

# Dynamic Native Optimization of Interpreters

IVME'03, June 12, 2003

Gregory T. Sullivan , Derek L. Bruening,  
Iris Baron, Timothy Garnett,  
Saman Amarasinghe

presentation on 24.11.2011

by Stephan Zalewski <[stephan.zalewski@uni-duesseldorf.de](mailto:stephan.zalewski@uni-duesseldorf.de)>

# DynamoRIO

*"DynamoRIO records sequences of native instructions, which are subsequently partially evaluated with respect to the in-memory representation of the program being interpreted."*

# Motivation

- interpreted implementation of highly dynamic languages
  - translate language into Intermediate Representation (IR)
  - interpreter loop bytecode dispatcher
  - $\Rightarrow$  overhead
- DynamoRIO combines JIT compiler and partial evaluation
  - provided as framework to interpreter writers

# DynamoRIO

program represented as immutable vector of bytecodes

- indexed by the Logical PC (LPC)

processed like

- fetch bytecode by LPC
- optional dereference other bytecodes e.g. arguments
  - constant folding
- conditional control flow based on bytecode
- jump to bytecode instructions
  - conditional branches based on constant values can be removed
- increment LPC
  - track LPC increments, continue partial evaluation

# DynamoRIO Dynamic Optimization Framework

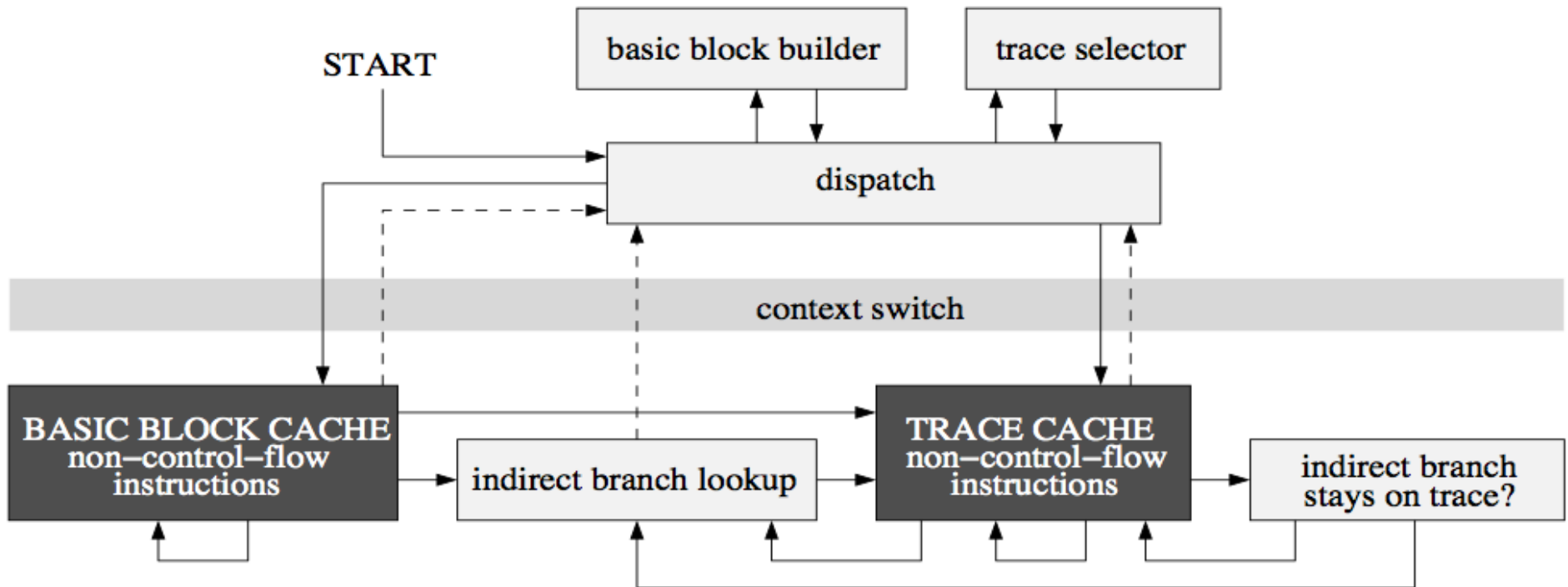


Figure 1. Flow chart of the DynamoRIO System. Dark shading indicates application code.

