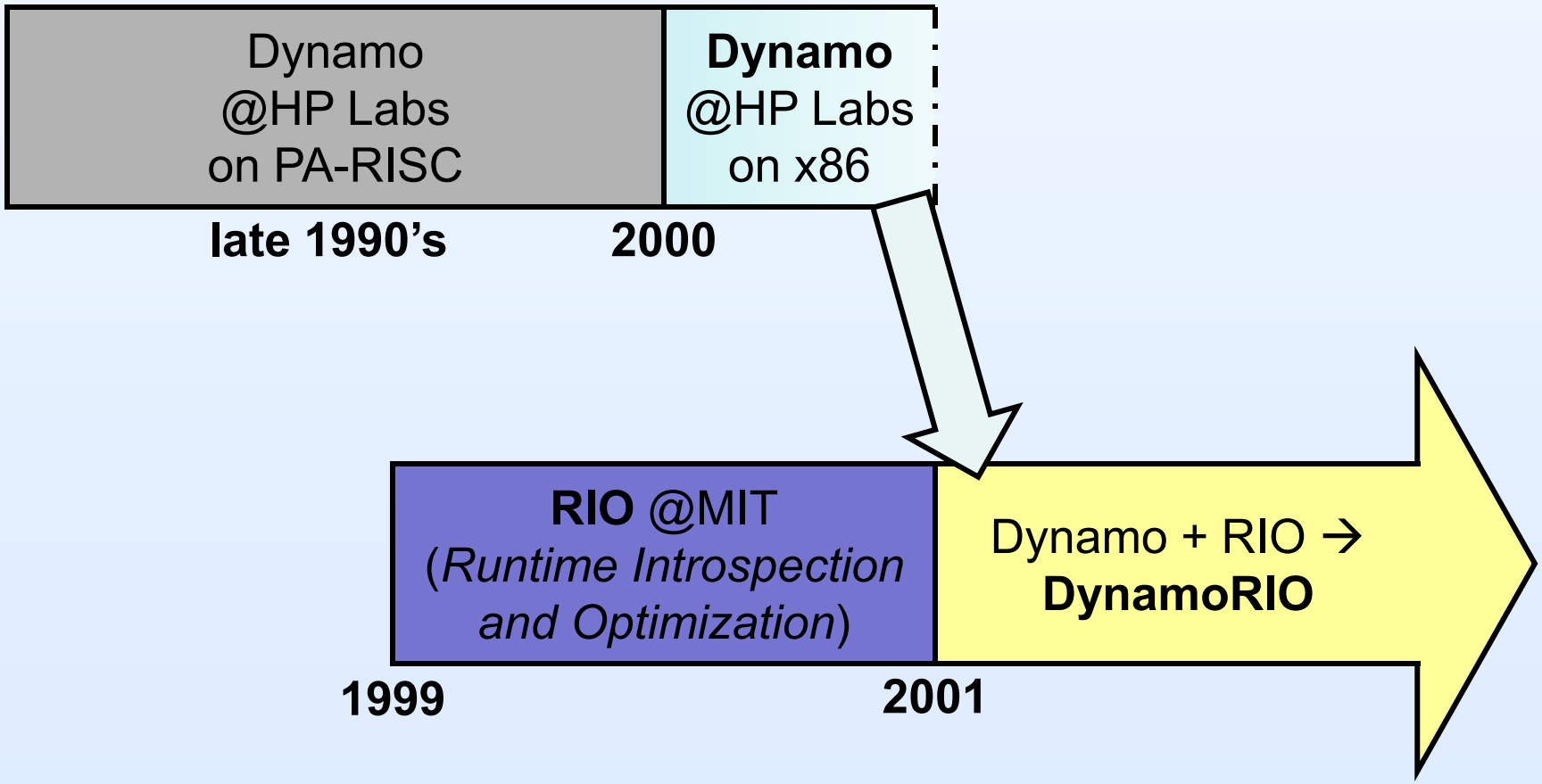
Building Dynamic Tools with DynamoRIO on x86 and ARM

Derek Bruening and Qin Zhao
Google

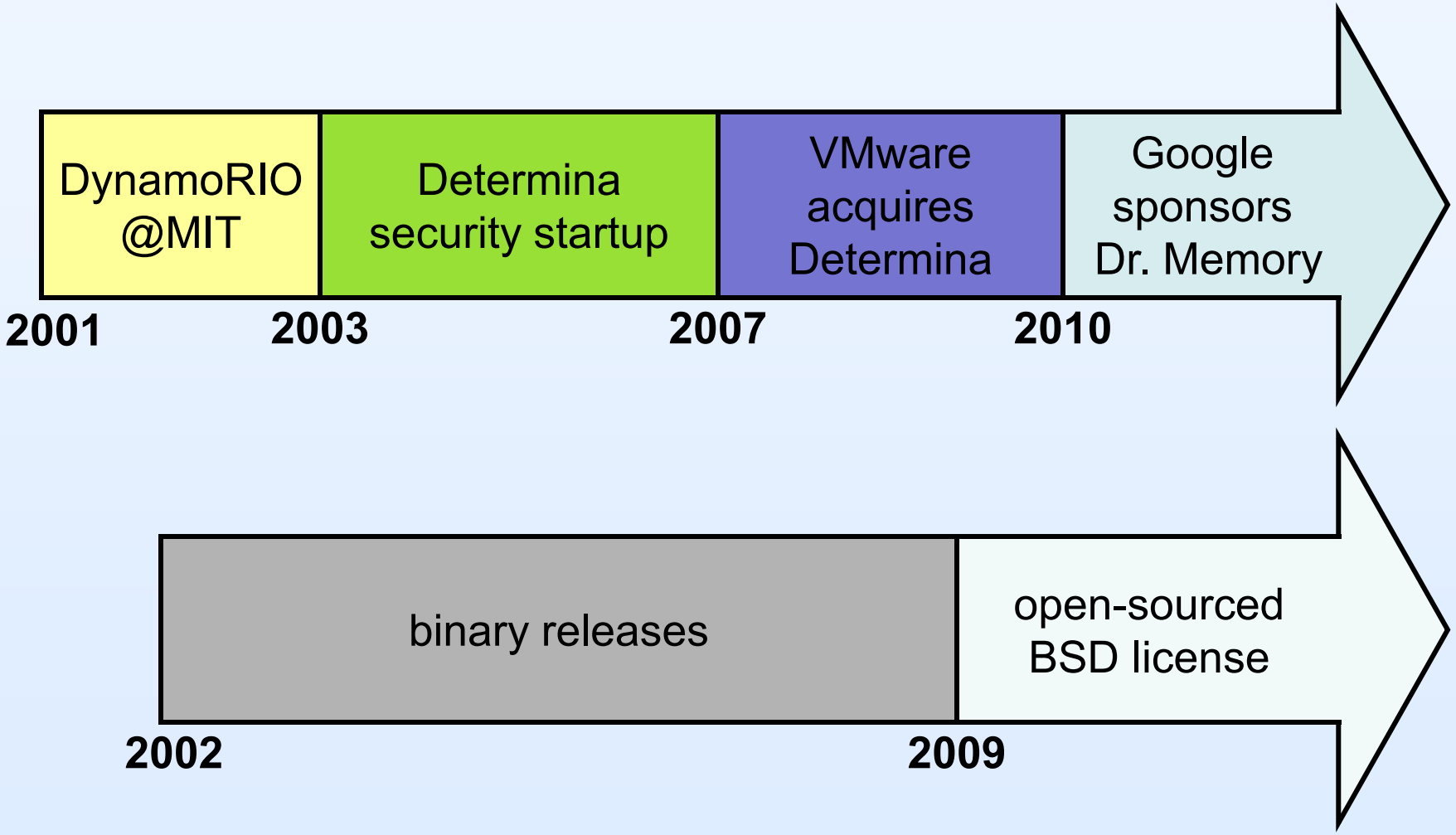
Tutorial Outline

- 2:00-2:10 Welcome + DynamoRIO History
- 2:10-3:10 DynamoRIO Overview
- 3:10-3:45 Creating Simple Tools
- *3:45-4:00 Break*
- 4:00-4:30 Creating Complex Tools
- 4:30-4:50 DynamoRIO API
- 4:50-5:15 DynamoRIO on ARM
- 5:15-5:30 Q & A

DynamoRIO


Dynamo
@HP Labs
on PA-RISC
Dynamo
@HP Labs
on x86
late 1990's
2000
RIO @MIT
(Runtime Introspection
and Optimization)
Dynamo + RIO →
DynamoRIO
1999
2001

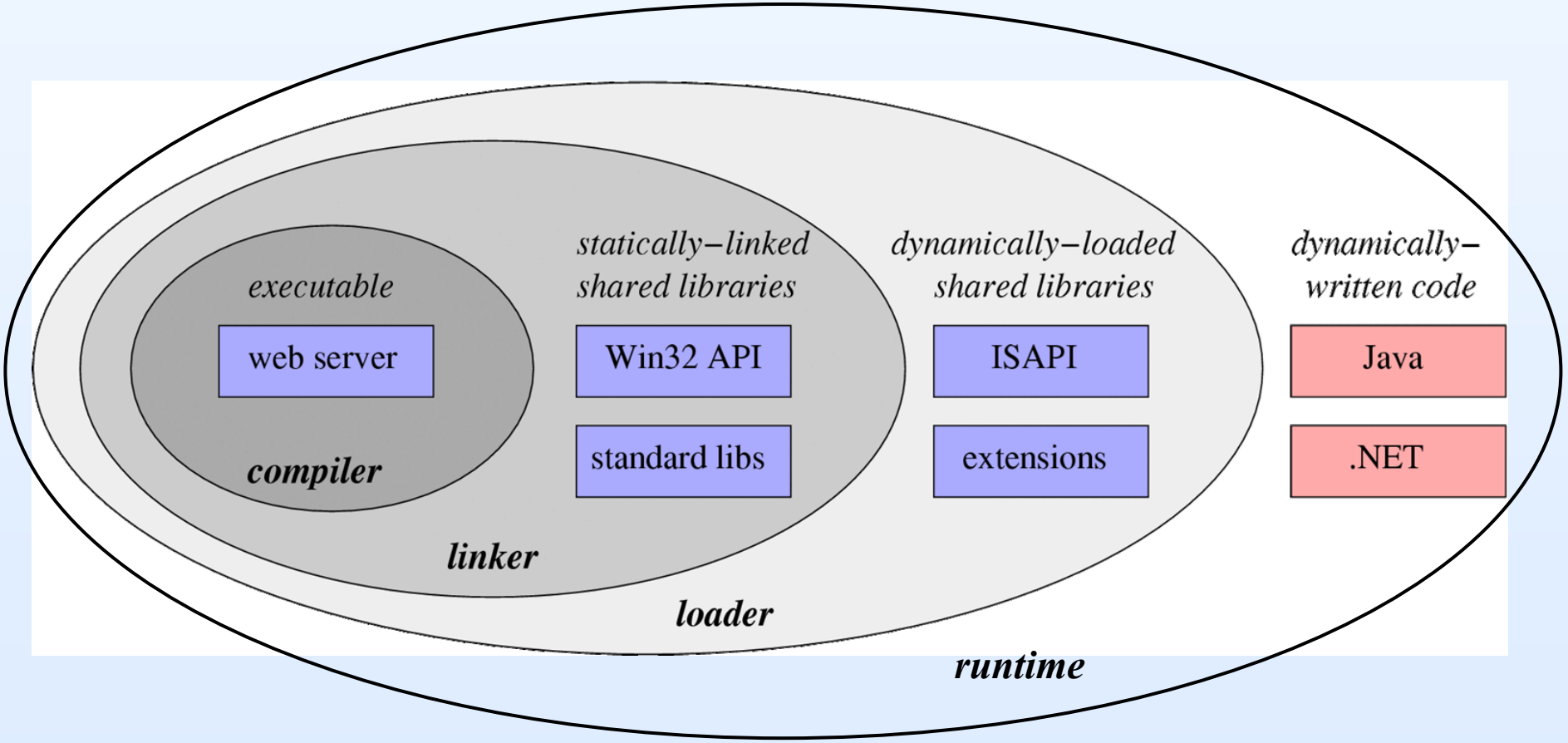
DynamoRIO History



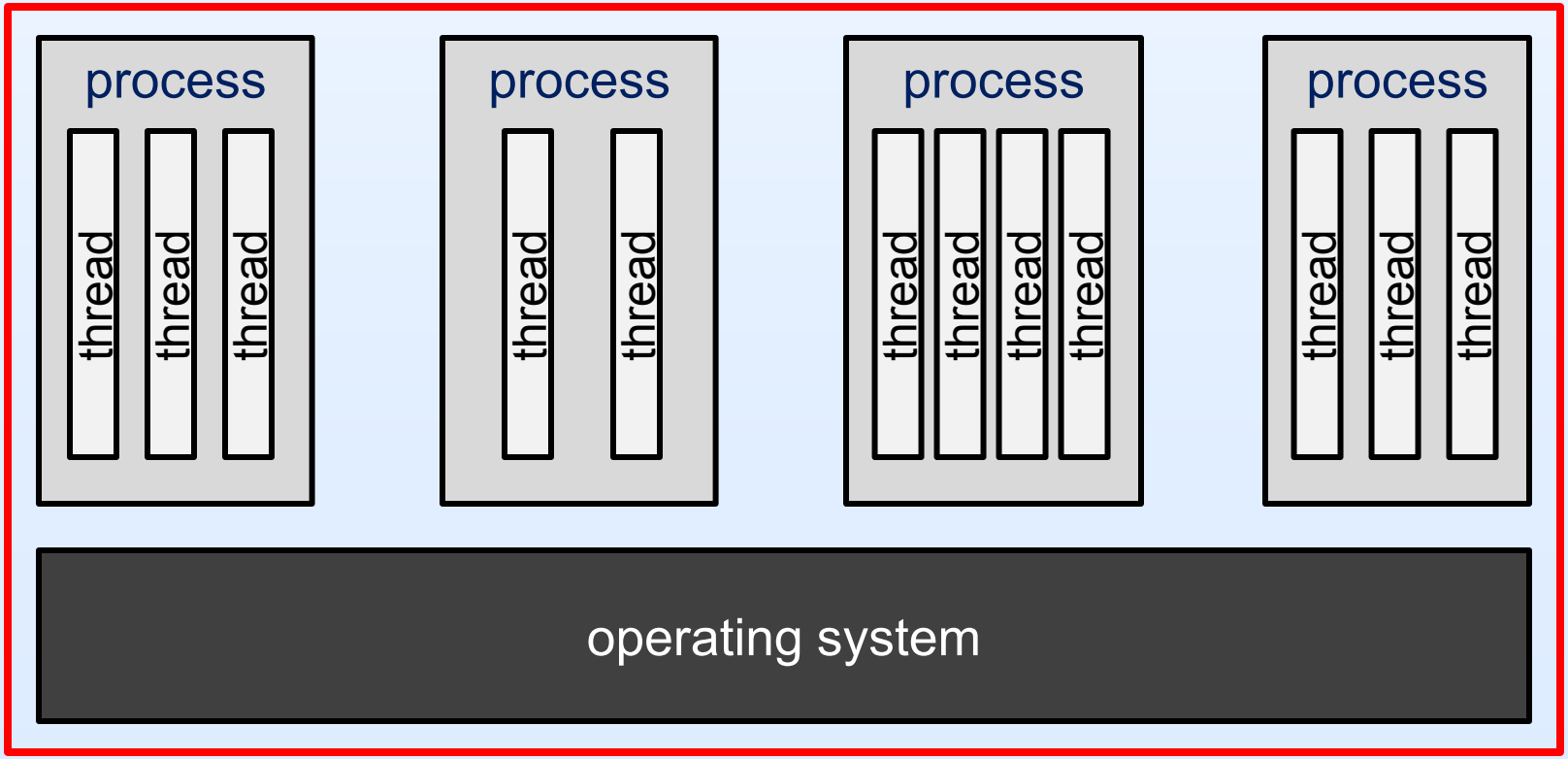
DynamoRIO Overview

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A

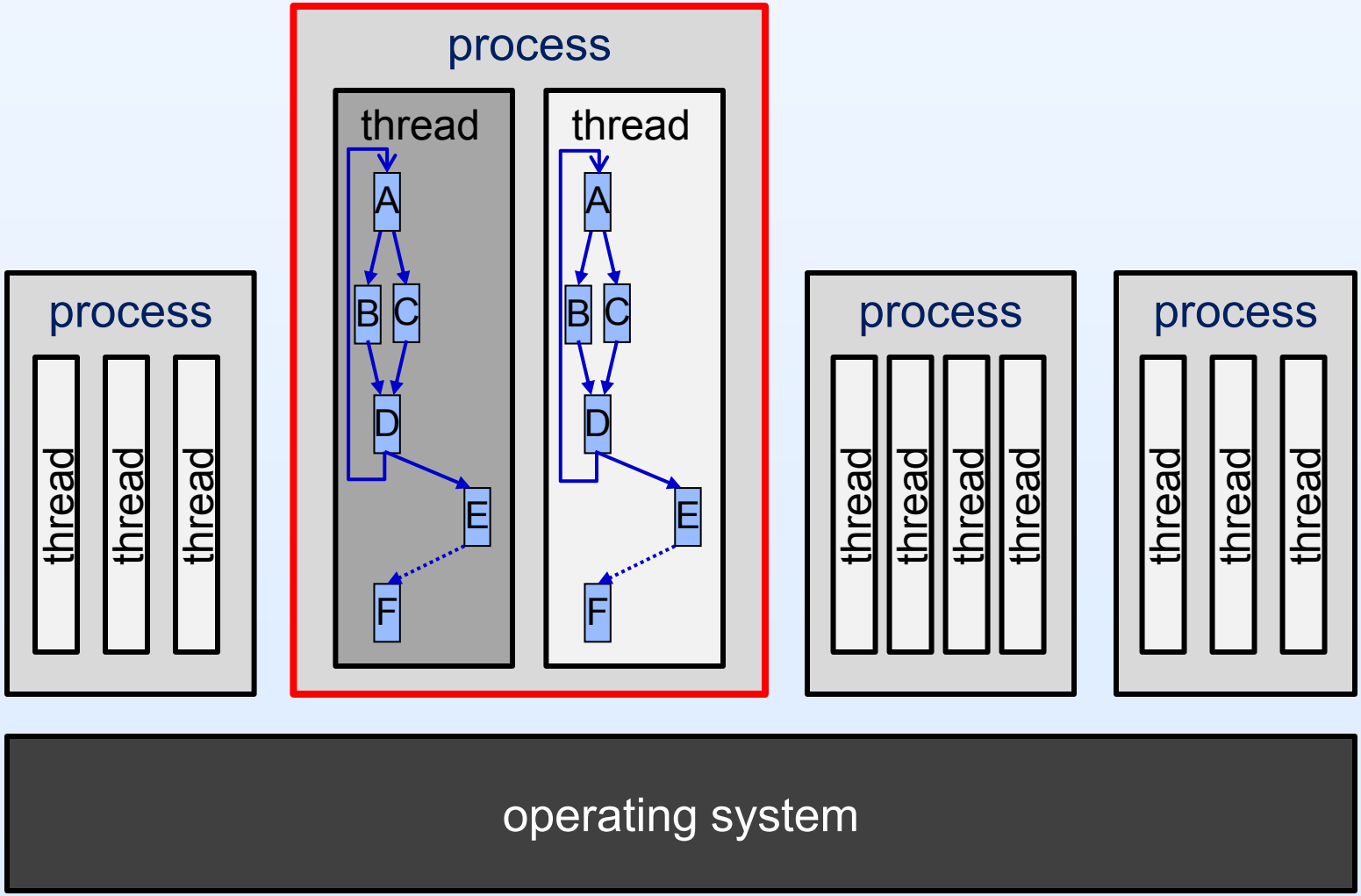
Reach of Toolchain Control Points



System Virtualization



Process Virtualization



Design Goals

- Efficient
  - Near-native performance
- Transparent
  - Match native behavior
- Comprehensive
  - Control every instruction, in any application
- Customizable
  - Adapt to satisfy disparate tool needs

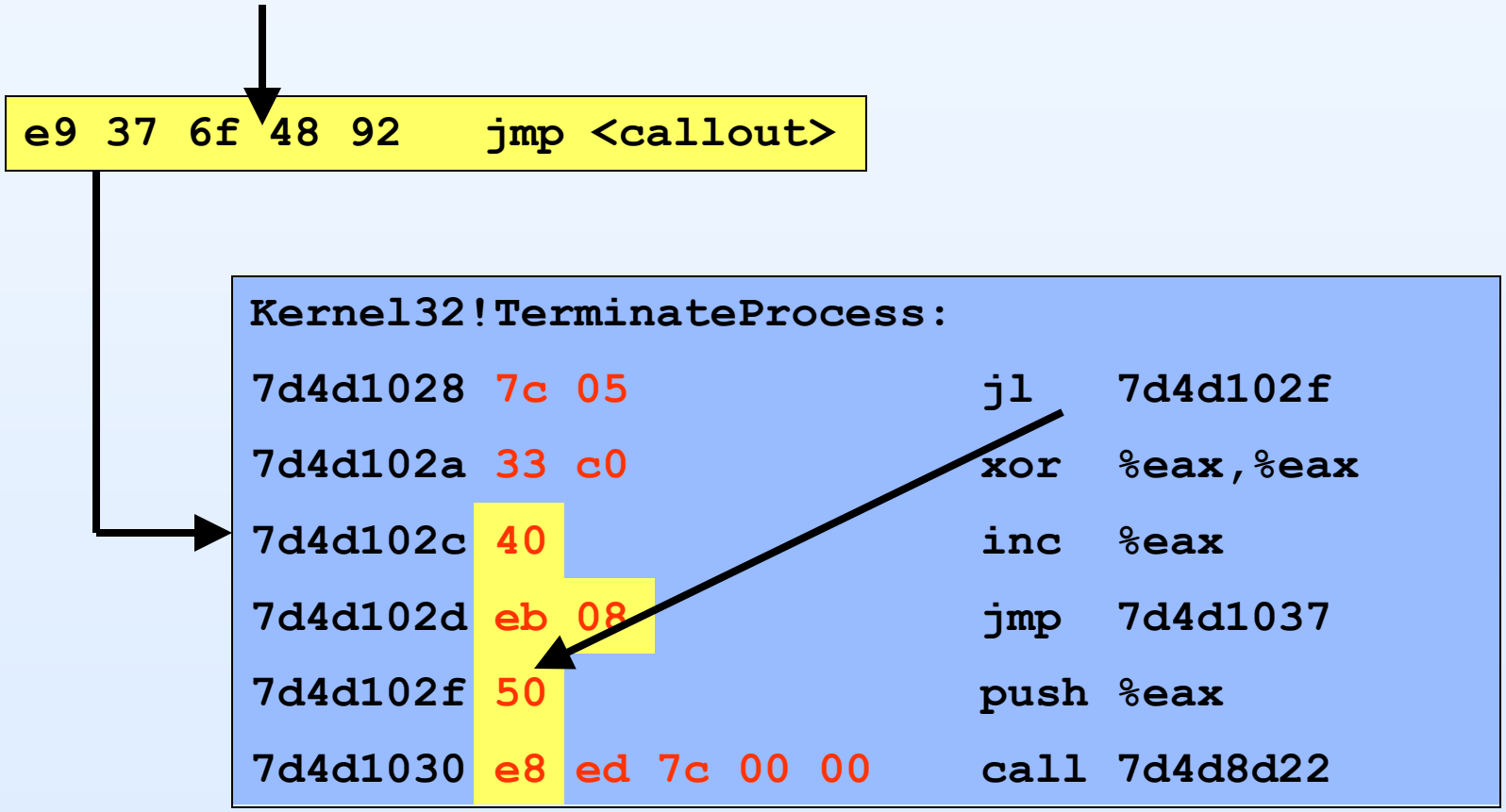
Challenges of Real-World Apps

- Multiple threads
  - Synchronization
- Application introspection
  - Reading of return address
- Transparency corner cases are the norm
  - Example: access beyond top of stack
- Scalability
  - Must adapt to varying code sizes, thread counts, etc.
- Dynamically generated code
  - Performance challenges

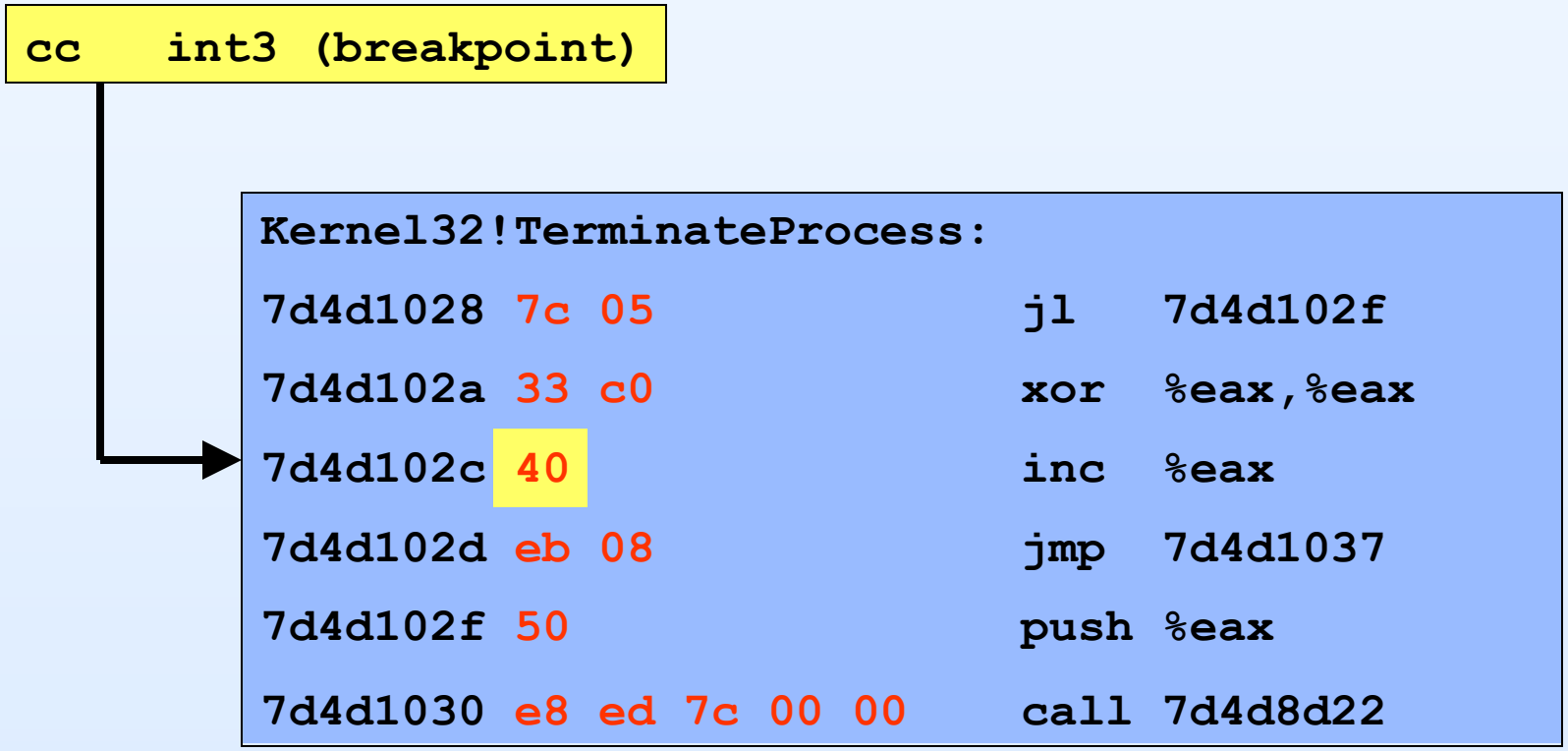
Overview Outline

- Efficient
  - Software code cache overview
  - Thread-shared code cache
- Transparent
- Comprehensive
- Customizable

Direct Code Modification



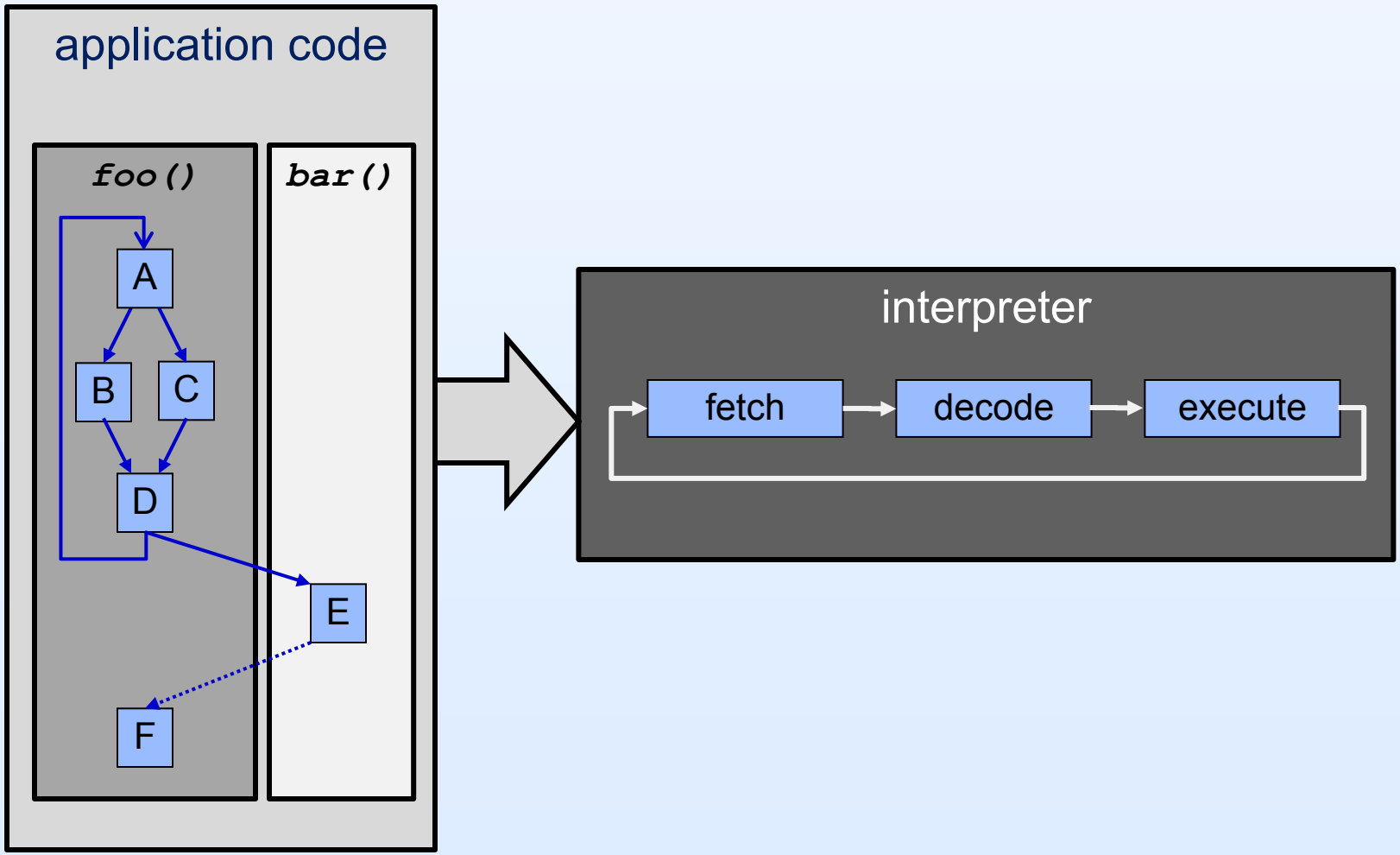
Debugger Trap Too Expensive



We Need Indirection

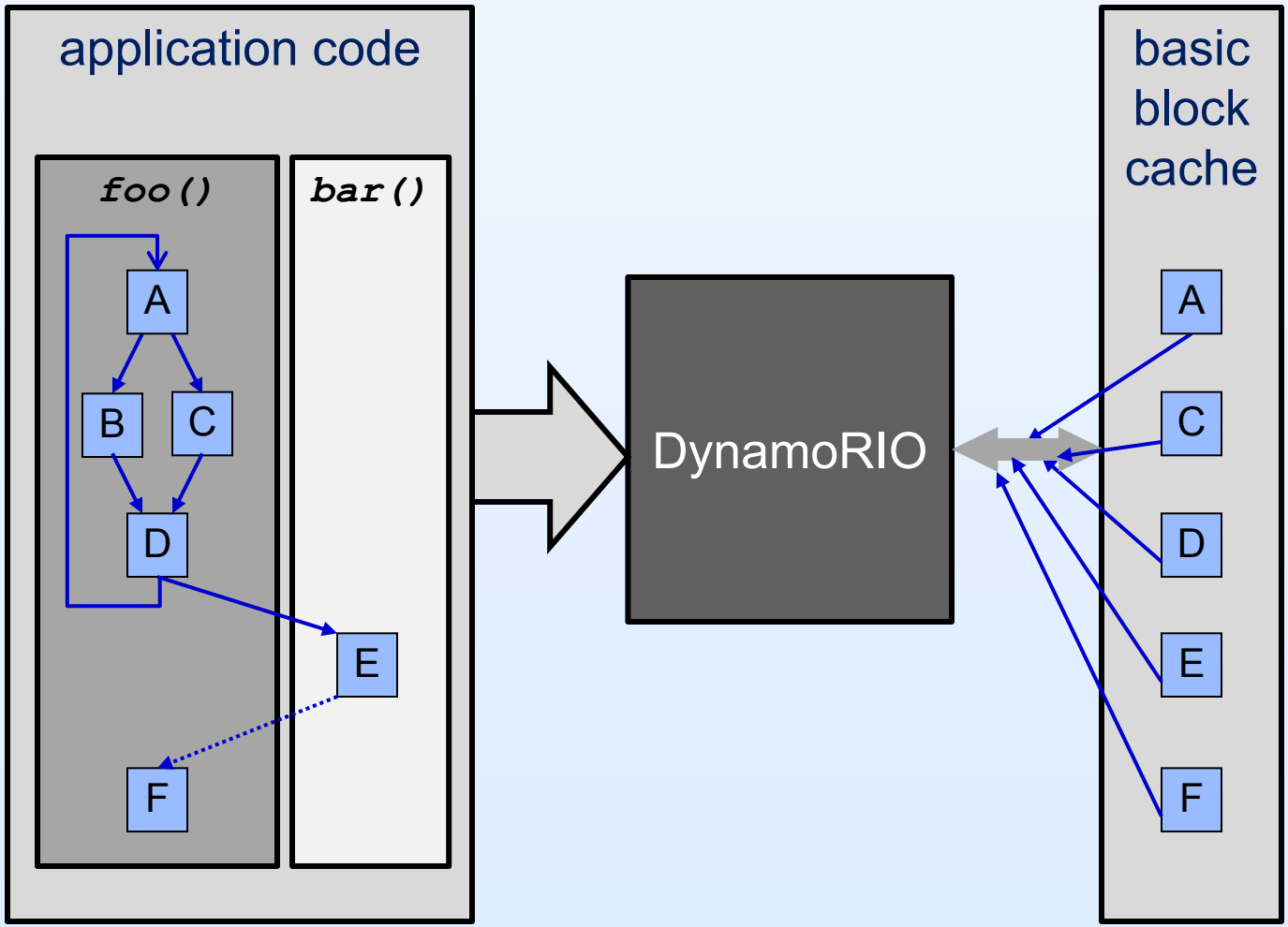
- Avoid transparency and granularity limitations of directly modifying application code
- Allow arbitrary modifications at unrestricted points in code stream
- Allow systematic, fine-grained modifications to code stream
- Guarantee that all code is observed

