

Reconstructing Control Flow from Predicated Assembly Code

Björn Decker, Saarland University

Daniel Kästner, AbsInt GmbH

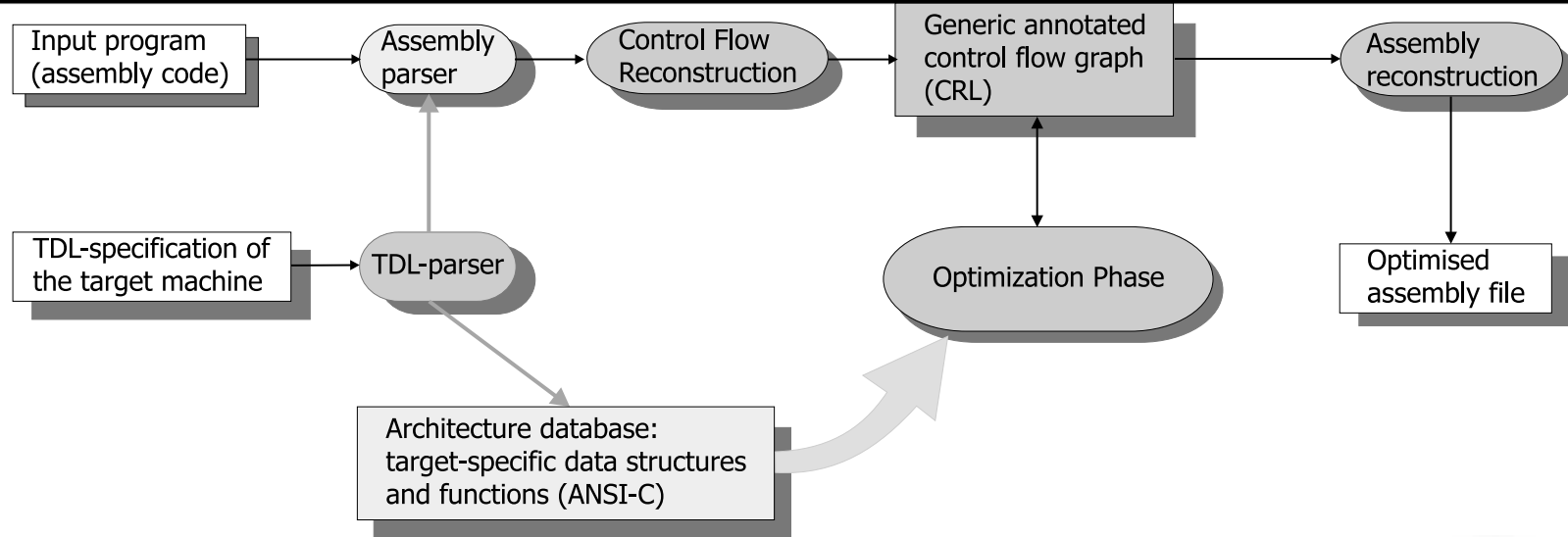


Motivation

- Many contemporary microprocessors use **instruction-level parallelism** to achieve high performance.
- **Predicated instructions** provide better performance due to the elimination of branches and better utilization of hardware resources: the issue slots of long instruction words can be filled with (sub-) operations from **different** control paths.
- However: predicated instructions make **postpass optimizations** more difficult, since the control dependences have been transformed to data dependences.
- **Goal**: Precise reconstruction of control flow from assembly / executable files for processors with predicated instructions in a **retargetable** way.



The PROPAN System



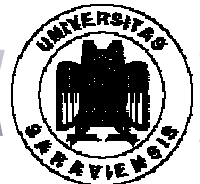
Advantage of Postpass Approach

- **Easy integration** into existing tool chains.
- Appropriate format for doing **processor-specific optimizations**. This is especially important for processors with **irregular hardware architectures**, a feature typical for embedded processors and DSPs.
- **Enhanced optimization potential** compared to standard compiler techniques:
 - cross-file optimizations
 - optimizations across inline assembly



Control Flow Reconstruction

- Many postpass optimizations requires the **control flow graph** of the input program to be **known**. Examples: transformations based on dataflow analysis like postpass instruction scheduling, register renaming, ...
- In order to enable high quality optimizations the CFG has to be very **precise**.
- Control flow must be reconstructed from the assembly code:
 - Phase 1: **Explicit** control flow reconstruction: computing the call graph, determining targets of direct and indirect jumps. In our framework based on extended program slicing of [Kästner, Wilhelm:LCTES02].
 - Phase 2: **Implicit** control flow reconstruction: [This article](#).



Control Flow Reconstruction

- This control flow graph has to be **safe**: all control paths of the input program) must be represented in the reconstructed graph.
- Due to information not statically computable, the reconstructed control flow graph may contain too many control flow edges: **conservative approximation**. (If the target of a branch is unknown, edges to all potential targets are inserted.)
- However, the reconstructed graph **should** be **as precise as possible**, i.e. the number of control paths that actually cannot occur in the input program should be minimized.