# DynamoRIO Dynamic Optimization Framework

observe instructions prior to execution

- interpretation
- cache translated instructions

cache basic blocks

- sequence ending with single control transfer
- context switch to interpreter after each block
- direct jump target → link blocks
- indirect jump → translate based on original address

# DynamoRIO Dynamic Optimization Framework

trace frequently executed sequence of blocks

- connect basic blocks along on branch path
- assume execution stays on trace
  - insert check and fall back to lookup on branch

popular jump targets as starting points for trace

- does not work so great for interpreter main loop
  - frequent jump target but...
  - leaves trace early due to bytecode dispatch

⇒ support DynamoRIO trace generation by supplying interpreter information

# Abstract PC

neither logical nor native PC suffice to uniquely identify position in control flow

⇒ Abstract PC = (logical PC, native PC)

# Logical Control Flow

identify APC changes, supports trace generation

- `logical_direct_jump(new_logpc)`
  - jump to next instruction based on compile time constant
  - (current NPC, current LPC)  $\Rightarrow$  (NPC + 1, new\_logpc)
- `logical_indirect_jump(new_logpc)`
  - jump is based on runtime information (e.g. return address on stack)
- `logical_relative_jump(offset)`
  - like `logical_direct_jump` but
  - "uninteresting" control transfers

# Immutable Program Data

required for constant propagation and folding

- `set_region_immutable(start, end)`
  - memory region immutable
- `add_trace_constant_address(addr)`
  - identify an address as constant for current APC
  - neat for pc and dereferencing bytecode
- `set_trace_constant_stack_address(addr, val)`
  - restrict constant folding to current stack frame

# Example: TinyVM

instruction decoding:

loop:

```
op = instrs[pc].op;  
arg = instrs[pc].arg;  
switch (op) { ...
```

allow constant folding for bytecode

```
set region immutable(instrs,  
    (instrs + num instrs*sizeof(ByteCode)1));  
dynamorio add trace constant address(&pc);
```

# Example: TinyVM

## CALL opcode

case CALLOP:

... setup new call frame ...

pc = arg; /\* go to start of function body \*/

**dynamorio\_logical\_direct\_jump**(pc);

goto loop;

# Example: TinyVM

## RETURN opcode

case RETOP:

... clean up stack ...

pc = pop\_raw(); /\* pop the return PC \*/

**dynamorio\_logical\_indirect\_jump(pc);**

\*(++sp) = val; /\* put return value back on stack \*/

goto loop;

# Example: TinyVM

BEQ bytecode (args is constant instrs[pc].arg)

case BEQOP:

```
n = pop int(); /* top must be an int */
if (n) { /* not equal 0 */
    pc++; /* just continue */
    dynamorio_logical_relative_jump(1);
} else /* equal 0 */
    pc = arg; /* do the branch */
    dynamorio_logical_direct_jump(pc);
}
goto loop;
```