Basic Interpreter



Slowdown: 300x

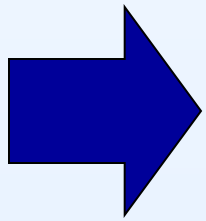
Improvement #1: Basic Block Cache



Slowdown: ~~300x~~ 25x

Example Basic Block Fragment

```
add   %eax, %ecx
cmp   $4, %eax
jle   $0x40106f
```

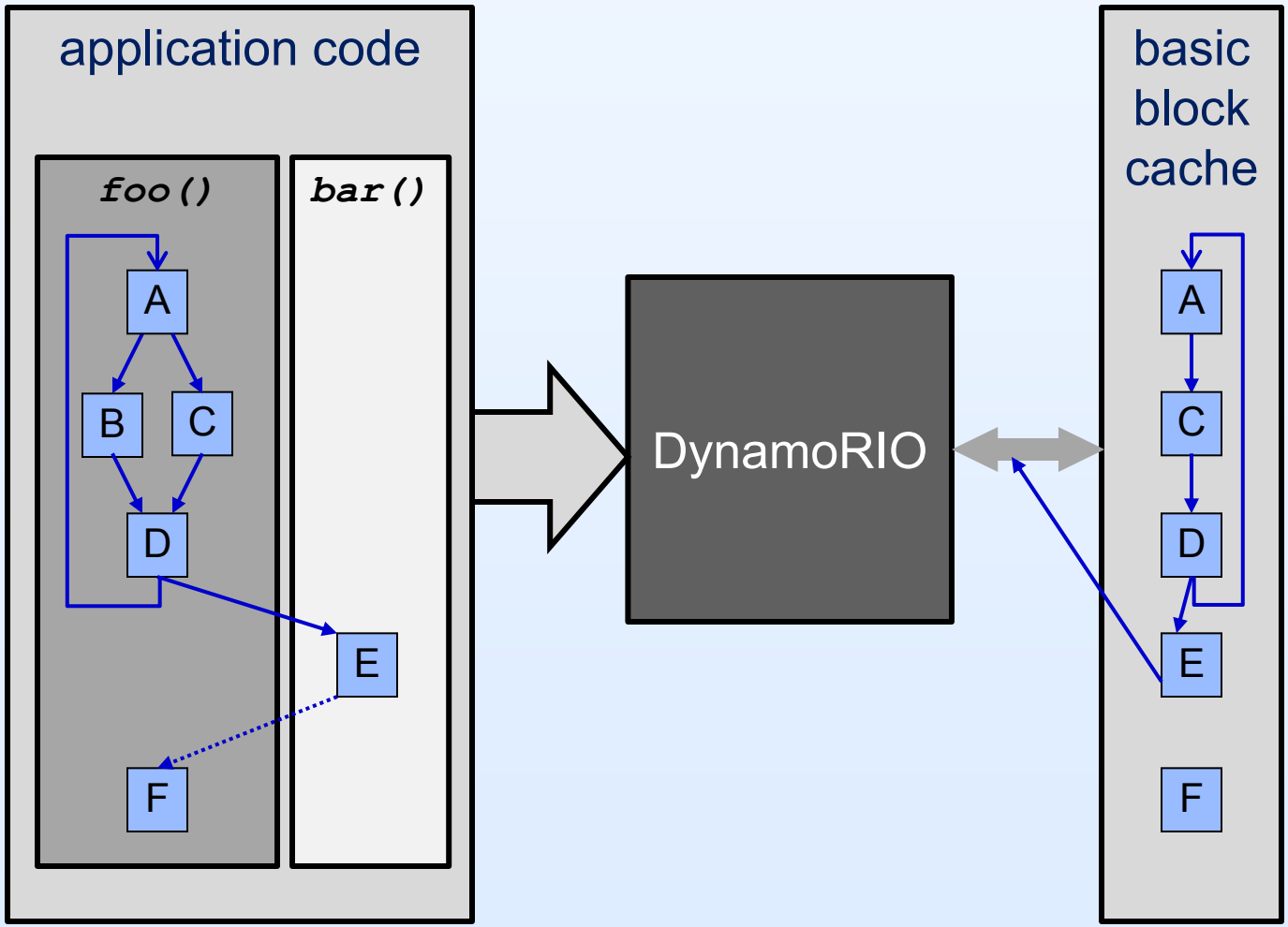


```
frag7: add   %eax, %ecx
       cmp   $4, %eax
       jle   <stub0>
       jmp   <stub1>
stub0: mov   %eax, eax-slot
       mov   &dstub0, %eax
       jmp   context_switch
stub1: mov   %eax, eax-slot
       mov   &dstub1, %eax
       jmp   context_switch
```

```
dstub0
target: 0x40106f
```

```
dstub1
target: fall-thru
```

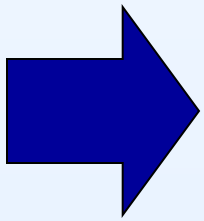
Improvement #2: Linking Direct Branches



Slowdown: ~~300x~~ ~~25x~~ 3x

Direct Linking

```
add   %eax, %ecx
cmp   $4, %eax
jle   $0x40106f
```

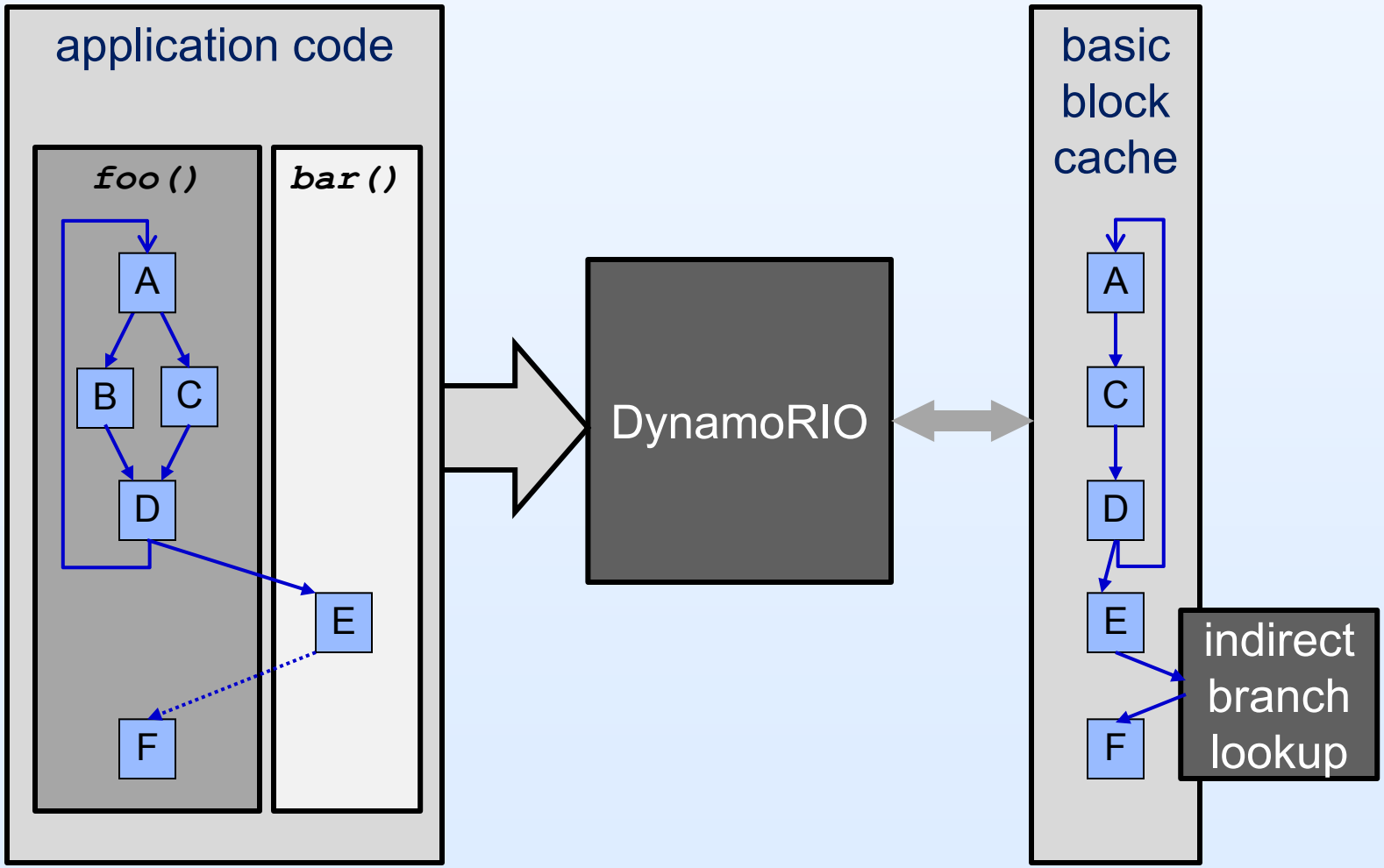


```
frag7: add   %eax, %ecx
       cmp   $4, %eax
       jle   <frag8>
       jmp   <stub1>
stub0: mov   %eax, eax-slot
       mov   &dstub0, %eax
       jmp   context_switch
stub1: mov   %eax, eax-slot
       mov   &dstub1, %eax
       jmp   context_switch
```

dstub0
target: 0x40106f

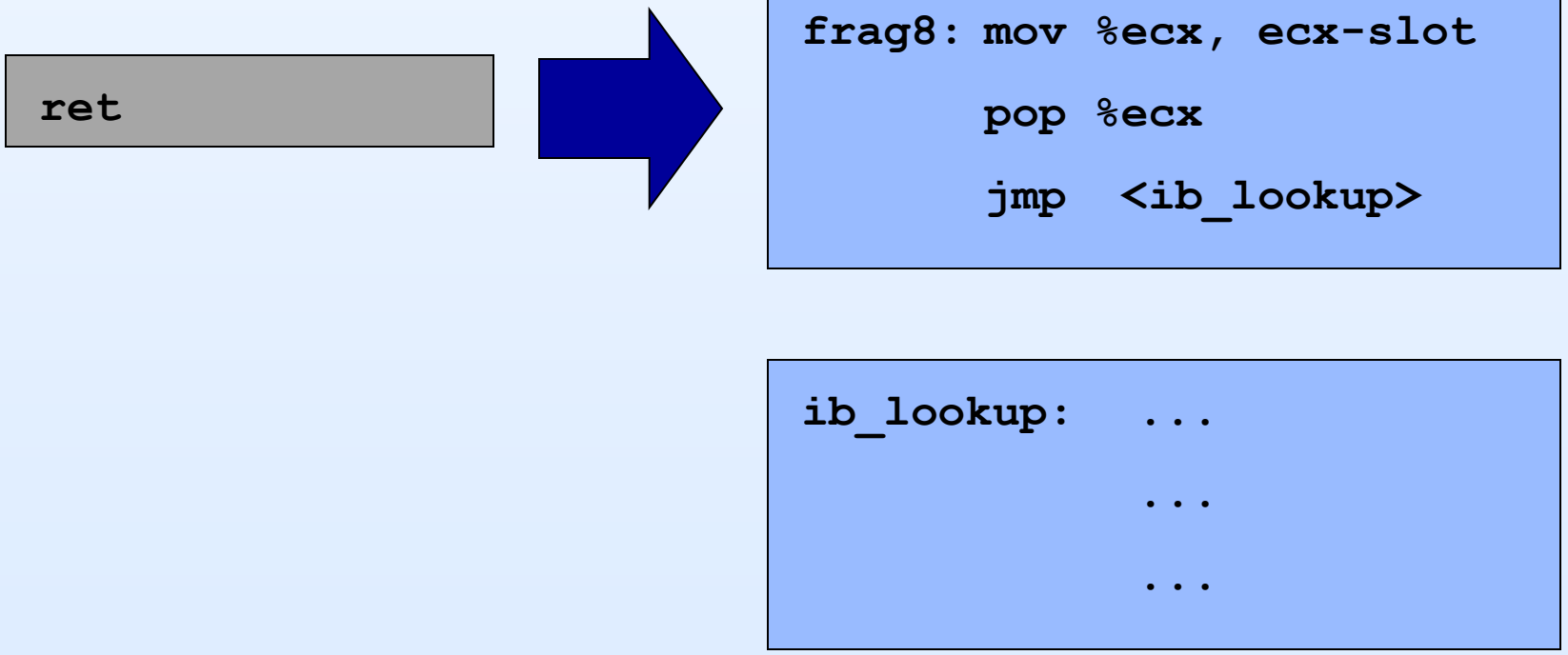
dstub1
target: fall-thru

Improvement #3: Linking Indirect Branches

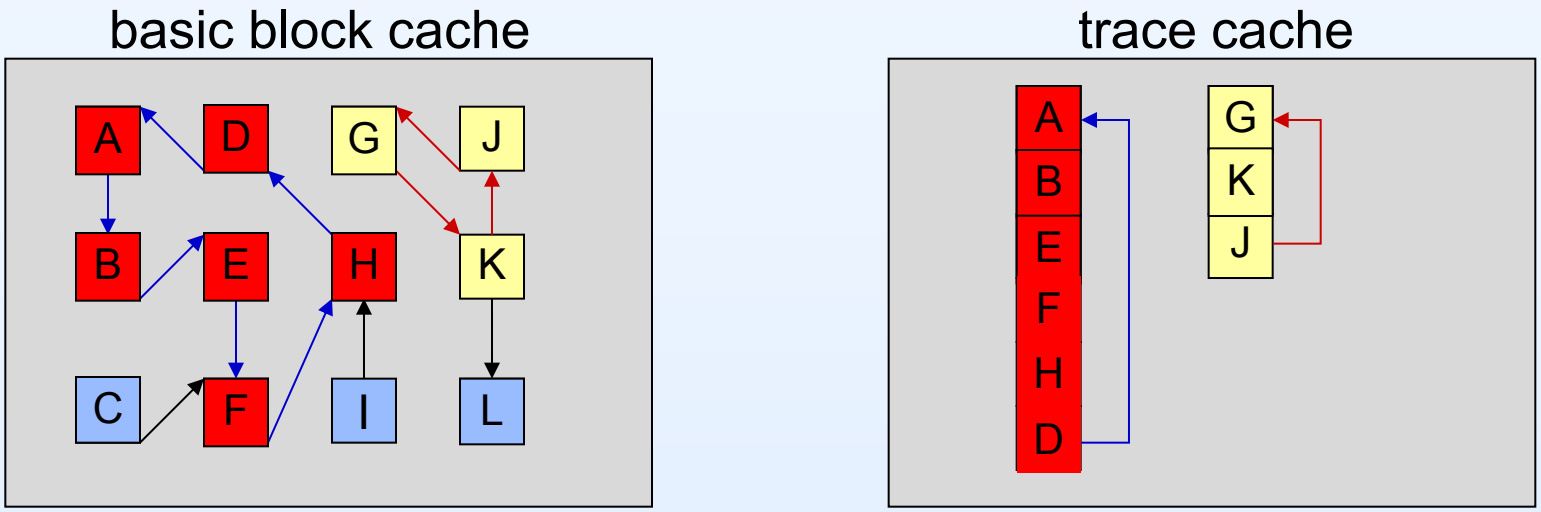


Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ 1.2x

Indirect Branch Transformation

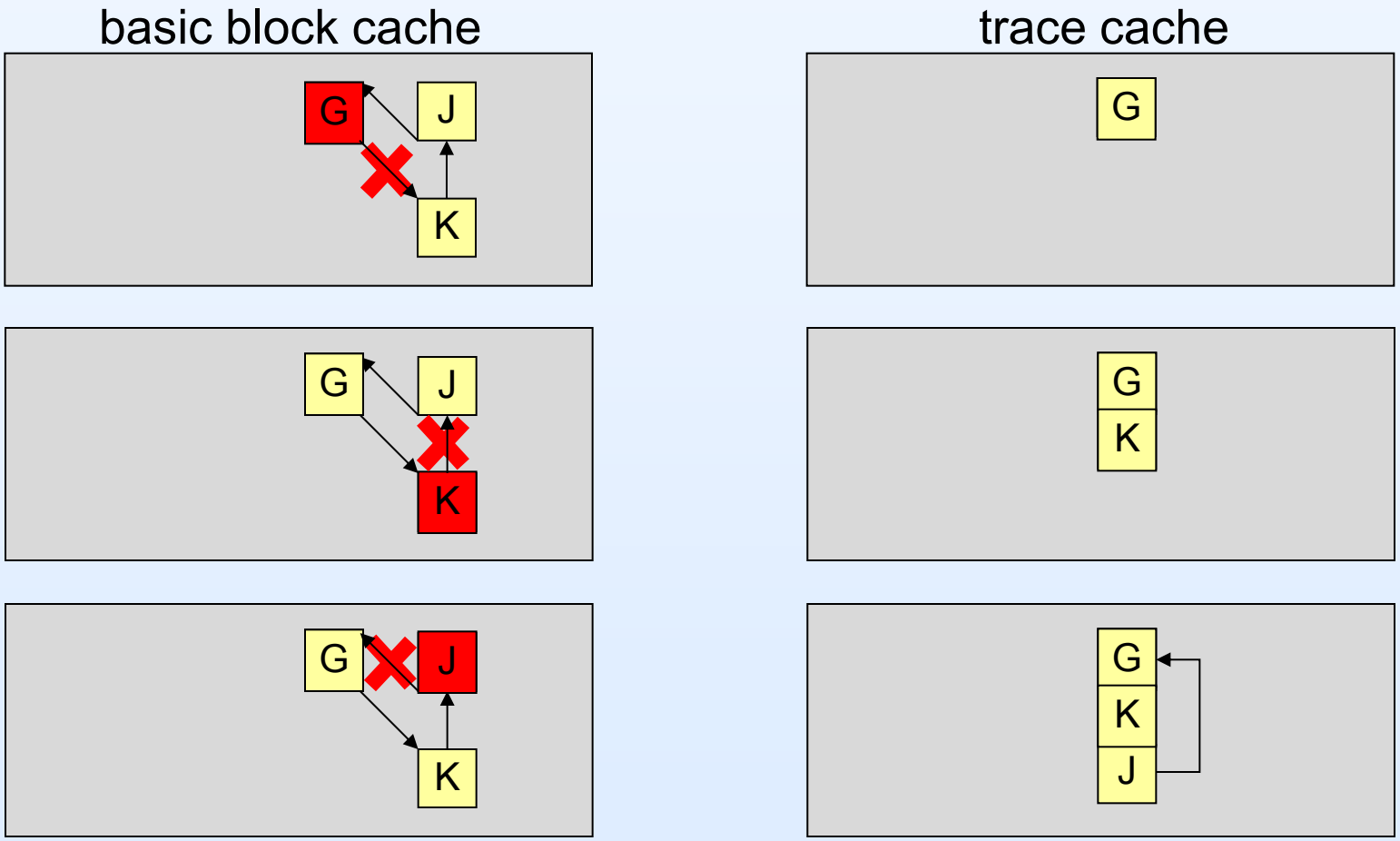


Improvement #4: Trace Building

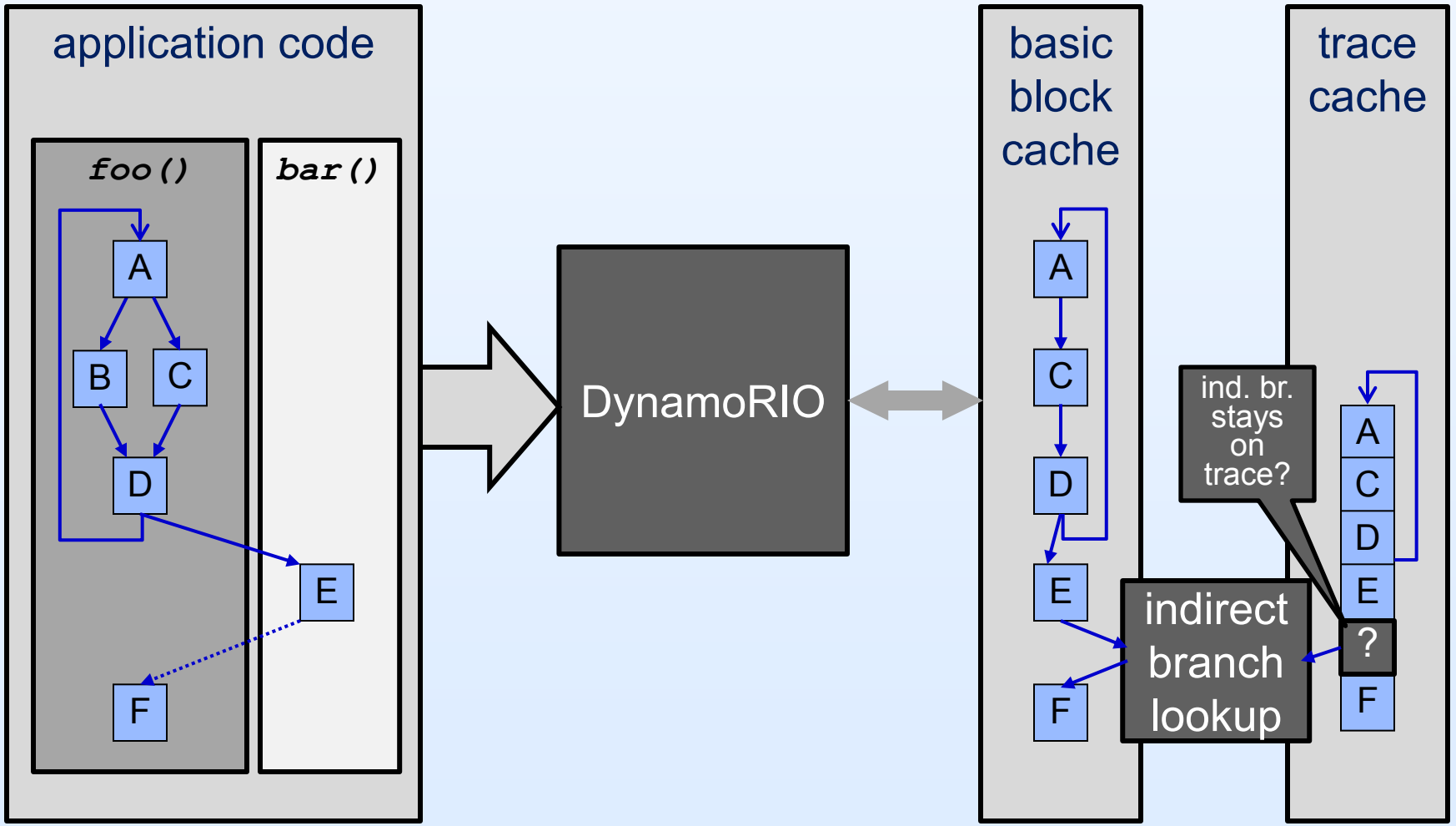


- Traces reduce branching, improve layout and locality, and facilitate optimizations across blocks
  - We avoid indirect branch lookup
- Next Executing Tail (NET) trace building scheme [Duesterwald 2000]

Incremental NET Trace Building

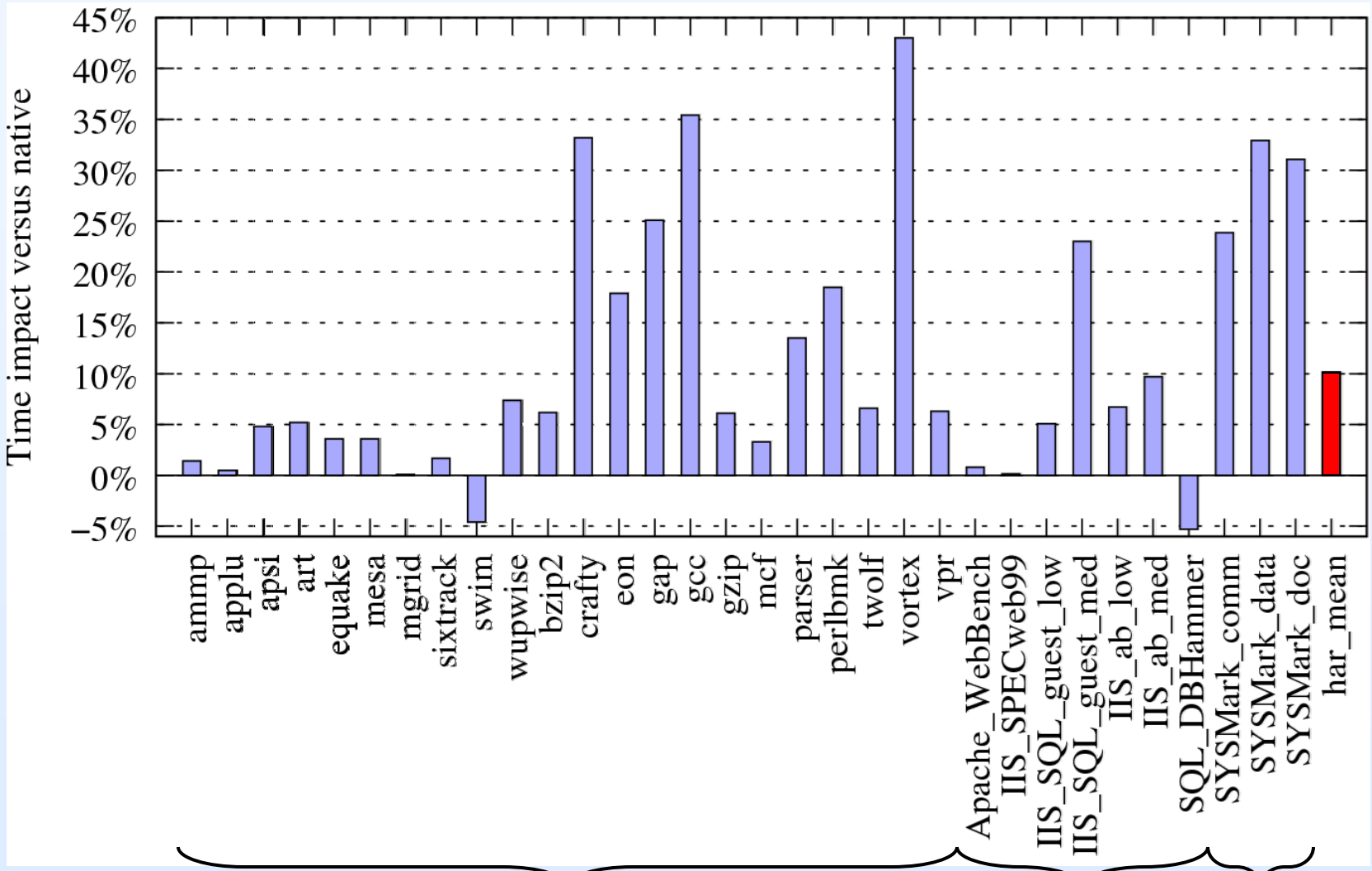


Improvement #4: Trace Building

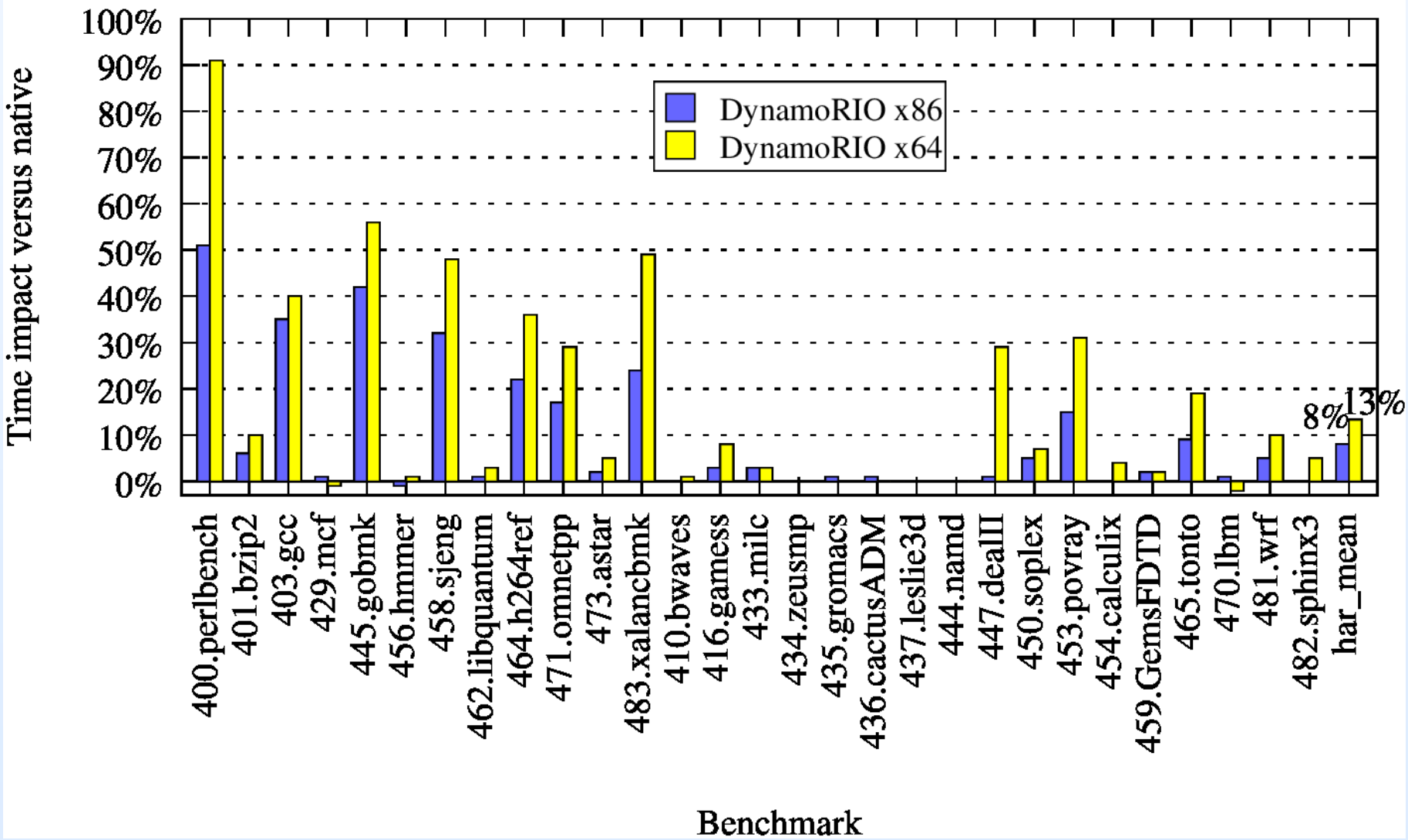


Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ ~~1.2x~~ 1.1x

Base Performance



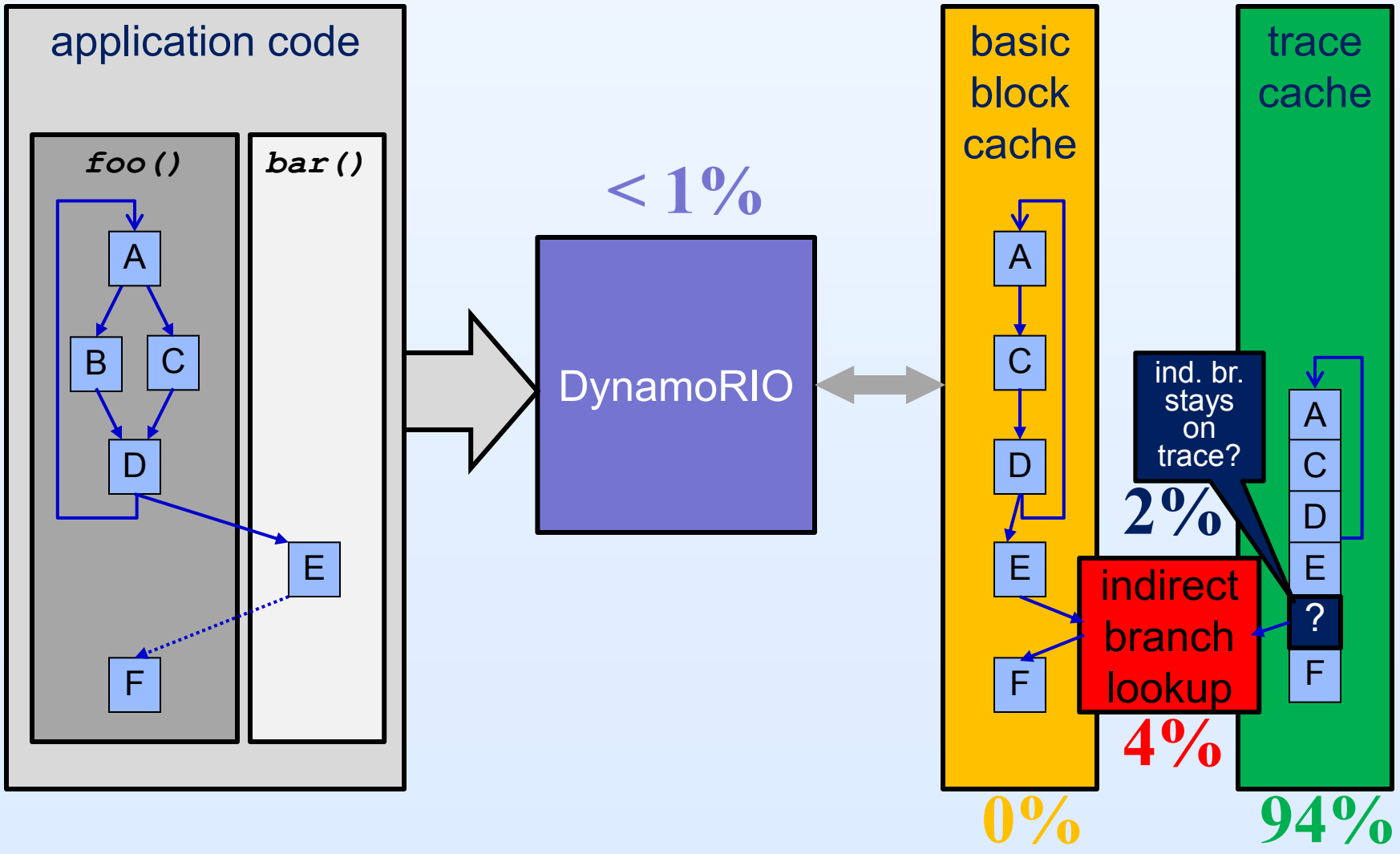
Base Performance: SPEC 2006



Sources of Overhead

- Extra instructions
  - Indirect branch target comparisons
  - Indirect branch hashtable lookups
- Extra data cache pressure
  - Indirect branch hashtable
- Branch mispredictions
  - ret becomes jmp*
- Application code modification

