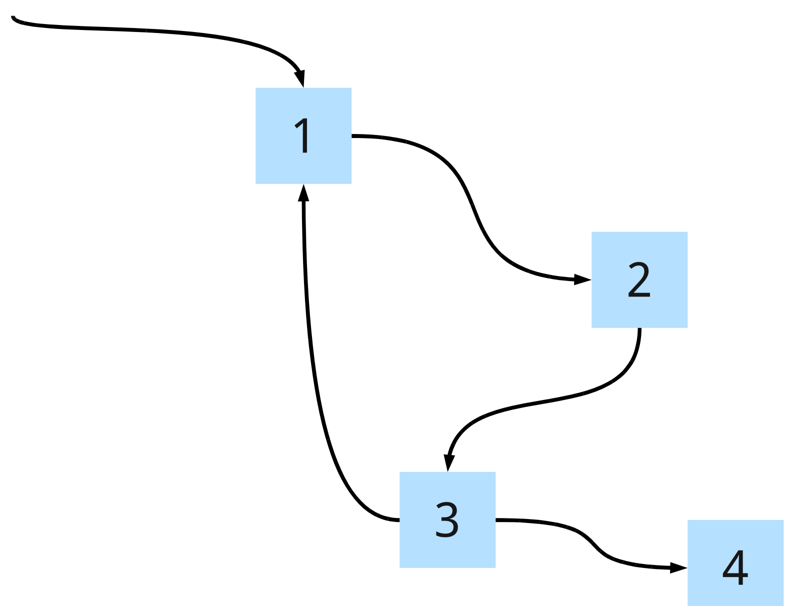
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# LLDSAL
# A Low-Level Domain-Specific Aspect Language for Dynamic Code-Generation and Program Modification

**Mathias Payer, Boris Bluntschli, & Thomas R. Gross**
Department of Computer Science
ETH Zürich, Switzerland

# Motivation: program instrumentation

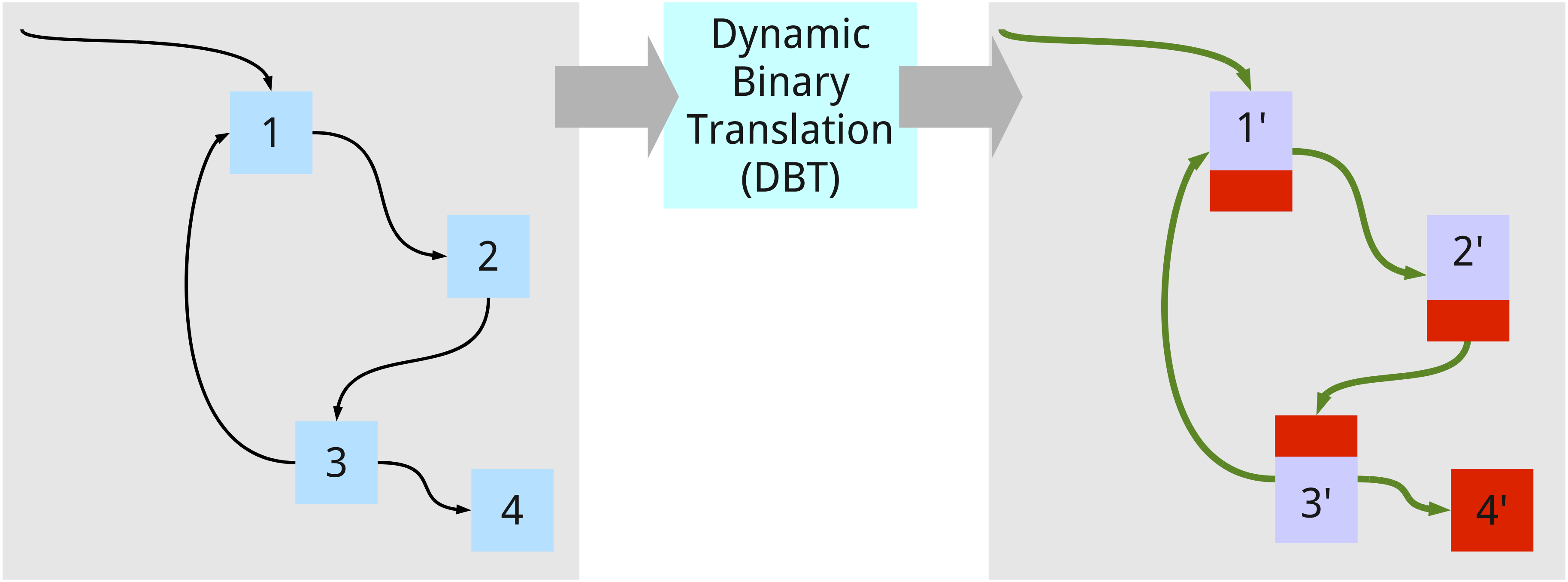
LLDSAL enables runtime code generation in a Dynamic Binary Translator (DBT)



