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

- Processor Model
- Documentation
- Compiler
- Simulator
- Assembler/Linker
- HDL Model
- Test Cases
- Encoding

Diagram showing the application of Processor Description Language.
Application of Processor Description Language

![Diagram showing the process of generating documentation, compiler, simulator, assembler/linker, HDL model, and test cases from a processor model.]

- Processor Model
  - Documentation
    - Compiler
  - Generator
    - Simulator
    - Assembler/Linker
  - Encoding
    - HDL Model
    - Test Cases
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, ...)

- 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
    - ...
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
  - ...
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
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:

- \{e_1, e_2, e_3, e_6, e_8, e_{10}\}
- \{e_1, e_2, e_4, e_7, e_9, e_{10}\}
- \{e_1, e_2, e_3, e_6, e_8, e_{11}\}
- \{e_1, e_2, e_4, e_7, e_9, e_{11}\}
- \{e_1, e_2, e_5\}
Example: MIPS Model

Instruction paths:

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

PC:

[Diagram of MIPS pipeline stages: fetch (FE), instruction code (IC), decode (DE), execute (EX), memory (MEM), writeback (WB).]

- FE: IC: DE: EX
- ...

ImmW, ImmJ, addr, load, sextb, zextb, !load, fwd, bypass.
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
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 \textit{acc}
- \textbf{Instruction selector}
  - Derive tree patterns from instruction set model
  - Extend coverage
  - Verify completeness
- \textbf{Instruction scheduler}
  - Instruction paths resemble possible execution flow
  - Correspond to resource tables for scheduling in LLVM
  - Additional information required for \textit{Operation Tables} in \textit{acc}
- \textbf{Register allocator}
  - Derive register classes from register files/ports
Instruction Selection using Tree Pattern Matching

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
<th>Emit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $r \rightarrow ar$</td>
<td>1</td>
<td>mov $r = ar$</td>
</tr>
<tr>
<td>(2) $ar \rightarrow r$</td>
<td>1</td>
<td>mov $ar = r$</td>
</tr>
<tr>
<td>(3) $r \rightarrow V$</td>
<td>0</td>
<td>V</td>
</tr>
<tr>
<td>(4) $imm \rightarrow C$</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>(5) $r \rightarrow imm$</td>
<td>1</td>
<td>ldi $r = imm$</td>
</tr>
<tr>
<td>(6) $r \rightarrow *(r_1, r_2)$</td>
<td>3</td>
<td>mul $r = r_1 * r_2$</td>
</tr>
<tr>
<td>(7) $r \rightarrow +(r_1, r_2)$</td>
<td>1</td>
<td>add $r = r_1 + r_2$</td>
</tr>
<tr>
<td>(8) $r \rightarrow +(r_1, imm)$</td>
<td>1</td>
<td>add $r = r_1 + imm$</td>
</tr>
<tr>
<td>(9) $r \rightarrow LD(+([ar_1, imm]))$</td>
<td>5</td>
<td>ld $r = [ar_1 + imm]$</td>
</tr>
</tbody>
</table>
Covering the Intermediate Representation

\( V_b \)

- \( \text{imm}: \infty \)
- \( r: 0 \quad (3) \)
- \( ar: 1 + 0 \quad (2\&3) \)

\( + \)

\( C_{60} \)

- \( \text{imm}: 0 \quad (4) \)
- \( r: 1 \quad (4\&5) \)
- \( ar: 1 + 1 \quad (2\&4\&5) \)

\( \ast \)

\( * \)

\( \text{imm}: \infty \)
- \( r: 3 + 0 + 6 \quad (6) \)
- \( ar: 1 + 9 \quad (2\&6) \)

\( V_a \)

- \( \text{imm}: \infty \)
- \( r: 0 \quad (3) \)
- \( ar: 1 + 0 \quad (2\&3) \)

\( LD \)

- \( \text{imm}: \infty \)
- \( r: 1 + 0 + 0 \quad (8) \)
- \( ar: 1 + 1 \quad (2\&8) \)

\( \text{imm}: 0 \quad (4) \)
- \( r: 1 \quad (4\&5) \)
- \( ar: 1 + 1 \quad (2\&4\&5) \)