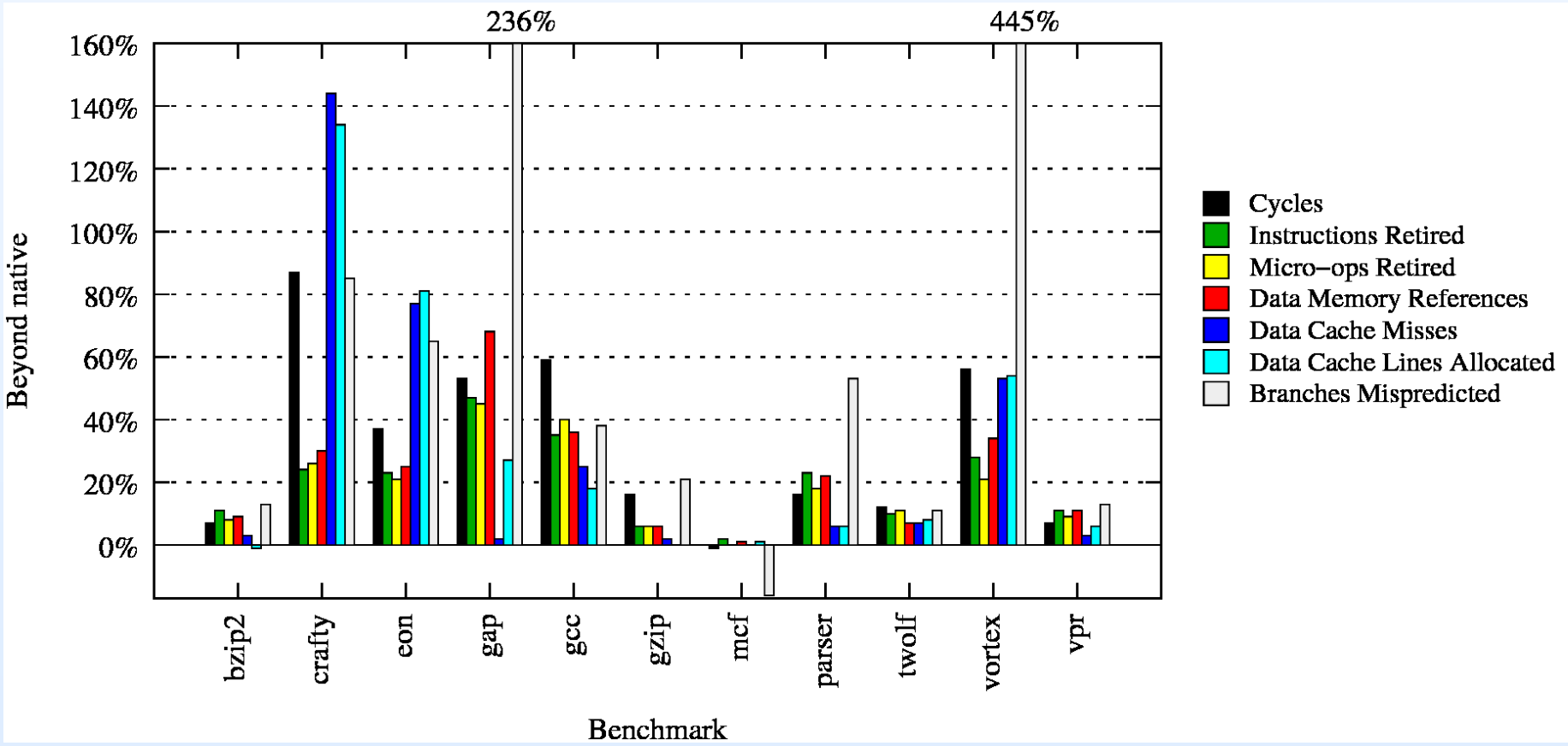
Time Breakdown for SPEC CPU INT



Not An Ordinary Application

- An application executing in DynamoRIO's code cache looks different from what the underlying hardware has been tuned for
- The hardware expects:
  - Little or no dynamic code modification
    - Writes to code are expensive
  - call and ret instructions
    - Return Stack Buffer predictor

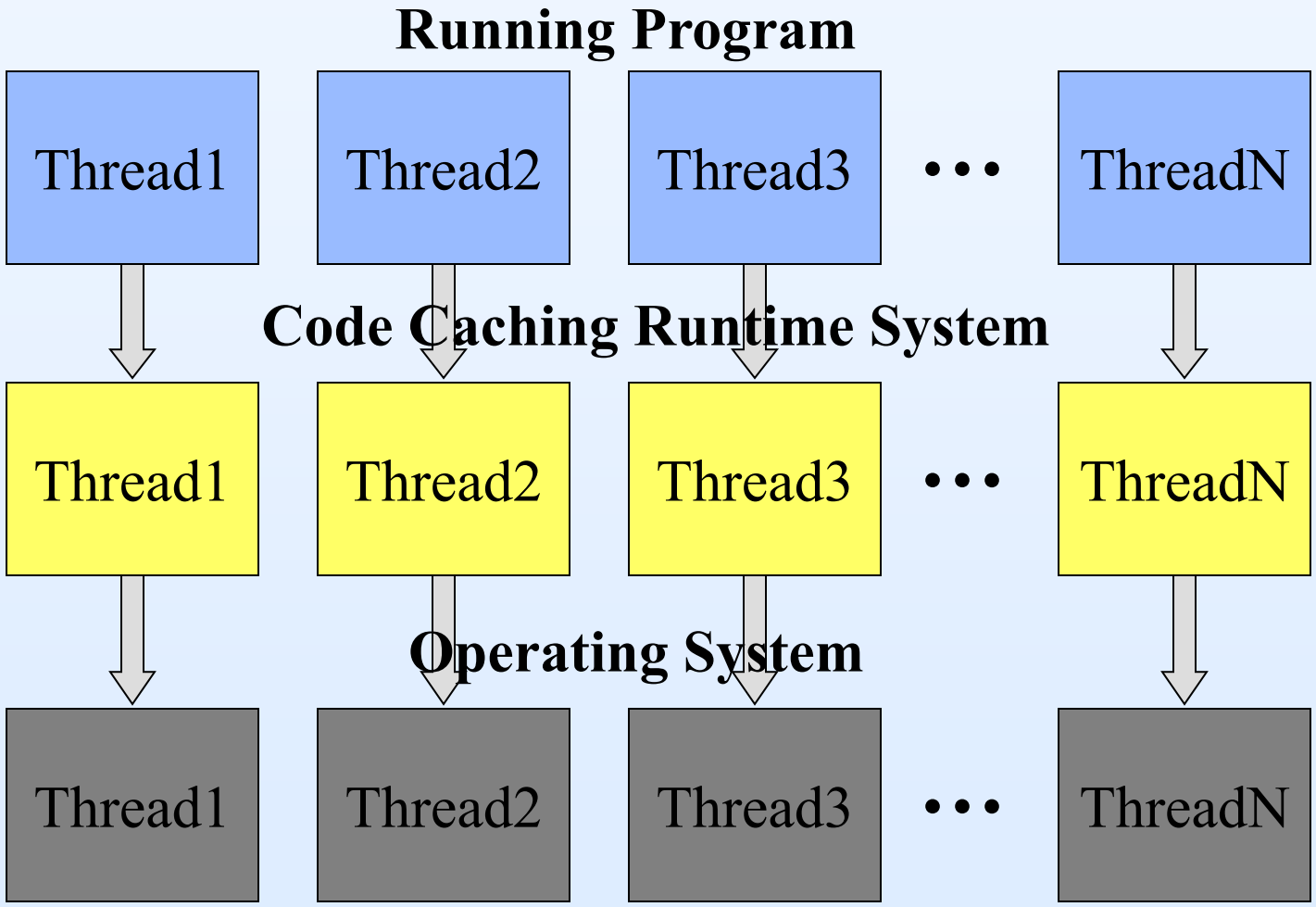
Performance Counter Data



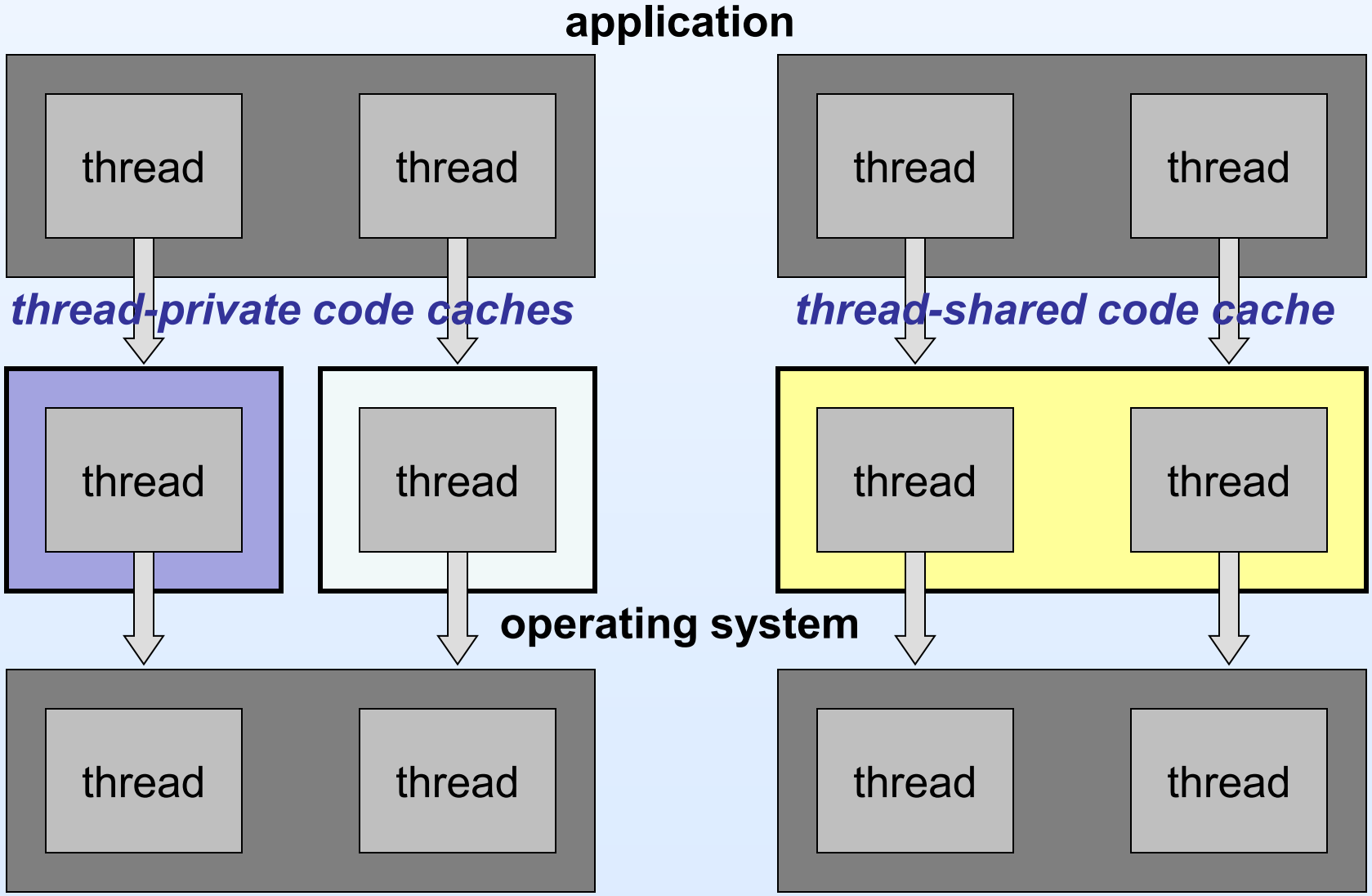
Overview Outline

- Efficient
  - Software code cache overview
  - Thread-shared code cache
- Transparent
- Comprehensive
- Customizable

Threading Model



Code Cache Threading Models



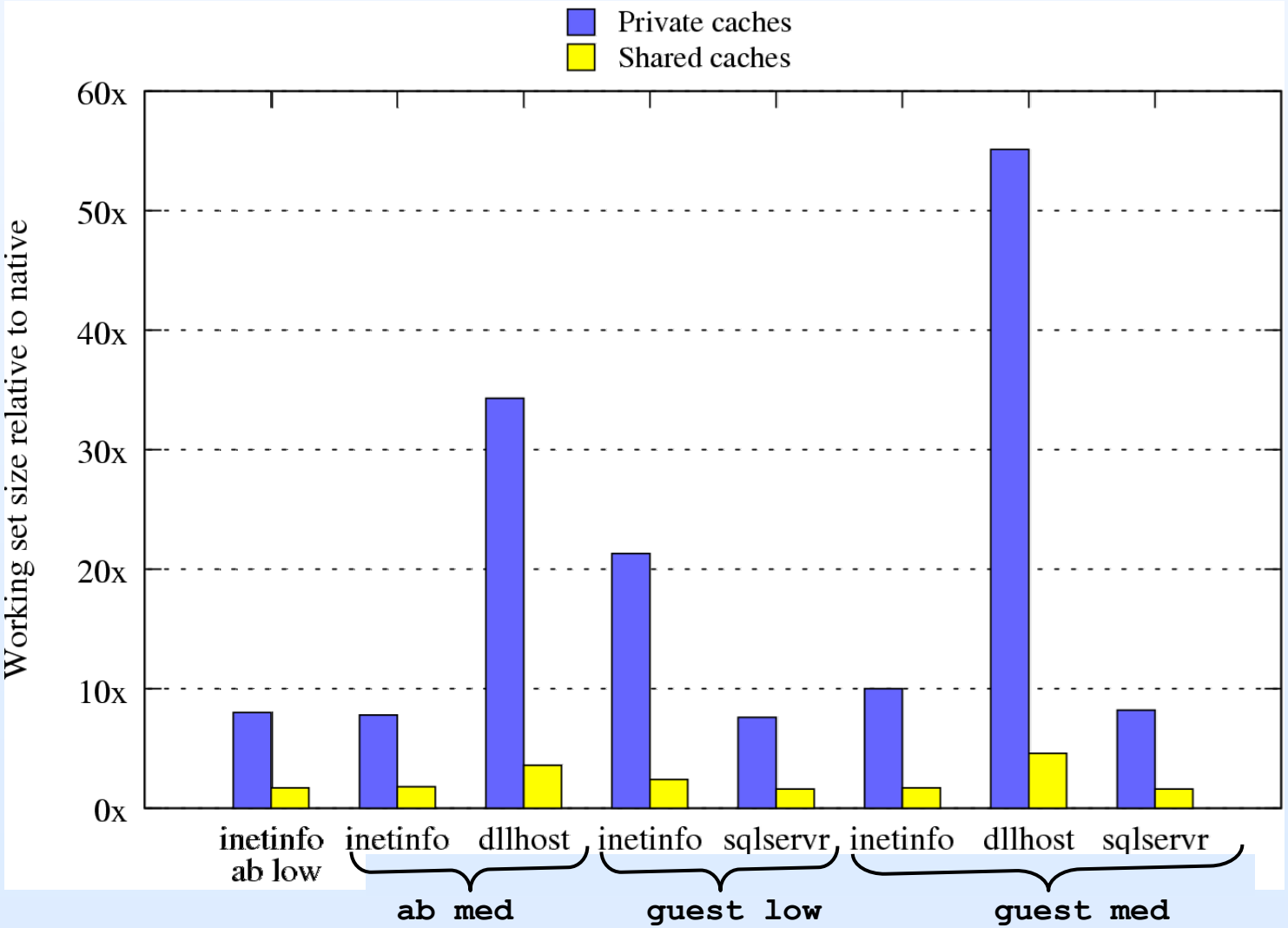
Thread-Private versus Thread-Shared

- Thread-private
  - Less synchronization needed
  - Absolute addressing for thread-local storage
  - Thread-specific optimization and instrumentation
- Thread-shared
  - Scales to many-threaded apps

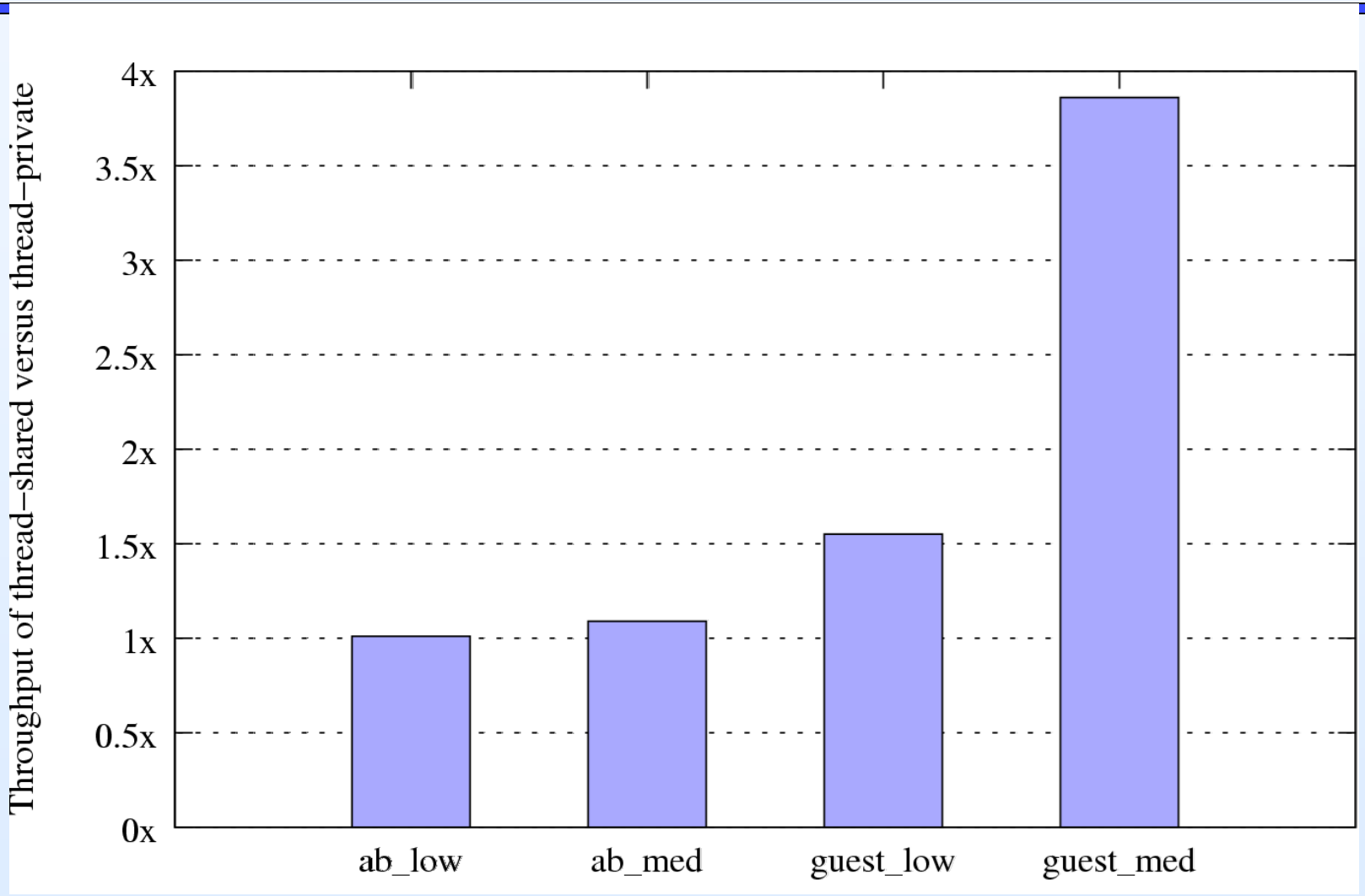
Database and Web Server Suite

Benchmark	Server	Processes
ab low	IIS low isolation	inetinfo.exe
ab med	IIS medium isolation	inetinfo.exe, dllhost.exe
guest low	IIS low isolation, SQL Server 2000	inetinfo.exe, sqlservr.exe
guest med	IIS medium isolation, SQL Server 2000	inetinfo.exe, dllhost.exe, sqlservr.exe

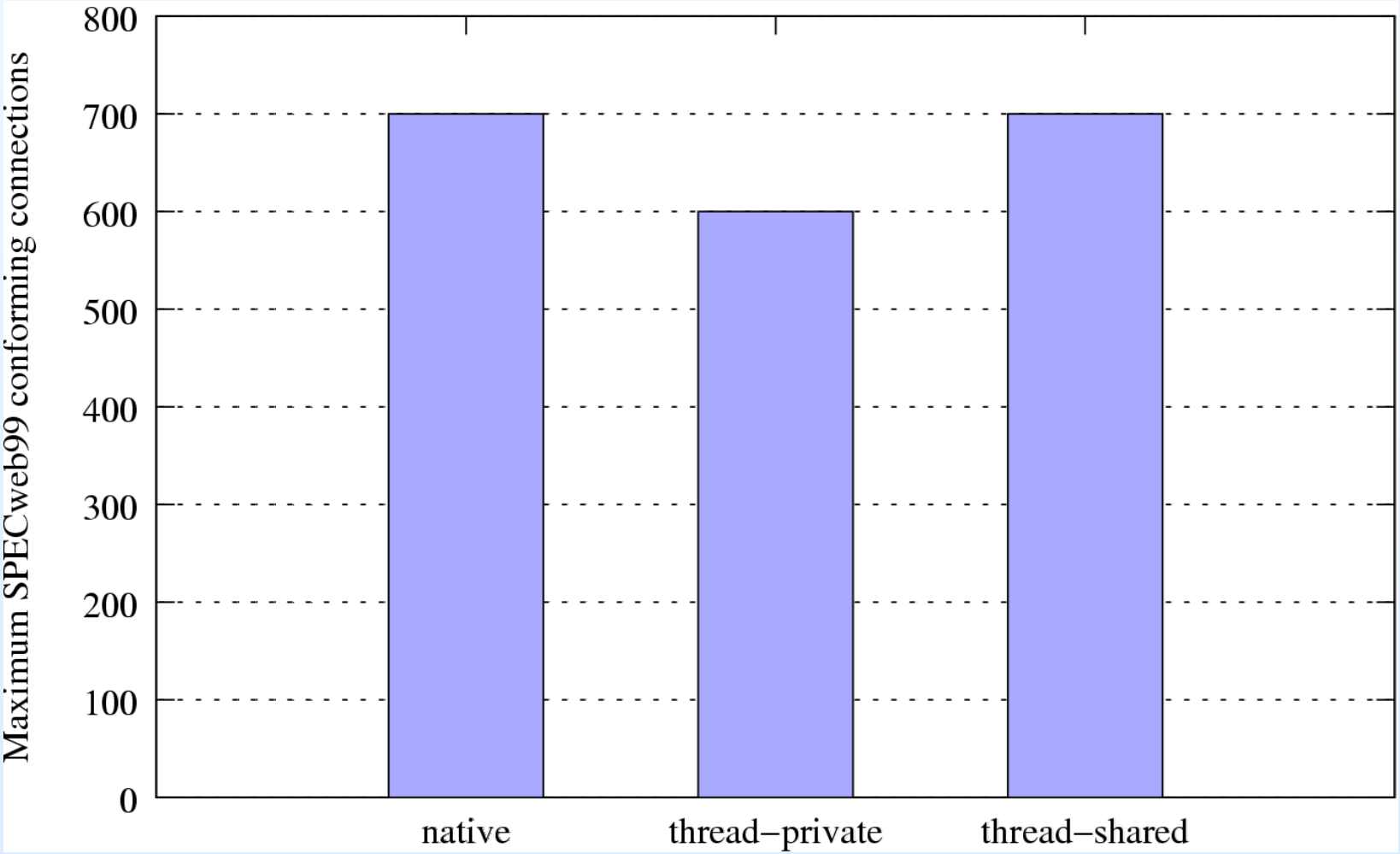
Memory Impact



Performance Impact



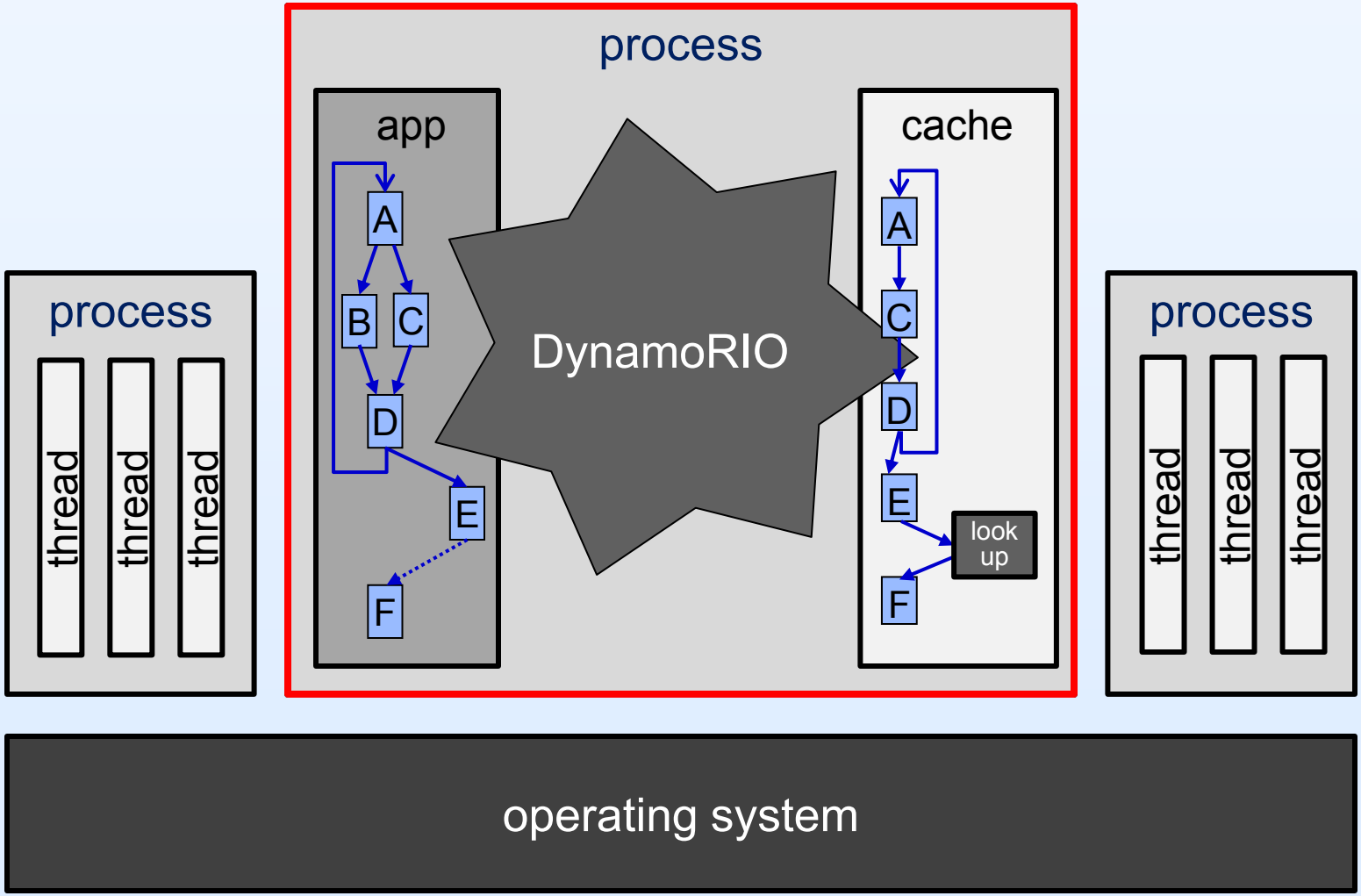
Scalability Limit



Overview Outline

- Efficient
- Transparent
  - Transparency principles
  - Cache consistency
  - Synchronization
- Comprehensive
- Customizable

Unavoidably Intrusive



Transparency

- Do not want to interfere with the semantics of the program
- Dangerous to make any assumptions about:
  - Register usage
  - Calling conventions
  - Stack layout
  - Memory/heap usage
  - I/O and other system call use

Painful, But Necessary

- Difficult and costly to handle corner cases
- Many applications will not notice…
- …but some will!
  - Microsoft Office: Visual Basic generated code, stack convention violations
  - COM, Star Office, MMC: trampolines
  - Adobe Premiere: self-modifying code
  - VirtualDub: UPX-packed executable
  - etc.

Transparency Principles

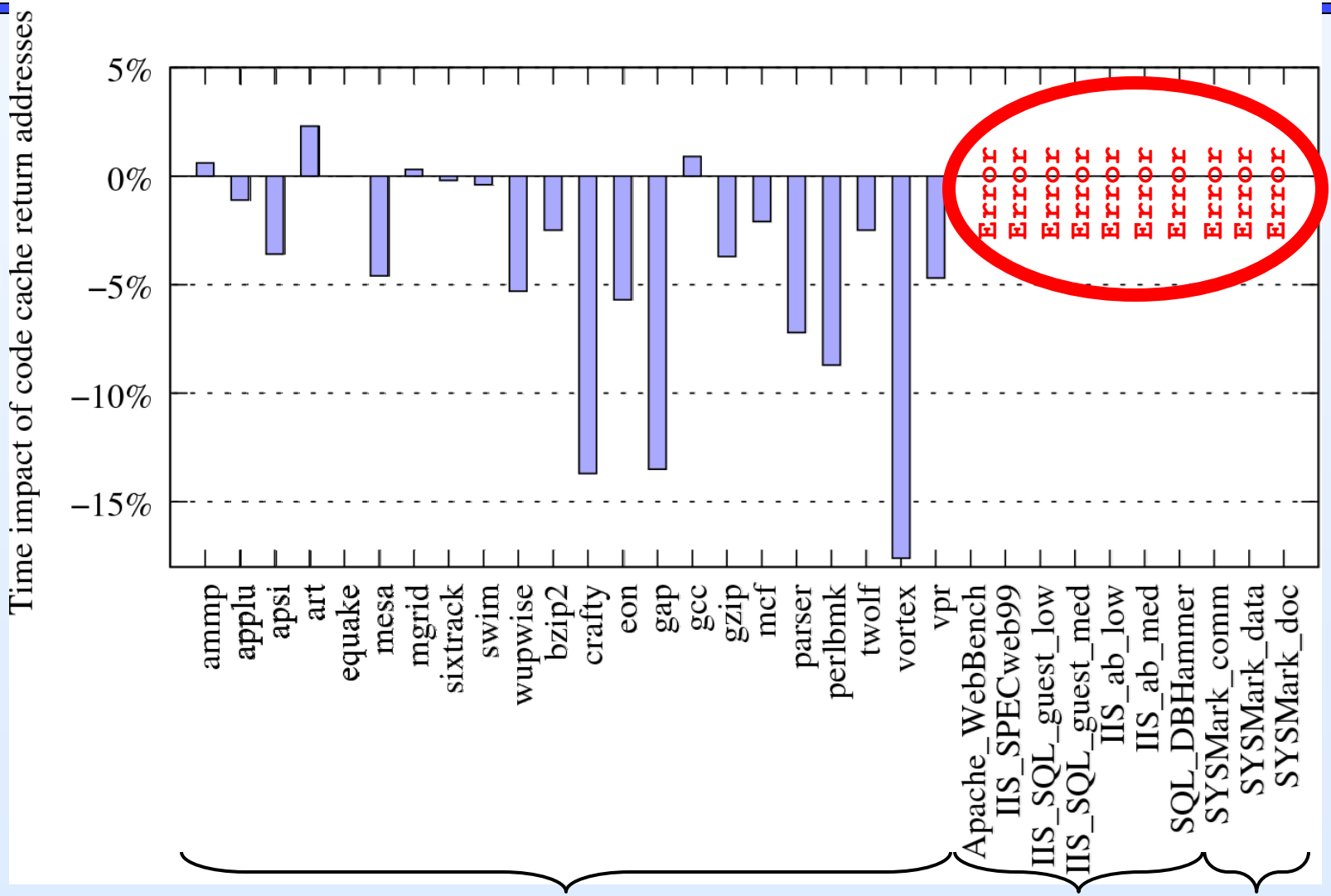
- **Principle 1: *As few changes as possible***
  - Set a high bar for value before changing the native environment
- **Principle 2: *Hide necessary changes***
  - Whatever is valuable enough to change must be hidden
  - Changes that cannot be hidden should not be made
- **Principle 3: *Separate resources***
  - Avoid intra-process resource conflicts

Bruening et al. "Transparent Dynamic Instrumentation" VEE'12

Principle 1: *As few changes as possible*

- Application code
- Executable on disk
- Stored addresses
- Threads
- Application data
  - Including the stack!