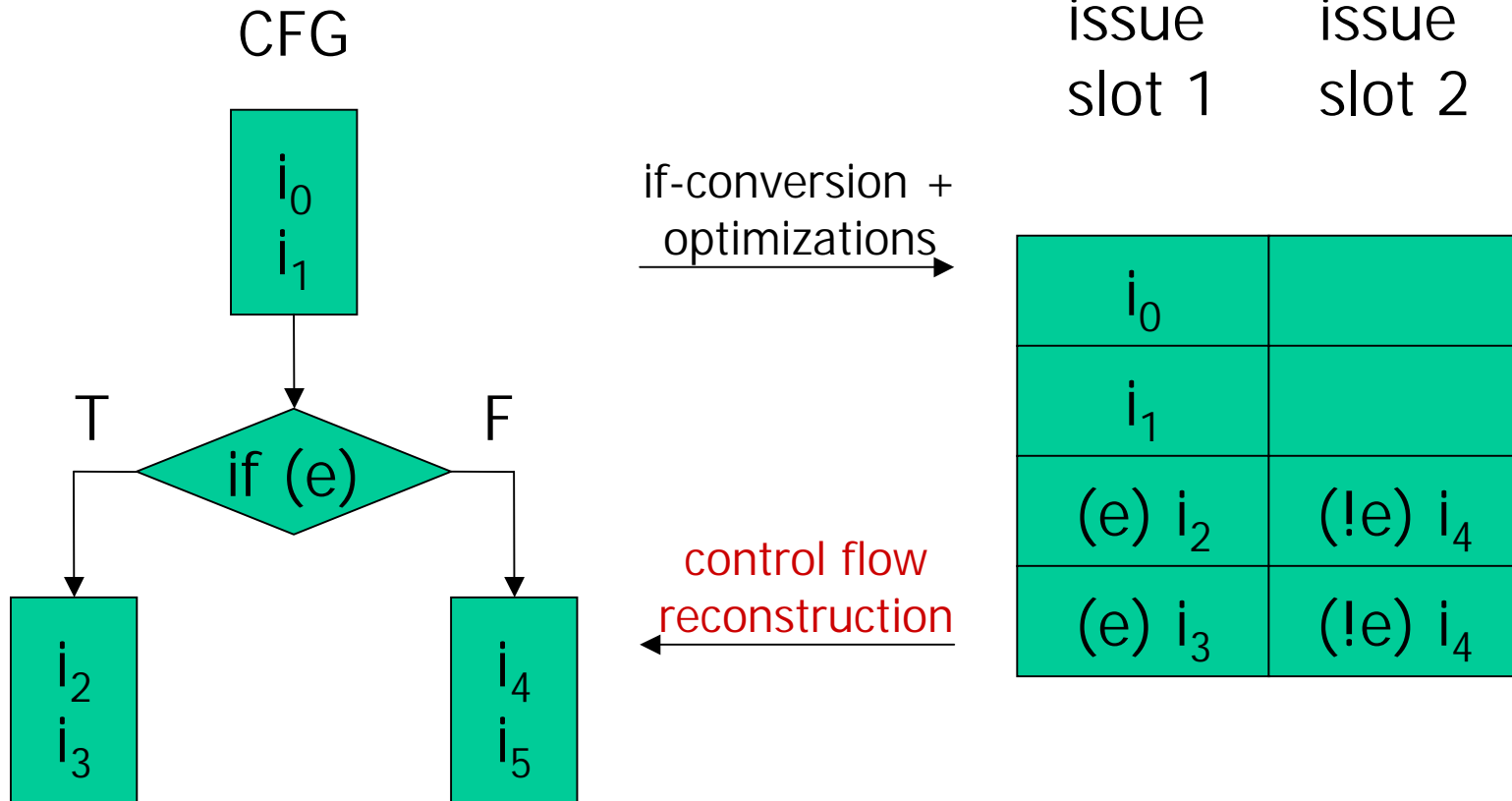
Predicated Instructions

Guarded (predicated) Code:

- Each assembly operation is associated with a guard that determines whether the operation is executed or not.
- Example: `IF r39 iaddi(0x4) r5 -> r34`
Adds the immediate value `0x4` to register `r5` and stores results in `r34`, but **only if** register `r39` evaluates to **TRUE**, otherwise, a nop is executed.
- Advantages:
 - Improved code density by enabling to fill more issue slots of the same instruction.
 - Reduced number of conditional branch operations.



Predicated Instructions



Precision of Control Flow Reconstruction for Predicated Code

- Consider two successive long instructions:
 - (i1) IF r39 iaddi(0x4) r5 -> r34;
 - (i2) IF !r39 iaddi(0x4) r34 -> r37;
- If the predicates are ignored:
 - A data dependence between i1 and i2 wrt r34 has to be assumed: i1 and i2 cannot be parallelized.
 - Assume r5=2, r34=7, r39=1, r37=9 immediately before i1. After i2, constant propagation yields r34=unknown, r37=unknown.
- If the implicit control flow is reconstructed:
 - The conditions r39 and !r39 are disjoint.
 - No data dependence between i1 and i2.
 - Assume r5=2, r34=7, r39=1, r37=9 immediately before i1. After i2, constant propagation yields r34=6, r37=9.



