ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fast Binary Translation: Translation Efficiency and Runtime Efficiency

Mathias Payer and Thomas R. Gross
Department of Computer Science
ETH Zürich

Motivation

- Goal: User-Space BT for Software Virtualization
  - fastBT as a system to analyze cost of BT
  - We are interested in
    - Flexibility of code generation
    - Efficiency of translation
    - Efficiency of generated runtime image
    - Limits of dynamic software BT
- Problem:
  - Flexibility of dynamic software BT comes at a cost
  - Especially indirect control transfers incur high overhead
- What is the lowest possible overhead (w/o HW support)?

Outline

- Introduction
- Design and Implementation
  - Translator
  - Table generation
- Optimization
  - How to reduce overhead
  - Benchmarks
- Related Work
- Conclusion

Introduction

- Design of a fast and flexible dynamic binary translator
  - Table driven translation approach
  - Master (indirect) control transfers
    - Indirect jumps, indirect calls, and function returns
  - Use a code cache and inlining
  - High level interface to generate translation tables at compile time
    - Manual table construction is hard & cumbersome
    - Use automation and high level description!

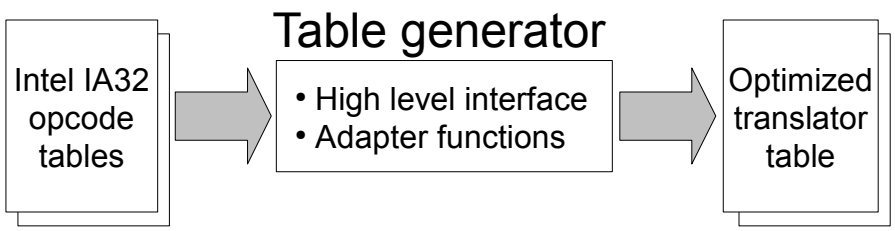
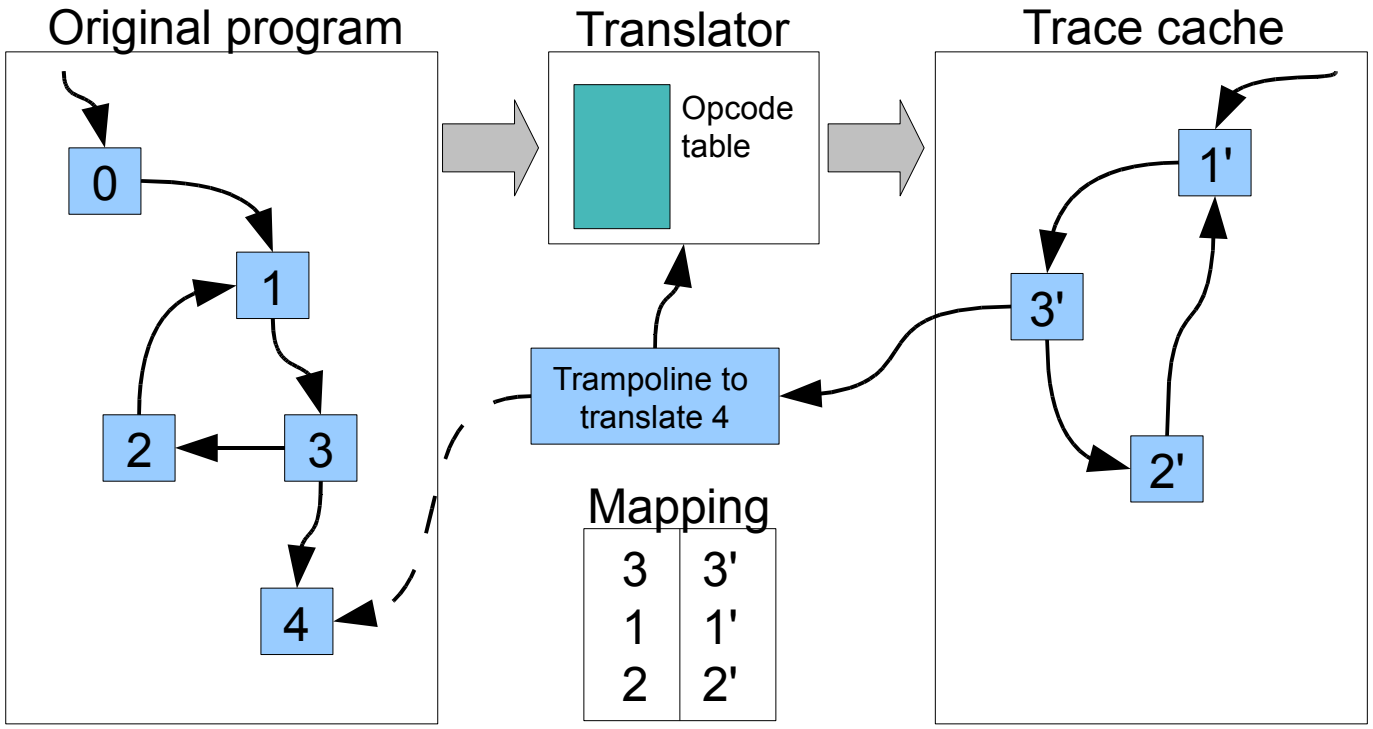


