Worst-Case Execution Time Analysis of Predicated Architectures

Florian Brandner
LTCI
Telecom ParisTech

Amine Naji
U2IS
ENSTA ParisTech

This work is supported by the Digiteo project PM-TOP.

What is Predication?

Predication:

- Combination of architectural and compilation techniques.
- Converts control dependencies to data dependencies.
- Conditional execution based on a guard bit called Predicate.
- It allows the compiler to eliminate branches (and their side effects).

What is Predication?

Predication:

- Combination of architectural and compilation techniques.
- Converts control dependencies to data dependencies.
- Conditional execution based on a guard bit called Predicate.
- It allows the compiler to eliminate branches (and their side effects).

Branch side effects:

- High penalties (pipelines depth).
  - Branch prediction: Conflicts and miss-prediction.
  - Branch delay-slots: Explicit Nops and code size increase.
- Limited Instruction-Level Parallelism (ILP).
  - Mostly a problem for VLIWs.
  - Cannot bundle instructions before and after branches.

Patmos Architecture

Overview:

- Dual-issue VLIW.
- Fully predicated.
  - All instructions can be predicated.
  - 8 predicate registers (`p0`, ..., `p7`)
  - `p0` is always true.
  - Predicates can be inverted (`!p0`).
- Branch variants:
  - Non-delayed: 2 or 3 cycles penalty.
  - Delayed: Execute 2 or 3 bundles in branch delay slots.

Example: if-conversion

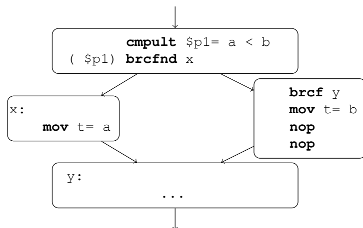
```c
int foo(int a, int b) {
  return t = a < b ? a : b;
}
```

Function in C code

Example: if-conversion

```
int foo(int a, int b) {
  return t = a < b ? a : b;
}
```

Function in C code

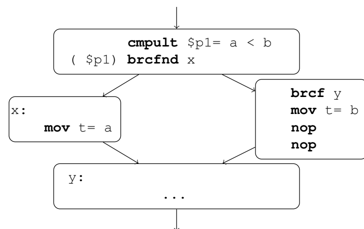


Control-flow graph with branches

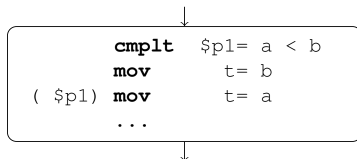
Example: if-conversion

```
int foo(int a, int b) {
  return t = a < b ? a : b;
}
```

Function in C code



Control-flow graph with branches



Control-flow with predicates

Predication in Real-Time Systems

Some benefits:

- Eliminates branch penalties:
  - Simpler analysis (eliminates branches).
  - Fewer conflicts between branches, since fewer branches.
- Better cache locality due to fewer control-flow transfers.
- This promises more predictable code.
- Single-Path Programming:
  - Extreme approach: Remove branches almost completly.
  - Goal: Eliminate timing variations.

Predication in Real-Time Systems

Some benefits:

- Eliminates branch penalties:
  - Simpler analysis (eliminates branches).
  - Fewer conflicts between branches, since fewer branches.
- Better cache locality due to fewer control-flow transfers.
- This promises more predictable code.
- Single-Path Programming:
  - Extreme approach: Remove branches almost completly.
  - Goal: Eliminate timing variations.

Predicated instructions have to be analyzed.

Timing Analysis With Predicates

Challenges:

- The execution of instructions depends on the predicate register value.
  - Predicate register values are needed to build program's CFG.
  - Program's CFG is needed to analyze predicate registers.
- Handling of nested branches in branch delay slots.
- All underlying analyses have to be aware of predicates.
- Expressing flow constraints on predicated code.

Timing Analysis With Predicates

Challenges:

- The execution of instructions depends on the predicate register value.
  - Predicate register values are needed to build program's CFG.
  - Program's CFG is needed to analyze predicate registers.
- Handling of nested branches in branch delay slots.
- All underlying analyses have to be aware of predicates.
- Expressing flow constraints on predicated code.