Reconstructing Explicit Control Flow

- Input: Assembly code
- Program slicing and value analysis are used to
 - reconstruct procedures
 - reconstruct intraprocedural control flow via call, return, jump and branch operations
- Output: roughly reconstructed CFG representing procedures and **explicit** control flow



Reconstructing Explicit Control Flow

1. For each **jump**, **call**, and **branch** operation assembly **slices** are computed containing exactly those operations influencing the target operand of the jump operation.
2. Assembly slices are evaluated in an **abstract** manner yielding an abstract value of the target address.
3. Abstract values of address targets represent sets of addresses of **possible successor** operations. Thus, edges in the CFG are introduced from the jump operation to **all** operations residing at addresses of possible successor operations.

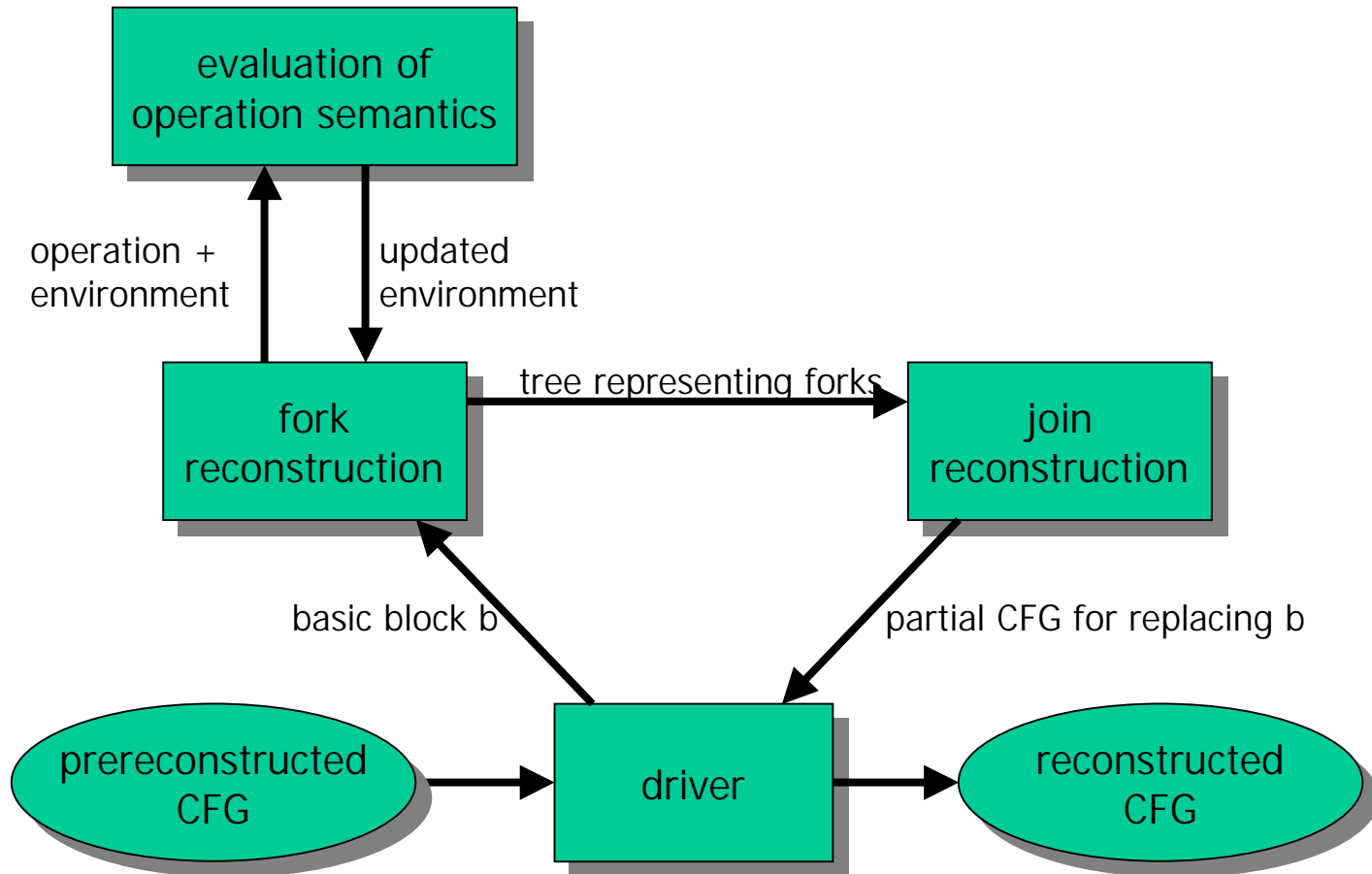


Reconstructing **Implicit** Control Flow

- Input: Assembly code of basic blocks in prereconstructed CFG.
- Examining boolean relations between guard registers.
- Refining control flow graph by arranging operations according to the relation of their guard registers.