Table Generation

- Use enriched opcode tables
  - Information about opcodes, possible encodings, and properties
  - Specify default translation actions
- Use table generator to offer high-level interface
  - Transforming opcode tables into runtime translation tables
  - Add analysis functions to control the table generation
    - Memory access?
    - What are src, dst, aux parameters?
    - FPU usage?
    - What kind of opcode?
    - Immediate value as pointer?
    - ...

Design and Implementation

- BT in a nutshell:



Optimization

- Various optimizations explored for IA32
  - Performance limited by indirect control flow transfers
  - Optimize indirect call/jump and function returns
    - Require runtime lookup and dispatching
  - BT replaces indirect control transfers with software traps
    - Calculate target address from original instruction
    - Lookup target (translated?)
    - Redirect to target

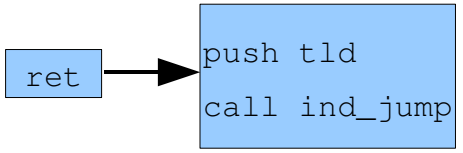
Optimization

- Various optimizations explored for IA32
  - Performance limited by indirect control flow transfers
  - Optimize indirect call/jump and function returns
    - Require runtime lookup and dispatching
  - BT replaces indirect control transfers with software traps
    - Calculate target address from original instruction
    - Lookup target (translated?)
    - Redirect to target

A naive approach translates one instruction into ~30 instructions (+function call)

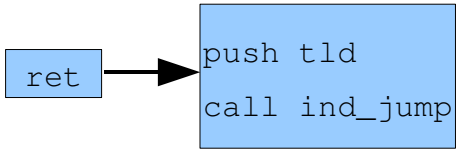
Optimization: Return instructions, naive approach

- Treat a return instruction like an indirect jump
- Use return IP on stack and branch to `ind_jump`
- `ind_jump` pseudocode:
  - Lookup target
  - Call to mapping table lookup function
  - Translate target if not in code cache
  - Return to translated target



Optimization: Return instructions, naive approach

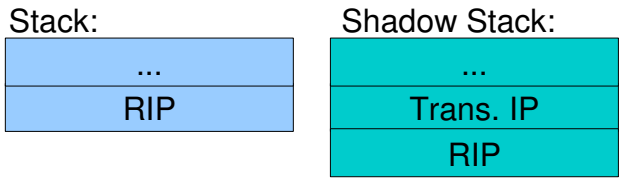
- Treat a return instruction like an indirect jump
- Use return IP on stack and branch to `ind_jump`
- `ind_jump` pseudocode:
  - Lookup target
  - Call to mapping table lookup function
  - Translate target if not in code cache
  - Return to translated target



- Results in ~30 instructions
  - 2-3 function calls (`ind_jump`, lookup, maybe translation)
  - No distinction between fast path and slow path