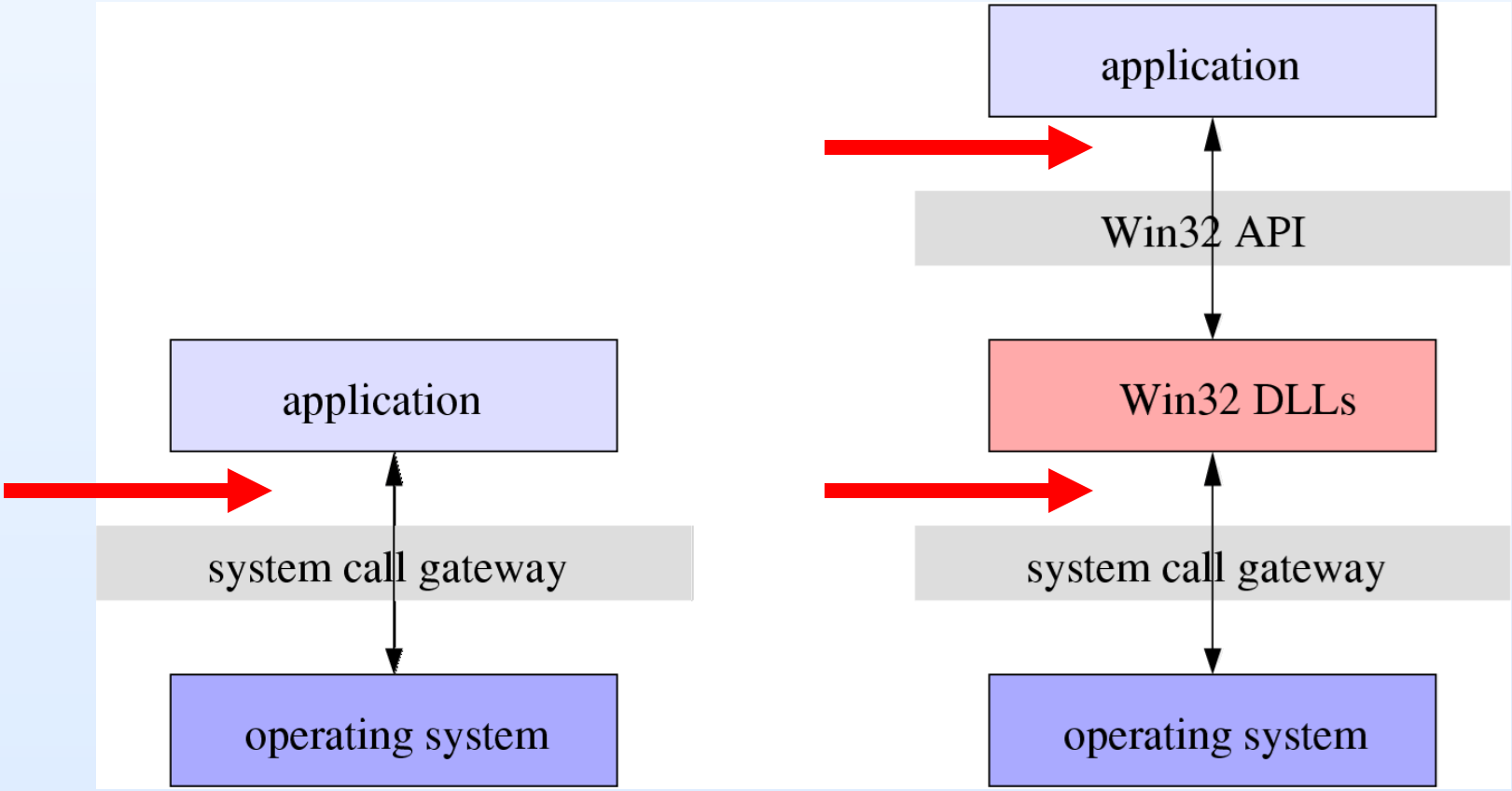
Return Address Transparency



Principle 2: *Hide necessary changes*

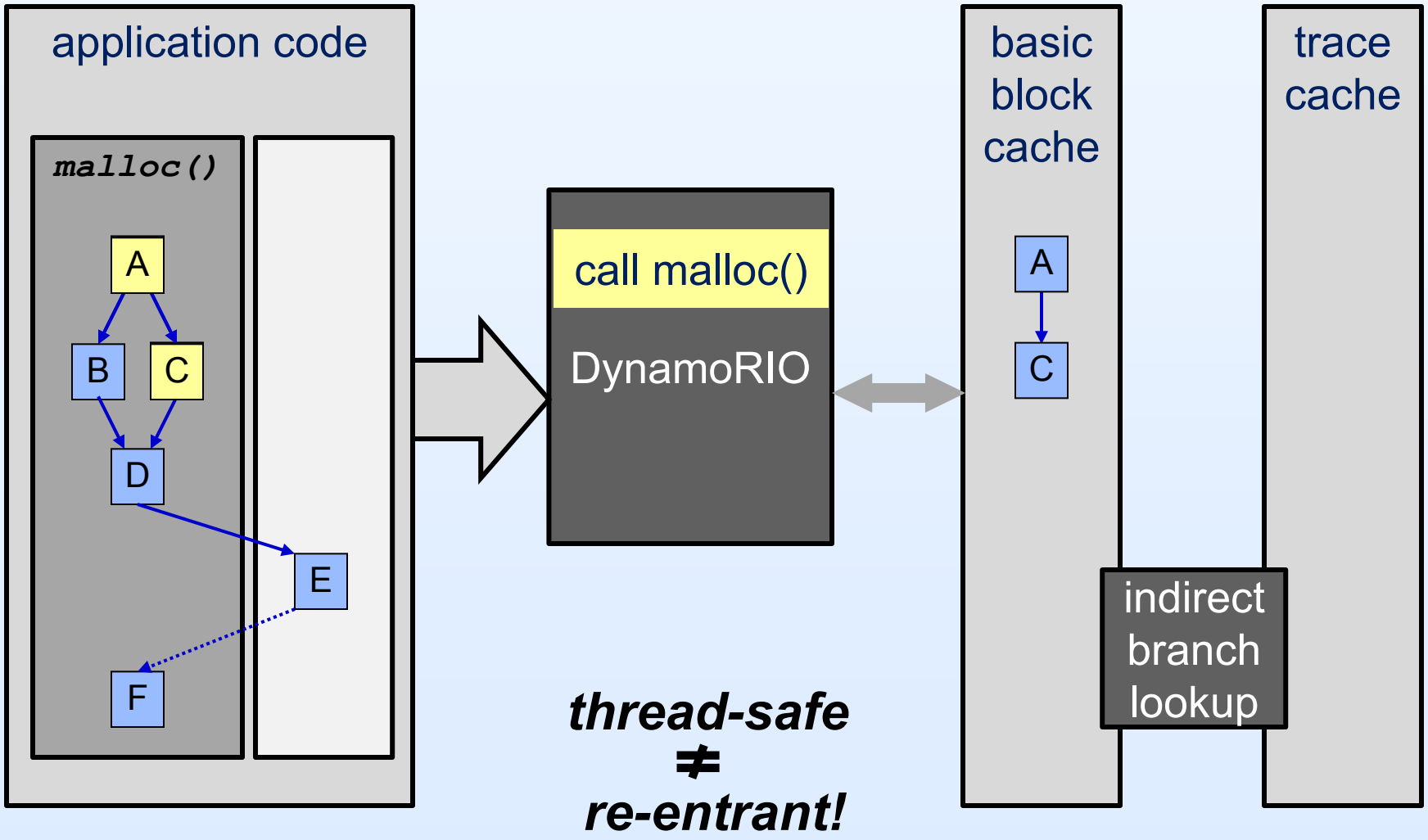
- Application addresses
- Address space
- Error transparency
- Code cache consistency

Principle 3: *Separate resources*

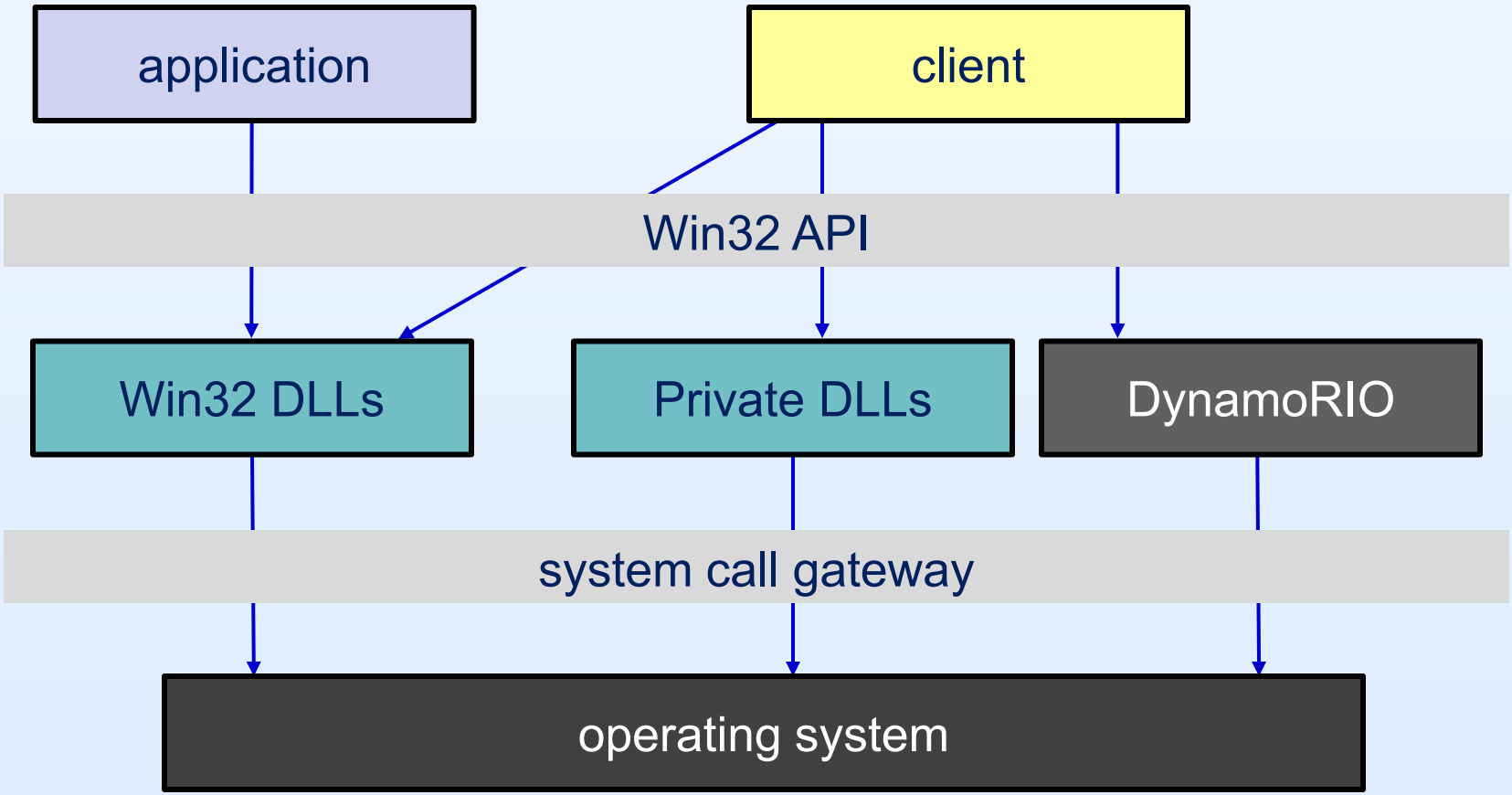


Linux ≠ Windows

Arbitrary Interleaving



Private Libraries



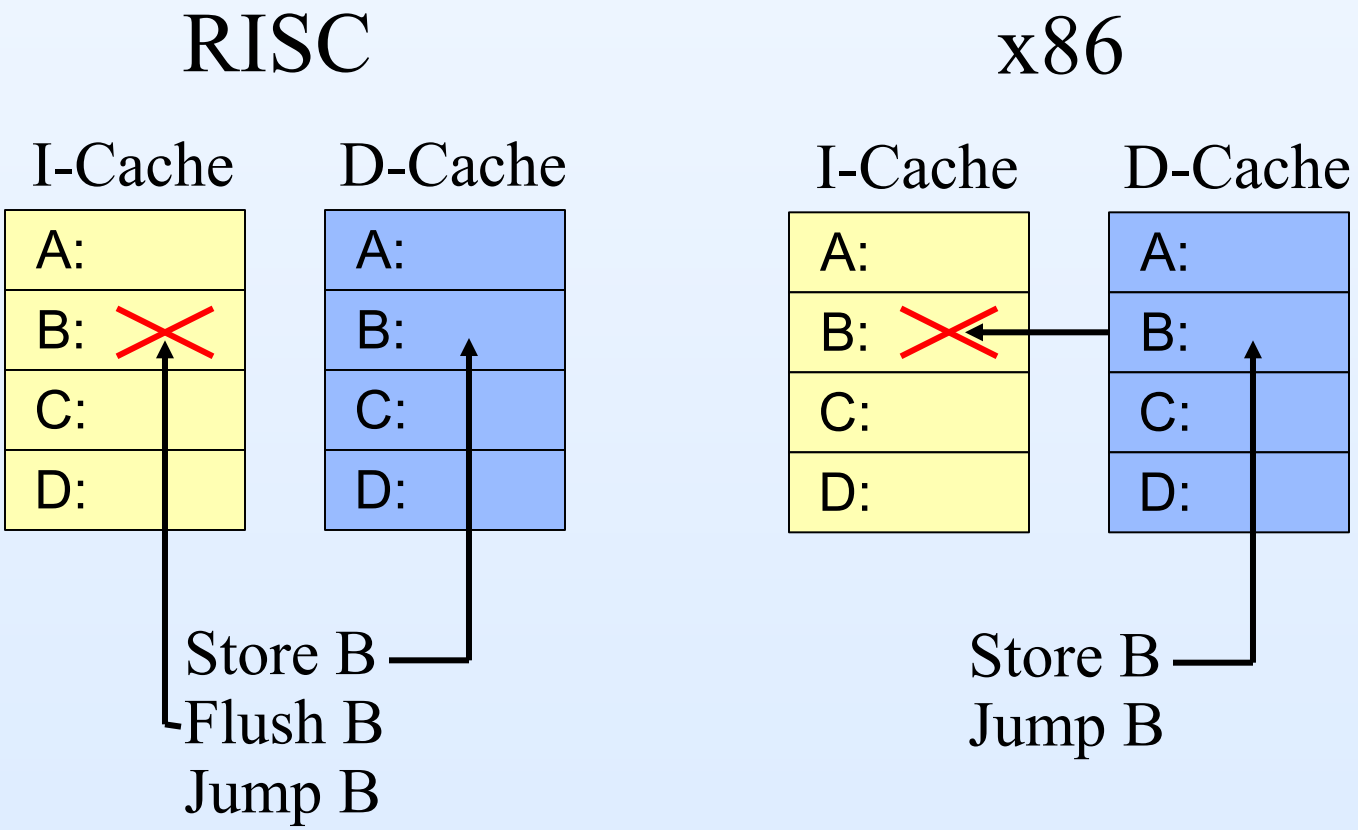
Transparency Landscape

	Principle 1: ***As few changes as possible***	**Principle 2:** ***Hide necessary changes***	**Principle 3:** ***Separate resources***
Code	application code, stored addresses	machine context, cache consistency	
Data	stack, heap, registers, condition flags		separate stack, heap, context, i/o
Concurrency	threads, memory ordering		disjoint locks
Other		preserve errors	

Overview Outline

- Efficient
- Transparent
  - Transparency principles
  - Cache consistency
  - Synchronization
- Comprehensive
- Customizable

Code Change Mechanisms



How Often Does Code Change?

- Not just modification of code!
- Removal of code
  - Shared library unloading
- Replacement of code
  - JIT region re-use
  - Trampoline on stack

Code Change Events

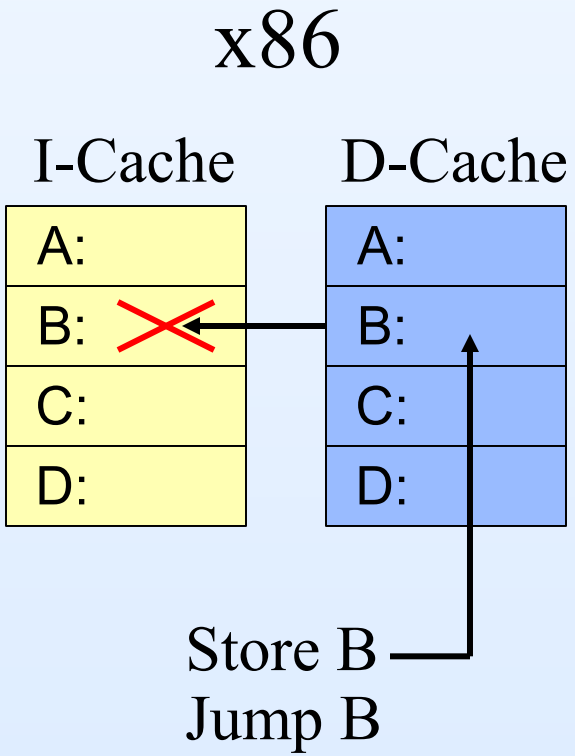
	Memory Unmappings	Generated Code Regions	Modified Code Regions
SPECFP	112	0	0
SPECINT	29	0	0
SPECJVM	7	3373	4591
Excel	144	21	20
Photoshop	1168	40	0
Powerpoint	367	28	33
Word	345	20	6

Detecting Code Removal

- Example: shared library being unloaded
- Requires explicit request by application to operating system
- Detect by monitoring system calls (munmap, NtUnmapViewOfSection)

Detecting Code Modification

- On x86, no explicit app request required, as the icache is kept consistent in hardware – so any memory write could modify code!



Page Protection Plus Instrumentation

- Invariant: application code copied to code cache must be read-only
  - If writable, hide read-only status from application
- Some code cannot or should not be made read-only
  - Self-modifying code
  - Windows stack
  - Code on a page with frequently written data
- Use per-fragment instrumentation to ensure code is not stale on entry and to catch self-modification

Adaptive Consistency Algorithm

- Use page protection by default
  - Most code regions are always read-only
- Subdivide written-to regions to reduce flushing cost of write-execute cycle
  - Large read-only regions, small written-to regions
- Switch to instrumentation if write-execute cycle repeats too often (or on same page)
  - Switch back to page protection if writes decrease

Bruening et al. "Maintaining Consistency and Bounding Capacity of Software Code Caches" CGO'05

Overview Outline

- Efficient
- Transparent
  - Transparency principles
  - Cache consistency
  - Synchronization
- Comprehensive
- Customizable

Synchronization Transparency

- Application thread management should not interfere with the runtime system, and vice versa
  - Cannot allow the app to suspend a thread holding a runtime system lock
  - Runtime system cannot use app locks

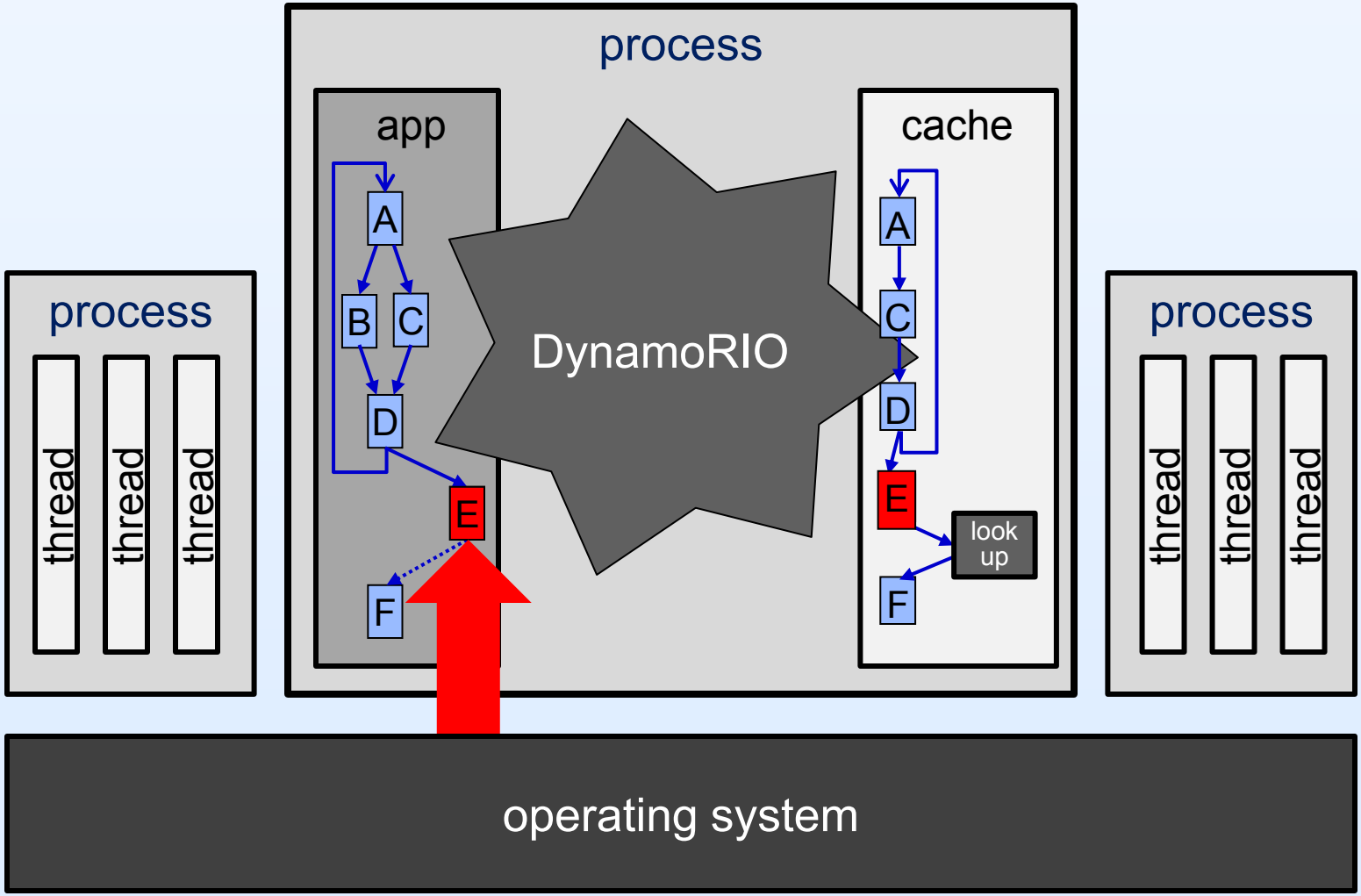
Disjoint Locks

- App thread suspension requires safe spots where no runtime system locks are held
- Time spent in the code cache can be unbounded
- → Our invariant: no runtime system lock can be held while executing in the code cache

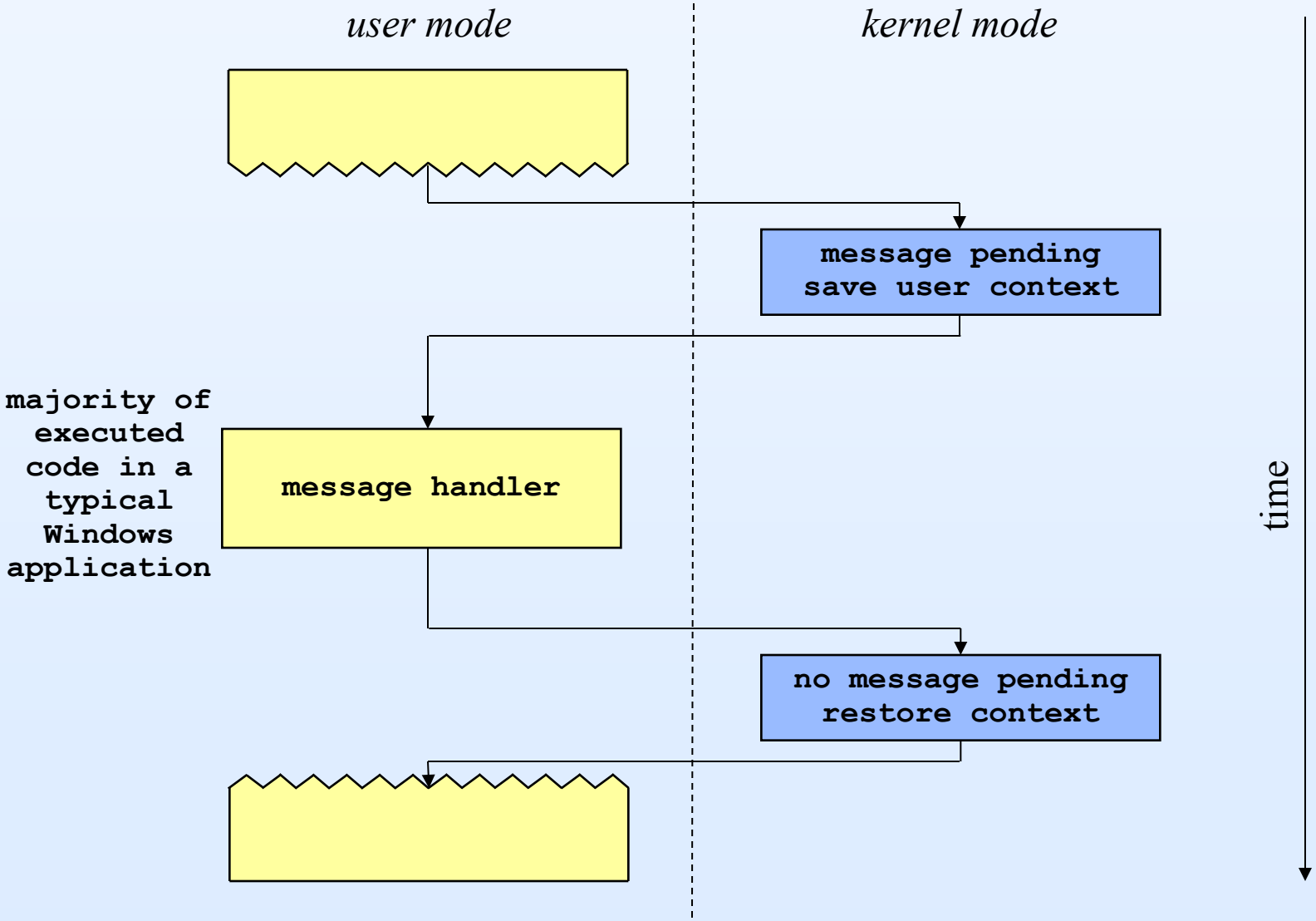
Overview Outline

- Efficient
- Transparent
- Comprehensive
- Customizable

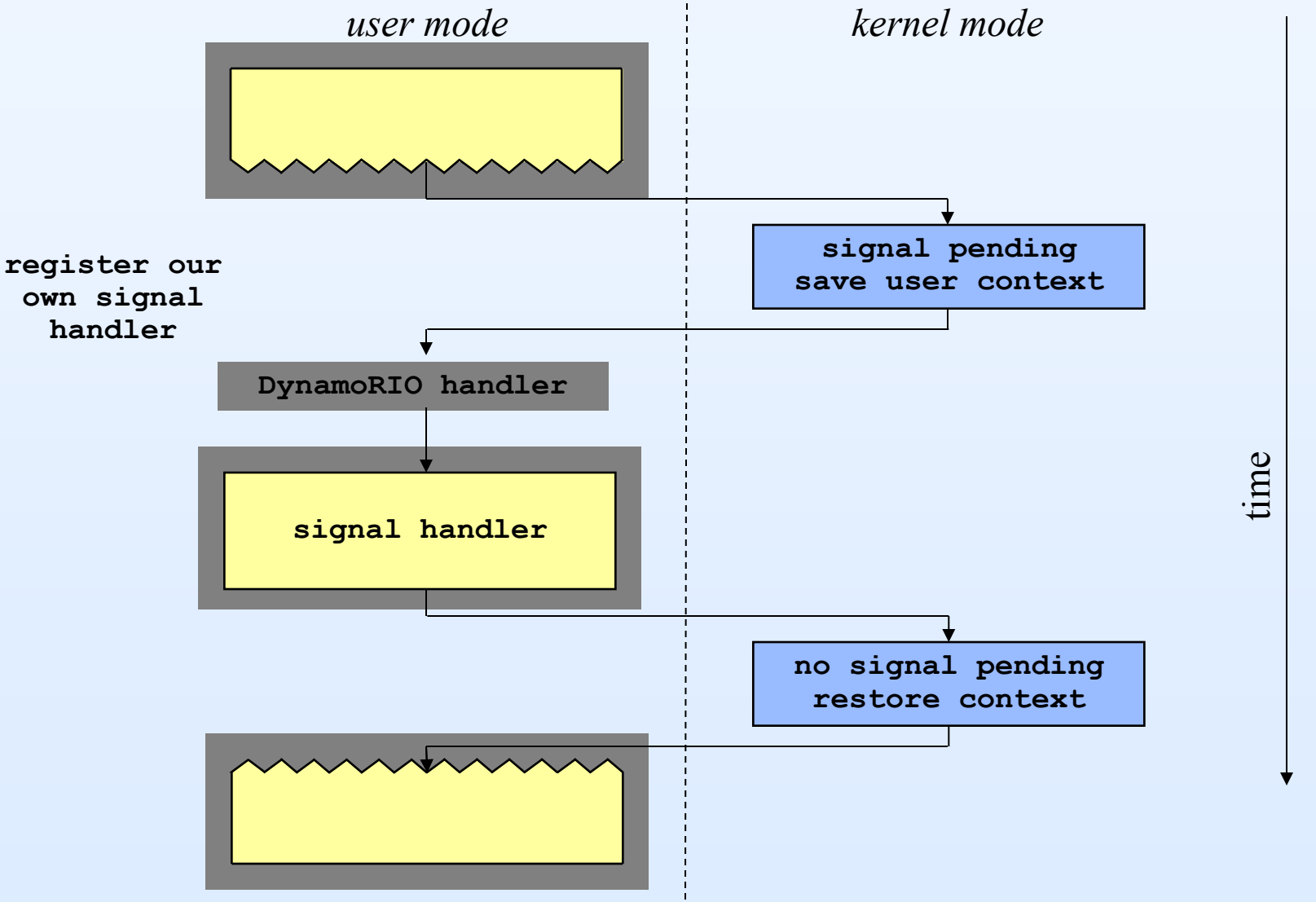
Above the Operating System



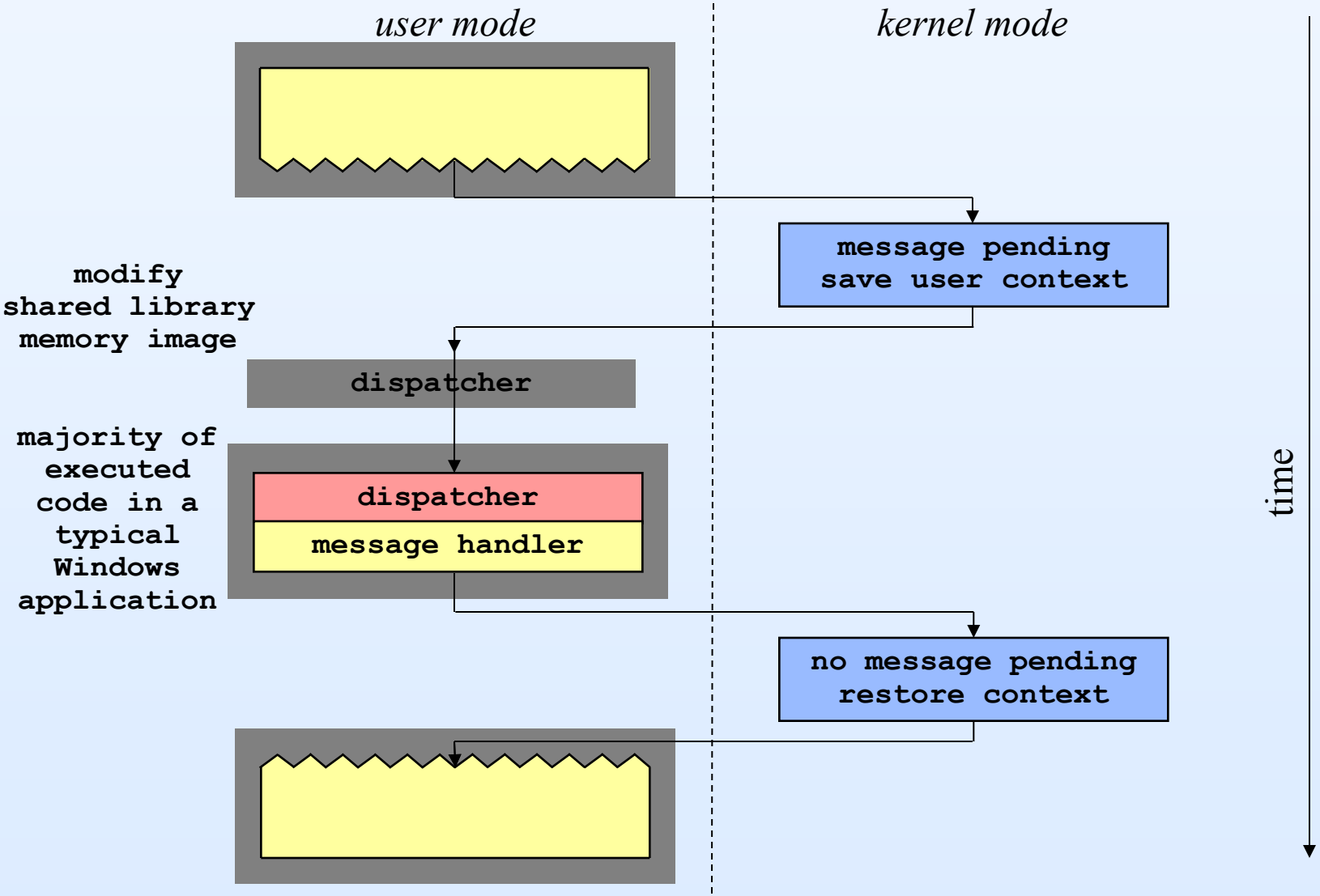
Kernel-Mediated Control Transfers



Intercepting Linux Signals



Intercepting Windows Messages



Must Monitor System Calls

- To maintain control:
  - Calls that affect the flow of control: register signal handler, create thread, set thread context, etc.
- To maintain transparency:
  - Queries of modified state app should not see
- To maintain cache consistency:
  - Calls that affect the address space
- To support cache eviction:
  - Interruptible system calls must be redirected

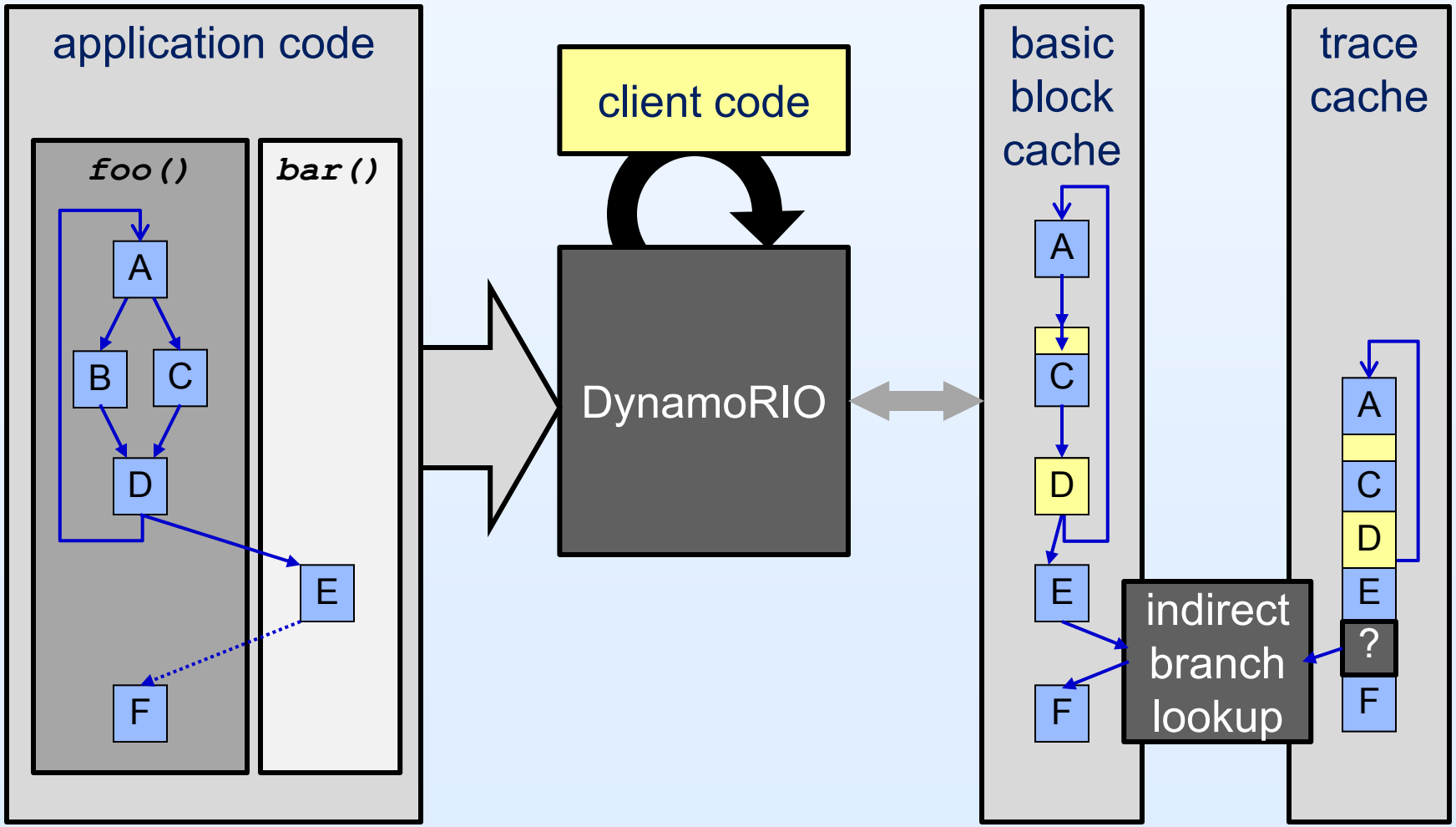
Operating System Dependencies

- System calls and their numbers
  - Monitor application's usage, as well as for our own resource management
  - Windows changes the numbers each major rel
- Details of kernel-mediated control flow
  - Must emulate how kernel delivers events
- Initial injection
  - Once in, follow child processes