Reconstructing Implicit Control Flow



Fork Reconstruction (Input)

- Input: basic block.
- From now on: TriMedia TM1000 as example processor.
- Instructions have five issue slots filled with so-called operations.
- Registers r1 and r0 are hardwired to 1 resp. 0.
- Processor implements the least-significant-bit truth-value representation, i.e. the least significant bit of register contents indicate whether it is interpreted as true or false.

```
(r1) r9 := r8 > r0  
(r1) r6 := r8 <= r0  
(r1) r7 := r1 + r0  
(r1) nop  
(r1) nop
```

```
(r6) r8 := r7 + r0  
(r9) r8 := r7 + r0  
(r1) nop  
(r1) nop  
(r1) nop
```

```
(r8) r5 := r0 + r1  
(r1) nop  
(r1) nop  
(r1) nop  
(r1) nop
```



Fork Reconstruction

- During fork reconstruction a **block tree** is created representing forks of the control flow of the input block.
- Successively arrange instructions in leaf blocks of the tree:
 - Examine whether each guard of the instruction uniformly evaluates to true or false in a certain leaf block.
 - Whenever a guard register does not uniformly evaluate: introduce two new successors for this block and restrict their environments. In one of them the violating guard register has to evaluate to true; in the other it must be false. Then the new blocks are considered for instruction arrangement.
 - Otherwise, the instruction is placed into the block. Operations whose guard evaluates to false are replaced by nop-operations.



Fork Reconstruction Example (1)

Input block

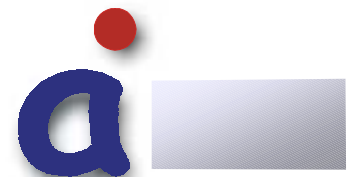
```
(r1) r9 := r8 > r0  
(r1) r6 := r8 <= r0  
(r1) r7 := r1 + r0  
(r1) nop  
(r1) nop
```

```
(r6) r8 := r7 + r0  
(r9) r8 := r7 + r0  
(r1) nop  
(r1) nop  
(r1) nop
```

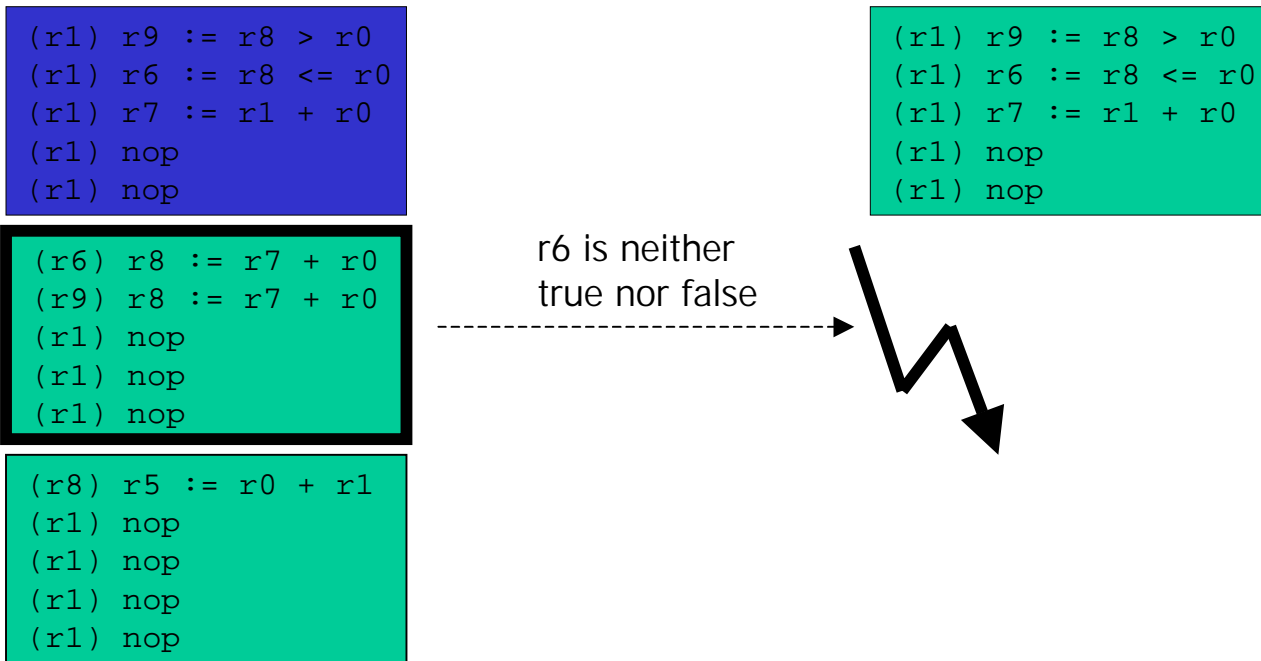
```
(r8) r5 := r0 + r1  
(r1) nop  
(r1) nop  
(r1) nop  
(r1) nop
```

Block tree

```
(r1) r9 := r8 > r0  
(r1) r6 := r8 <= r0  
(r1) r7 := r1 + r0  
(r1) nop  
(r1) nop
```



Fork Reconstruction Example (2)



Fork Reconstruction Example (3)