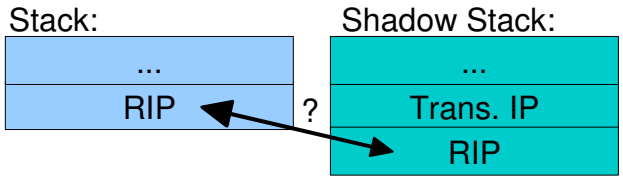
Optimization: Shadow Stack

- Use relationship between call/ret
- CALL
  - Push return IP and translated IP on shadow stack



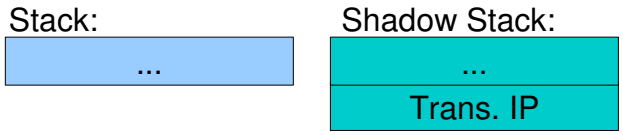
Optimization: Shadow Stack

- Use relationship between call/ret
- CALL
  - Push return IP and translated IP on shadow stack
- RET
  - Compare return IP on stack with shadow stack



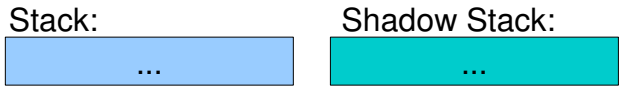
Optimization: Shadow Stack

- Use relationship between call/ret
- CALL
  - Push return IP and translated IP on shadow stack
- RET
  - Compare return IP on stack with shadow stack
  - If it matches, return to translated IP on shadow stack



Optimization: Shadow Stack

- Use relationship between call/ret
- CALL
  - Push return IP and translated IP on shadow stack
- RET
  - Compare return IP on stack with shadow stack
  - If it matches, return to translated IP on shadow stack



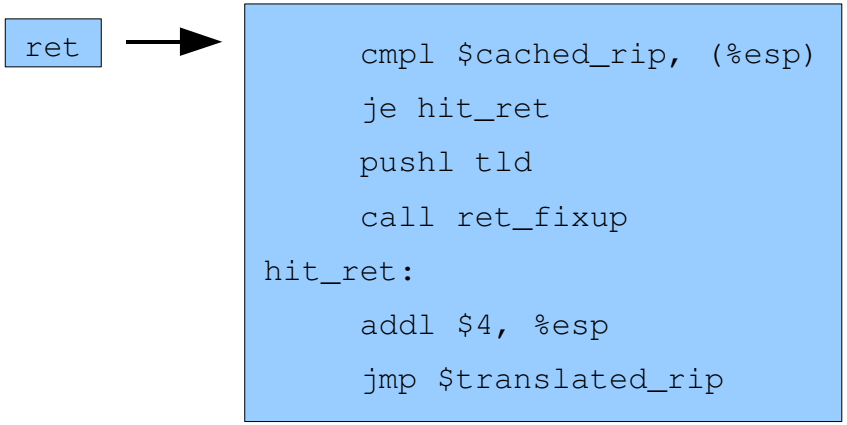
Optimization: Shadow Stack

- Use relationship between call/ret
- CALL
  - Push return IP and translated IP on shadow stack
- RET
  - Compare return IP on stack with shadow stack
  - If it matches, return to translated IP on shadow stack

- Results in ~18 instructions
  - 1 additional function call, if target is untranslated
  - Overhead results from stack synchronization

Optimization: Return Prediction

- Save last target IP and translated IP in inline cache
  - Compare inline cache with actual IP branch to translated IP if correct
  - Otherwise recover through indirect jump and backpatch cached entries