# Motivation: program instrumentation

LLDSAL enables runtime code generation in a Dynamic Binary Translator (DBT)

- **External aspects** extend program functionality
- **Internal aspects** to implement the instrumentation framework



# Problem: code generation in DBT

DBT needs aspects that bridge between (translated) application and DBT world

- No calling conventions, must store everything
- Dynamic environment, no static addresses or locations
- Code must be fast (JIT-able)

```
char *code = ...;

BEGIN_ASM(code)
  addl $5, %eax
  movl %eax, %edx
  incl {&my_var}
END_ASM
```

# Solution: LLDSAL

Low-level Domain Specific Aspect/Assembly Language

- Aspects have access to high-level language constructs
- Aspects adhere to low-level conventions

DBT and LLDSAL enable AOP without any hooks

- JIT binary rewriting adds aspects on the fly

LLDSAL status: implemented and in use

- LLDSAL used for **internal aspects** of a BT (fastBT)
- LLDSAL guarantees **security properties** (libdetox security framework)

# Outline

Motivation

Background: Binary Translation (BT)

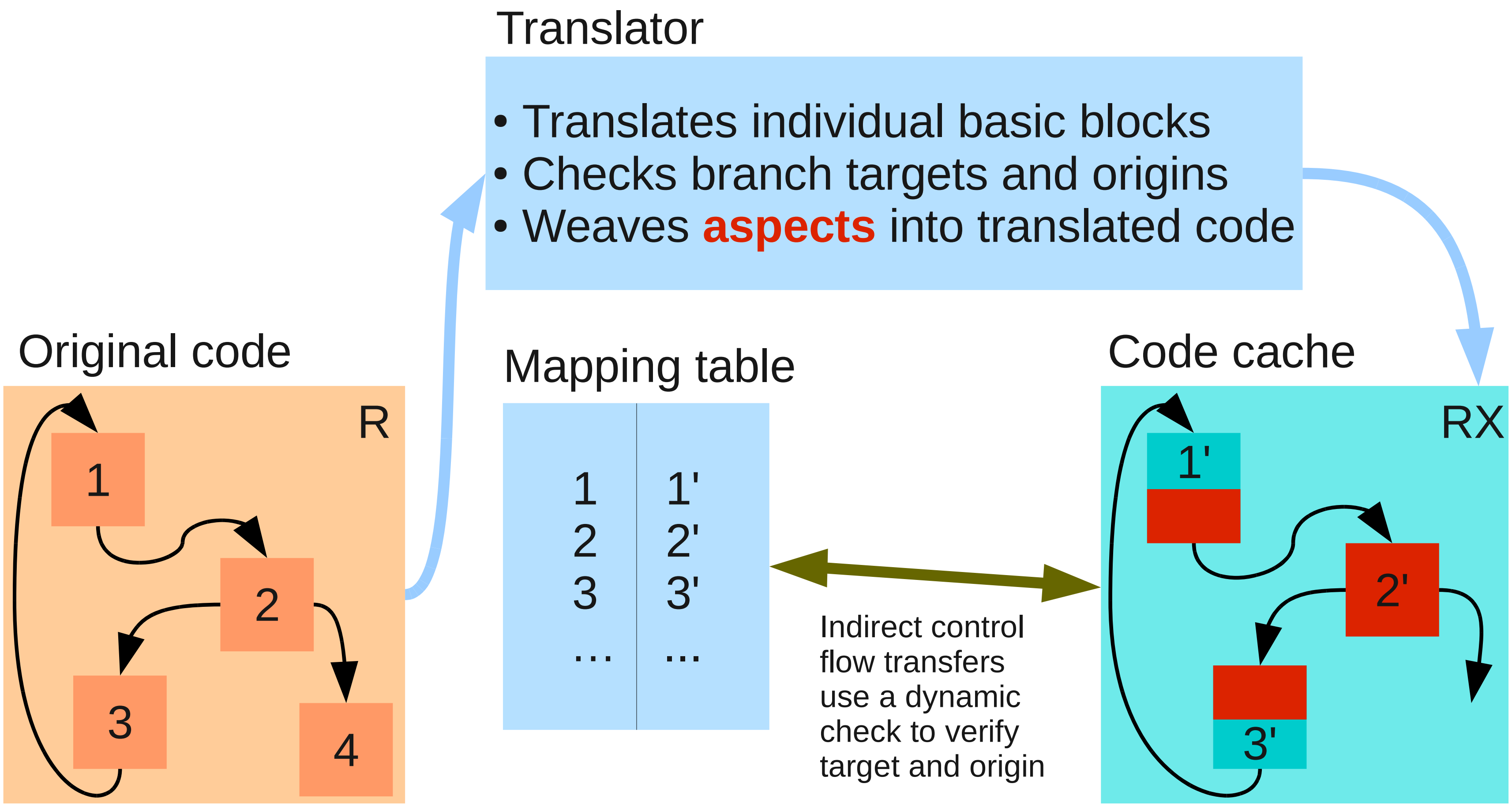
Language design

Implementation

Related work

Conclusion

# Binary Translation in a nutshell



# Outline

Motivation

Binary Translation (BT)

Language design

- Dynamic assembly language
- Data (variable) access
- Example: Dynamic code generation