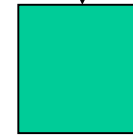
```
(r1) r9 := r8 > r0  
(r1) r6 := r8 <= r0  
(r1) r7 := r1 + r0  
(r1) nop  
(r1) nop
```

```
(r6) r8 := r7 + r0  
(r9) r8 := r7 + r0  
(r1) nop  
(r1) nop  
(r1) nop
```

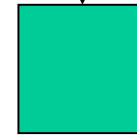
```
(r8) r5 := r0 + r1  
(r1) nop  
(r1) nop  
(r1) nop  
(r1) nop
```

```
(r1) r9 := r8 > r0  
(r1) r6 := r8 <= r0  
(r1) r7 := r1 + r0  
(r1) nop  
(r1) nop
```

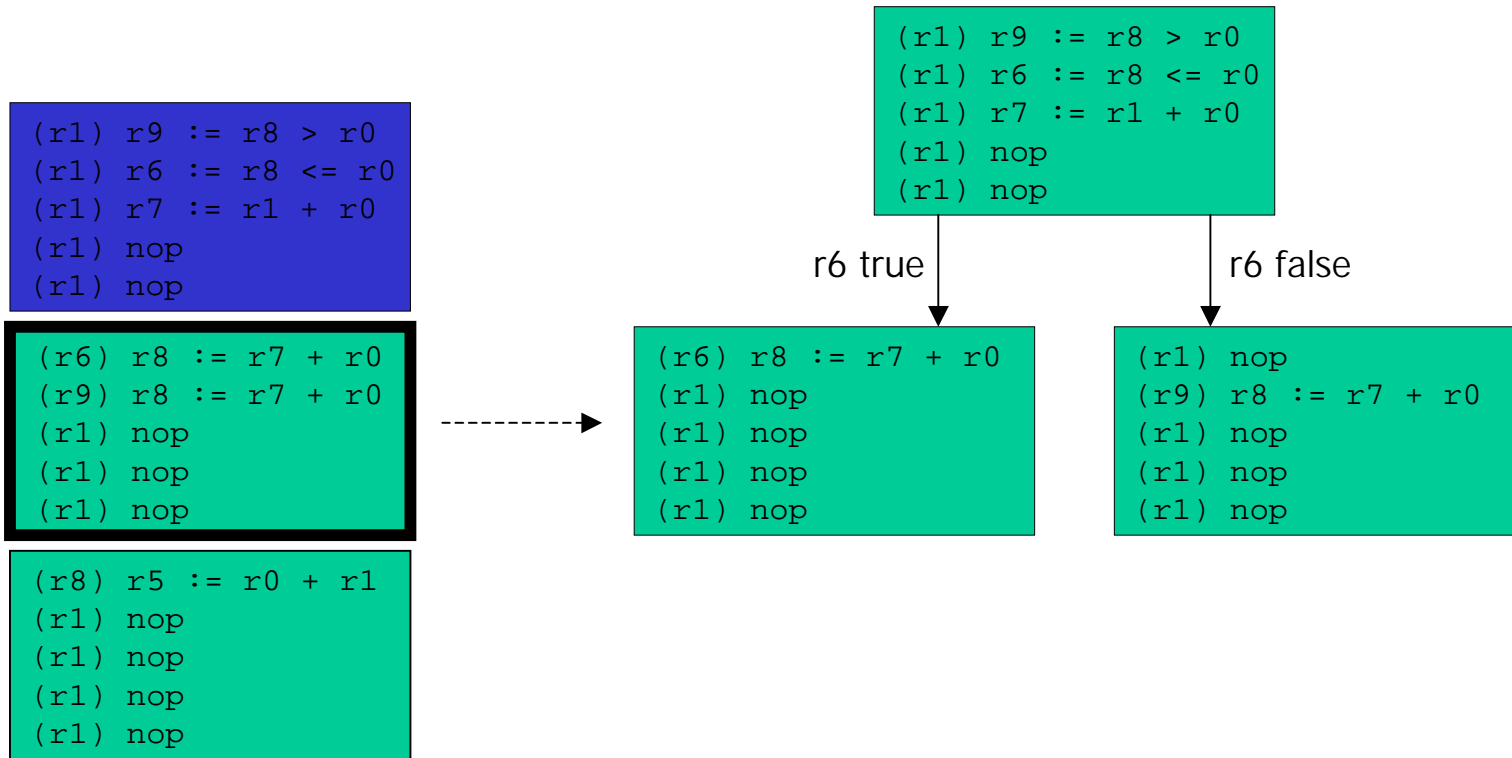
r6 true



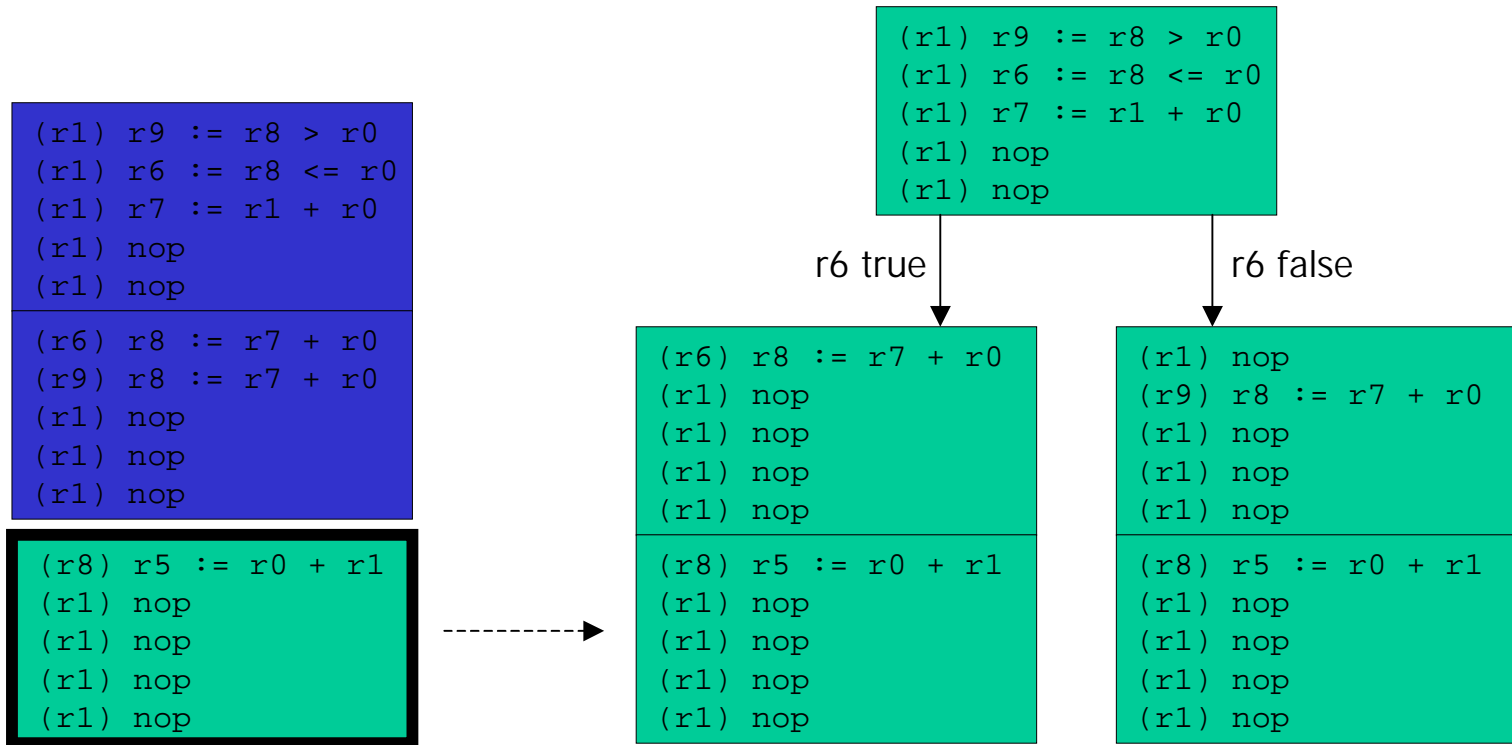
r6 false



Fork Reconstruction Example (4)



Fork Reconstruction Example (5)