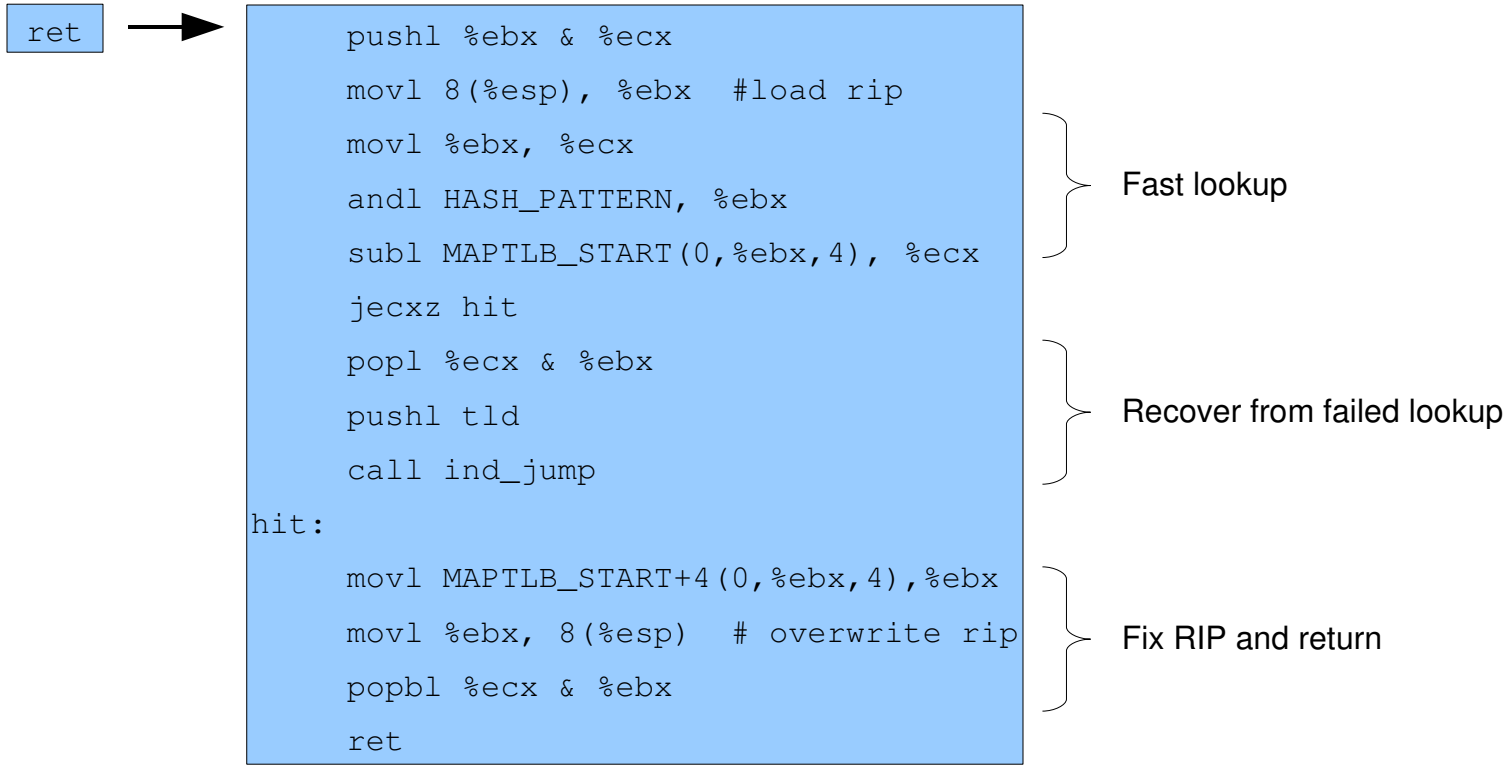
Optimization: Return Prediction

- Save last target IP and translated IP in inline cache
  - Compare inline cache with actual IP branch to translated IP if correct
  - Otherwise recover through indirect jump and backpatch cached entries

- Results in 4/43 (hit/miss) instructions
  - 1 additional function call, if target is untranslated
    - Only possible for misses
  - Optimistic approach that speculates on a high hit-rate
    - Recovery is more expensive than even the naive approach

Optimization: Inlined Fast Return

- Inline a fast mapping table lookup into the code cache
  - Branch to target if already translated
  - Otherwise branch to ind_jump



Optimization: Inlined Fast Return

- Inline a fast mapping table lookup into the code cache
  - Branch to target if already translated
  - Otherwise branch to ind_jump

- Results in 12 instructions
  - 1 additional function call, if target is untranslated
    - Only possible for misses
  - Faster than shadow stack and naive approach
  - For most benchmarks faster than the return prediction