Overview Outline

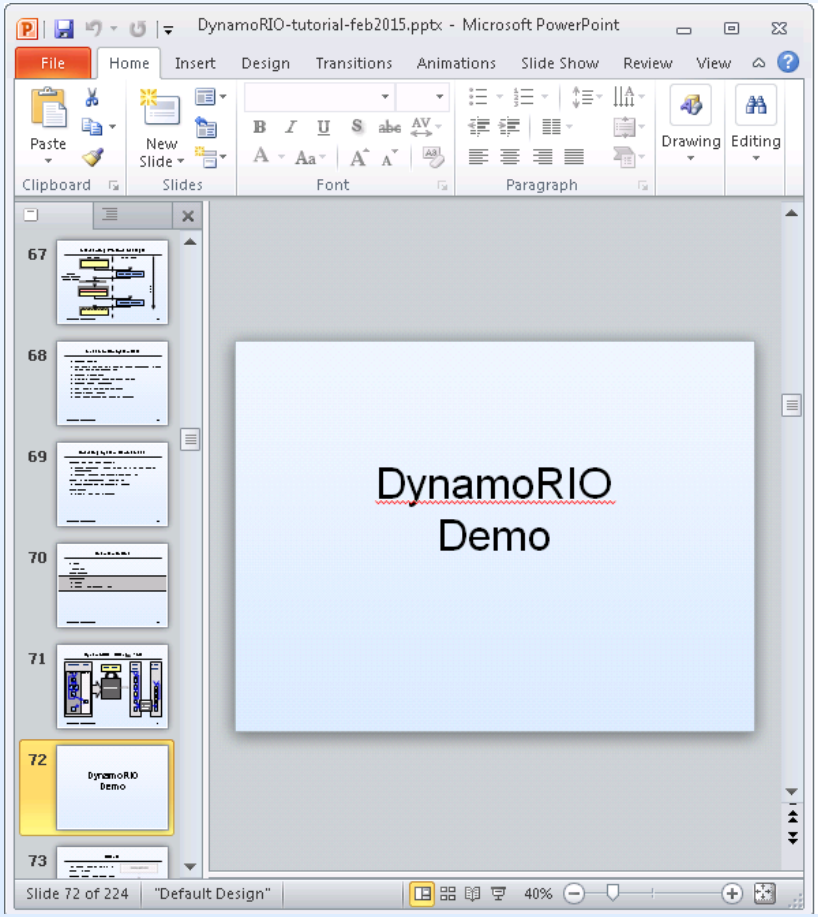
- Efficient
- Transparent
- Comprehensive
- Customizable
  - Clients
  - Building and Deploying Tools

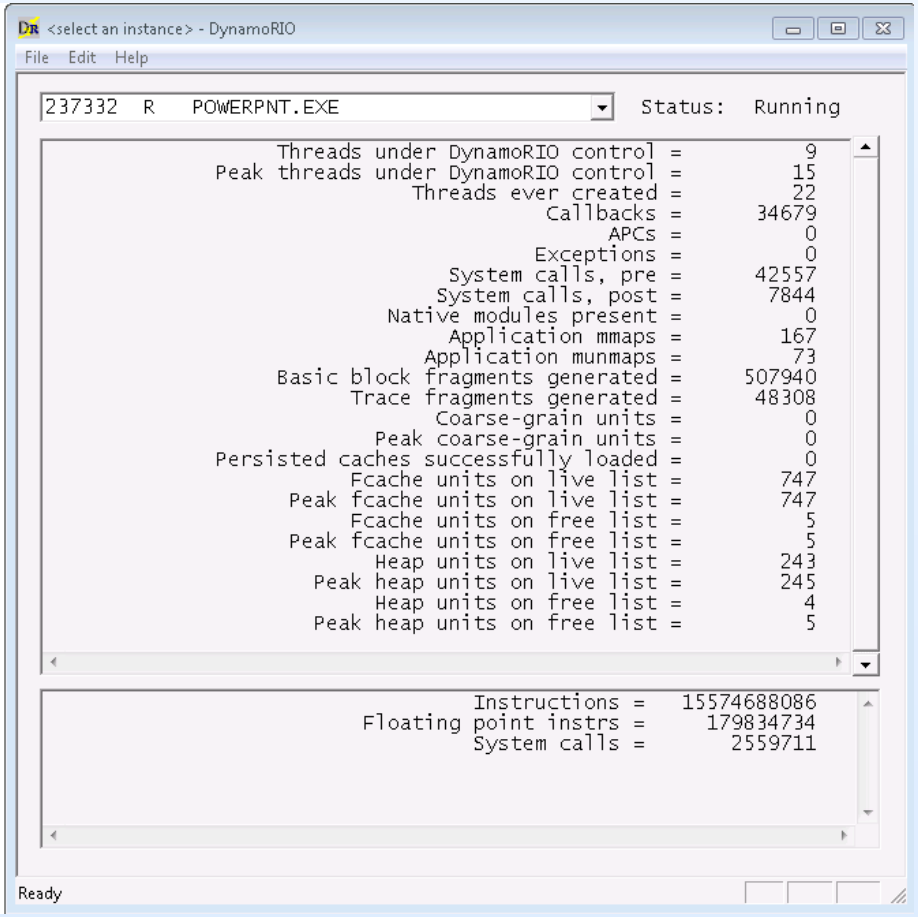
DynamoRIO + Client ⇒ Tool



DynamoRIO Demo

DynamoRIO Demo

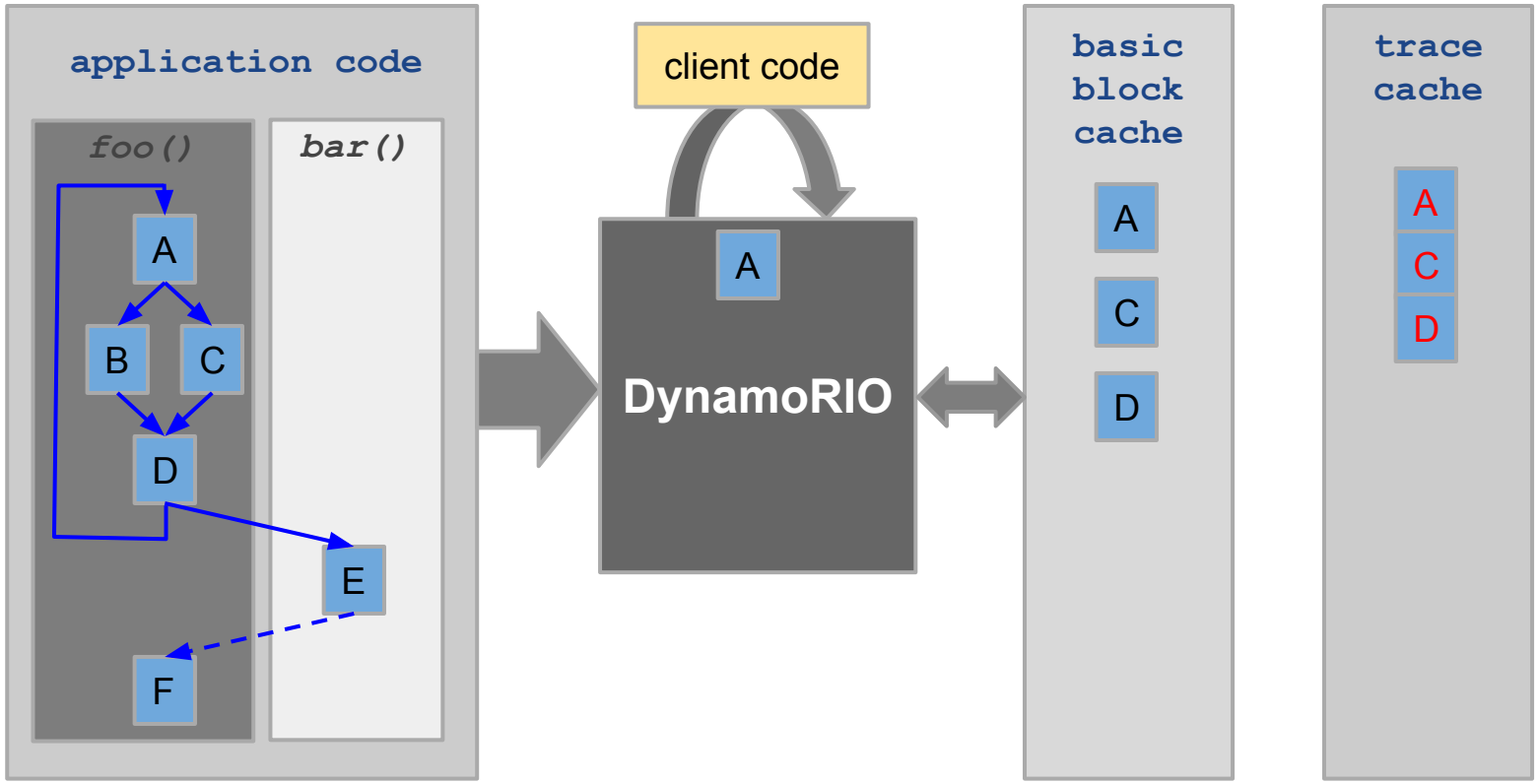




Creating Simple Tools

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A

Tools = DynamoRIO + Clients



DynamoRIO Client

- Event Driven
  - Application, DynamoRIO, Client
  - Events
- Write Simple DynamoRIO Clients
  - Part 1: registration event callbacks
  - Part 2: implementation event callbacks
  - Part 3: instrumentation
- Config, Build, and Run
  - CMake

Event Driven

- Application, DynamoRIO, and Client
  - DynamoRIO "*interprets*" application execution
  - A client takes actions on interested events
- Events
  - Application execution events
    - process start/exit
    - thread start/exit
    - module load/unload
    - pre/post system call execution
    - signal/exception
  - DynamoRIO system svents
    - new basic block
    - new trace
    - delete a code fragment

DynamoRIO Client

- Event Driven
  - Application, DynamoRIO, Client
  - Events
- Write Simple DynamoRIO Clients
  - Part 1: registration event callbacks
  - Part 2: implementation event callbacks
  - Part 3: instrumentation
- Config, Build, and Run
  - CMake

Write Simple DynamoRIO Clients

- Part 1: Register Event Callbacks
  - dr_init
  - dr_register_*_event()
    - thread_init, module_load, pre_syscall, etc.
    - bb, trace, etc.
- Part 2: Implement Event Callbacks
  - Profiling
  - Bookkeepting
- Part 3: Instrumentation
  - BB/Trace
  - Instruction list

Part 1: Register Event Callbacks

```
#include “dr_api.h”

DR_EXPORT void dr_init(client_id_t id) {

    dr_register_thread_init_event(event_thread_init);
    dr_register_thread_exit_event(event_thread_exit);

    dr_register_module_load_event(event_module_load);
    dr_register_module_unload_event(event_module_unload);

    dr_register_pre_syscall_event(event_pre_syscall);
    dr_register_post_syscall_event(event_post_syscall);


    dr_register_bb_event(event_bb);
    dr_register_trace_event(event_trace);
    dr_register_delete_event(event_delete);

}
```

Write Simple DynamoRIO Clients

- Part 1: Register Event Callbacks
  - dr_init
  - dr_register_*_event()
    - thread_init, module_load, pre_syscall, etc.
    - bb, trace, etc.
- Part 2: Implement Event Callbacks
  - Profiling
  - Bookkeepting
- Part 3: Instrumentation
  - BB/Trace
  - Instruction list

Part 2: Implement Event Callback - strace

```
DR_EXPORT void dr_init(client_id_t id) {
    dr_register_post_syscall_event(event_post_syscall);
}
```

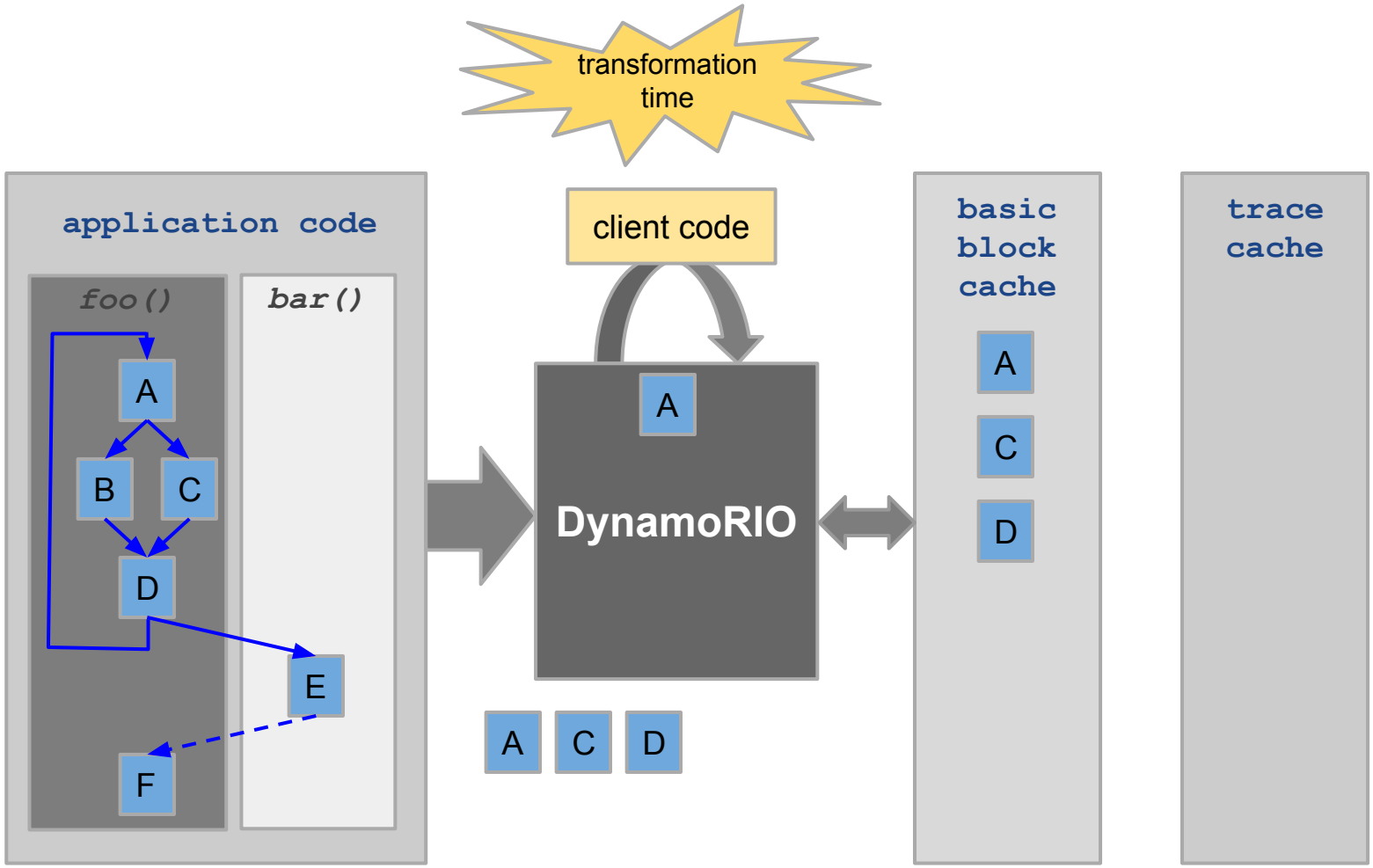
```
static void event_post_syscall(void *drcontext, int sysnum) {
    reg_t result = dr_syscall_get_result(drcontext);
    dr_printf("syscall %d: %d\n", sysnum, result);
}
```

Part 2: Implement Event Callback - strace

```
DR_EXPORT void dr_init(client_id_t id) {
   dr_register_pre_syscall_event(event_pre_syscall);
}
```

```
static bool event_pre_syscall(void *drcontext, int sysnum) {
   if (sysnum ==  SYS_write &&
      dr_syscall_get_param(drcontext, 0) == (reg_t) STDERR){
      dr_syscall_set_result(drcontext, 0);
      dr_printf("skip syscall %d\n", sysnum);
      return false;
    }
   return true;
}
```

Part 2: Implement Event Callbacks - drcov



Part 2: Implement Event Callbacks - drcov

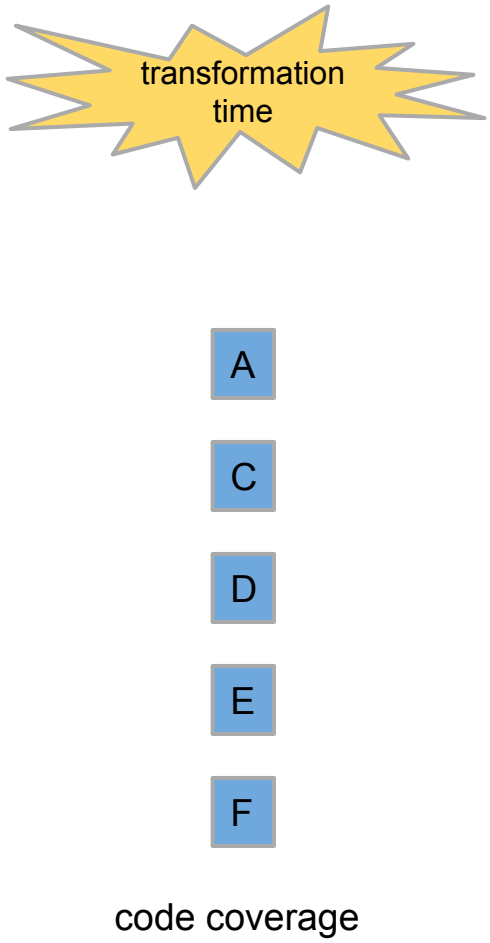
```
DR_EXPORT void dr_init(client_id_t id) {
    dr_register_module_load_event(event_module_load);
    dr_register_module_unload_event(event_module_unload);
    dr_register_bb_event(event_basic_block);
}
```

```
static void event_module_load(void *drcontext, const module_data_t *info, bool loaded) {
    module_table_load(module_table, info);
}

static void event_module_unload(void *drcontext, const module_data_t *info) {
    module_table_unload(module_table, info);
}

static bool
event_basic_block(void *drcontext, void *tag, instrlist_t *bb, bool trace, bool translating) {
    app_pc start_pc = dr_fragment_app_pc(tag);
    bb_table_entry_add(drcontext, start_pc)
    return DR_EMIT_DEFAULT;
}
```

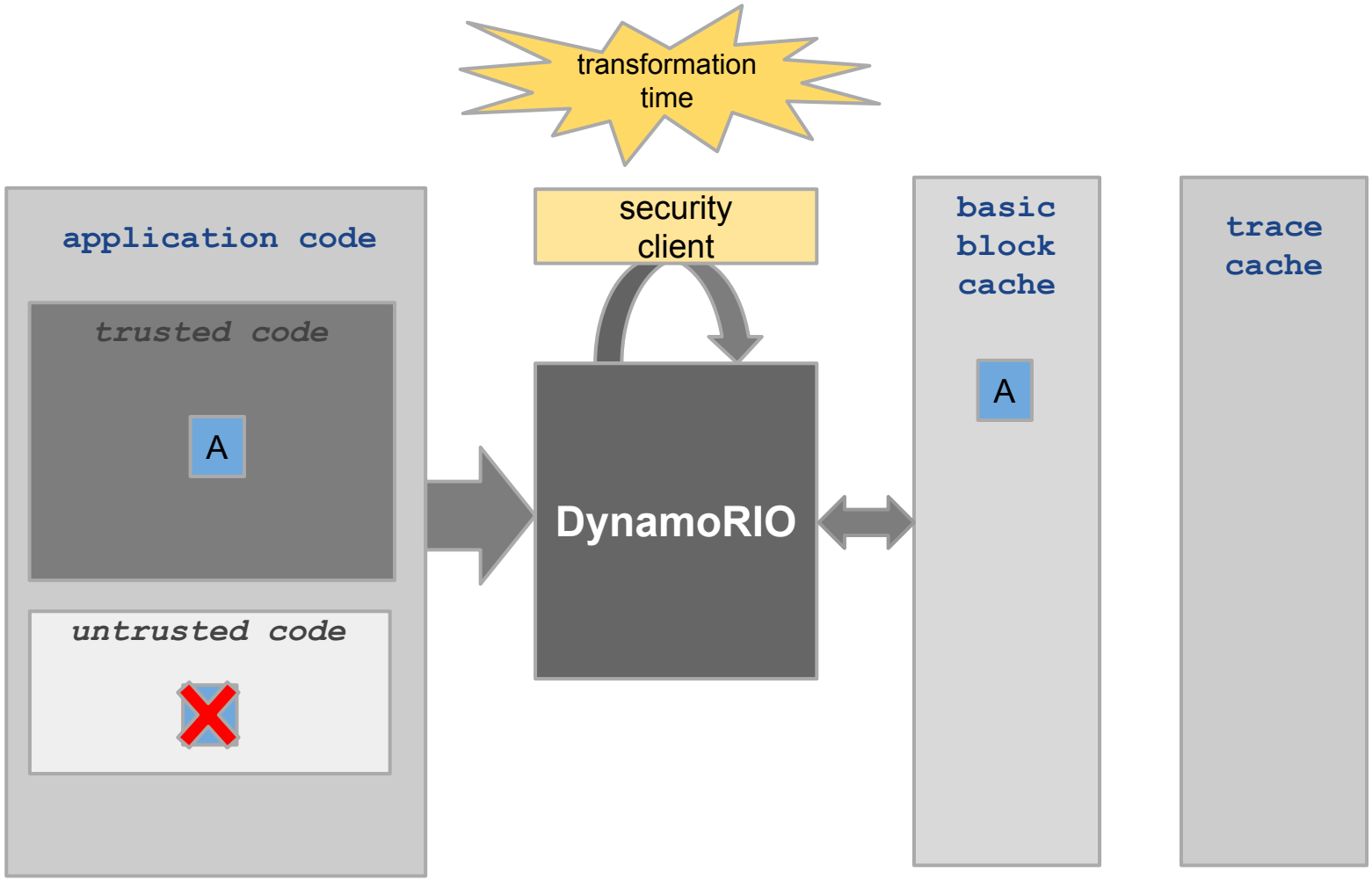
Code Coverage via DynamoRIO - drcov

Why code coverage via DynamoRIO?

- No special re-compilation
- No source code requirement
- More information
  - Code discovery order
    - Cold start optimization
      - Starting Chrome, switching tabs
    - Per-test coverage info
- Special handling
  - Low integrity (Windows)
  - Unexpected exit



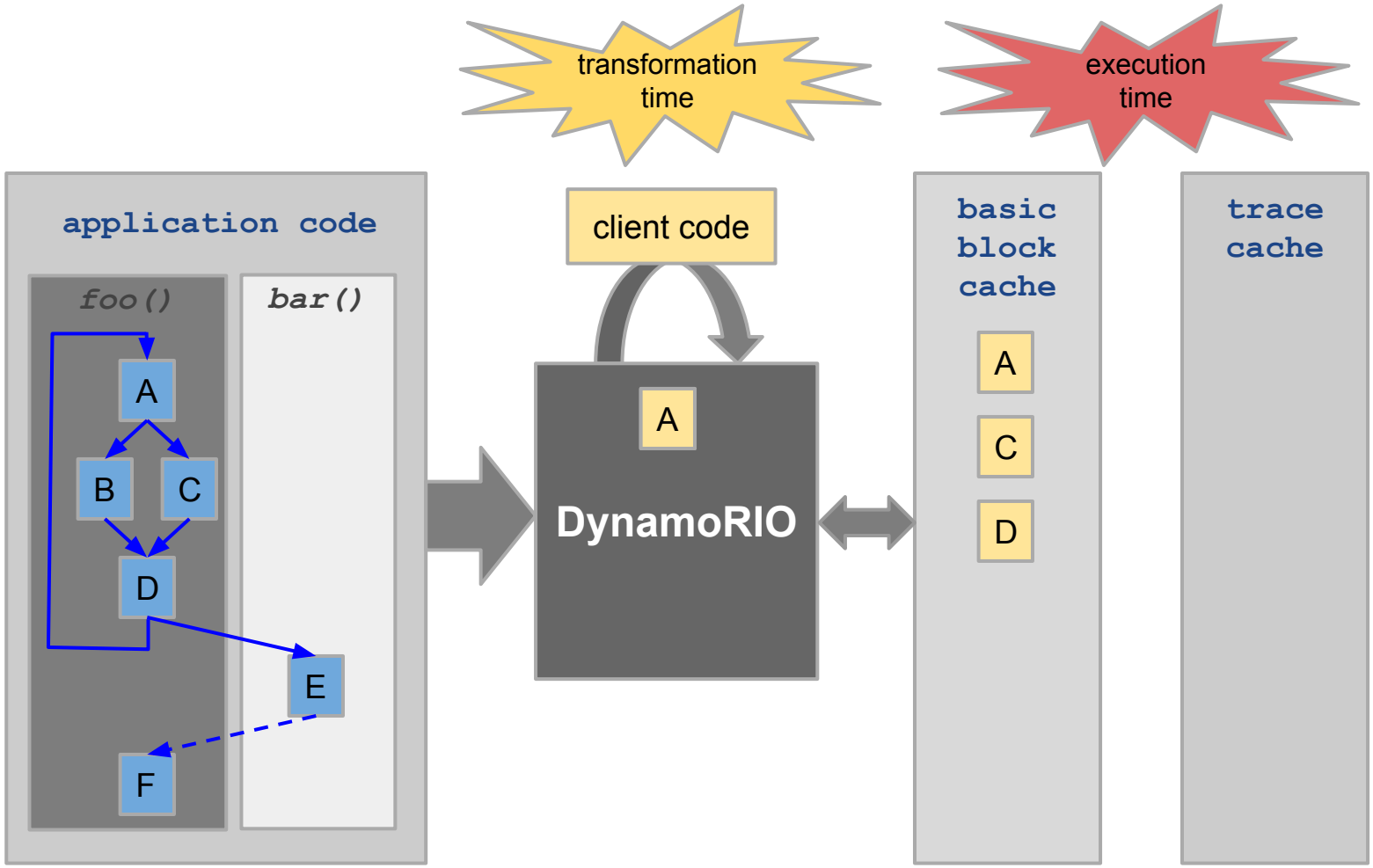
Part 2: Implement Event Callbacks - security



Write Simple DynamoRIO Clients

- Part 1: Register Event Callbacks
  - dr_init
  - dr_register_*_event()
    - thread_init, module_load, pre_syscall, etc.
    - bb, trace, etc.
- Part 2: Implement Event Callbacks
  - Profiling
  - Bookkeepting
- Part 3: Instrumentation
  - BB/Trace
  - Instruction list

Part 3: Instrumentation



Part 3: Instrumentation: instruction counting

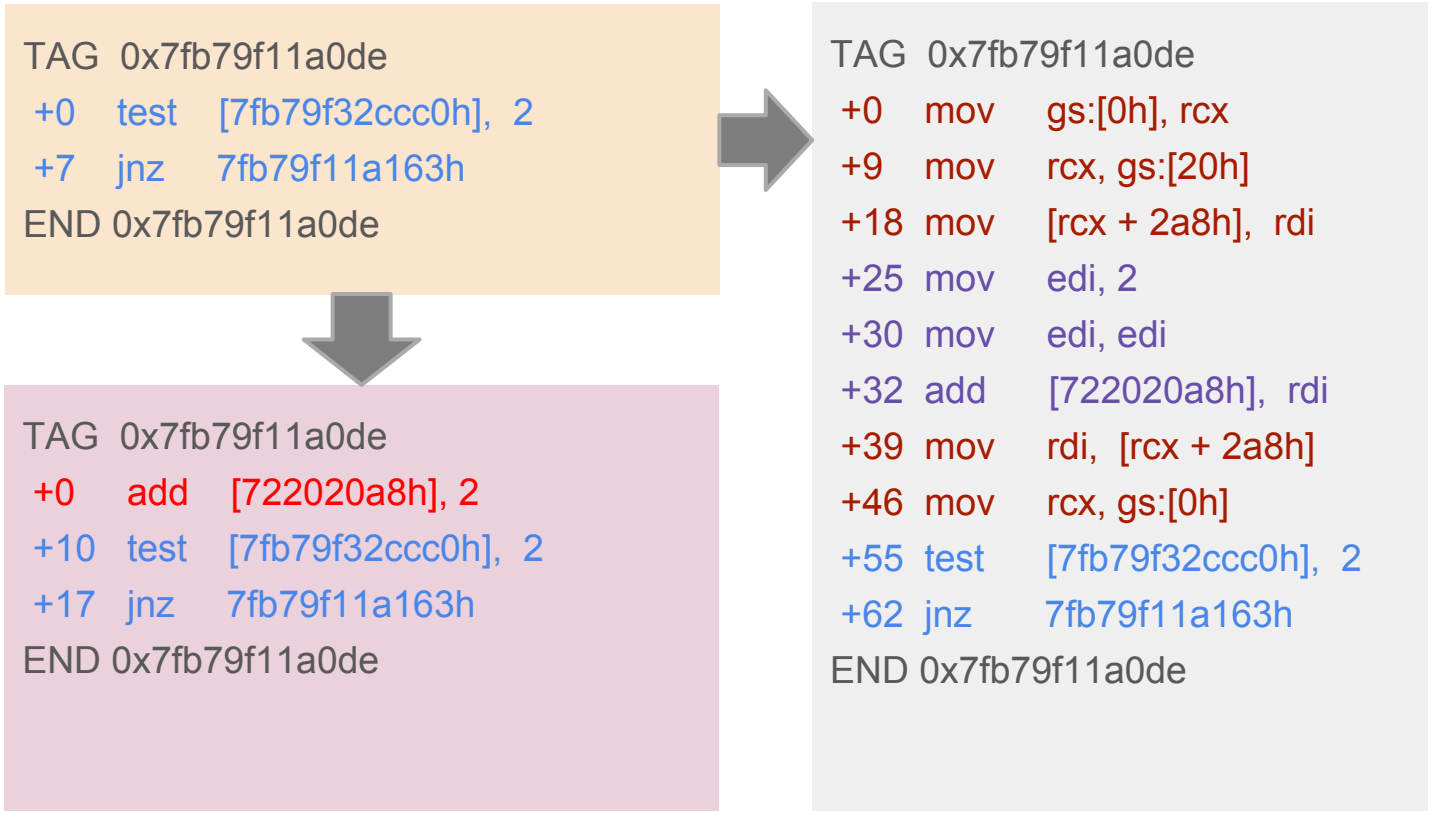
```
static void inscount(uint num_instrs) { count += num_instrs; }
```


```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb, bool trace, bool translating)
{
    instr_t *instr;
    uint num_instrs = 0;

    for (instr = instrlist_first_app(bb); instr != NULL; instr = instr_get_next_app(instr))
        num_instrs++;

    dr_insert_clean_call(drcontext, bb, instrlist_first_app(bb),
                         (void *)inscount, false /* save fpstate */,
                         1, OPND_CREATE_INT32(num_instrs));

    return DR_EMIT_DEFAULT;
}
```

Clean Calls

- C-code Callout
  - dr_insert_clean_call{_ex}
  - dr_insert_{call,mbr,ubr,cbr}_instrumentation{_ex}
  - Inserted clean callee is called from code cache on every execution of the fragment
- Full context switch
  - Save/restore full app state (expensive!)
- Automatic Optimization & Inlining
  - Clean callees are analyzed, optimized & potentially inlined
  - Optimization
    - partial context switch
    - Inlining if simple enough

Part 3: Instrumentation: instruction counting



DynamoRIO Client

- Event Driven
  - Application, DynamoRIO, Client
  - Events
- Write Simple DynamoRIO Clients
  - Part 1: registration event callbacks
  - Part 2: implementation event callbacks
  - Part 3: instrumentation
- Config, Build, and Run
  - CMake

Config, Build, and Run

- CMake
  - http://www.cmake.org/
  - Cenerates build files for native compiler of choice
    - Makefiles for UNIX, nmake, etc.
    - Visual Studio project files
- CMakeLists.txt

```
add_library(myclient SHARED myclient.c)

find_package(DynamoRIO)
if (NOT DynamoRIO_FOUND)
   message(FATAL_ERROR "DynamoRIO package required to build")
endif(NOT DynamoRIO_FOUND)

configure_DynamoRIO_client(myclient)
```

Config, Build, and Run

- Config
  - cmake /path/to/your/client/
  - ccmake, cmake-gui
- Build
  - make
  - cmake --build .
- Run
  - Method 1 (one step)
    - drrun -c <client> <client options> -- <app cmdline>
  - Method 2 (two steps)
    - drconfig -reg <appname> -c <client> <client options>
    - drinject <app cmdline>

Creating Complex Tools

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A

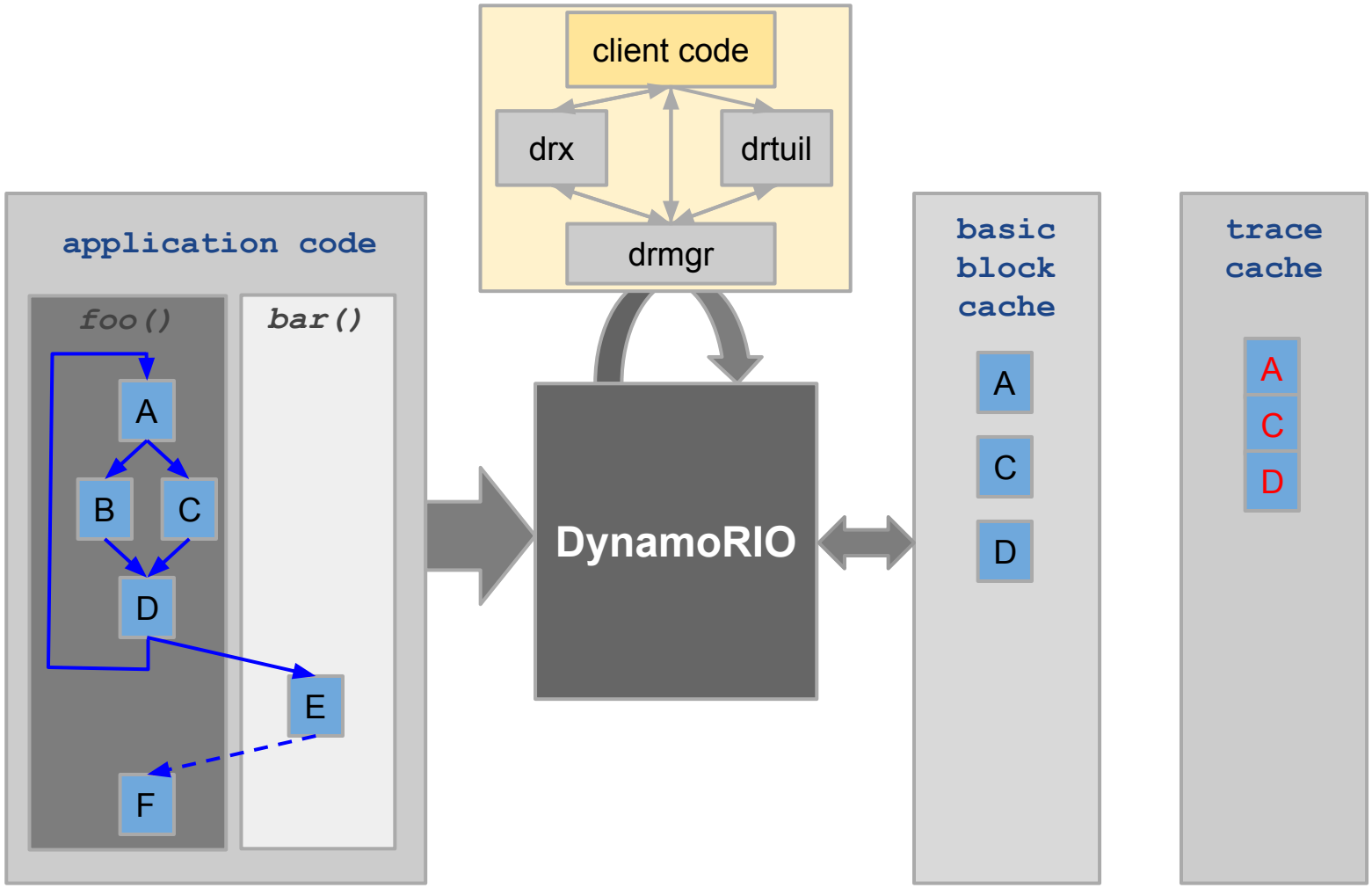
Write Complex DynamoRIO Clients

- Challenges (while building Dr. Memory)
  - Multiple components
    - system call database, shadow memory managment, memory allocation tracking, etc.
  - Manipulate the same instruction list
    - instrumentation may interfere with each other
  - Good performance
    - avoid unnecessary save/restore
    - instrumentation cross multiple instructions (e.g. shadow xl8)
  - Code reuse
- Solution
  - DynamoRIO extension
    - drmgr