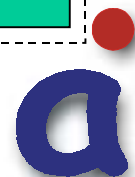
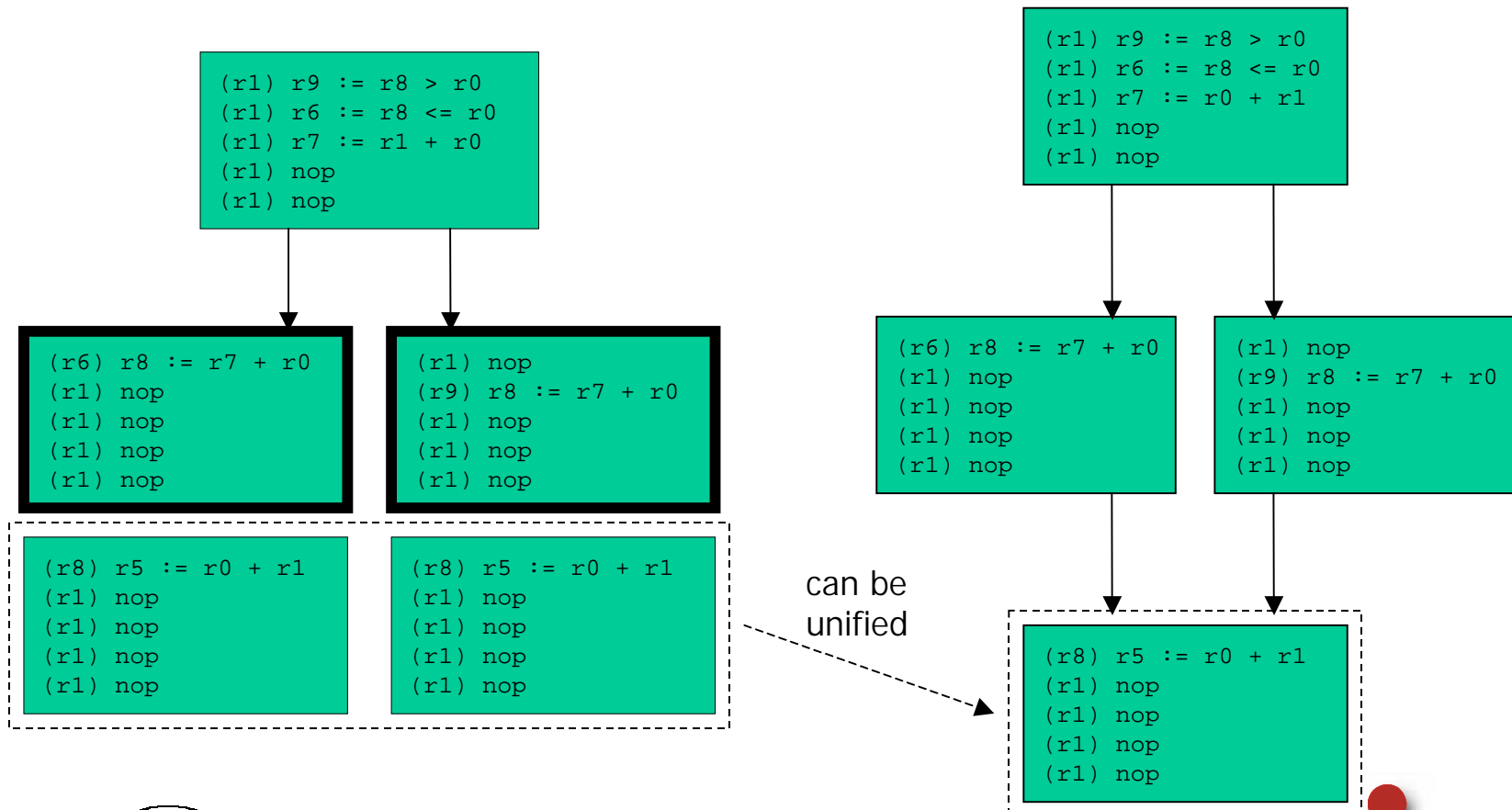


Join Reconstruction

- A join of control flow after two instructions exists iff they are indistinguishable with regard on leaving control flow paths.
- The following algorithm is used to recognize control flow joins in the result of the fork reconstruction phase:
 - For every pair of instruction instances (instructions in the tree that are created from the same instruction of the input block), determine whether the sets of paths reaching instances of the last instruction are equivalent.
 - Sets of paths A, B are equivalent iff for each path in A there is a path in B that contains equivalent instruction instances and vice versa.
 - Whenever such a pair is found we unify the subpaths after the two instructions.



Join Reconstruction Example



Instruction Semantics Evaluation

- The **domain** used in our analysis contains concrete (e.g. 0, 1, 1.0,...) and abstract values (e.g. true, false, not(.), or(...),...).
- **Abstract values** reflect boolean and arithmetic relations between registers. Based on those relations guard registers belonging to **disjoint** control paths are identified.
- In our analyses **memory cells** are supposed to contain unknown values.
- The **truth-value representation** implemented by the processor significantly impacts instruction semantic evaluation (see examples).



Instruction Semantics Evaluation

- In order to achieve maximum precision our evaluation process is divided into two parts:
 - Target-independent, **generic evaluation**: Applies whenever an operation has only concrete operands.
 - Machine-dependent, **generative evaluation** (generated from the TDL machine description of the target processor).



Instruction Semantics Evaluation (Examples)

Generic	Least- significant-bit	Zero (true iff different from 0)
$r2 \rightarrow 3$ $r3 \rightarrow 4$ $r2 + r3 \Rightarrow 7$	$r2 \rightarrow \text{true}$ $r3 \rightarrow 1$ $r2 + r3 \Rightarrow \text{false}$	$r2 \rightarrow \text{true}$ $r3 \rightarrow 1$ $r2 + r3 \Rightarrow \text{true}^*$
$r2 \rightarrow 3$ $r3 \rightarrow 4$ $r2 < r3 \Rightarrow \text{true}$	$r2 \rightarrow \text{false}$ $r3 \rightarrow 1$ $r2 < r3 \Rightarrow \text{unknown}$	$r2 \rightarrow \text{false}$ $r3 \rightarrow 1$ $r2 < r3 \Rightarrow \text{true}$