\( \text{imm}: 0 \quad (4) \)
- \( r: 1 \quad (4\&5) \)
- \( ar: 1 + 1 \quad (2\&4\&5) \)
Covering the Intermediate Representation

\[ \text{imm: } \infty \]
\[ r: 0 \quad (3) \]
\[ ar: 1 + 0 \quad (2&3) \]

\[ V_b \]

\[ \text{imm: } \infty \]
\[ r: 1 + 0 + 0 \quad (8) \]
\[ ar: 1 + 1 \quad (2&8) \]

\[ C_{60} \]

\[ \text{imm: } \infty \]
\[ r: 5 + 1 + 0 \quad (9) \]
\[ ar: 1 + 6 \quad (2&9) \]

\[ LD \]

\[ \text{imm: } \infty \]
\[ r: 3 + 0 + 6 \quad (6) \]
\[ ar: 1 + 9 \quad (2&6) \]

\[ V_a \]

\[ \text{imm: } \infty \]
\[ r: 0 \quad (3) \]
\[ ar: 1 + 0 \quad (2&3) \]

\[ * \]
Covering the Intermediate Representation

<table>
<thead>
<tr>
<th>Code</th>
<th>Rule Number</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>(2)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>(3)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(4)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>(5)</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>(6)</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

- (1) mov $ar_1 = b$
- (2) $ar = 1 + 0$
- (3) $imm: \infty$
- (4) ld $r_1 = [ar_1 + 60]$
- (5) mul $r_2 = a \times r_1$
- (6) $r: 3 + 0 + 6$
- (7) $ar: 1 + 9$
- (8) $imm: \infty$
- (9) $r: 5 + 1 + 0$
- (10) $ar: 1 + 6$

12/27
Cost Functions and Dynamic Checks

Dynamic cost functions

- The cost of covering a tree fragment can be computed \textit{dynamically} at compile time
- May inspect compiler options, the compilation context, or the covered tree fragment
- Usually, specified using \texttt{code}, and is thus hardly analyzable
Cost Functions and Dynamic Checks

Dynamic cost functions

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

• What happens when a cost function returns infinity?
  • The rule is effectively \texttt{disabled}
  • We call such cost functions \texttt{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.

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

Check emptiness of $L_{ir} \cap L_{is}$

Problem: Dynamic checks cannot be modeled!
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
  \[\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{\text{char, short, int}\}\]

- Term with properties
  \[
  \text{CST}(\{60\} \times \{GPR\} \times \{\text{int}\})
  \]

- Rules with conditions
  - Unconditional:
    \[r \rightarrow \text{CST}(\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{\text{char, short, int}\})\]
  - Unsatisfiable:
    \[r \rightarrow \text{CST}(\{-\infty, \ldots, \infty\} \times \{AR, GPR\} \times \{\}\})\]
  - Conditional:
    \[r \rightarrow \text{CST}(\{0, \ldots, 65535\} \times \{GPR\} \times \{\text{short}\})\]
Representing Emit Functions

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

\textbf{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.
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, ...

<table>
<thead>
<tr>
<th>µ-op.</th>
<th>Tree pattern</th>
<th>Operand Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6)</td>
<td>RC_R_32 → OR(RC_R_32, ZEXT_{16}(immediate))</td>
<td>(ImmW, 21)</td>
</tr>
<tr>
<td>(7)</td>
<td>RC_R_32 → OR(RC_R_32, ZEXT_{16}(_))</td>
<td>(Rs, 1)</td>
</tr>
<tr>
<td>(11)</td>
<td>RC_R_32 → OR(_, ZEXT_{16}(_))</td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>RC_R_32 → OR(_, _)</td>
<td></td>
</tr>
<tr>
<td>(19)</td>
<td>RC_R_32 → _</td>
<td>(Rd, new_reg)</td>
</tr>
</tbody>
</table>
Extending the Coverage

The computed rule set is typically not complete

- Extend the coverage using \textit{specialization} and \textit{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