DynamoRIO Extensions

- DynamoRIO Extensions
  - Extending DynamoRIO API is extended via libraries
  - Built and packaged with DynamoRIO
  - Easy for a client to use
    - use_DynamoRIO_extension(myclient extensionname)
- Examples
  - drmgr: multi-instrumentation mediation
  - drsyms: symbol lookup
  - drcontainers: hashtable
  - drwrap: function wrapping and replacing
  - drutil: memory address calculation, string loop expansion
  - drx: multi-process management, misc utilities
  - drsyscall: system call names, numbers, parameter types

DynamoRIO Client with Extensions



Multi-Instrumentation Mediation: drmgr

- Mediation Among Multiple Components
  - TLS/CLS access
  - Basic block instrumentation
- Four Phases of Instrumentation
  - Application-to-application transformation
  - Application analysis
  - Instrumentation insertion
  - Instrumentation optimization
- Priority
  - Numeric priority (lower number, early in order)
  - Relative order

drmgr: TLS Access

- TLS in DynamoRIO
  - TLS field
    - dr_{set,get}_tls_field
  - TLS slots
    - SPILL_SLOT_1, ..., SPILL_SLOT_MAX
    - dr_raw_tls_{calloc,cfree}
- drmgr
  - drmgr_register_tls_field
  - drmgr_get_tls_field
  - drmgr_set_tls_field

drmgr: Instrumentation Phases

- Application-To-Application
- Application Analysis
- Instrumentation Insertion
- Instrumentation Optimization

```
bool
drmgr_register_bb_instrumentation_ex_event(drmgr_app2app_ex_cb_t app2app_func,
                                           drmgr_ilist_ex_cb_t   analysis_func,
                                           drmgr_insertion_cb_t  insertion_func,
                                           drmgr_ilist_ex_cb_t   instru2instru_func,
                                           drmgr_priority_t      *priority)
```

drmgr: Instrumentation Phases

- Application-To-Application

```
bool drmgr_register_bb_app2app_event(drmgr_xform_cb_t func,
                                     drmgr_priority_t *priority);
```

- Application Analysis
- Instrumentation Insertion

```
bool drmgr_register_bb_instrumentation_event(drmgr_analysis_cb_t analysis_func,
                                             drmgr_insertion_cb_t insertion_func,
                                             drmgr_priority_t *priority);
```

- Instrumentation Optimization

```
bool drmgr_register_bb_instru2instru_event(drmgr_xform_cb_t func,
                                           drmgr_priority_t *priority);
```

drmgr: Priority

- drmgr_priority_t
  - const char *name
  - const char *before
  - const char *after
  - int priority

```
drmgr_priority_t priority = {
        sizeof(priority),  /* size of struct */
        "memtrace",        /* name of our operation */
        NULL,              /* optional name of operation we should precede */
        NULL,              /* optional name of operation we should follow */
        0};                /* numeric priority */
```

Client Using drmgr: memtrace

```
DR_EXPORT void dr_init(client_id_t id) {
    drmgr_priority_t priority = {
        sizeof(priority), /* size of struct */
        "memtrace",       /* name of our operation */
        NULL,             /* optional name of operation we should precede */
        NULL,             /* optional name of operation we should follow */
        0};               /* numeric priority */

    drmgr_init();
    drutil_init();

    dr_register_exit_event(event_exit);
    if (!drmgr_register_thread_init_event(event_thread_init) ||
        !drmgr_register_thread_exit_event(event_thread_exit) ||
        !drmgr_register_bb_app2app_event(event_bb_app2app, &priority) ||
        !drmgr_register_bb_instrumentation_event
                (event_bb_analysis, event_bb_insert, &priority))
        return;

    tls_index = drmgr_register_tls_field();
    ....
}
```

Client Using drmgr: memtrace

```
static void event_exit() {
    drutil_exit();
    drmgr_exit();
}

static dr_emit_flags_t
event_bb_app2app(void *drcontext, void *tag, instrlist_t *bb, bool trace, bool translating) {
    if (!drutil_expand_rep_string(drcontext, bb)) {
        /* error handle */
    }
    return DR_EMIT_DEFAULT;
}

static dr_emit_flags_t
event_bb_analysis(void *drcontext, void *tag, instrlist_t *bb, bool trace, bool translating,
                  OUT void **user_data) {
    return DR_EMIT_DEFAULT;
}
```

Client Using drmgr: memtrace

```
static dr_emit_flags_t
event_bb_insert(void *drcontext, void *tag, instrlist_t *bb, instr_t *instr, bool trace,
                bool translating, void *user_data) {
    int i;

    if (instr_reads_memory(instr)) {
        for (i = 0; i < instr_num_srcs(instr); i++) {
            if (opnd_is_memory_reference(instr_get_src(instr, i))) {
                instrument_mem(drcontext, bb, instr, i, false);
            }
        }
    }
    if (instr_writes_memory(instr)) {
        for (i = 0; i < instr_num_dsts(instr); i++) {
            if (opnd_is_memory_reference(instr_get_dst(instr, i))) {
                instrument_mem(drcontext, bb, instr, i, true);
            }
        }
    }
    return DR_EMIT_DEFAULT;
}
```

Client Using drmgr: memtrace

```
static void
instrument_mem(void *drcontext, instrlist_t *ilist, instr_t *where, int pos, bool write)
{
    dr_save_reg(drcontext, ilist, where, reg1, SPILL_SLOT_2);
    dr_save_reg(drcontext, ilist, where, reg2, SPILL_SLOT_3);
    drutil_insert_get_mem_addr(drcontext, ilist, where, ref, reg1, reg2);
    drmgr_insert_read_tls_field(drcontext, tls_index, ilist, where, reg2);
    ...
    /* Store address in memory ref */
    opnd1 = OPND_CREATE_MEMPTR(reg2, offsetof(mem_ref_t, addr));
    opnd2 = opnd_create_reg(reg1);
    instr    = INSTR_CREATE_mov_st(drcontext, opnd1, opnd2);
    instrlist_meta_preinsert(ilist, where, instr);
    …
    dr_restore_reg(drcontext, ilist, where, reg1, SPILL_SLOT_2);
    dr_restore_reg(drcontext, ilist, where, reg2, SPILL_SLOT_3);
}
```

Transparency

- Instrumentation Transparency
  - Preserve application state
    - machine context
- Library Transparency
  - Application stack
  - TLS
  - Private loader & libraries
    - should work with most libraries
    - except pthread

Debugging Your DynamoRIO Client

- Run With Debug Build DynamoRIO
  - drrun -debug …
  - -stderr_mask 0xN
  - -ignore_assert_list ‘*’
  - -disable_traces
- Run With Logging
  - drrun -debug -loglevel 3
- Run With Debugger
  - drrun -debug -loglevel 3 -msgbox_mask 0xf …
  - -no_hide (windows)
  - Attach debugger to the process

Debugging Your DynamoRIO Client

- Debug Client Code
  - Load debug symbol for your client
    - add-symbol-file ‘/path/to/your/client/libname.so’ 0xxxxxxx
  - Normal debugging
- Debug Instrumented Code
  - printf debugging
    - {opnd, instrlist, instr}_disassemble
  - breakpoint
    - build_basic_block_fragment
    - build_bb_ilist
    - enter_fcache
  - watchpoint (data breakpoint)

Build Your Own Tools

- Profiling
  - Path profiling
  - Memory tracing
  - Data flow tracing
- Debugging & Testing
  - Reverse execution
  - Software breakpoint
  - Data race detection
  - Fuzzing
- Security
  - Program shepherding
  - Taint tracking
  - Code de-obfuscation
- Your ideas
  - ?
  - ?
  - ?

DynamoRIO API

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A

DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

Cross-Platform Clients

- DynamoRIO API presents a consistent interface that works across platforms
  - Windows versus Linux
  - 32-bit versus 64-bit
  - Thread-private versus thread-shared
- Same client source code generally works on all combinations of platforms
- Some exceptions, noted in the documentation

Building a Client

- Include DR API header file
  - **`#include "dr_api.h"`**
- Set platform defines
  - WINDOWS or LINUX
  - X86_32 or X86_64
- Export a dr_init function
  - **`DR_EXPORT void dr_init (client_id_t client_id)`**
- Build a shared library

Auto-Configure Using CMake

```
add_library(myclient SHARED myclient.c)
find_package(DynamoRIO)
if (NOT DynamoRIO_FOUND)
  message(FATAL_ERROR "DynamoRIO package
      required to build")
endif(NOT DynamoRIO_FOUND)
configure_DynamoRIO_client(myclient)
```

CMake

- Build system converted to CMake when open-sourced
  - Switch from frozen toolchain to supporting range of tools
- CMake generates build files for native compiler of choice
  - Makefiles for UNIX, nmake, etc.
  - Visual Studio project files
- http://www.cmake.org/

DynamoRIO Extensions

- DynamoRIO API is extended via libraries called Extensions
- Both static and shared supported
- Built and packaged with DynamoRIO
- Easy for a client to use
  - **`use_DynamoRIO_extension(myclient extensionname)`**

Current DynamoRIO Extensions

- Current Extensions:
  - *drsyms*: symbol table and debug information lookup
  - *drcontainers*: hashtable, vector, and table
  - *drmgr*: multi-instrumentation mediation
  - *drwrap*: function wrapping and replacing
  - *drutil*: memory tracing, string loop expansion
  - *drx:* multi-process management, misc utilities
  - *drsyscall:* system call monitoring: system call names, numbers, parameter types, memory references
  - *drdecode*: standalone IA32/AMD64/ARM/Thumb decoding/encoding library
  - *umbra*: shadow memory framework

Forthcoming DynamoRIO Extensions

- Coming soon:
  - *drreg:* register stealing and allocating
  - *drtrack: s*hadow value propagation (e.g., for taint tracking)
  - *drmalloc*: malloc tracking
  - *drcallstack: e*fficient callstack walking
  - *Your utility library or framework contribution!*

Application Configuration

- File-based scheme
- Per-user local files
  - $HOME/.dynamorio/ on Linux
  - $USERPROFILE/dynamorio/ on Windows
- Global files
  - /etc/dynamorio/ on Linux
  - Registry-specified directory on Windows
- Files are lists of var=value

Deploying Clients

- One-step configure-and-run usage model:
  - drrun –c <client> <client options> -- <app cmdline>
  - Uses an invisible temporary first-priority one-time config file
- Two-step usage model giving more control over children:
  - drconfig –reg <appname> -c <client> <client options>
  - drinject <app cmdline>
- Systemwide injection:
  - drconfig –syswide_on –reg <appname> -c <client> <options>
  - <run app normally>
- Polished tool invocation:
  - drrun –t <tool> <tool options> -- <app cmdline>
  - Uses a pre-existing tool config file

Deploying Clients On Linux

- drrun and drinject scripts: LD_PRELOAD-based
  - Take over after statically-dependent shared libs but before exe
- Suid apps ignore LD_PRELOAD
  - Place libdrpreload.so's full path in /etc/ld.so.preload
  - Copy libdynamorio.so to /usr/lib
- In the future:
  - Attach
  - Earliest injection

Deploying Clients On Windows

- drinject and drrun injection
  - Currently after all shared libs are initialized
- From-parent injection
  - Early: before any shared libs are loaded
- Systemwide injection via –syswide_on
  - Requires administrative privileges
  - Launch app normally: no need to run via drinject/drrun
  - Moderately early: during user32.dll initialization
- In the future:
  - Earliest injection for drrun/drinject and from-parent

Following Child Processes

- Runtime option –follow_children
  - Default on: follow all children
- Whitelist
  - -no_follow_children and configure files for whitelist
- Blacklist
  - -follow_children and configure files –norun for blacklist
  - *drconfig -norun* to create do-not-follow config file

Non-Standard Deployment

- drdecode
  - Static IA-32/AMD64 decoding/encoding/instruction manipulation library
- Standalone API
  - Use DynamoRIO as a library of IA-32/AMD64 manipulation routines plus cross-platform file i/o, locks, etc.
- Start/Stop API
  - Can instrument source code with where DynamoRIO should control the application

Runtime Options

- Pass options to drconfig/drrun
- A large number of options; the most relevant are:
  - -code_api
  - -c <client lib> <client options>
  - -thread_private
  - -follow_children
  - -opt_cleancall
  - -tracedump_text and –tracedump_binary
  - -prof_pcs

Runtime Options For Debugging

- Notifications:
  - -stderr_mask 0xN
  - -msgbox_mask 0xN
- Windows:
  - -no_hide
- Debug-build-only:
  - -loglevel N
  - -ignore_assert_list ‘*’

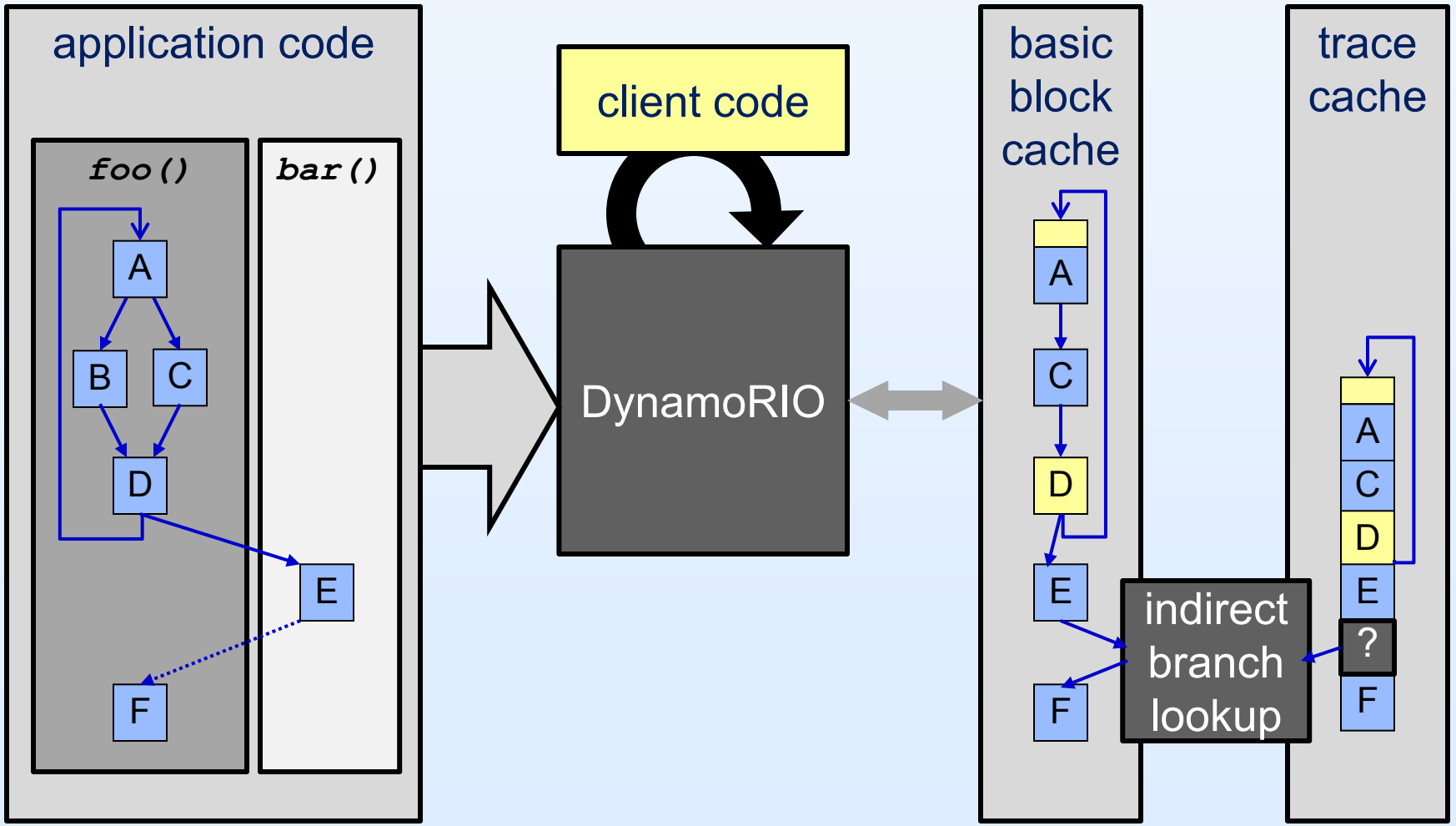
Provided Tools

- In addition to numerous sample clients, DynamoRIO ships with the following polished end-user tools:
  - Dr. Memory (*drmemory*): identifies memory errors (use-after-frees, buffer overflows, uninitialized reads, memory leaks, etc.)
  - Dr. Cov (*drcov*): code coverage tool
  - Dr. Strace (*drstrace*): system call tracer for Windows
  - Dr. Ltrace (*drltrace*): library call tracer
- Run these tools via *drrun –t <toolname>*
  - E.g.: *bin32/drrun –t drmemory -- <app cmdline>*

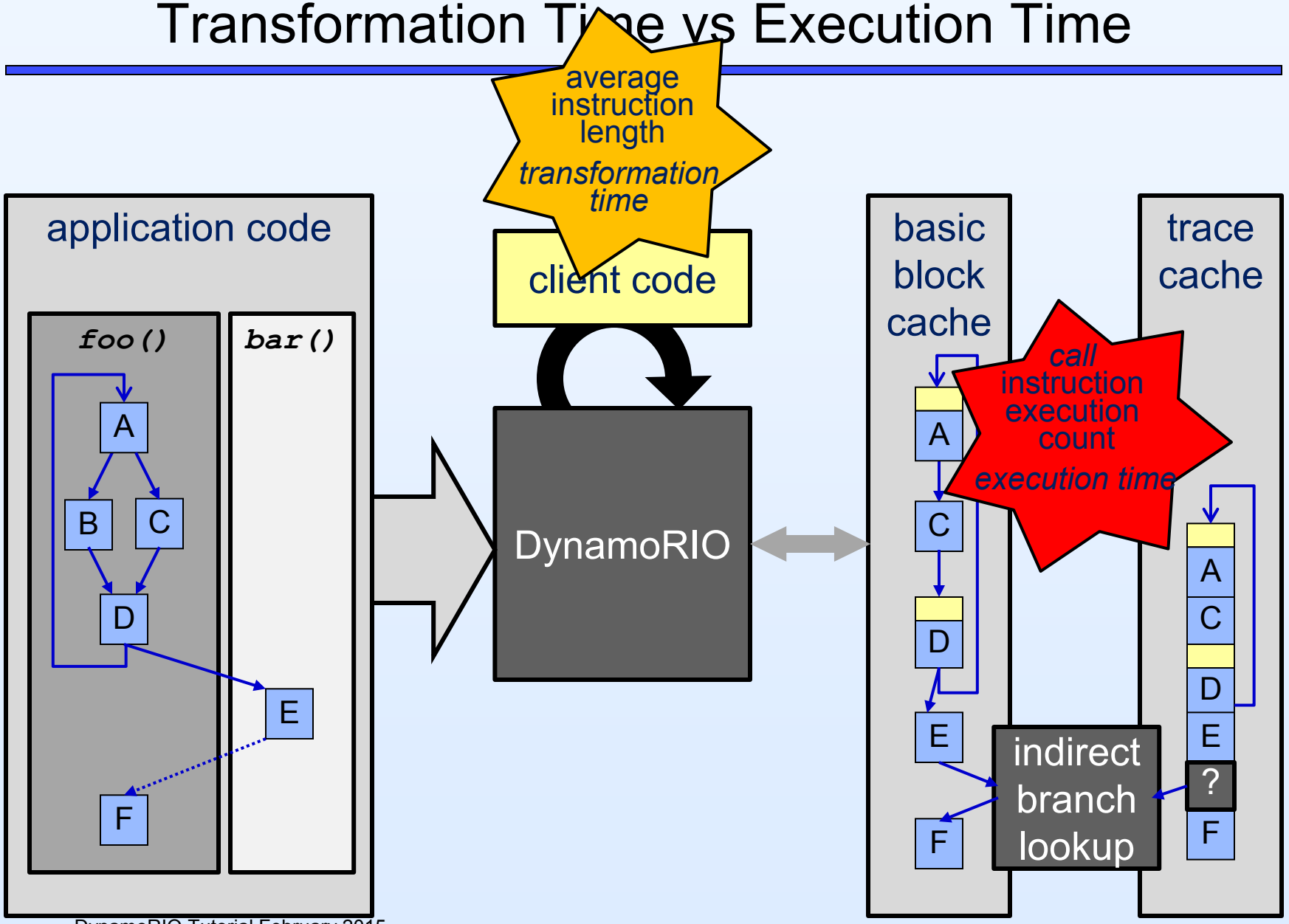
DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

DynamoRIO + Client ⇒ Tool



Transformation Time vs Execution Time



Client Events: Code Stream

- Client has opportunity to inspect and potentially modify every single application instruction, immediately before it executes
  - Event happens at transformation time
  - Modifications or inserted code will operate at execution time
- Entire application code stream
  - Basic block creation event: can modify the block
  - For comprehensive instrumentation tools
- Or, focus on hot code only
  - Trace creation event: can modify the trace
  - Custom trace creation: can determine trace end condition
  - For optimization and profiling tools

Simplifying Client View

- Several optimizations disabled
  - Elision of unconditional branches
  - Indirect call to direct call conversion
  - Shared cache sizing
  - Process-shared and persistent code caches
- Future release will give client control over optimizations

Basic Block Event

```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag,
                  instrlist_t *bb, bool for_trace,
                  bool translating) {
    instr_t *inst;
    for (inst = instrlist_first(bb);
         inst != NULL;
         inst = instr_get_next(inst)) {
        /* … */
    }
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
```

Trace Event

```
static dr_emit_flags_t
event_trace(void *drcontext, void *tag,
            instrlist_t *trace, bool translating) {
    instr_t *inst;
    for (inst = instrlist_first(trace);
         inst != NULL;
         inst = instr_get_next(inst)) {
        /* … */
    }
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_trace_event(event_trace);
}
```

Client Events: Application Actions

- Application thread creation and deletion
- Application library load and unload
- Application exception (Windows)
  - Client chooses whether to deliver or suppress
- Application signal (Linux)
  - Client chooses whether to deliver, suppress, bypass the app handler, or redirect control

Client Events: Application System Calls

- Application pre- and post- system call
  - Platform-independent system call parameter access
  - Client can modify:
    - Return value in post-, or set value and skip syscall in pre-
    - Call number
    - Params
  - Client can invoke an additional system call as the app

Client Events: Bookkeeping

- Initialization and Exit
  - Entire process
  - Each thread
  - Child of fork (Linux-only)
- Basic block and trace deletion during cache management
- Nudge received
  - Used for communication into client
- Itimer fired (Linux-only)

Multiple Clients

- It is each client's responsibility to ensure compatibility with other clients
  - Instruction stream modifications made by one client are visible to other clients
- At client registration each client is given a priority
  - dr_init() called in priority order (priority 0 called first and thus registers its callbacks first)
- Event callbacks called in reverse order of registration
  - Gives precedence to first registered callback, which is given the final opportunity to modify the instruction stream or influence DynamoRIO's operation
- drmgr Extension provides mediation among multiple components

DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

DynamoRIO API: General Utilities

- DynamoRIO provides safe utilities for transparency support
  - Separate stack
  - Separate memory allocation
  - Separate file I/O
- Utility options
  - Use DynamoRIO-provided utilities directly
  - Use shared libraries via DynamoRIO private loader
    - Malloc, etc. redirected to DynamoRIO-provided utilities
  - Use static libraries with dependencies redirected
- Risky for client to directly invoke system calls

DynamoRIO Heap

- Three flavors:
  - Thread-private: no synchronization; thread lifetime
  - Global: synchronized, process lifetime
  - “Non-heap”: for generated code, etc.
  - No header on allocated memory: low overhead but must pass size on free
- Leak checking
  - Debug build complains at exit if memory was not deallocated

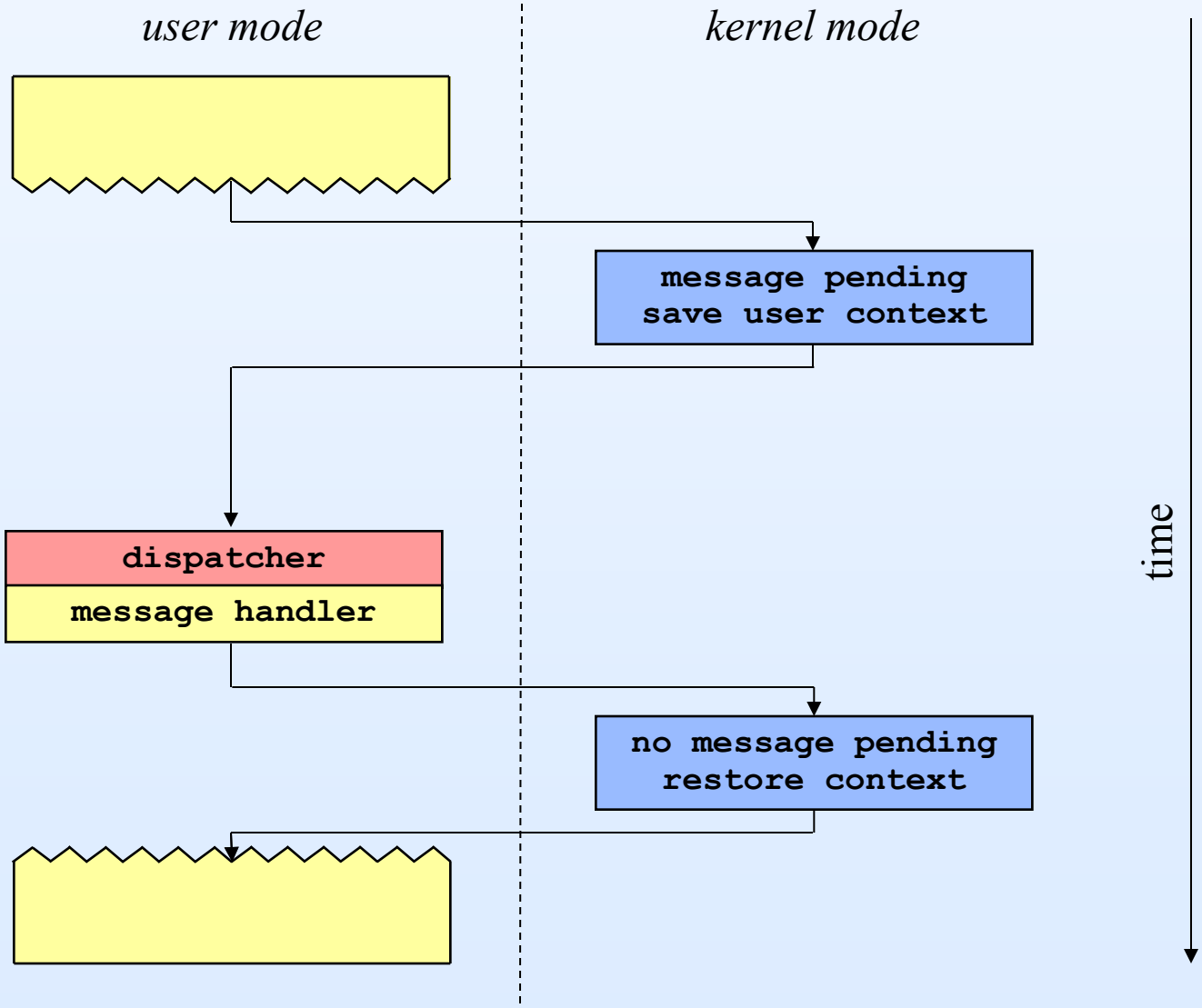
Thread Support

- Thread support
  - Thread-local storage
  - Callback-local storage
  - Simple mutexes
  - Read-write locks
  - Thread-private code caches, if requested
- Sideline support
  - Create new client-only thread
  - Thread-private itimer (Linux-only)
- Suspend and resume all other threads
  - Cannot hold locks while suspending

Thread-Local Storage (TLS)

- Absolute addressing
  - Thread-private only
- Application stack
  - Not reliable or transparent
- Stolen register
  - Performance hit
- Segment
  - Best solution for thread-shared

Callback-Local Storage (CLS)



Callback-Local Storage (CLS)

- Windows callbacks interrupt execution to process an event and later resume the suspended context
- TLS data from the suspended context will be overwritten during callback execution
- CLS data is saved at the interruption point and restored at the resumption point
- Whenever keeping persistent data specific to one context rather than overall execution, use CLS instead of TLS
  - Usually only needed when storing data specific to a system call in pre-syscall event and reading it back in post-syscall event
- Can be used for Linux signals as well
- Provided by the drmgr Extension

DynamoRIO API: General Utilities, Cont'd

- Communication
  - *Nudges:* ping from external process
  - File creation, reading, and writing
  - File descriptor isolation on Linux
- Safe read/write
  - Fault-proof read/write routines
  - Try/except facility

DynamoRIO API: General Utilities, Cont'd

- Application inspection
  - Address space querying
  - Module iterator
  - Processor feature identification
  - Symbol lookup
  - Function replacing and wrapping

Symbol Table Access

- The *drsyms* Extension provides access to symbol tables and debug information
- Currently supports the following:
  - Windows PDB
  - Linux ELF + DWARF2
  - Windows PECOFF + DWARF2
  - OSX MachO + DWARF2
- API includes:
  - Address to symbol and line information
  - Symbol to address
  - Symbol enumeration and searching
  - Symbol demangling
  - Symbol types

Function Replacing and Wrapping

- *drwrap* Extension provides function replacing and wrapping
- Use dr_get_proc_address() to find library exports or drsyms Extension to find internal functions
- Function replacing replaces with *application code*
- Function wrapping calls pre and post callbacks that execute as client code around the target application function
- Arguments, return value, and whether the function is executed can all be examined and controlled

Third-Party Libraries

- Private loader inside DynamoRIO will load any external shared libraries a client imports from
  - Loads a duplicate copy of each library and tries to isolate from the application's copy
- On Windows, private loader does not support locating SxS libraries, so use static libc with VS2005 or VS2008
- C++ clients are built normally
- C clients by default do not link with libc
  - Set DynamoRIO_USE_LIBC variable prior to invoking configure_DynamoRIO_client() to use libc with a C client

Private Libraries

- Private loader on Windows
  - Not easy to fully isolate system data structures
    - PEB and key TEB fields are isolated
    - Some libraries like ntdll.dll are shared
  - To examine application state while in client code, use dr_switch_to_app_state()
- Private loader on Linux
  - Isolation is simpler and more complete

Optimal Transparency

- For best transparency: completely self-contained client
  - Imports only from DynamoRIO API
  - -nodefaultlibs or /nodefaultlib
- Alternatives to dynamic libc on Windows:
  - String and utility routines provided by forwards to ntdll
    - ntdll contains “mini-libc”
  - Cl.exe /MT static copy of C/C++ libraries
- Alternatives to dynamic libc on Linux:
  - For static C/C++ lib, use ld –wrap to redirect malloc to DR’s heap
  - Newer distributions don’t ship suitable static C/C++ lib

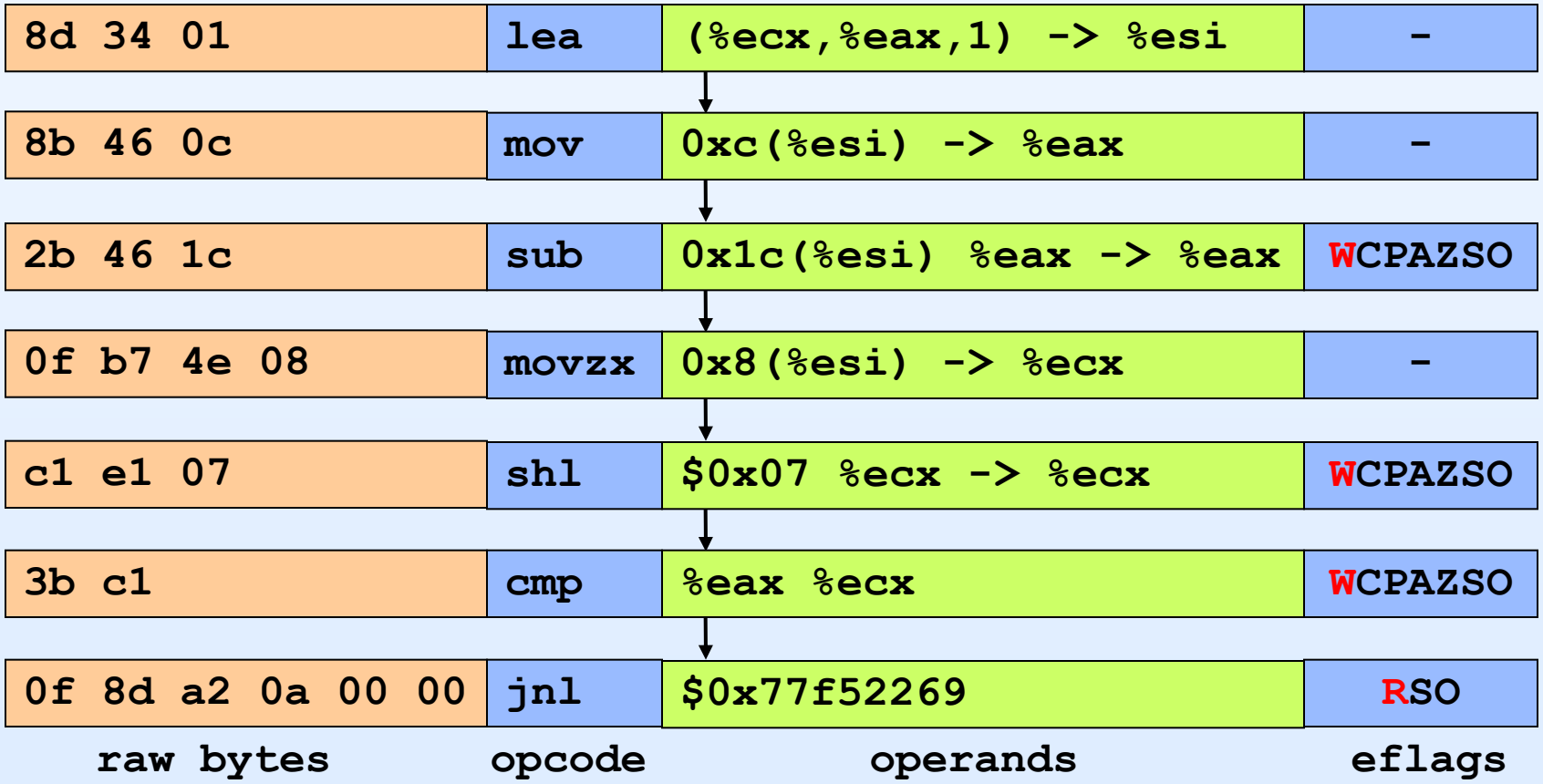
DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