Implementation

Related work

Conclusion

# Language design

## Usability: low-level / high-level trade-off

- Mix assembly code plus access to high-level language constructs

## Integration into host language

- DSL integrates naturally into the host language

## No runtime dependencies

- Source-to-source translation (LLDSAL to C code)

## LLDSAL defines a dynamic assembly language

- Enables dynamic low-level code generation at runtime

# Dynamic assembly language

LLDSAL combines assembly code with access to high-level data structures

- Expressiveness and syntax comparable to inline assembler
- JIT code generation at runtime, optimization for data-accesses
- Parameters encoded (inlined) into instructions

Assembly block

```
char *code = ...;

BEGIN_ASM(code)
  addl $5, %eax
  movl %eax, %edx
  incl {&my_var}
END_ASM
```

Pointer to code

Variable access

# Comparison LLDSAL vs. inline asm

## Code generation

- Inline asm ***executes*** code inline
- LLDSAL ***generates*** code inline

## Access to dynamic or thread local data

- Inline asm uses ***indirect memory references*** (pointer chasing)
- LLDSAL embeds ***direct pointers*** in generated code

```
asm ("incl %0\n"
      : "=a" (myvar)
      : "0" (myvar));
```

```
char *code = ...;

BEGIN_ASM(code)
  addl $5, %eax
  movl %eax, %edx
  incl {&my_var}
END_ASM
```