<table>
<thead>
<tr>
<th>Tree Pattern</th>
<th>Operand Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) RC.R_32 \rightarrow OR(RC.R_32, immediate{0,\ldots,65535})</td>
<td>(Rs, 1), (ImmW, 2)</td>
</tr>
<tr>
<td>(2) RC.R_32 \rightarrow OR(immediate{0,\ldots,65535}, RC.R)</td>
<td>(Rs, 2), (ImmW, 1)</td>
</tr>
<tr>
<td>(3) RC.R_32 \rightarrow immediate{0,\ldots,65535}</td>
<td>(Rs, 0), (ImmW, \epsilon)</td>
</tr>
<tr>
<td>(4) RC.R_32 \rightarrow RC.R_32</td>
<td>(Rs, \epsilon), (ImmW, 0)</td>
</tr>
</tbody>
</table>
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 specialization.
  - **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.

<table>
<thead>
<tr>
<th>Tree Pattern</th>
<th>Operand Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) RC.R.32 → ( OR(\text{RC.R.32, immediate{0, \ldots, 65535}}) )</td>
<td>((Rs, 1), ((ImmW, 2))</td>
</tr>
<tr>
<td>(2) RC.R.32 → ( OR(\text{immediate{0, \ldots, 65535}, RC.R}) )</td>
<td>((Rs, 2), ((ImmW, 1))</td>
</tr>
<tr>
<td>(3) RC.R.32 → ( \text{immediate{0, \ldots, 65535}} )</td>
<td>((Rs, 0), ((ImmW, \epsilon))</td>
</tr>
<tr>
<td>(4) RC.R.32 → ( \text{RC.R.32} )</td>
<td>((Rs, \epsilon), ((ImmW, 0))</td>
</tr>
</tbody>
</table>
Instruction Selector Completeness

Based on tree automata theory:

• Express both the instruction selector \((G_{is})\) and intermediate representation \((G_{ir})\) using \textit{normalized} tree grammars with conditions

• Transform these grammars into \textit{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 $\text{term}(r_1) = \text{term}(r_2)$ the following condition holds:

$$\text{cond}(r_1) = \text{cond}(r_2) \lor \neg \text{overlap} (\text{cond}(r_1), \text{cond}(r_2)).$$

$\text{cond}(r) \rightarrow D \ldots$ condition of rule $r$ \hspace{1cm} $\text{term}(r) \rightarrow F \ldots$ 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.

\[
\begin{align*}
    v & \rightarrow \text{INT}_\text{CONST} \quad \{-\infty, \ldots, \infty\} \\
    v & \rightarrow +(v, v) \quad \{-\infty, \ldots, \infty\} \\
    r & \rightarrow \text{INT}_\text{CONST} \quad \{-32768, \ldots, 32767\} \\
    r & \rightarrow \text{INT}_\text{CONST} \quad \{0, \ldots, 65535\} \\
    r & \rightarrow +(r, r) \quad \{-\infty, \ldots, \infty\}
\end{align*}
\]

(a) Intermediate Representation

(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

\[
\begin{align*}
    v & \rightarrow INT\_CONST \quad \{-\infty, \ldots, -32769\} \\
    v & \rightarrow INT\_CONST \quad \{-32768, \ldots, -1\} \\
    v & \rightarrow INT\_CONST \quad \{0, \ldots, 32767\} \\
    v & \rightarrow INT\_CONST \quad \{32768, \ldots, 65535\} \\
    v & \rightarrow INT\_CONST \quad \{65536, \ldots, \infty\} \\
    v & \rightarrow +(v, v) \quad \{-\infty, \ldots, \infty\}
\end{align*}
\]

(a) Intermediate Representation

\[
\begin{align*}
    r & \rightarrow INT\_CONST \quad \{-32768, \ldots, -1\} \\
    r & \rightarrow INT\_CONST \quad \{0, \ldots, 32767\} \\
    r & \rightarrow INT\_CONST \quad \{32768, \ldots, 65535\} \\
    r & \rightarrow +(r, r) \quad \{-\infty, \ldots, \infty\}
\end{align*}
\]

