldots, c_n) \in D = \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n)$
  - Simple tests: c_i of a condition
  - Dynamic check: $\forall i \in \{1, \ldots, n\} : \bigwedge p_i \in c_i$

Example

- Domain
  $\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{char, short, int\}$
- Term with properties
  $\mathbf{CST}(\{60\} \times \{GPR\} \times \{int\})$
- Rules with conditions
  - Unconditional:
    $r \rightarrow \mathbf{CST}(\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{char, short, int\})$
  - Unsatisfiable:
    $r \rightarrow \mathbf{CST}(\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{\})$
  - Conditional:
    $r \rightarrow \mathbf{CST}(\{0, \ldots, 65535\} \times \{GPR\} \times \{short\})$

Representing Emit Functions

- Sequence of instructions with *operand bindings*
- Bindings:
  - Fresh virtual register
  - Constant Immediate values
  - Output of another instruction
  - Associate with sub-terms of the tree pattern

Representing Emit Functions

- Sequence of instructions with *operand bindings*
- Bindings:
  - Fresh virtual register
  - Constant Immediate values
  - Output of another instruction
  - Associate with sub-terms of the tree pattern

Note: Operand bindings can be used to express non-regular operand constraints, e.g., the equality of nodes in the intermediate representation, using the last kind of bindings.

Discovering Instruction Selection Patterns

Construct tree patterns during a backward traversal:

- Process the micro-operations of the instruction set model
- Derive pattern, conditions, and bindings
- Additionally: Derive non-terminals, construct conversion rules, special handling of branches and memory operations, ...

Discovering Instruction Selection Patterns

Construct tree patterns during a backward traversal:

- Process the micro-operations of the instruction set model
- Derive pattern, conditions, and bindings
- Additionally: Derive non-terminals, construct conversion rules, special handling of branches and memory operations, ...

μ-op.	Tree pattern	Operand Bindings
(6)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\texttt{RC_R_32}, \mathit{ZEXT}_{16}(\mathit{immediate}))$	$(\mathit{ImmW}, \underline{21})$
(7)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\texttt{RC_R_32}, \mathit{ZEXT}_{16}(_))$	$(\mathit{Rs}, \underline{1})$
(11)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(_, \mathit{ZEXT}_{16}(_))$	
(14)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(_, _)$	
(19)	$\texttt{RC_R_32} \rightarrow _$	$(\mathit{Rd}, \mathit{new_reg})$

Example: *or immediate* instruction

abort on BEX

bypasses

```
FE::pc_i = move(pc::p_fe)                    [st: 0, op: fe]
FE::pc_o = add(FE::pc_i, const_4)            [st: 0, op: fe]
pc::p_fe = move(FE::pc_o)                    [st: 0, op: fe]
ICache::@read = move(FE::pc_o)               [st: 0]
ICache::read = read(ICache::@read)           [st: 0]
--------------------------------------------------------------
DE::ImmW_i = move(ImmW)                      [st: 1, op: de]
DE::Rs_i = move(R::Rs[0,31])                 [st: 1, op: de]
DE::IW_i = move(ICache::read)                [st: 1, op: de]
decode(IW_i)                                 [st: 1, op: de]
DE::Rs_o = move(DE::Rs_i)                    [st: 1, op: de]
DE::ImmWu_o = zext(DE::ImmW_i)               [st: 1, op: de]
--------------------------------------------------------------
EX::ImmWu_i = move(DE::ImmWu_o)              [st: 2, op: ori]
EX::Rs_i = move(DE::Rs_o)                    [st: 2, op: ori]
EX::Rd_o = or(EX::Rs_i, EX::ImmWu_i)         [st: 2, op: ori]
--------------------------------------------------------------
MEM::Rd_i = move(EX::Rd_o)                   [st: 3, op: fwd]
MEM::Rd_o = move(MEM::Rd_i)                  [st: 3, op: fwd]
--------------------------------------------------------------
WB::Rd_i = move(MEM::Rd_o)                   [st: 4, op: wb]
WB::Rd_o = move(WB::Rd_i)                    [st: 4, op: wb]
R::Rd[0,31] = move(WB::Rd_o)                 [st: 4, op: wb]
```

Extending the Coverage

The computed rule set is typically not complete

- Extend the coverage using *specialization* and *templates*
- Specialization: Eliminate pattern fragments using algebraic laws and special operand bindings
- Templates: Create new patterns by combining existing rules