# Data (variable) access

JIT-compiled code enables new data access patterns

- LLDSAL enables variable access in host space using **`{variable}`**

Variable addresses directly encoded in emitted code

- No parameters are passed
- No indirection or pointer chasing

```
// inside indirect_call action
BEGIN_ASM(code)
  incl {&tld->stat->nr_ind_calls}
END_ASM
```

# Dynamic code generation

```
typedef void (*void_func)();
long my_func(long a) { return a * a; }
```

```
long result = 5;
char *target = ...;
void_func f = (void_func)target; {
  BEGIN_ASM(target)
    pushl ${result}
    call_abs {my_func}
    movl %eax, {&result}
    addl $4, %esp
    ret
  END_ASM
}

f(); // result == 25
```

pushes $5 to the stack

my_func(5)

result = my_func(5)

Clean-up and return

Execute dynamic code

# Outline

Motivation

Binary Translation (BT)

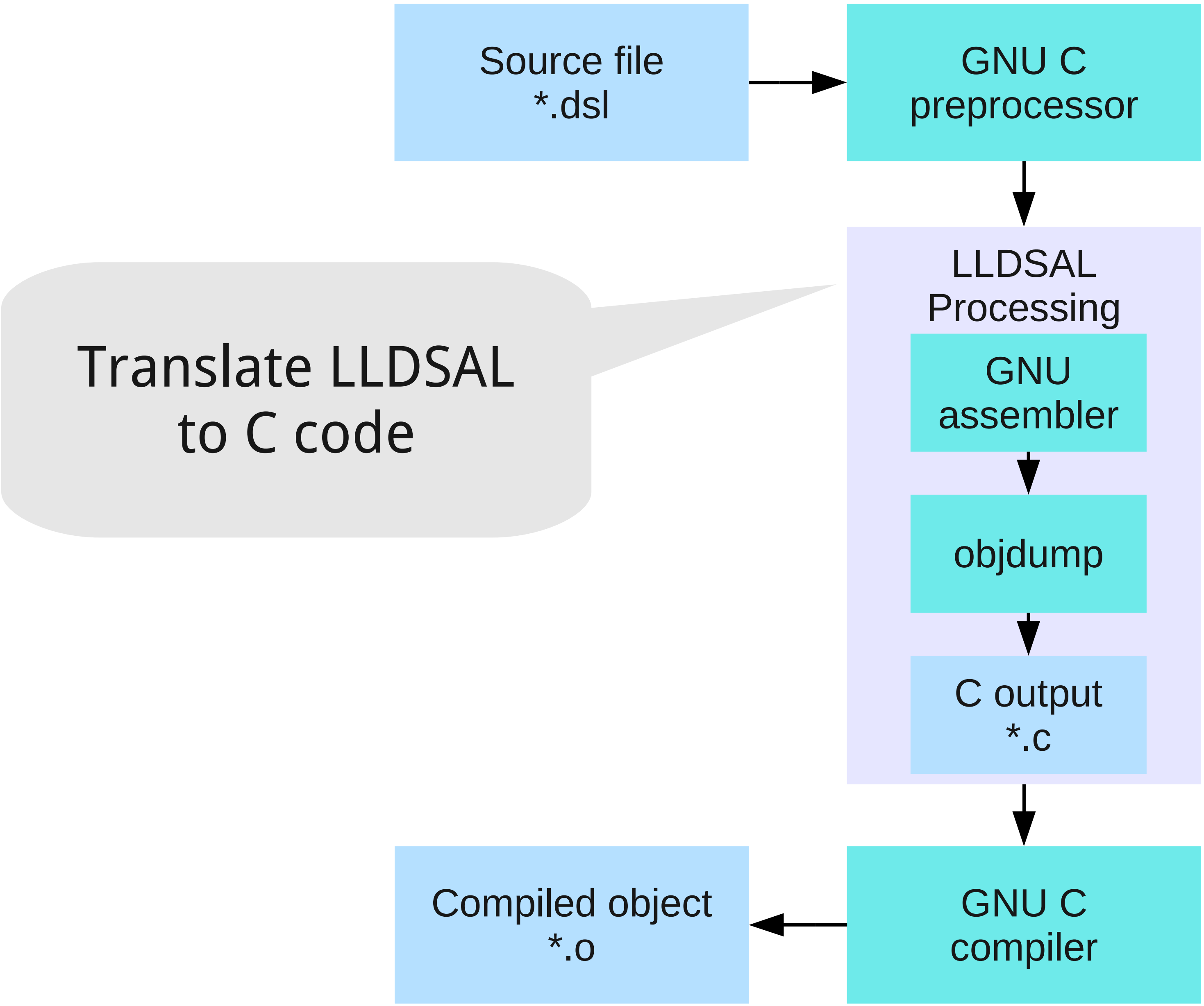
Language design

Implementation

Related work

Conclusion

# LLDSAL implementation



# LLDSAL alternatives

Macro-based approach

```
#define PUSHL_IMM32(dst, imm) \\
   *dst++=0x68; *((int_32_t*)dst)=imm; dst+=4
...
PUSHL_IMM32(code, 0xdeadbeef);
```

- No additional compilation pass needed
- Error prone, manual encoding

JIT code generation (GNU lightning, asmjit)

- Very flexible, dynamic register allocation
- High overhead, library dependencies

# Outline

Motivation

Binary Translation (BT)

Language design

Implementation

Related work

Conclusion

# Related work

Compile-time DSL parsing [Porkolab et al., GPCE'10]

- LLDSAL first dynamic low-level DSAL for BT

Guyer and Lin describe an approach to optimize libraries for different environments [DSL'99]

- Annotation based, LLDSAL uses assembly code with high-level data access

Khepora is an approach to s2s DSLs [Faith et al., DSL'97]