(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

<table>
<thead>
<tr>
<th>Model</th>
<th>LOC</th>
<th>LOC</th>
<th>#Tpl.</th>
<th>LOC</th>
<th>#Tpl.</th>
<th>LOC</th>
<th>#Ty.</th>
<th>LOC</th>
<th>#Ists.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILI-v2</td>
<td>1580</td>
<td>191</td>
<td>12</td>
<td>141</td>
<td>6</td>
<td>800</td>
<td>20</td>
<td>350</td>
<td>14</td>
</tr>
<tr>
<td>CHILI-v4</td>
<td>1739</td>
<td>220</td>
<td>12</td>
<td>156</td>
<td>6</td>
<td>830</td>
<td>20</td>
<td>454</td>
<td>24</td>
</tr>
<tr>
<td>MIPS</td>
<td>1143</td>
<td>183</td>
<td>14</td>
<td>134</td>
<td>9</td>
<td>592</td>
<td>14</td>
<td>157</td>
<td>12</td>
</tr>
<tr>
<td>SPEAR</td>
<td>1298</td>
<td>109</td>
<td>5</td>
<td>223</td>
<td>12</td>
<td>733</td>
<td>13</td>
<td>172</td>
<td>14</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model</th>
<th>LOC</th>
<th>LOC</th>
<th>#Tmpl.</th>
<th>LOC</th>
<th>#Tmpl.</th>
<th>LOC</th>
<th>#Ty.</th>
<th>LOC</th>
<th>#Ists.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILI-v2</td>
<td>1580</td>
<td>191</td>
<td>12</td>
<td>141</td>
<td>6</td>
<td>800</td>
<td>20</td>
<td>350</td>
<td>14</td>
</tr>
<tr>
<td>CHILI-v4</td>
<td>1739</td>
<td>220</td>
<td>12</td>
<td>156</td>
<td>6</td>
<td>830</td>
<td>20</td>
<td>454</td>
<td>24</td>
</tr>
<tr>
<td>MIPS</td>
<td>1143</td>
<td>183</td>
<td>14</td>
<td>134</td>
<td>9</td>
<td>592</td>
<td>14</td>
<td>157</td>
<td>12</td>
</tr>
<tr>
<td>SPEAR</td>
<td>1298</td>
<td>109</td>
<td>5</td>
<td>223</td>
<td>12</td>
<td>733</td>
<td>13</td>
<td>172</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>#Uts.</th>
<th>#Ops.</th>
<th>#Uts.</th>
<th>#Ops.</th>
<th>#Paths</th>
<th>#Insts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILI-v2</td>
<td>19</td>
<td>77</td>
<td>31</td>
<td>129</td>
<td>15</td>
<td>886</td>
</tr>
<tr>
<td>CHILI-v4</td>
<td>19</td>
<td>77</td>
<td>60</td>
<td>253</td>
<td>27</td>
<td>1672</td>
</tr>
<tr>
<td>MIPS</td>
<td>7</td>
<td>61</td>
<td>7</td>
<td>61</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>SPEAR</td>
<td>7</td>
<td>62</td>
<td>7</td>
<td>62</td>
<td>3</td>
<td>104</td>
</tr>
</tbody>
</table>
Processor Models

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

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

<table>
<thead>
<tr>
<th>Model</th>
<th>LOC</th>
<th>LOC</th>
<th>#Tmpl.</th>
<th>LOC</th>
<th>#Tmpl.</th>
<th>LOC</th>
<th>#Ty.</th>
<th>LOC</th>
<th>#Ists.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILI-v2</td>
<td>1580</td>
<td>191</td>
<td>12</td>
<td>141</td>
<td>6</td>
<td>800</td>
<td>20</td>
<td>350</td>
<td>14</td>
</tr>
<tr>
<td>CHILI-v4</td>
<td>1739</td>
<td>220</td>
<td>12</td>
<td>156</td>
<td>6</td>
<td>830</td>
<td>20</td>
<td>454</td>
<td>24</td>
</tr>
<tr>
<td>MIPS</td>
<td>1143</td>
<td>183</td>
<td>14</td>
<td>134</td>
<td>9</td>
<td>592</td>
<td>14</td>
<td>157</td>
<td>12</td>
</tr>
<tr>
<td>SPEAR</td>
<td>1298</td>
<td>109</td>
<td>5</td>
<td>223</td>
<td>12</td>
<td>733</td>
<td>13</td>
<td>172</td>
<td>14</td>
</tr>
</tbody>
</table>

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.
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

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

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