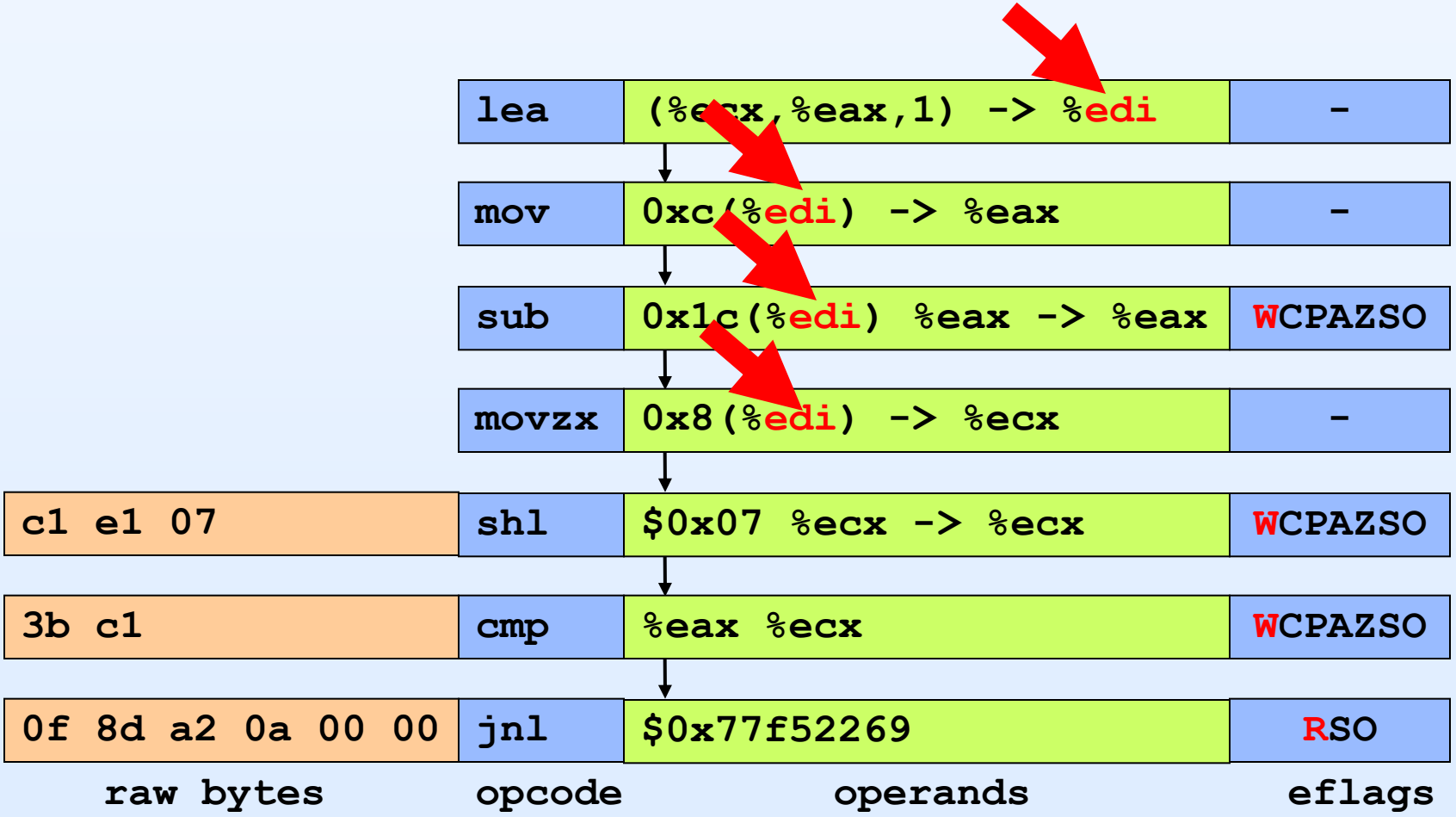
DynamoRIO API: Instruction Representation

- Full IA-32/AMD64 instruction representation
- Instruction creation with auto-implicit-operands
- Operand iteration
- Instruction lists with iteration, insertion, removal
- Decoding at various levels of detail
- Encoding

Instruction Representation



Instruction Representation



Instruction Creation

- Method 1: use the INSTR_CREATE_opcode macros that fill in implicit operands automatically:

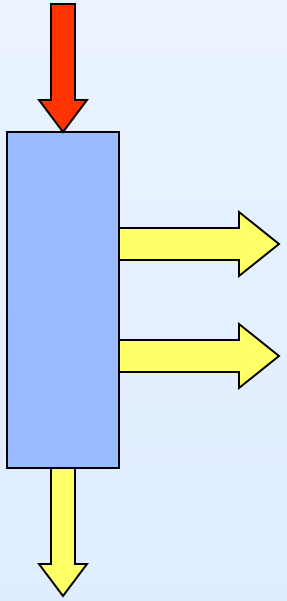
```
instr_t *instr = INSTR_CREATE_dec(dcontext,
   opnd_create_reg(DR_REG_EDX));
```

- Method 2: specify opcode + all operands (including implicit operands):

```
instr_t *instr = instr_create(dcontext);
instr_set_opcode(instr, OP_dec);
instr_set_num_opnds(dcontext, instr, 1, 1);
instr_set_dst(instr, 0, opnd_create_reg(DR_REG_EDX));
instr_set_src(instr, 0, opnd_create_reg(DR_REG_EDX));
```

Linear Control Flow

- Both basic blocks and traces are linear
- Instruction sequences are all single-entrance, multiple-exit
- Greatly simplifies analysis algorithms



64-Bit Versus 32-Bit

- 32-bit build of DynamoRIO only handles 32-bit code
- 64-bit build of DynamoRIO decodes/encodes both 32-bit and 64-bit code
  - Current release does not support executing applications that mix the two
- IR is universal: covers both 32-bit and 64-bit
  - Abstracts away underlying mode

64-Bit Thread and Instruction Modes

- When going to or from the IR, the thread mode and instruction mode determine how instrs are interpreted
- When decoding, current thread's mode is used
  - Default is 64-bit for 64-bit DynamoRIO
  - Can be changed with set_x86_mode()
- When encoding, that instruction's mode is used
  - When created, set to mode of current thread
  - Can be changed with instr_set_x86_mode()

64-Bit Clients

- Define X86_64 before including header files when building a 64-bit client
- Convenience macros for printf formats, etc. are provided
  - E.g.:
    - `printf("Pointer is "PFX"\n", p);`
- Use "X" macros for cross-platform registers
  - DR_REG_XAX is DR_REG_EAX when compiled 32-bit, and DR_REG_RAX when compiled 64-bit

DynamoRIO API: Code Manipulation

- Processor information
- State preservation
  - Eflags, arith flags, floating-point state, MMX/SSE state
  - Spill slots, TLS, CLS
- Clean calls to C code
- Dynamic instrumentation
  - Replace code in the code cache
- Branch instrumentation
  - Convenience routines

Processor Information

- Processor type
  - proc_get_vendor(), proc_get_family(), proc_get_type(), proc_get_model(), proc_get_stepping(), proc_get_brand_string()
- Processor features
  - proc_has_feature(), proc_get_all_feature_bits()
- Cache information
  - proc_get_cache_line_size(), proc_is_cache_aligned(), proc_bump_to_end_of_cache_line(), proc_get_containing_page()
  - proc_get_L1_icache_size(), proc_get_L1_dcache_size(), proc_get_L2_cache_size(), proc_get_cache_size_str()

State Preservation

- Spill slots for registers
  - 3 fast slots, 6/14 slower slots
  - dr_save_reg(), dr_restore_reg(), and dr_reg_spill_slot_opnd()
  - From C code: dr_read_saved_reg(), dr_write_saved_reg()
- Dedicated TLS field for thread-local data
  - dr_insert_read_tls_field(), dr_insert_write_tls_field()
  - From C code: dr_get_tls_field(), dr_set_tls_field()
  - Parallel routines for CLS fields
- Arithmetic flag preservation
  - dr_save_arith_flags(), dr_restore_arith_flags()
- Floating-point/MMX/SSE state
  - dr_insert_save_fpstate(), dr_insert_restore_fpstate()

Clean Calls

```
if (instr_is_mbr(instr)) {
  app_pc address = instr_get_app_pc(instr);
  uint opcode = instr_get_opcode(instr);
  instr_t *nxt = instr_get_next(instr);
  dr_insert_clean_call(drcontext, ilist, nxt, (void *) at_mbr,
                       false/*don't need to save fp state*/,
                       2 /* 2 parameters */,
                       /* opcode is 1st parameter */
                       OPND_CREATE_INT32(opcode),
                       /* address is 2nd parameter */
                       OPND_CREATE_INTPTR(address));
}
```

- Saved interrupted application state can be accessed using dr_get_mcontext() and modified using dr_set_mcontext()

Clean Call Inlining

- Simple clean callees will be automatically optimized and potentially inlined
- -opt_cleancall runtime option controls aggressiveness
- Current requirements for inlining:
  - Leaf routine (may call PIC get-pc thunk)
  - Zero or one argument
  - Relatively short
- Compile the client with optimizations to improve clean call optimization
- Look in debug logfile for “CLEANCALL” to see results

Dynamic Instrumentation

- Thread-shared: flush all code corresponding to application address and then re-instrument when re-executed
  - Can flush from clean call, and use dr_redirect_execution() since cannot return to potentially flushed cache fragment
- Thread-private: can also replace particular fragment (does not affect other potential copies of the source app code)
  - dr_replace_fragment()

Flushing the Cache

- Immediately deleting or replacing individual code cache fragments is available for thread-private caches
  - Only removes from that thread's cache
- Two basic types of thread-shared flush:
  - Non-precise: remove all entry points but let target cache code be invalidated and freed lazily
  - Precise/synchronous:
    - Suspend the world
    - Relocate threads inside the target cache code
    - Invalidate and free the target code immediately

Flushing the Cache

- Thread-shared flush API routines:
  - dr_unlink_flush_region(): non-precise flush
  - dr_flush_region(): synchronous flush
  - dr_delay_flush_region():
    - No action until a thread exits code cache on its own
    - If provide a completion callback, synchronous once triggered
    - Without a callback, non-precise

Multi-Instrumentation Mediation

- The *drmgr* Extension provides mediation among multiple agents for basic block instrumentation and TLS/CLS access
- Divides instrumentation into four stages and orders the callbacks for each stage:
  - Application-to-application transformations
  - Application analysis
  - Instrumentation insertion
  - Instrumentation optimization
- Enables multi-library frameworks and modular clients

Memory Tracing

- *drutil* Extension provides utilities for memory address tracing:
  - Address acquisition
  - String loop expansion

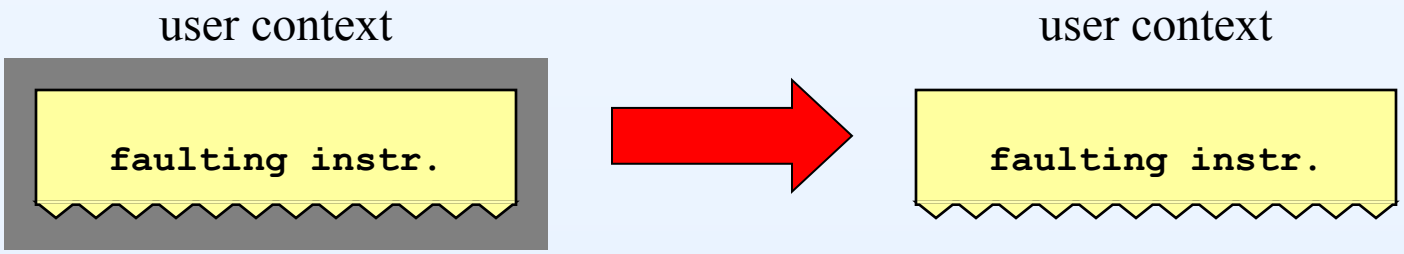
DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

DynamoRIO API: Translation

- Translation refers to the mapping of a code cache machine state (program counter, registers, and memory) to its corresponding application state
  - The program counter always needs to be translated
  - Registers and memory may also need to be translated depending on the transformations applied when copying into the code cache

Translation Case 1: Fault



- Exception and signal handlers are passed machine context of the faulting instruction.
- For transparency, that context must be translated from the code cache to the original code location
- Translated location should be where the application would have had the fault or where execution should be resumed

Translation Case 2: Relocation

- If one application thread suspends another, or DynamoRIO suspends all threads for a synchronous cache flush:
  - Need suspended target thread in a safe spot
  - Not always practical to wait for it to arrive at a safe spot (if in a system call, e.g.)
- DynamoRIO forcibly relocates the thread
  - Must translate its state to the proper application state at which to resume execution

Translation Approaches

- Two approaches to program counter translation:
  - Store mappings generated during fragment building
    - High memory overhead (> 20% for some applications, because it prevents internal storage optimizations) even with highly optimized difference-based encoding. Costly for something rarely used.
  - Re-create mapping on-demand from original application code
    - Cache consistency guarantees mean the corresponding application code is unchanged
    - Requires idempotent code transformations
- DynamoRIO supports both approaches
  - The engine mostly uses the on-demand approach, but stored mappings are occasionally needed

Instruction Translation Field

- Each instruction contains a translation field
- Holds the application address that the instruction corresponds to
- Set via instr_set_translation()

Context Translation Via Re-Creation

```
A1: mov   %ebx, %ecx
A2: add   %eax, (%ecx)
A3: cmp   $4, (%eax)
A4: jle   710349fb
```

```
C1: mov   %ebx, %ecx
C2: add   %eax, (%ecx)
C3: cmp   $4, (%eax)
C4: jle   <stub0>
C5: jmp   <stub1>
```

```
D1: (A1) mov   %ebx, %ecx
D2: (A2) add   %eax, (%ecx)
D3: (A3) cmp   $4, (%eax)
D4: (A4) jle   <stub0>
D5: (A4) jmp   <stub1>
```

Application vs. Meta Instructions

- By default, instructions are treated as application instructions
  - Must have translations: `instr_set_translation(), INSTR_XL8()`
  - Control-flow-changing app instructions are modified to retain DynamoRIO control and result in cache populating
- *Meta* instructions are added instrumentation code
  - Not treated as part of the application (e.g., calls run natively)
  - Usually cannot fault, so translations not needed
  - Created via `instr_set_meta()` or `instrlist_meta_append()`
- Meta instructions can reference application memory, or deliberately fault
  - A meta instruction that might fault must contain a translation
  - The client should handle any such fault

Client Translation Support

- Instruction lists passed to clients are annotated with translation information
  - Read via instr_get_translation()
  - Clients are free to delete instructions, change instructions and their translations, and add new tool and app instructions (see dr_register_bb_event() for restrictions)
  - An idempotent client that restricts itself to deleting app instructions and adding non-faulting meta instructions can ignore translation concerns
  - DynamoRIO takes care of instructions added by API routines (insert_clean_call(), etc.)
- Clients can choose between storing or regenerating translations on a fragment by fragment basis.

Client Regenerated Translations

- Client returns DR_EMIT_DEFAULT from its bb or trace event callback
- Client bb & trace event callbacks are re-called when translations are needed with translating==true
- Client must exactly duplicate transformations performed when the block was generated
- Client must set translation field for all added app instructions and all meta instructions that might fault
  - This is true even if translating==false since DynamoRIO may decide it needs to store translations anyway

Client Stored Translations

- Client returns DR_EMIT_STORE_TRANSLATIONS from its bb or trace event callback
- Client must set translation field for all added app instructions and all meta instructions that might fault
- Client bb or trace hook will not be re-called with translating==true

Register State Translation

- Translation may be needed at a point where some registers are spilled to memory
  - During indirect branch or RIP-relative mangling, e.g.
- DynamoRIO walks fragment up to translation point, tracking register spills and restores
  - Special handling for stack pointer around indirect calls and returns
- DynamoRIO tracks client spills and restores *implicitly* added by API routines
  - Clean calls, etc.
  - Explicit spill/restore (e.g., dr_save_reg()) client's responsibility

Client Register State Translation

- If a client adds its own register spilling/restoring code or changes register mappings it must register for the restore state event to correct the context
- The same event can also be used to fix up the application's view of memory
- DynamoRIO does not internally store this kind of translation information ahead of time when the fragment is built
  - The client must maintain its own data structures

DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

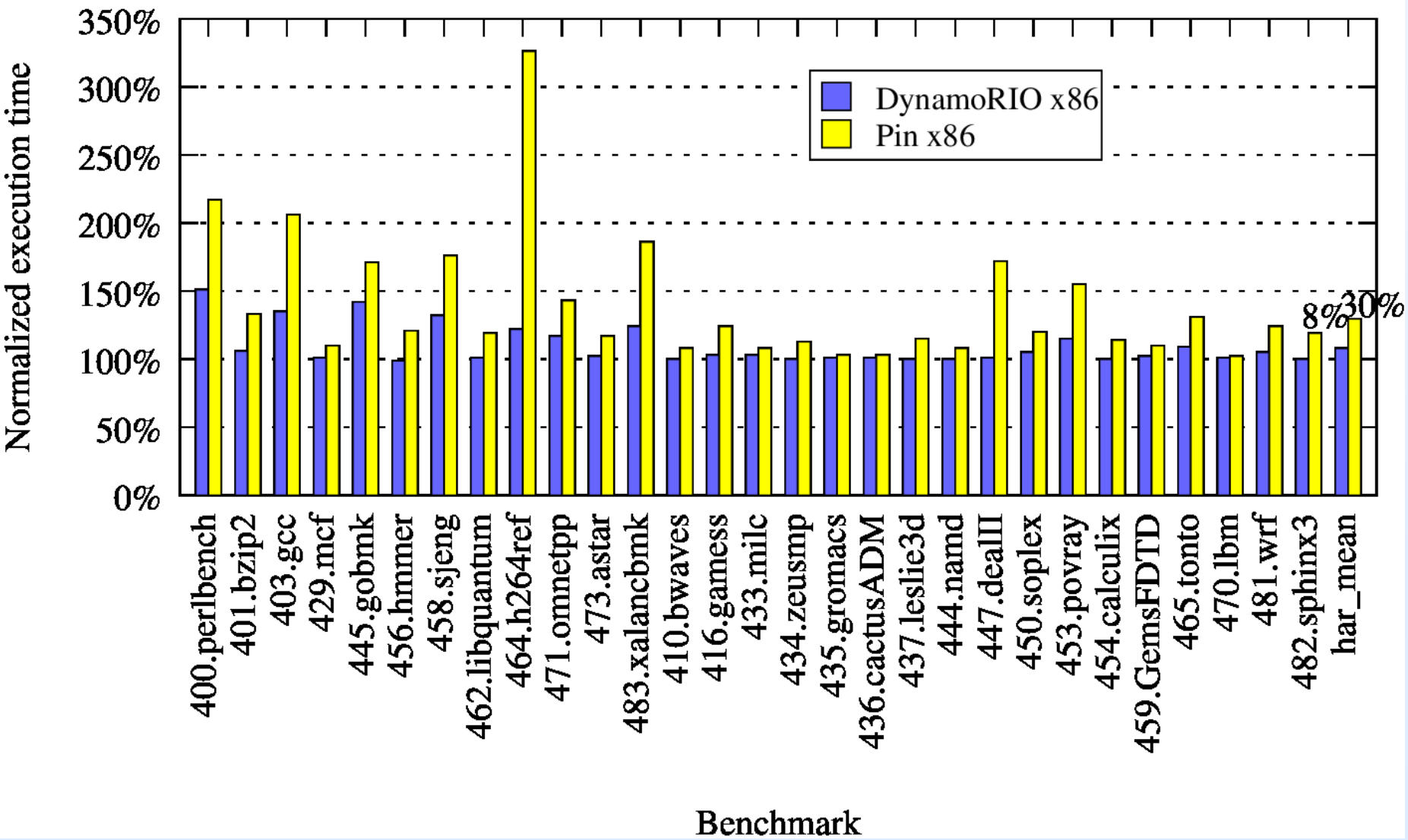
DynamoRIO versus Pin

- Basic interface is fundamentally different
- Pin = insert callout/trampoline only
  - Not so different from tools that modify the original code: Dyninst, Vulcan, Detours
  - Uses code cache only for transparency
  - *Proprietary*
- DynamoRIO = arbitrary code stream modifications
  - Only feasible with a code cache
  - Takes full advantage of power of code cache
  - General IA-32/AMD64 decode/encode/IR support
  - *Open-source*

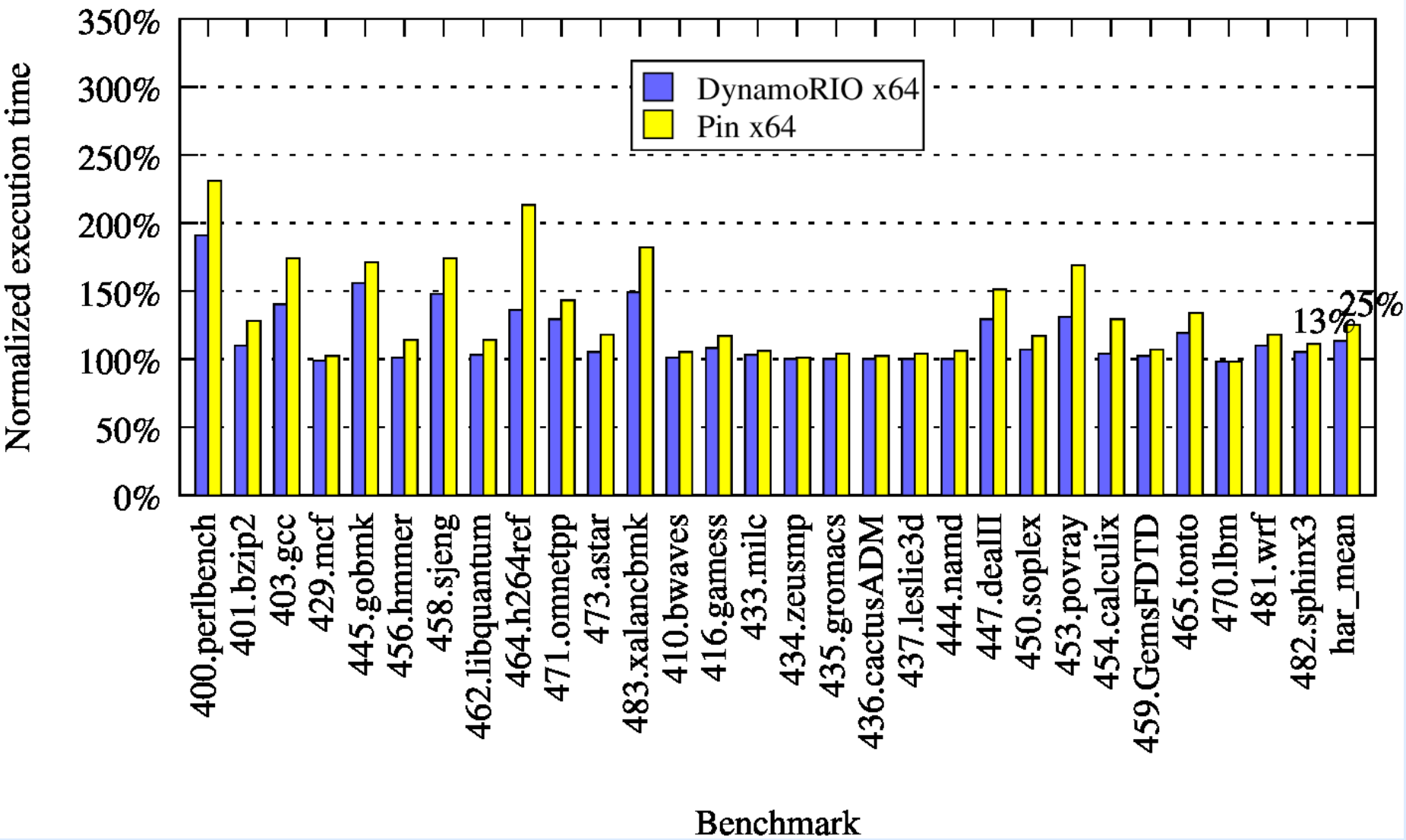
DynamoRIO versus Pin

- Pin = insert callout/trampoline only
  - Pin tries to inline and optimize
  - Client has little control or guarantee over final performance
- DynamoRIO = arbitrary code stream modifications
  - Client has full control over all inserted instrumentation
  - Result can be significant performance difference
    - PiPA Memory Profiler + Cache Simulator:
      3.27x speedup w/ DynamoRIO vs 2.6x w/ Pin
  - DynamoRIO also performs callout (“clean call”) optimization and inlining just like Pin for less performance-focused clients

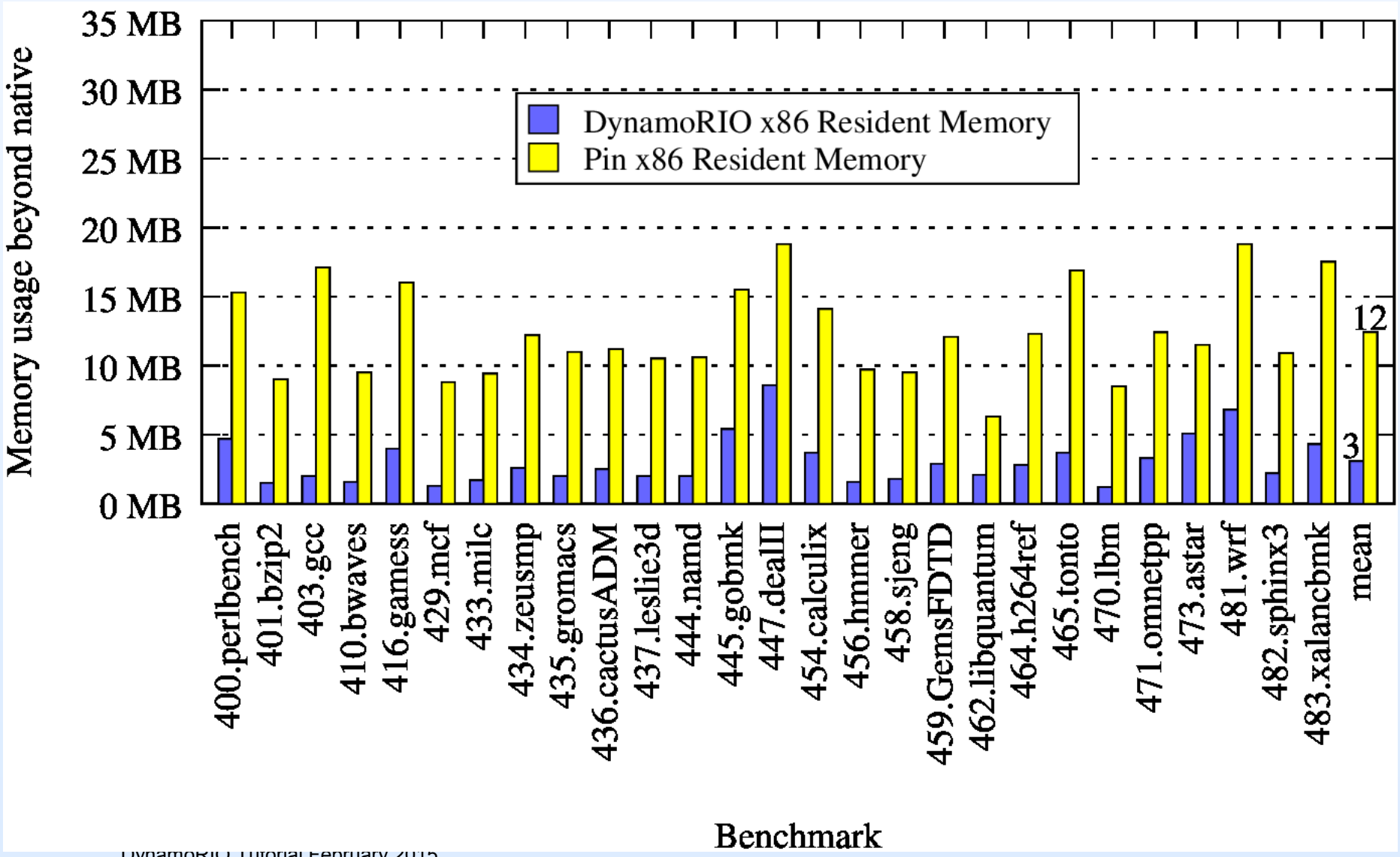
Base Performance Comparison (No Tool)



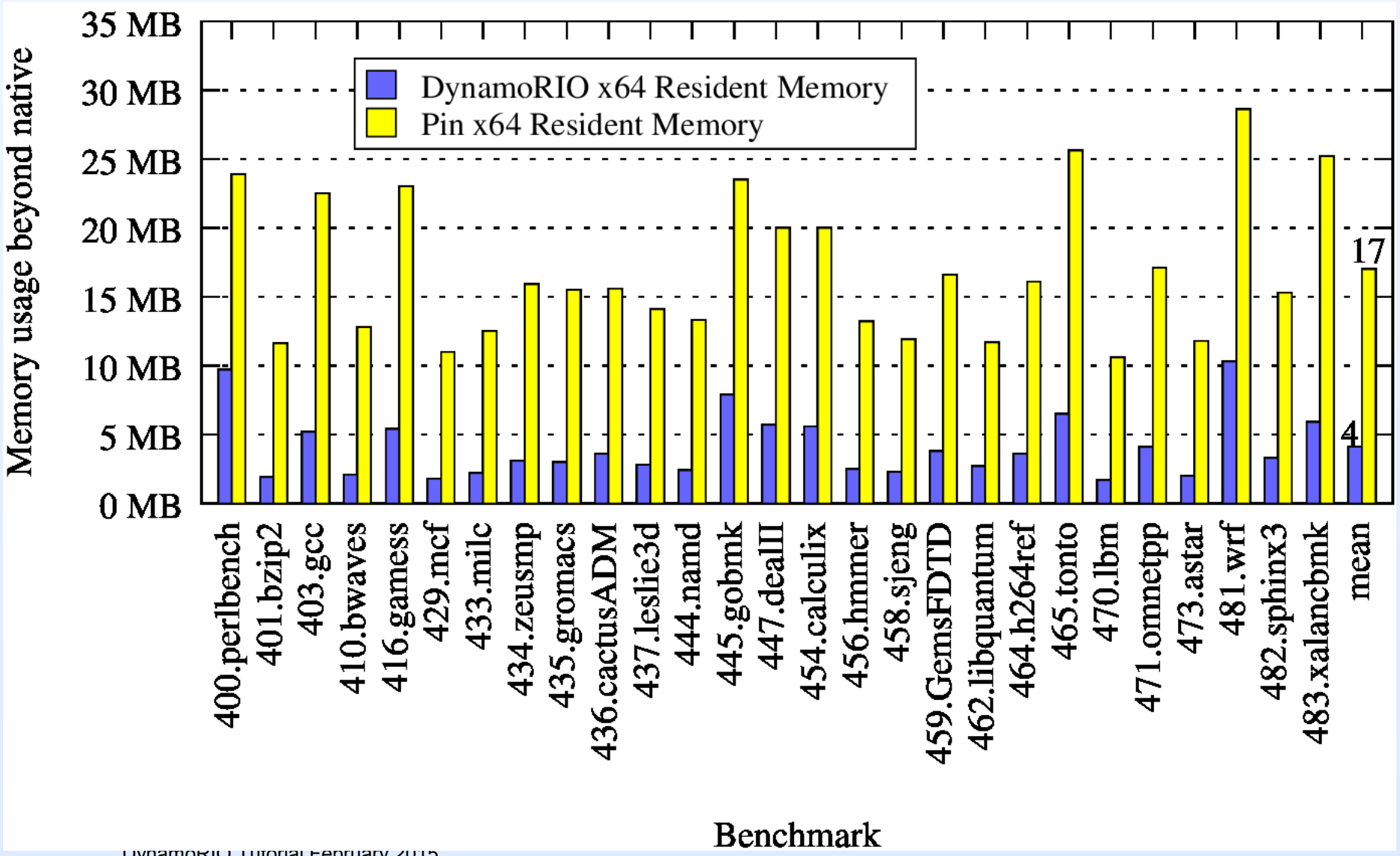
Base Performance Comparison (No Tool)



Base Memory Comparison (No Tool)



Base Memory Comparison (No Tool)



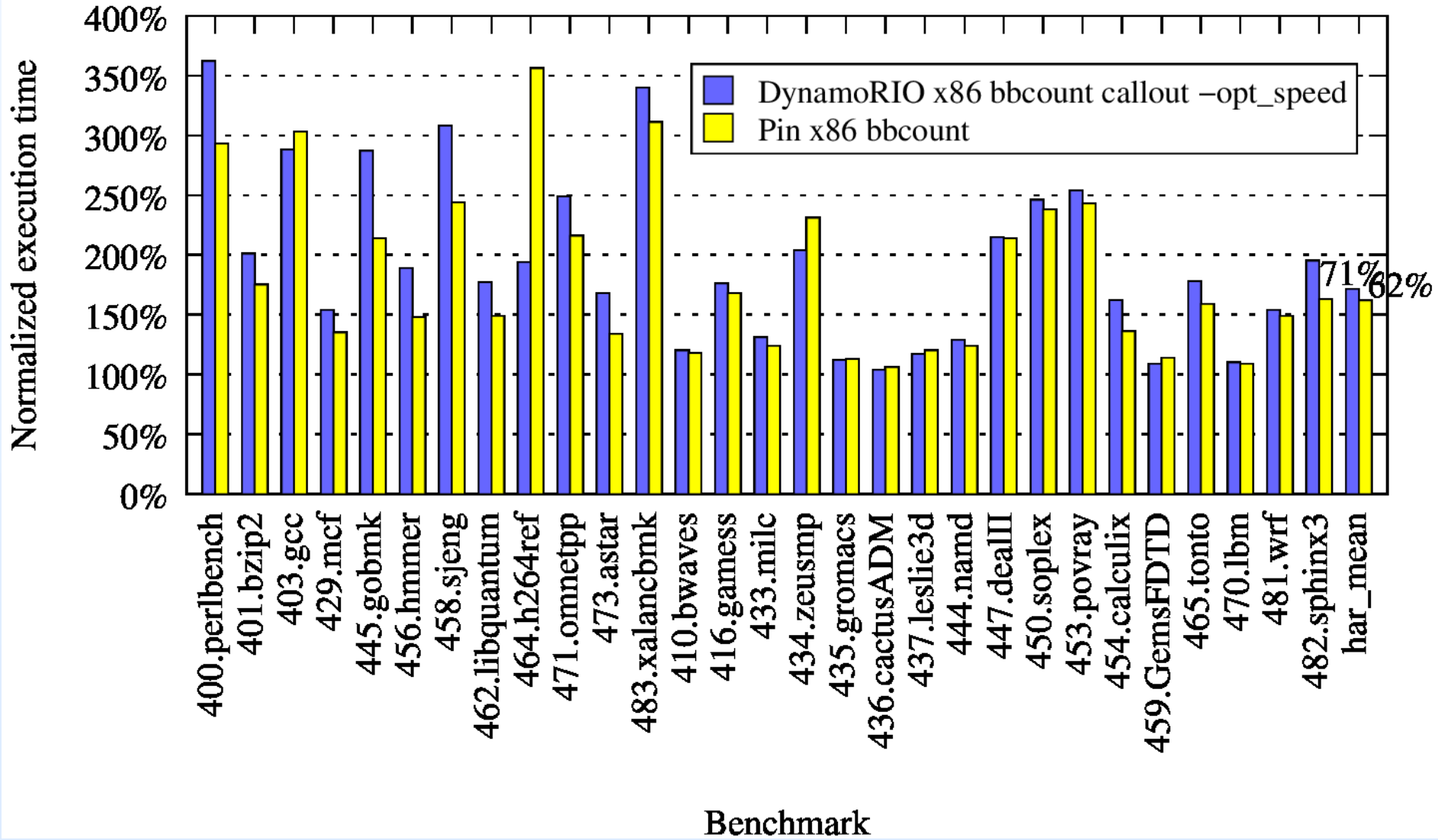
BBCount Pin Tool

```
static int bbcount;

VOID PIN_FAST_ANALYSIS_CALL docount() { bbcount++; }

VOID Trace(TRACE trace, VOID *v) {
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, AFUNPTR(docount),
                       IARG_FAST_ANALYSIS_CALL, IARG_END);
    }
}

int main(int argc, CHAR *argv[]) {
    PIN_InitSymbols();
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_StartProgram();
    return 0;
}
```