Simple solution:

- Consider predicate to be true and false for each instruction.
- Conservatively perform join after each instruction.

Timing Analysis With Predicates

Challenges:

- The execution of instructions depends on the predicate register value.
  - Predicate register values are needed to build program's CFG.
  - Program's CFG is needed to analyze predicate registers.
- Handling of nested branches in branch delay slots.
- All underlying analyses have to be aware of predicates.
- Expressing flow constraints on predicated code.

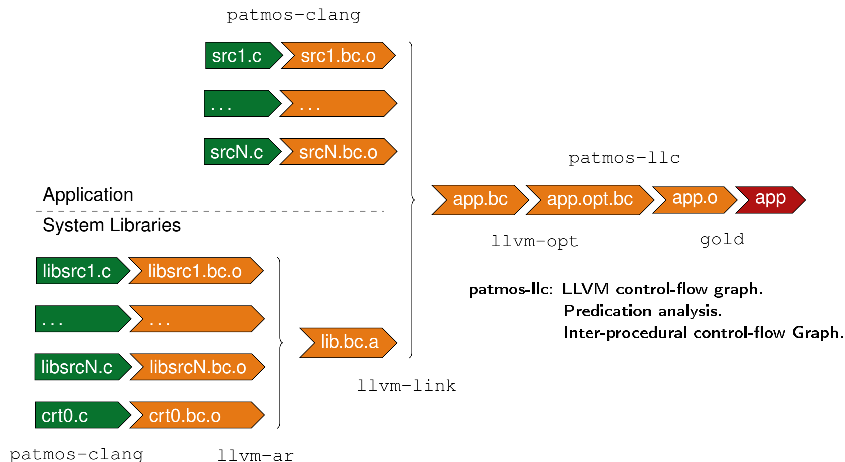
Simple solution:

- Consider predicate to be true and false for each instruction.
- Conservatively perform join after each instruction.

Our approach:

Recover control-flow from predicated code through unfolding.

Compilation/Analysis Flow

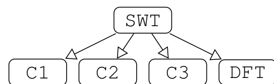


The Patmos toolchain (LLVM Compiler).

Motivating Example: Load Side-Effects

```
switch(x) {
  case  0: ... break;
  case  1: ... break;
  case  2: ... break;
  default: ... break;
}
```

C code.



LLVM control-flow graph.

```
        cmpult $p1=x, 3
( $p1)  shl    $r1=x, 2
(!$p1)  brcf   DFT
( $p1)  lwc    $r1=[$r1+jt]
        nop
( $p1)  brcfnd $r1
```

Assembly code of `SWT` block using jump table.

High-level Overview

Construct an inter-procedural CFG (iCFG):

- Split LLVM's basic blocks.
    - Proceeds in two phases (next slide).
    - According to branches and predicate definitions.
    - Considering branch delay slots.
- iCFG nodes:
    - Wrapper around LLVM's basic blocks.
    - Associated with a set of predicates known to be true.
    - All instruction become unconditional (predicates are removed).
    - Replace nullified instructions by `nop`.

Algorithm: UnFold Basic Block

Find the split point.

1. Scan instructions in LLVM's basic block and check for splits:
   - Track live predicates.
   - Case 1: The instruction defines a predicate:
     - Track the predicate.
     - Split the control-flow immediately.
   - Case 2: The instruction is a branch.
     - Track branch delay slots.
     - Track successors.
   - Split the control-flow after branch delay slots.

Algorithm: UnFold Basic Block

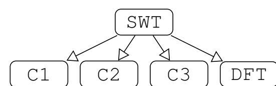
Find the split point.

1. Scan instructions in LLVM's basic block and check for splits:
    - Track live predicates.
    - Case 1: The instruction defines a predicate:
        - Track the predicate.
        - Split the control-flow immediately.
    - Case 2: The instruction is a branch.
        - Track branch delay slots.
        - Track successors.
    - Split the control-flow after branch delay slots.

At this point the control-flow is split, now we build the iCFG.