Extending the Coverage

The computed rule set is typically not complete

- Extend the coverage using *specialization* and *templates*
- Specialization: Eliminate pattern fragments using algebraic laws and special operand bindings
- Templates: Create new patterns by combining existing rules

	Tree Pattern	Operand Binding
(1)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\texttt{RC_R_32}, \mathit{immediate}\{0, \ldots, 65535\})$	$(\mathit{Rs}, \underline{1}), (\mathit{ImmW}, \underline{2})$
(2)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\mathit{immediate}\{0, \ldots, 65535\}, \mathit{RC_R})$	$(\mathit{Rs}, \underline{2}), (\mathit{ImmW}, \underline{1})$
(3)	$\texttt{RC_R_32} \rightarrow \mathit{immediate}\{0, \ldots, 65535\}$	$(\mathit{Rs}, 0), (\mathit{ImmW}, \underline{\epsilon})$
(4)	$\texttt{RC_R_32} \rightarrow \texttt{RC_R_32}$	$(\mathit{Rs}, \underline{\epsilon}), (\mathit{ImmW}, 0)$

Extending the Coverage

The computed rule set is typically not complete

- Extend the coverage using *specialization* and *templates*
- Specialization: Eliminate pattern fragments using algebraic laws and special operand bindings
- Templates: Create new patterns by combining existing rules

Note: The extended rule set may still be incomplete, due to restrictions of the instruction set of the target processor, missing specialization and/or template patterns, et cetera.

	Tree Pattern	Operand Binding
(1)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\texttt{RC_R_32}, \mathit{immediate}\{0,\ldots,65535\})$	$(\mathit{Rs}, \underline{1}), (\mathit{ImmW}, \underline{2})$
(2)	$\texttt{RC_R_32} \rightarrow \mathit{OR}(\mathit{immediate}\{0,\ldots,65535\}, \mathit{RC_R})$	$(\mathit{Rs}, \underline{2}), (\mathit{ImmW}, \underline{1})$
(3)	$\texttt{RC_R_32} \rightarrow \mathit{immediate}\{0,\ldots,65535\}$	$(\mathit{Rs}, 0), (\mathit{ImmW}, \underline{\epsilon})$
(4)	$\texttt{RC_R_32} \rightarrow \texttt{RC_R_32}$	$(\mathit{Rs}, \underline{\epsilon}), (\mathit{ImmW}, 0)$

Instruction Selector Completeness

Based on tree automata theory:

- Express both the instruction selector (G_{is}) and intermediate representation (G_{ir}) using *normalized* tree grammars with conditions
- Transform these grammars into regular tree automata
- Examine languages accepted by the respective automata

Instruction Selector Completeness

Based on tree automata theory:

- Express both the instruction selector (G_{is}) and intermediate representation (G_{ir}) using *normalized* tree grammars with conditions
- Transform these grammars into regular tree automata
- Examine languages accepted by the respective automata

Terminal Splitting:
Split the rules in the grammars such that for any two rules r_1 in G_{ir} and r_2 in G_{is}, where $term(r_1) = term(r_2)$ the following condition holds:

$$cond(r_1) = cond(r_2) \vee \neg overlap(cond(r_1), cond(r_2)).$$

$cond(r) \rightarrow D$ ... condition of rule r $term(r) \rightarrow \mathcal{F}$ ... terminal symbol of rule r.

Instruction Selector Completeness (2)

After terminal splitting:

- Construct equivalent automata using dedicated terminal symbols representing the conditions and terminal symbols of the rules in the original grammars
- The alphabets of the automata are guaranteed compatible

Instruction Selector Completeness (2)

After terminal splitting:

- Construct equivalent automata using dedicated terminal symbols representing the conditions and terminal symbols of the rules in the original grammars
- The alphabets of the automata are guaranteed compatible

$$v \to INT_CONST \quad \{-\infty, \ldots, \infty\}$$
$$v \to +(v, v) \quad \{-\infty, \ldots, \infty\}$$

(a) Intermediate Representation

$$r \to INT_CONST \quad \{-32768, \ldots, 32767\}$$
$$r \to INT_CONST \quad \{0, \ldots, 65535\}$$
$$r \to +(r, r) \quad \{-\infty, \ldots, \infty\}$$

(b) Instruction Selector

Instruction Selector Completeness (2)

After terminal splitting:

- Construct equivalent automata using dedicated terminal symbols representing the conditions and terminal symbols of the rules in the original grammars
- The alphabets of the automata are guaranteed compatible

$v \rightarrow INT_CONST$	$\{-\infty, \dots, -32769\}$
$v \rightarrow INT_CONST$	$\{-32768, \dots, -1\}$
$v \rightarrow INT_CONST$	$\{0, \dots, 32767\}$
$v \rightarrow INT_CONST$	$\{32768, \dots, 65535\}$
$v \rightarrow INT_CONST$	$\{65536, \dots, \infty\}$
$v \rightarrow +(v, v)$	$\{-\infty, \dots, \infty\}$

(a) Intermediate Representation

$r \rightarrow INT_CONST$	$\{-32768, \dots, -1\}$
$r \rightarrow INT_CONST$	$\{0, \dots, 32767\}$
$r \rightarrow INT_CONST$	$\{32768, \dots, 65535\}$
$r \rightarrow +(r, r)$	$\{-\infty, \dots, \infty\}$

(b) Instruction Selector

Instruction Selector Completeness (2)

After terminal splitting:

- Construct equivalent automata using dedicated terminal symbols representing the conditions and terminal symbols of the rules in the original grammars
- The alphabets of the automata are guaranteed compatible

$v \rightarrow INT_CONST_{\{-\infty,\ldots,-32769\}}$
$v \rightarrow INT_CONST_{\{-32768,\ldots,-1\}}$
$v \rightarrow INT_CONST_{\{0,\ldots,32767\}}$
$v \rightarrow INT_CONST_{\{32768,\ldots,65535\}}$
$v \rightarrow INT_CONST_{\{65536,\ldots,\infty\}}$
$v \rightarrow +(v, v)_{\{-\infty,\ldots,\infty\}}$

(a) Intermediate Representation

$r \rightarrow INT_CONST_{\{-32768,\ldots,-1\}}$
$r \rightarrow INT_CONST_{\{0,\ldots,32767\}}$
$r \rightarrow INT_CONST_{\{32768,\ldots,65535\}}$
$r \rightarrow +(r, r)_{\{-\infty,\ldots,\infty\}}$

(b) Instruction Selector

Evaluation

- Architecture models
  - Subset of MIPS-I (RISC), SPEAR2 (RISC), CHILI
  - Model statistics, relate to other processor description systems

Evaluation

- Architecture models
  - Subset of MIPS-I (RISC), SPEAR2 (RISC), CHILI
  - Model statistics, relate to other processor description systems
- Compiler generation
  - Retarget LLVM compiler framework for CHILI
  - Code size and performance measurements
  - Comparison with hand-crafted production compilers
  - Results obtained using validated simulators
  - Subset of the MiBench benchmark suite

The CHILI Processor

Configurable media processor

- 32-bit data path
- 64 general purpose registers
- 2-way or 4-way parallel VLIW
- 7-stage pipeline
  - Large number of branch delay slots (4 cycles)
  - Long load delay (5 cycles)
  - Identical parallel pipelines
- Instruction set
  - Almost all instructions can be predicated
  - Rich set of predicated instruction variants
  - Dedicated instructions for video en-/decoding

Processor Models

Model	LOC	Syntax LOC	Syntax #Tmpl.	Encoding LOC	Encoding #Tmpl.	Types LOC	Types #Ty.	Components LOC	Components #Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12
SPEAR	1298	109	5	223	12	733	13	172	14

Processor Models

Model	LOC	Syntax LOC	Syntax #Tmpl.	Encoding LOC	Encoding #Tmpl.	Types LOC	Types #Ty.	Components LOC	Components #Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12
SPEAR	1298	109	5	223	12	733	13	172	14

Model	Definitions #Uts.	Definitions #Ops.	Expanded #Uts.	Expanded #Ops.	Instruction Set #Paths	Instruction Set #Insts.
CHILI-v2	19	77	31	129	15	886
CHILI-v4	19	77	60	253	27	1672
MIPS	7	61	7	61	3	57
SPEAR	7	62	7	62	3	104

Processor Models

Model	LOC	Syntax LOC	Syntax #Tmpl.	Encoding LOC	Encoding #Tmpl.	Types LOC	Types #Ty.	Components LOC	Components #Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12
SPEAR	1298	109	5	223	12	733	13	172	14

Model	Definitions #Uts.	Definitions #Ops.	Expanded #Uts.	Expanded #Ops.	Instruction Set #Paths	Instruction Set #Insts.
CHILI-v2	19	77	31	129	15	886
CHILI-v4	19	77	60	253	27	1672
MIPS	7	61	7	61	3	57
SPEAR	7	62	7	62	3	104

The specifications are compact, due to the use of types. Furthermore, the number of instruction paths is relatively low in comparison to the number of actual instructions.