# Collecting Traces

- traces start and end with control transfer
- trace is associated with APC

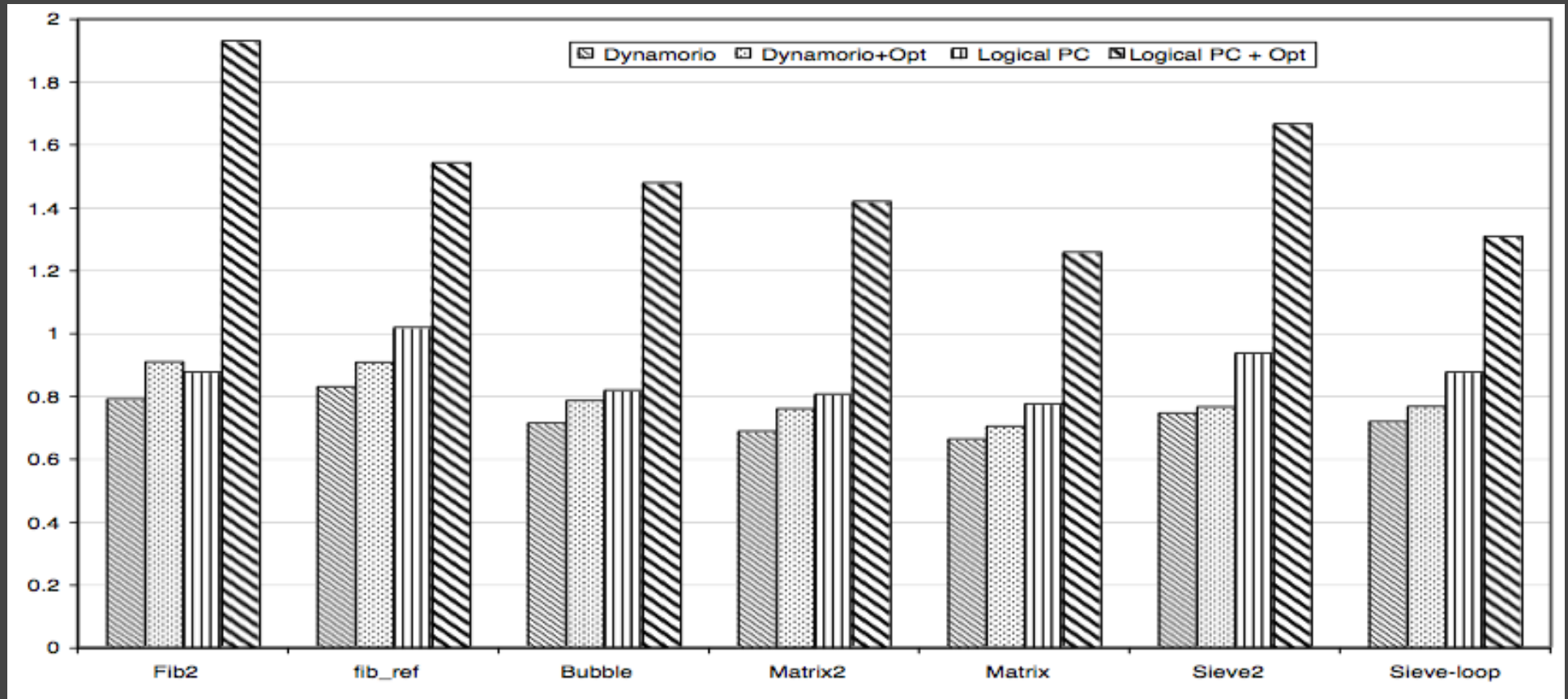
on logical jump

- if tracing, finish current trace
- if it is a direct jump from a trace, link blocks
- otherwise start tracing if exceeding threshold

# Optimizing Traces

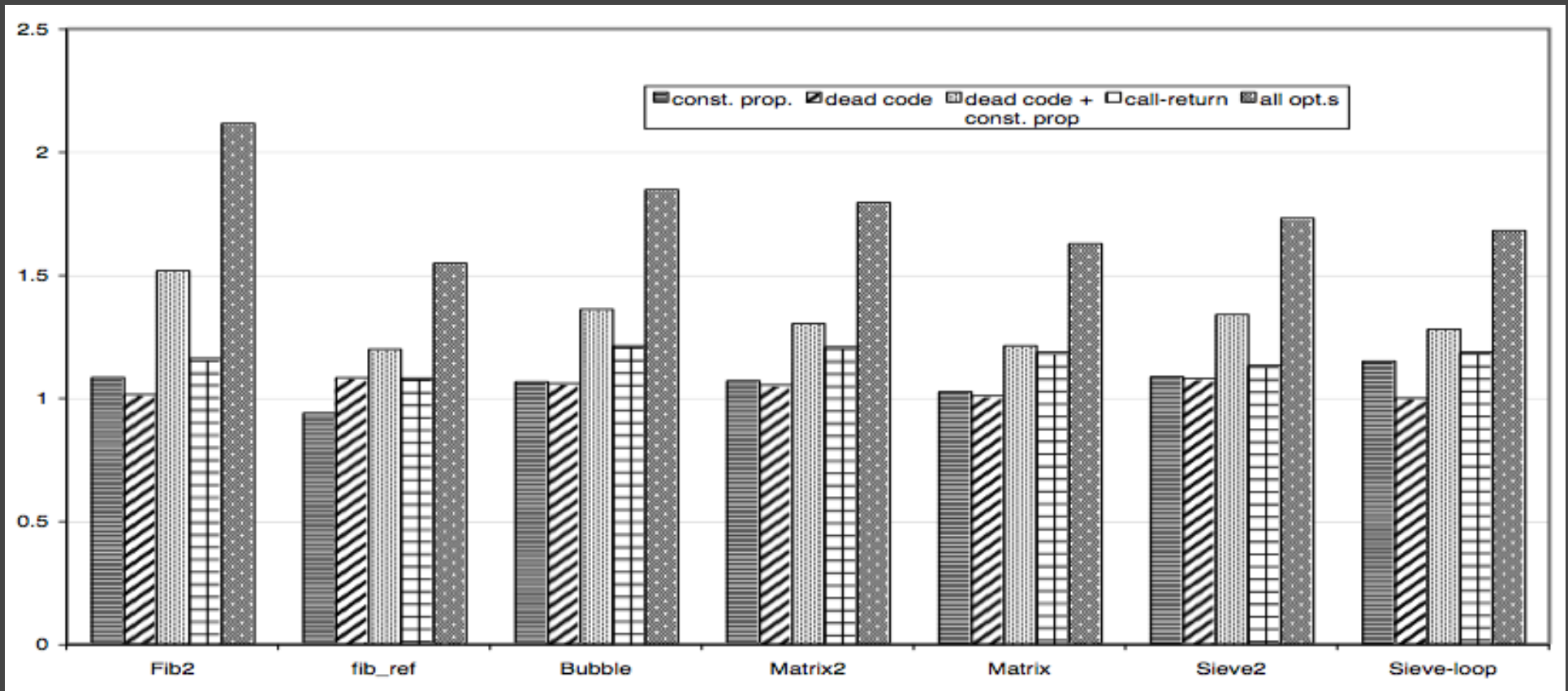
- constant propagation
  - use supplied constants information
- call-return matching
  - inlining
- dead code elimination

# Results



benchmarks on TinyVM, runtime normalized against native execution

# Results



contribution by optimizations, normalized to DynamoRIO with logical tracing