2. Build the inter-procedural control-flow graph:
    - Create iCFG Node.
        - Attach live predicates that are known to be true
    - Recursively scan branch targets (from phase 1.)
    - Create iCFG edges to branch targets

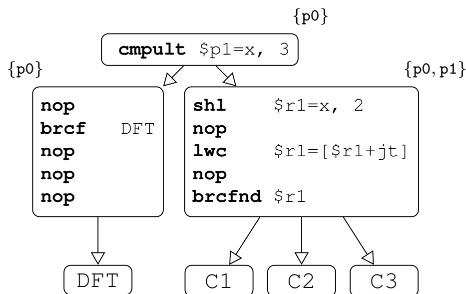
Motivating Example: Unfolded Control-Flow Graph



LLVM's CFG.

```
        cmpult $p1=x, 3
( $p1)  shl    $r1=x, 2
(!$p1)  brcf   DFT
( $p1)  lwc    $r1=[$r1+jt]
        nop
( $p1)  brcfnd $r1
```

Assembly code of SWT.



Unfolded iCFG.

Algorithm

Extensions:

- The support of multi-issue execution (VLIW processors ex. Patmos).
- The implementation handles calls and returns.
- Handling of branches nested in branch delay slots.
  - Support for non-disjoint predicates requires stack.

Algorithm

Extensions:

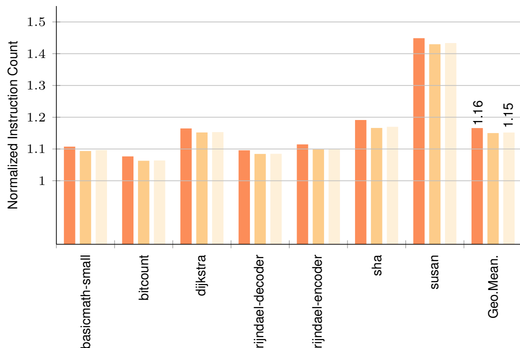
- The support of multi-issue execution (VLIW processors ex. Patmos).
- The implementation handles calls and returns.
- Handling of branches nested in branch delay slots.
  - Support for non-disjoint predicates requires stack.

Linear Complexity:

- The algorithm performs depth-first search on CFG.
- Every instruction is processed once for every set of potentially active predicates (up to 2^7).

Experiments: Increase in the number of instructions

Setup: Subset of TACLe benchmarks. LLVM compiler 3.5. Optimizations(-O2)

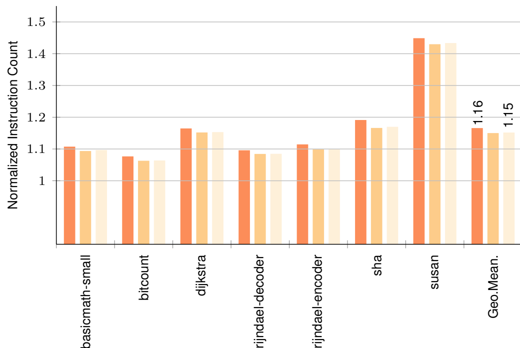


Increase in the number of instructions due to unfolding for the delayed (■), mixed (■), and non-delayed (■) configurations with VLIW instruction bundles, normalized to the size of LLVM's original CFG (lower is better).

- Usually low overhead induced by unfolding (between 10% and 20%).
- Susan benchmark shows higher increase (between 43% and 45%.)

Experiments: Increase in the number of instructions

Setup: Subset of TACLe benchmarks. LLVM compiler 3.5. Optimizations(-O2)



Increase in the number of instructions due to unfolding for the delayed ( ), mixed ( ), and non-delayed ( ) configurations with VLIW instruction bundles, normalized to the size of LLVM's original CFG (lower is better).

- Usually low overhead induced by unfolding (between 10% and 20%).
- Susan benchmark shows higher increase (between 43% and 45%.)
- There is no size explosion in the unfolded iCFG.

Conclusion

In this work:

- Lightweight approach to handle predicated code in WCET analysis.
  - Predicate definitions immediately lead to a control-flow split.
  - Subsequent instructions are analyzed depending on predicate value.
  - Control-flow dependencies are recovered and explicitly represented in the unfolded iCFG.
  - All instruction in iCFG are unconditional.
- Preliminary experiments show only a moderate iCFG size overhead.

Thanks for your attention

Any Questions ?