Optimization summary

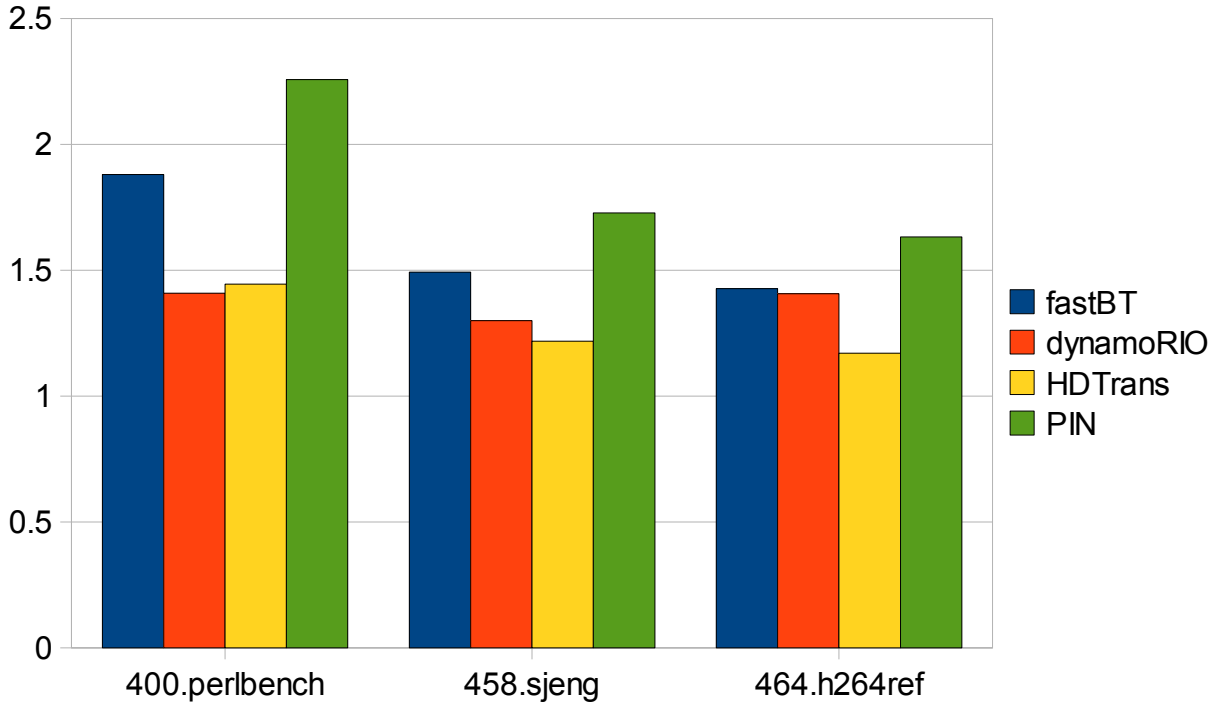
- Optimize different forms of indirect control transfers
  - Indirect jumps, indirect calls, and function returns
- fastBT uses:
  - Inlined fast return and inlining to reduce the cost of function returns
  - Indirect call prediction
    - Hit: 4, miss: 43 instructions
  - Inlined fast indirect jumps

Benchmarks

- Used SPEC CPU2006 benchmarks to evaluate different optimizations
- Compared against three dynamic BT systems
  - HDTrans version 0.4.1 (current version)
  - DynamoRIO version 0.9.4 (current version)
  - PIN version 2.4, revision 19012
- Used “null”-translation
- Machine: Intel Core2 Duo @ 3GHz, 2GB Memory