Processor Models

		Syntax		Encoding		Types		Components	
Model	LOC	LOC	#Tmpl.	LOC	#Tmpl.	LOC	#Ty.	LOC	#Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12
SPEAR	1298	109	5	223	12	733	13	172	14

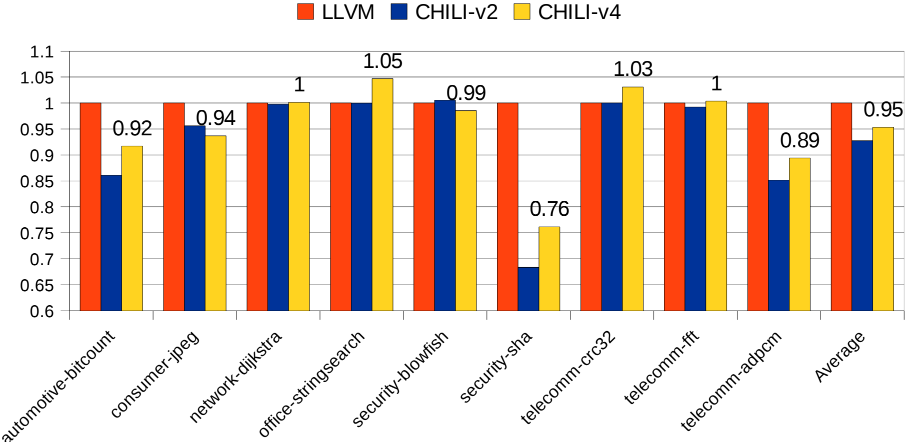
	Definitions		Expanded		Instruction Set	
Model	#Uts.	#Ops.	#Uts.	#Ops.	#Paths	#Insts.
CHILI-v2	[illegible]	77	31	129	15	88
CHILI-v4	[illegible]	77	60	253	27	1672
MIPS	7	61	7	61	3	57
SPEAR	7	62	7	62	3	104

		ISA		Behavior	Structure	Compiler	
Model	LOC	LOC	#Instrs	LOC	LOC	LOC	#Rules
R3000	2533	386	58	2121	–	–	–
acesMIPS	4184	828	85	–	533	2353	173

Low specification overhead compared to other processor description systems!

The specifications are compact, due to the use of types. Furthermore, the number of instruction paths is relatively low in comparison to the number of actual instructions.

LLVM Compiler Generator - Results for CHILI



Relative performance results of LLVM-based vs. xADL-generated LLVM compilers for two configurations of the CHILI VLIW.

LLVM Compiler Generator - Results for CHILI-v4

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	348,892	296,376	-15	881,144	1,183,104	+34
consumer-jpeg	2,341,904	1,241,408	-47	10,794,047	9,976,958	-8
network-dijkstra	485,560	470,744	-3	2,894,236	2,414,124	-17
office-stringsearch	334,004	303,384	-9	624,087	738,406	+18
security-blowfish	400,160	306,192	-23	1,541,883	1,491,519	-3
security-sha	351,860	327,608	-7	10,791,045	12,215,822	+13
telecomm-crc32	353,980	327,132	-8	8,637,327	9,520,911	+10
telecomm-fft	415,184	400,884	-3	187,968,275	188,243,462	+1
telecomm-adpcm	338,988	324,728	-4	10,116,131	9,755,433	-4

Runtime and code size results for GCC version 4.2.0 and the xADL-generated LLVM compiler for the four-way parallel CHILI.

Conclusion

- Structural xADL processor description language
  - Extensible types
  - Compact and intuitive specifications
  - Instruction set extraction along instruction paths
  - No redundant specification of behavior

Conclusion

- Structural xADL processor description language
  - Extensible types
  - Compact and intuitive specifications
  - Instruction set extraction along instruction paths
  - No redundant specification of behavior

- Generator tools
  - High-quality code generation
    - Competitive with production compilers
    - Slightly slower code by 5%
  - Automatic completeness test
    - Verifies completeness of the derived instruction selector
    - Provides valuable feedback using counter examples
  - In addition
    - High-speed simulation
    - ...