- Full DSL parsing using syntax trees, too heavy-weight for LLDSAL

# Conclusion

LLDSAL enables dynamic code generation for DBTs

- Direct access to host variables and data structures
- Low-overhead (no arguments passed, low-level encoding)
- No library dependencies

LLDSAL raises level of interaction between developer and BT framework

- Increased readability of code
- Better maintainability due to automatic translation

# Thank you for your attention

?

# Data (variable) access

Use the address of the variable `${&foo}`

- Instruction stores current address as immediate

Encode the (static) value of the variable `${foo}`

- Instruction stores current value as immediate

Use dynamic value of variable `{&foo}`

- Instruction stores address of variable and encodes memory dereference

Use dynamic value of the address of the variable `{foo}`

- Instruction stores value as immediate and encodes memory dereference

# Data (variable) access

pushl ${tld}

- Push current value of tld onto stack

movl {tld->stack-1}, %esp

- Read value from *(tld->stack-1) and store it in %esp

movl ${tld->stack-1}, %esp

- Store address of (tls->stack-1) in %esp

movl %eax, {&tld->saved_eax}

- Store %eax at &tld->saved_eax

# Example (indirect lookup, inside BT)

```
BEGIN_ASM(transl_instr)
  pushfl
  pushl %ebx
  pushl %ecx

  movl 12(%esp), %ebx       // Load target address
  movl %ebx, %ecx           // Duplicate RIP

  /* Load hashline (eip element) */
  andl ${MAPPING_PATTERN >> 3}, %ebx;
  cmpl {tld->mappingtable}(, %ebx, 8),  %ecx;
  jne nohit

hit:
  // Load target
  movl {tld->mappingtable+4}(, %ebx, 8), %ebx

  movl %ebx, {&tld->ind_target}
  popl %ecx
  popl %ebx
  popfl
  leal 4(%esp), %esp
  jmp *{&tld->ind_target}

nohit:
  // recover mode - there was no hit! ...
END_ASM
```