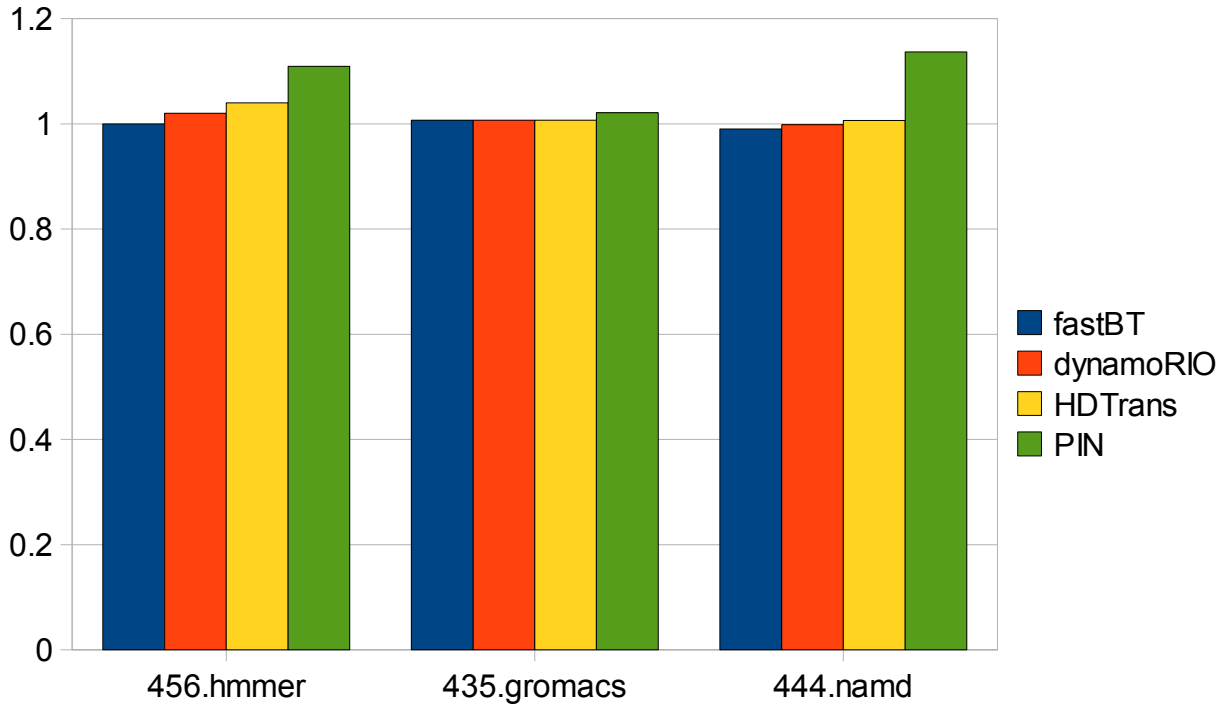
Benchmarks

Slowdown, relative to untranslated code



Benchmarks

Slowdown, relative to untranslated code



Benchmarks

- High overhead for SW BT:

	Map. Misses	(%miss)	Function calls	(%inl.)	Ind. Jumps	Ind. Calls	(%miss)
400.perlbench	246667	(0.00%)	**21909*10^6**	**(9.50%)**	**21930*10^6**	**3902*10^6**	**(89.14%)**
458.sjeng	1	(0.00%)	**21940*10^6**	**(1.25%)**	**109930*10^6**	**5070*10^6**	**(64.05%)**
464.h264ref	**11340*10^6**	**(42.64%)**	9148*10^6	(30.36%)	2317*10^6	28445*10^6	(1.20%)

- Low overhead for SW BT:

	Map. Misses	(%miss)	Function calls	(%inl.)	Ind. Jumps	Ind. Calls	(%miss)
456.hmmer	15	**(0.00%)**	219*10^6	(26.78%)	163*10^6	1*10^6	**(0.01%)**
435.gromacs	2	**(0.00%)**	**3510*10^6**	**(75.48%)**	27*10^6	3*10^6	**(0.86%)**
444.namd	2	**(0.00%)**	34*10^6	(20.47%)	15*10^6	2*10^6	**(0.00%)**

Benchmarks

- High overhead:
  - Many indirect control transfers
    - Combined w/ high number of mispredictions, or a low number of inlined methods
    - Overhead inherited from HW design, hard to reduce further with SW
  - High collision rate in mapping table
    - Leads to expensive recoveries
    - Could be fixed through an adaptive SW system
- Low overhead:
  - Few indirect control transfers
  - Cost of indirect control transfers is reduced by optimizations