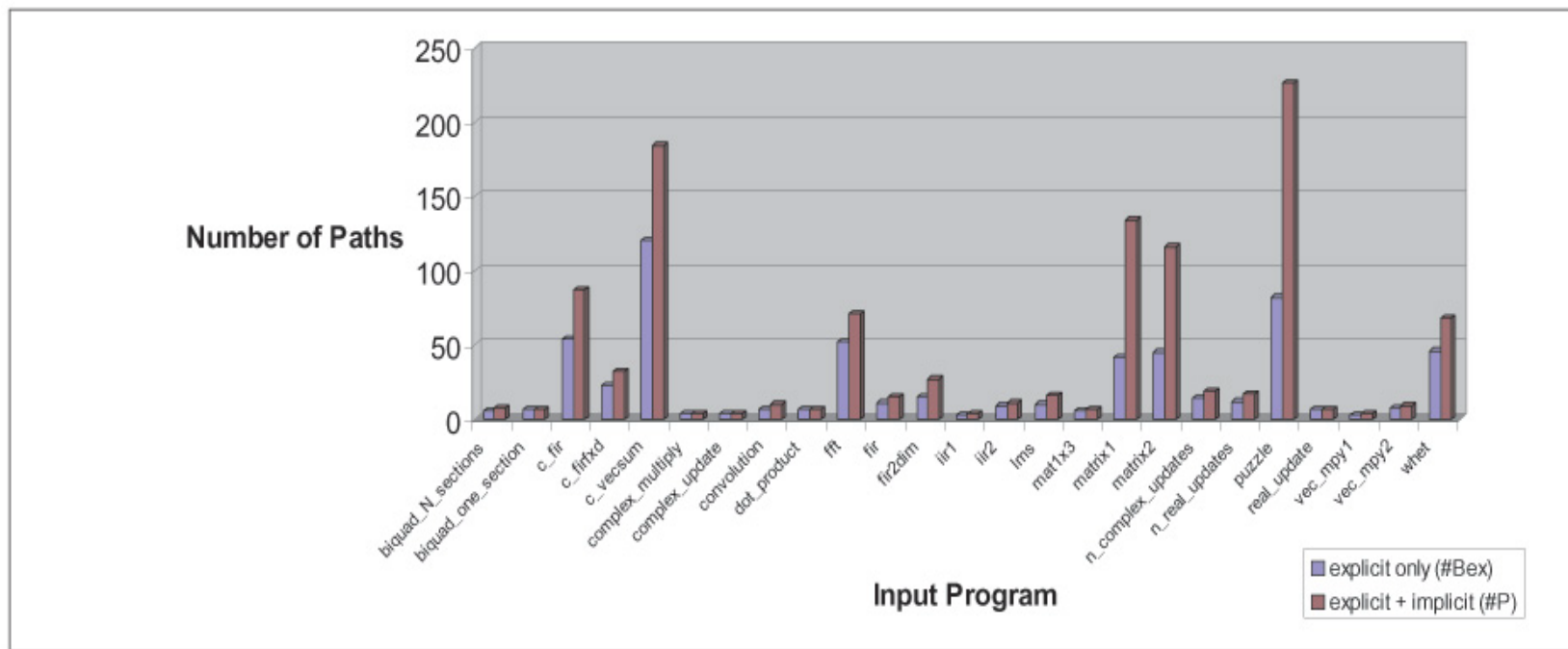
*: unless an overflow occurs

Experimental Results

Target processor: Philips TriMedia TM1000

Compiler: Philips tmcc (at highest optimization level)

Input files: DSPSTONE Benchmark



Experimental Results

file name	#I	# B_{ex}	# B_{em}	#P	#P/# B_{ex}	t [msec]
biquad_N_sections	56	6	12	8	1.33	27
biquad_one_section	28	7	7	7	1	12
c_fir	250	54	144	87	1.61	599
c_firfxd	168	23	50	32	1.39	249
c_vecsum	681	120	309	184	1.53	2,069
complex_multiply	10	4	4	4	1	2
complex_update	8	4	4	4	1	2
convolution	49	7	16	10	1.43	34
dot_product	25	7	7	7	1	11
fft	436	52	108	71	1.37	507
fir	56	11	23	15	1.36	41
fir2dim	193	15	50	27	1.8	830
iir1	27	3	6	4	1.33	26
iir2	27	9	15	11	1.22	26
lms	65	10	28	16	1.6	51
mat1x3	40	6	9	7	1.17	25
matrix1	202	42	226	134	3.19	781
matrix2	238	45	196	116	2.58	452
n_complex_updates	57	14	29	19	1.36	37
n_real_updates	70	12	27	17	1.42	44
puzzle	392	82	396	226	2.76	1,035
real_update	24	7	7	7	1	9
vec_mpy1	58	3	6	4	1.33	86
vec_mpy2	26	8	11	9	1.13	19
whet	648	46	108	68	1.48	1,618



Conclusion

- We presented an algorithm for precisely refining the prereconstructed control flow graph:
 - Phase 1: Detecting forks by extensive value analysis.
 - Phase 2: Reconstructing joins by identifying common subpaths.
 - At the end: implicit control flow has been made explicit.
- The algorithm is generic: all required information (e.g. instruction semantics) is taken from the TDL description of the target processor.
- The algorithm is based on a symbolic evaluation of instruction semantics taking into account the truth value representation of the target processor.



Conclusion

- Experimental results show that the precision of the reconstructed control flow is significantly higher than with approaches not taking predicated instructions into account.
- The experiments confirm the applicability of reconstruction algorithm for typical applications of digital signal processing.
- However: the worst-case complexity is exponential! This is due to the creation of new forks when contents of predicate registers are unknown.
- Future Work:
 - Refined value analysis based on memory disambiguation.
 - Further target architectures.