Benchmarks

- High overhead:
  - Many indirect control transfers
    - Combined w/ high number of mispredictions, or a low number of inlined methods
    - Overhead inherited from HW design, hard to reduce further with SW
  - High collision rate in mapping table
    - Leads to expensive recoveries
    - Could be fixed through an adaptive SW system
- Low overhead:
  - Few indirect control transfers
  - Cost of indirect control transfers is reduced by optimizations
- Competitive performance compared to other translation frameworks
  - Additional optimization opportunities might require more HW support

Related work

- HDTrans
  - S. Sridhar et al. HDTrans: A Low-Overhead Dynamic Translator. SIGARCH'07
  - Table based dynamic BT, no high level interface
- DynamoRIO
  - D. Bruening et al. Design and Implementation of a Dynamic Optimization Framework for Windows. In ACM Workshop Feedback-directed Dyn. Opt. (FDDO-4) (2001).
  - IR based optimizing BT, targets binary optimization
- PIN
  - C.-K. Luk et al. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In PLDI'05
  - IR based, offers high level interface

Conclusion

- fastBT as a low-overhead BT
  - Fast translation, resulting in an efficient program
  - Table based, but offers high-level interface at compile time
  - Overhead introduced by fastBT is tolerable
  - Used to investigate limits of BT performance
- Indirect control transfers limit performance of SW solutions
  - Cannot be overcome with software smartness alone

Thanks for your attention!

Future / current work

- Reduce collisions in mapping table
  - Only visible for some benchmarks
  - Reorder entries in mapping table
  - Reset hash function and adapt to program
- Reduce the cost of indirect jumps and indirect calls
  - Not all indirect jumps / indirect calls are the same
  - Different optimizations for different kinds of control transfers
    - Analyze during translation phase
    - Pick best strategy

fastBT basics

- Table generator code size: 3937 lines total
  - 2373 lines opcode definition tables
- Runtime code size: 8702 lines total
  - 4580 lines of code, comments, definitions
    - 1200 lines for default translation actions
  - 4122 lines automatically generated opcode tables
  - Library compiled to 88kB
- Machine code based translation tables constructed at compile time, no additional overhead at runtime
- Constant time needed to translate one instruction

Table Generator: Analysis function

```
bool isMemOp (const unsigned char* opcode,
  const instr& disInf, std::string& action)
{
 bool res;
 /* check for memory access in instruction */
 res  = mayOperAccessMemory(disInf.dstFlags);
 res |= mayOperAccessMemory(disInf.srcFlags);
 res |= mayOperAccessMemory(disInf.auxFlags);

 /* change the default action */
 if (res) { action = "handleMemOp"; }

 return res;
}

// in main function:
addAnalysFunction(isMemOp);
```

Translator: Action function (copy)

```
finalize_tu_t action_copy(translate_struct_t *ts) {
  unsigned char *addr = ts->cur_instr;
  unsigned char* transl_addr = ts->transl_instr;
  int length = ts->next_instr - ts->cur_instr;
  /* copy instruction verbatim to translated version */
  memcpy(transl_addr, addr, length);
  ts->transl_instr += length;
  return tu_neutral;
}
```

Translator: Action function (RET)

```
finalize_tu_t action_ret(translate_struct_t *ts) {
    unsigned char *addr = ts->cur_instr;
    unsigned char *first_byte_after_opcode = ts->first_byte_after_opcode;
    unsigned char* transl_addr = ts->transl_instr;
    int32_t jmp_target = (int32_t)&ind_jump;
    if(*addr == 0xC2) { /* this ret wants to pop some bytes of the stack */
     PUSHL_IMM32(transl_addr, *((int16_t*)first_byte_after_opcode));
     jmp_target = (int32_t)&ind_jump_remove;
    }
    PUSHL_IMM32(transl_addr, (int32_t)ts->tld);
    CALL_REL32(transl_addr, jmp_target);
    ts->transl_instr = transl_addr;
    return tu_close;
}
```