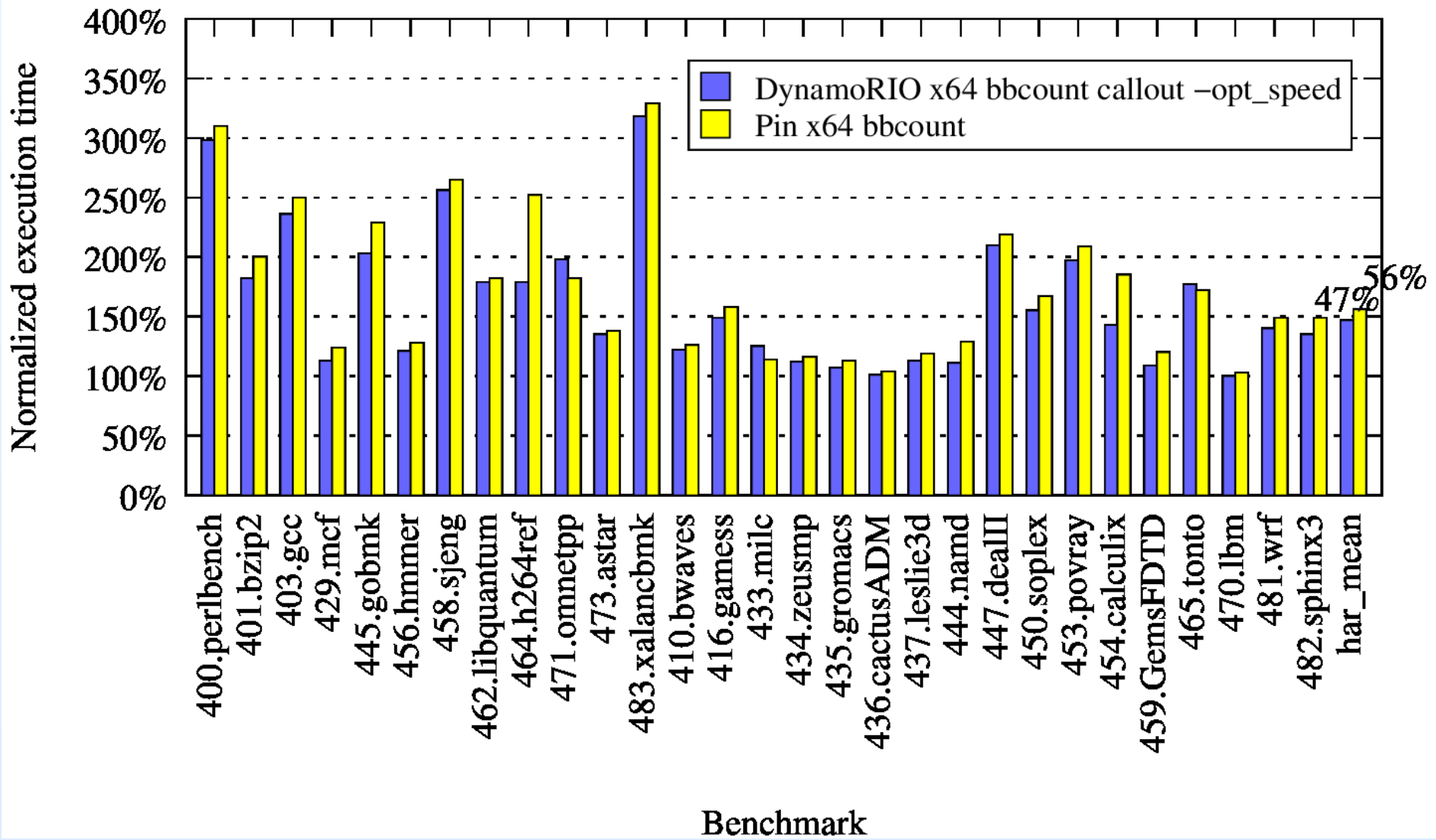
Simple BBCount DynamoRIO Tool

```
static int bbcount;

static void docount() { bbcount++; }

static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    dr_insert_clean_call(drcontext, bb, instrlist_first(bb), docount, false, 0);
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
```

BBCount Performance Comparison: Simple Tool



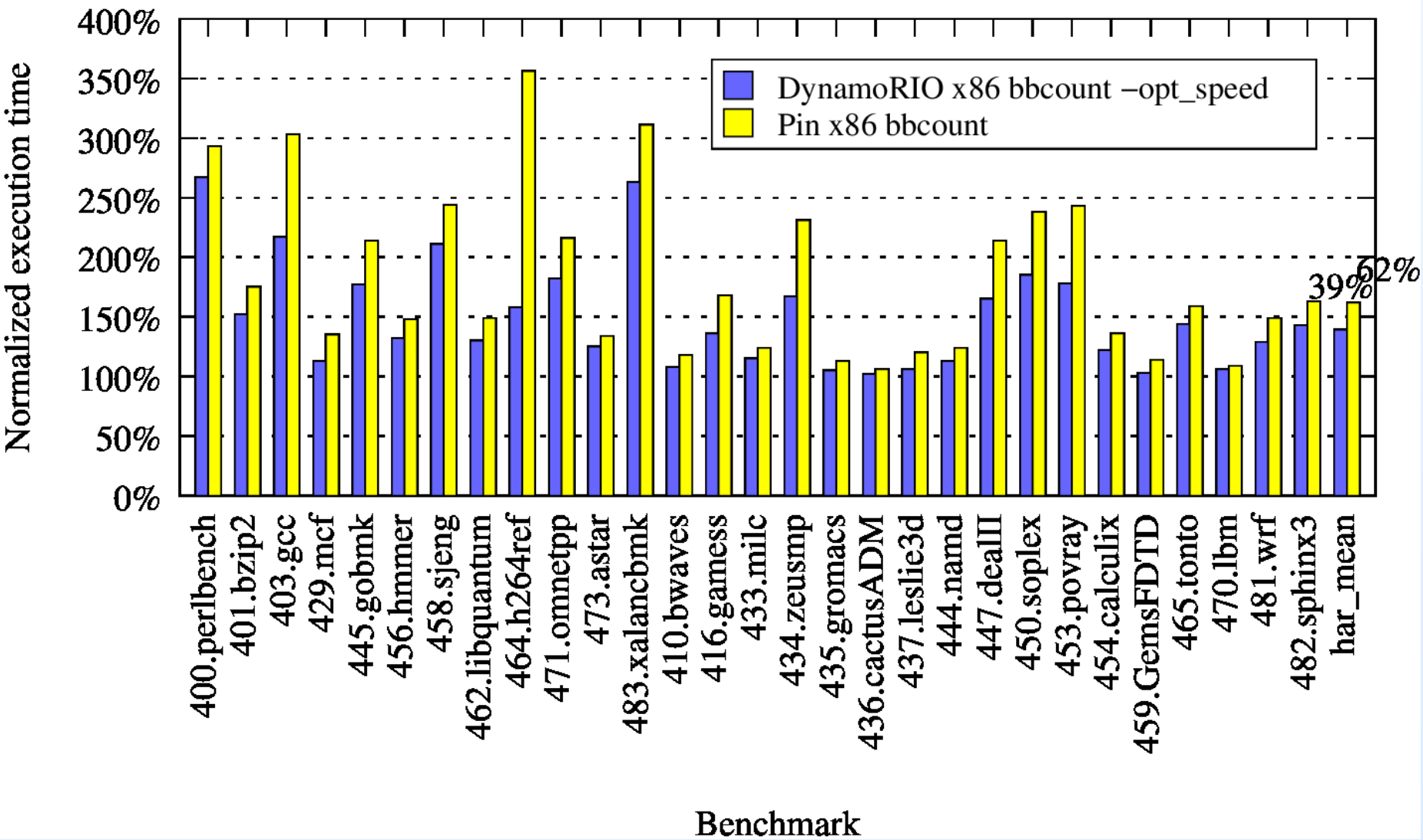
BBCount Performance Comparison: Simple Tool



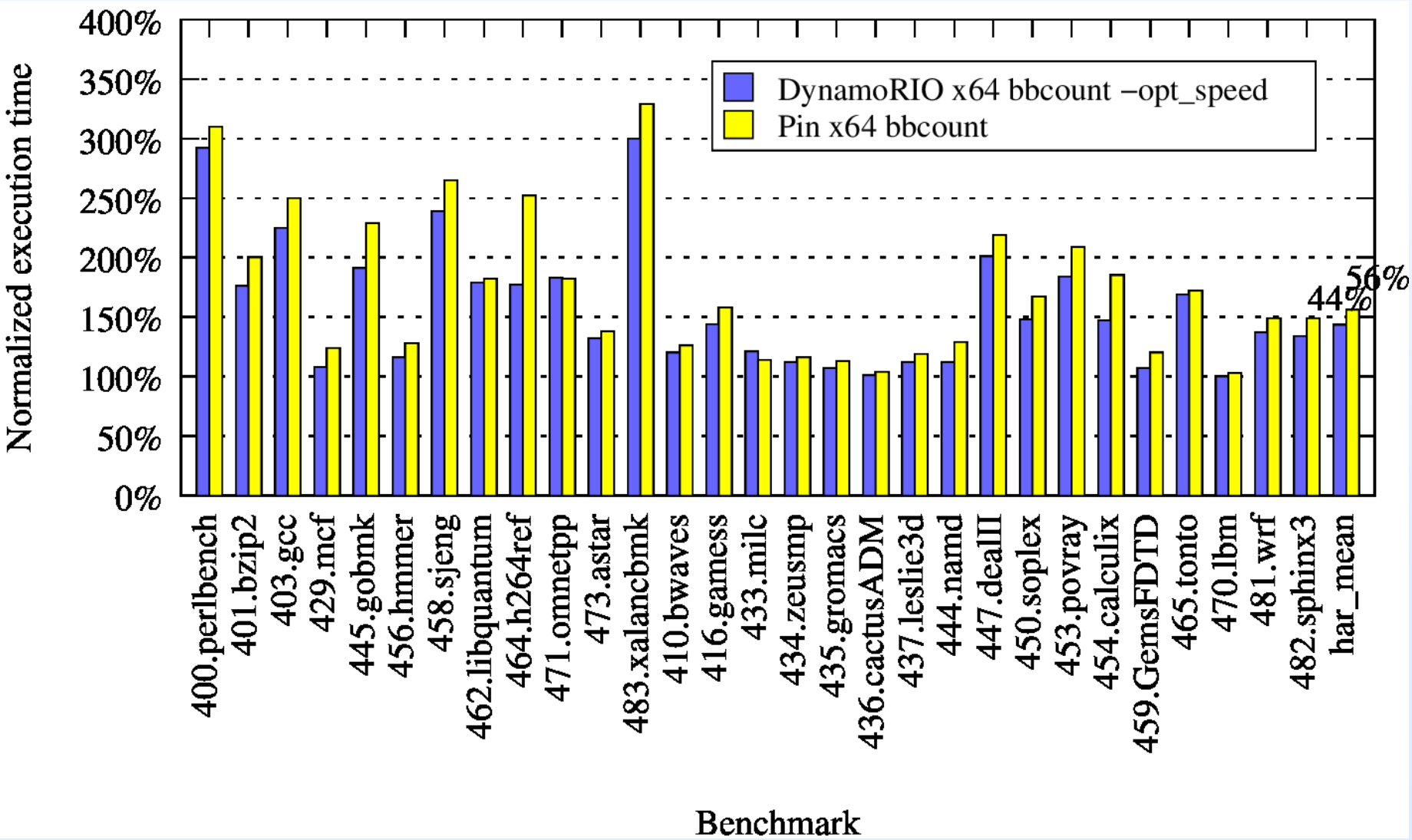
Optimized BBCount DynamoRIO Tool

```
static int global_count;

static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    instr_t *instr, *first = instrlist_first(bb);
    uint flags;
    /* Our inc can go anywhere, so find a spot where flags are dead.
     * Technically this can be unsafe if app reads flags on fault =>
     * stop at instr that can fault, or supply runtime op */
    for (instr = first; instr != NULL; instr = instr_get_next(instr)) {
        flags = instr_get_arith_flags(instr);
        /* OP_inc doesn't write CF but not worth distinguishing */
        if (TESTALL(EFLAGS_WRITE_6, flags) && !TESTANY(EFLAGS_READ_6, flags))
            break;
    }
    if (instr == NULL)
        dr_save_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    instrlist_tool_preinsert(bb, (instr == NULL) ? first : instr,
        INSTR_CREATE_inc(drcontext, OPND_CREATE_ABSMEM((byte *)&global_count, OPSZ_4)));
    if (instr == NULL)
        dr_restore_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
```

BBCount Performance Comparison: Opt Tool



BBCount Performance Comparison: Opt Tool



DynamoRIO API Outline

- Building and Deploying
- Events
- Utilities
- Instruction Manipulation
- State Translation
- Comparison with Pin
- Troubleshooting

Obtaining Help

- Read the documentation
  - http://dynamorio.org/docs/
- Look at the sample clients
  - In the documentation
  - In the release package: samples/
- Ask on the DynamoRIO Users discussion forum/mailing list
  - http://groups.google.com/group/dynamorio-users

Debugging Clients

- Use the DynamoRIO debug build for asserts
  - Often point out the problem
- Use logging
  - -loglevel N
  - stored in logs/ subdir of DR install dir
- Attach a debugger
  - gdb or windbg
  - -msgbox_mask 0xN
  - -no_hide
  - windbg: .reload myclient.dll=0xN
- More tips:
  - http://code.google.com/p/dynamorio/wiki/Debugging

Reporting Bugs

- Search the Issue Tracker off http://dynamorio.org first
  - http://code.google.com/p/dynamorio/issues/list
- File a new Issue if not found
- Follow conventions on wiki
  - http://code.google.com/p/dynamorio/wiki/BugReporting
  - CRASH, APP CRASH, HANG, ASSERT
- Example titles:
  - CRASH (1.3.1 calc.exe)
    vm_area_add_fragment:vmareas.c(4466)
  - ASSERT (1.3.0 suite/tests/common/segfault)
    study_hashtable:fragment.c:1745 ASSERT_NOT_REACHED

DynamoRIO on ARM

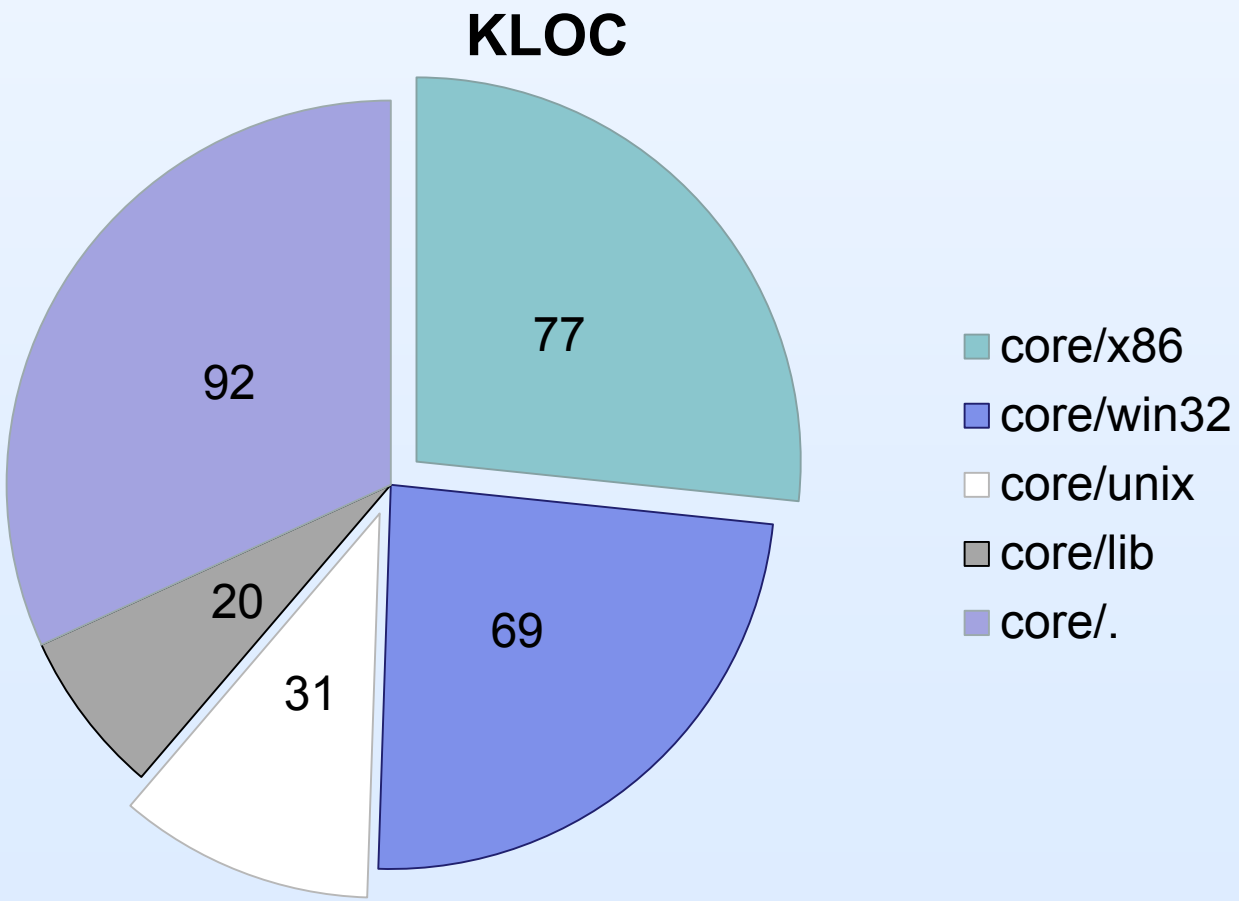
2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A

ARM Overview

- 32-bit has two separate ISA modes:
  - A32: “ARM”
    - 4-byte instructions
    - Can reference all 16 GPR registers r0-r15
    - Most GPR instructions can be predicated
    - Most GPR instructions can read or write PC
  - T32: “Thumb”
    - 2-byte or 4-byte instructions
    - 2-byte can typically only reference 8 GPR registers r0-r7
    - Predication limited to “IT blocks”
    - Very few instructions can read or write PC
- Typical Linux/ARM program is mostly T32 with some A32

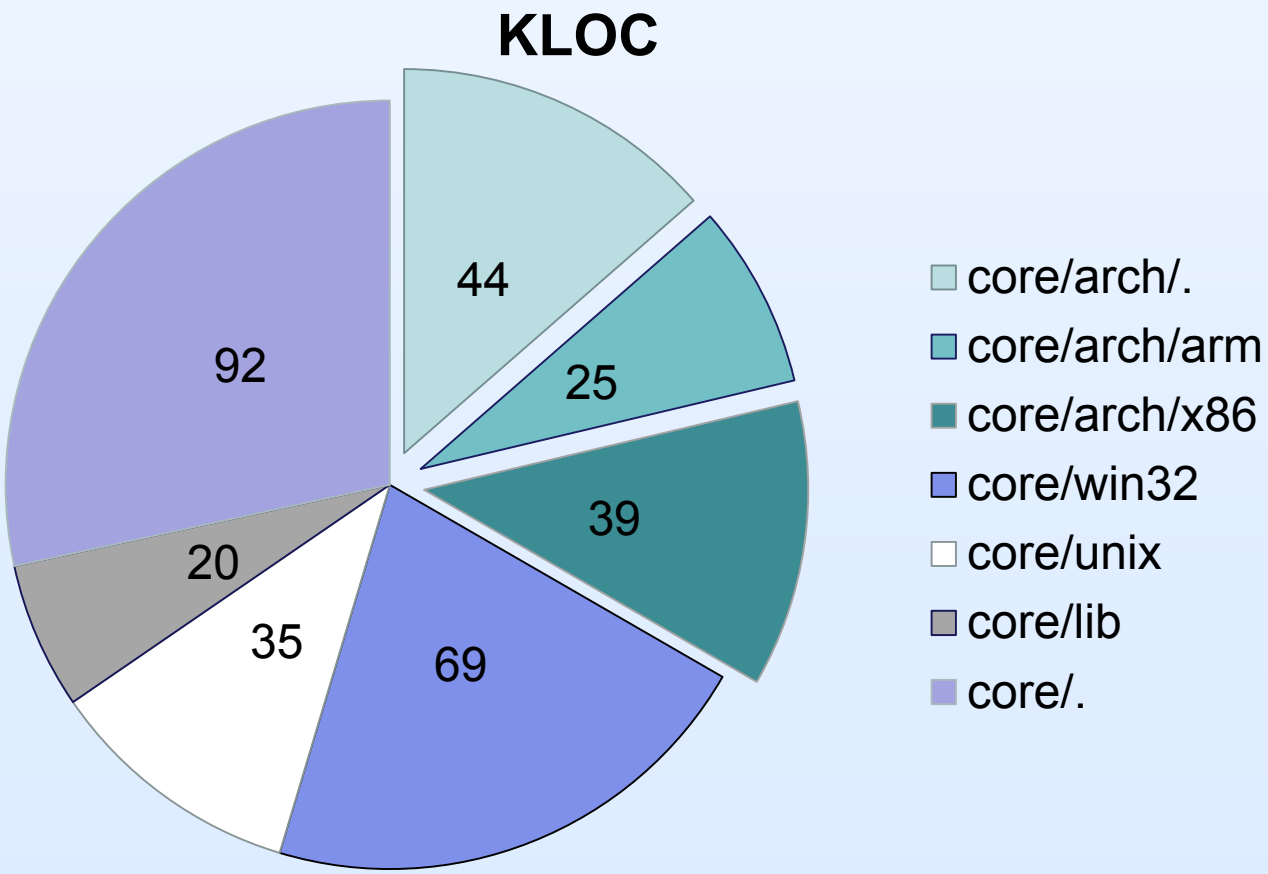
DynamoRIO ISA-Dependent Code

- 5.0.0 release breakdown of 290 KLOC:



DynamoRIO ISA-Dependent Code

- 5.1.0 release breakdown of 323 KLOC:



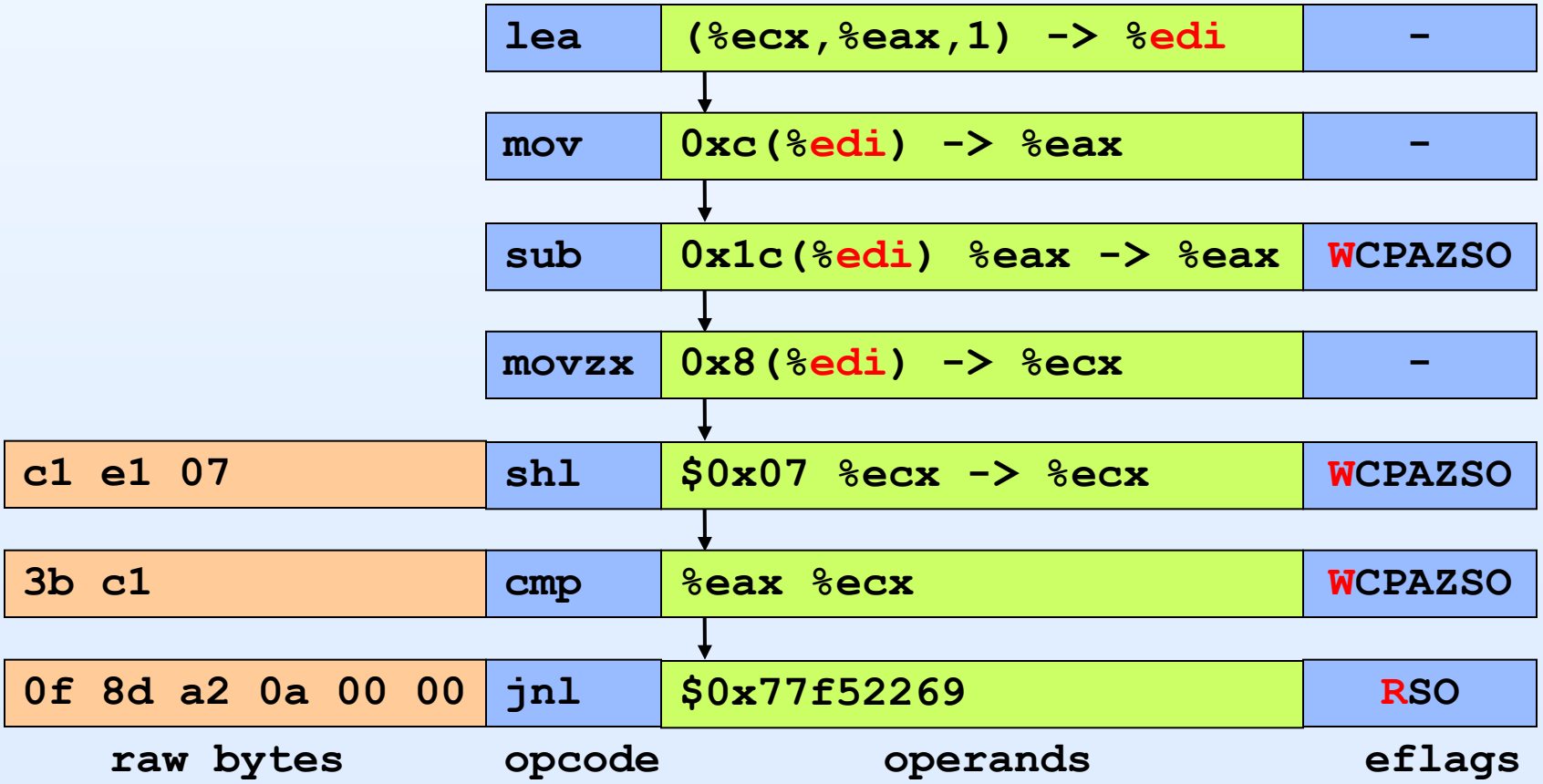
Porting to ARM Tasks

- Add ARM feature support to IR
- ARM decoder, encoder, disassembler
- Thumb decoder, encoder, disassembler
- Instruction generation
- Mangling of branches and PC references
- Port assembly code
- Port generated code
- Thread-local storage
- Cache consistency

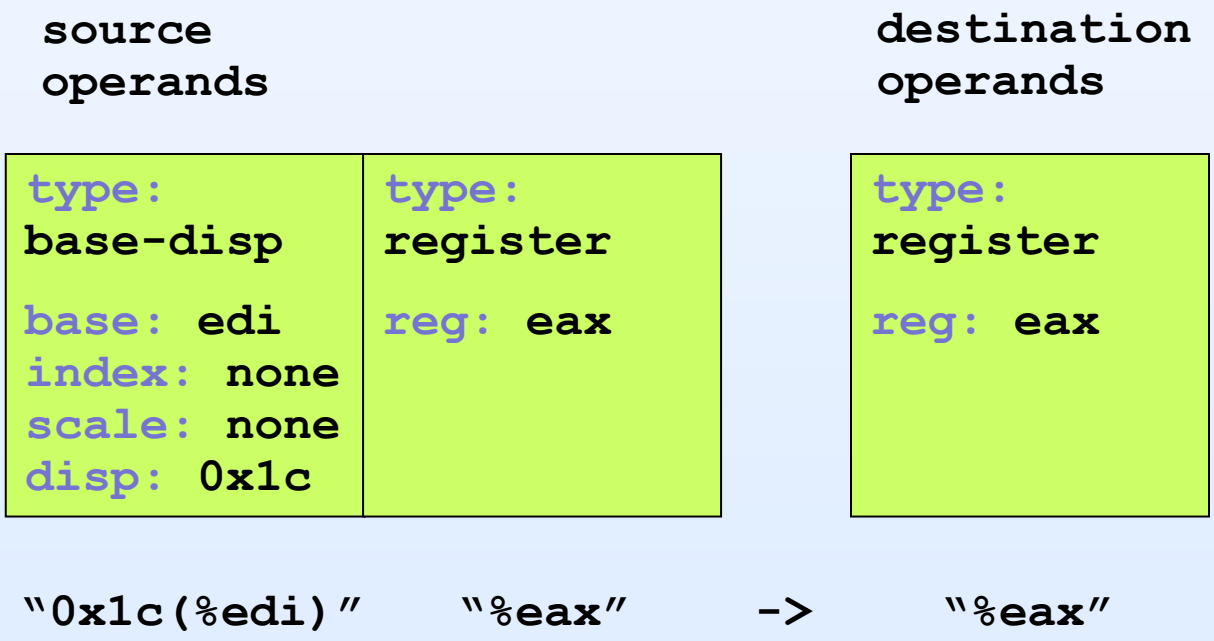
Cross-Platform Instrumentation

- Decided to keep our low-level IR for performance
  - Lack of intermediate layer is key for speed
- Instruction lists and operand arrays can be traversed and queried using the same general routines
  - Does this read memory? Does it write this register?
- New routines for generating cross-platform instrumentation
  - XINST_CREATE_load, XINST_CREATE_store
  - XINST_CREATE_add, XINST_CREATE_add_s
  - Etc.

Instruction Representation Review



Operand Representation



IR Minor Features for ARM

- Register lists
  - Simply series of regular register operands in operand list
  - Encoder uses a greedy + rollback mechanism
- Shifted registers
  - In an address: new operand fields for shift type and amount
  - In a register: shift type and amount are separate immeds
- Negated registers
  - In an address: new operand field for negation
  - In a register: informational flag on decoding
- Post-indexed and writeback addressing modes
  - Simply extra source and destination operands

Operand Representation Additions

source operands

type: base-disp base: r3 index: none shift-type: none shift-val: none negated: no disp: 0x1c	type: register reg: r3 negated: no	type: immed int val: 0x1c

destination operands

type: register reg: r7 negated: no	type: register reg: r3 negated: no

"0x1c(%r3)" "%r3" "$0x1c" -> "%r7" "%r3"

Application Predication

- Conditionally executed instructions do exist for x86
  - cmovcc, fcmovcc, bsr, bsf
  - Previously modeled by listing the destination as a source
- Several shades of predication
  - Complete conditional execution
  - Sources always read, destination written conditionally
  - Sources always read, destination written conditionally, but condition codes are always written (bsr and bsf, e.g.)

Application Predication Handling

- We added a predicate field to each instruction
- We added qualifiers to `instr_writes_to_reg()`, `instr_reads_from_reg()`, and other common analysis funcs
  - Support querying “might write” versus “definitely writes”: liveness and other analyses differ in what they want to know
- We back-ported this to x86 and removed the dst-as-src
  - Compatibility break
- Other attributes (e.g., number of dests) remain unchanged
  - User must handle predication there

Instrumentation Predication

- For pre-instruction instrumentation: use the same predicate as the application instruction
- For post-instruction instrumentation: the application instruction may have changed the condition codes
  - Open to suggestions for automated/assisted solutions
  - One approach would be to convert predicated execution to non-predicated with conditional branches
  - Instrumentation is already limited to pre-only for branches

IT Blocks

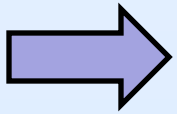
- Thumb allows predication only inside IT blocks, where an IT instruction specifies the predicate for 1-4 subsequent instrs
- Arithmetic instrs have different semantics inside IT vs outside
  - Outside they write condition codes while inside they do not
- Example:

```
cmp     r5, r2
ite     ls
movls   r3, #0
movhi   r3, #1
```

IT Block Decoding

- Complicates general decoding and encoding
  - Decode: if see IT instr, store the PC and only assume in IT block on subsequent decode calls if the PC matches
  - Encode: only support IT when encoding an entire list of instrs
- Complicates instrumentation

```
cmp     r5, r2
ite     ls
movls   r3, #0
movhi   r3, #1
```



```
cmp     r5, r2
ite     ls
movls   r3, #0
strhi   r3, <TLS-slot>
movs    r3, #1
```

IT Block Instrumentation

- Upon decoding, predicates are set for instrs inside IT block
- Tools that want to insert instrumentation in the IT block should call an API routine that removes the IT instrs
  - Inserted instrumentation can then ignore IT blocks
  - Mangling step will re-insert IT instrs for each predicated Thumb instruction
  - Non-default: some tools want to see original opcode mix
- For general instruction generation outside of app blocks, tool must insert IT instrs and set predicates on its own for successful encoding
- Open to suggestions for other ways of handling IT blocks

Thread-Local Storage

- DR requires directly-addressable TLS
- On x86 we steal a segment from the app
- On ARM there is hardware TLS support in coprocessor registers but they cannot be used as base registers in memory references
- We're forced to steal a register for TLS from the code cache
- We can use the coprocessor registers for DynamoRIO's own compiled code

Instrumentation and the Stolen Register

- DR's API is very low-level and it exposes machine registers to the tool writer
- Various ways to handle the stolen register: our solution is to have DR fully expose the stolen register to the tool
  - Add an API routine to access app value
  - Burden is on tool to not clobber TLS base
- Open to alternative suggestions
  - Auto-mangling tool's uses is complex

Branch Reachability

- Maximum branch reach is 64MB
- Fragment linking approach: use direct branches when they can reach; else link through the stub via indirect branch

Unlinked:

```
  b stub
stub:
  str r0, [r10, #r0-slot]
  movw r0, #bottom-half-&linkstub
  movt r0, #top-half-&linkstub
  ldr pc, [r10, #fcache-return-offs]
  <ptr-sized slot>
```

Linked, target < 64MB away:

```
  b target
stub:
  str r0, [r10, #r0-slot]
  movw r0, #bottom-half-&linkstub
  movt r0, #top-half-&linkstub
  ldr pc, [r10, #fcache-return-offs]
  <ptr-sized slot>
```

Linked, target > 64MB away:

```
  b stub
stub:
  ldr pc, [pc + 8]
  movw r0, #bottom-half-&linkstub
  movt r0, #top-half-&linkstub
  ldr pc, [r10, #fcache-return-offs]
  <target>
```

Current Status

- Add ARM feature support to IR
- ARM decoder, encoder, disassembler
- Thumb decoder, encoder, disassembler
- Instruction generation
- Mangling of branches and PC references
- Port assembly code
- Port generated code
- Thread-local storage
- Cache consistency

ARM vs x86

- Adding ARM support was not as simple as we expected
- Easier than x86:
  - Cache consistency
  - Instruction alignment and length
- Harder than x86:
  - Decoding and encoding: ARM is not really RISC!
    - Shifted sources, post-index + writeback, register lists, scattered opcode and immediate bits
  - Mixed ARM and Thumb mode: any indirect branch can change
  - App can read or write PC at any time
  - Branch reachability
  - Predication, IT blocks
  - Thread-local storage

Seeking ARM Contributors

- Many potential projects
  - Optimizing assembly code and generated code
  - Help port generated + assembly code in extension libraries
  - Help port generated code in DR's sample tools
  - Write a new tool and make sure it works on ARM, as a stress test of our IR and API design as well as contributing a new sample to DR.
  - Help port generated + assembly code in test suite
  - Add A64 support

DynamoRIO Roadmap: Platforms

- Current work in progress:
  - Linux/ARM
- Will require external contributions:
  - A64
  - Android/ARM
  - OSX/x86
- Hoping for external contributions:
  - Win10/x86

DynamoRIO Roadmap: API and Tools

- Add extension library
  - Register stealing coordination library
  - Shadow value propagation library
  - Callstack walking library
  - Malloc interception library
- Add awesome tools
  - Large or small

Potential DynamoRIO Collaborations

- Many potential arrangements
  - Summer internship
  - Google Summer of Code 2015
  - Industrial/academic collaboration
- Recent academic collaborations:
  - CGO 2015 paper
    - *Optimizing Binary Translation for Dynamically Generated Code,*
      Byron Hawkins and Brian Demsky (UC Irvine); Derek Bruening and Qin Zhao (Google)
  - PLDI 2015 paper
    - *Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code,*
      Charith Mendis, Jeffrey Bosboom, Kevin Wu, and Shoaib Kamil (MIT CSAIL); Jonathan Ragan-Kelley (Stanford); Sylvain Paris (Adobe); Qin Zhao (Google); Saman Amarasinghe (MIT CSAIL)

Q & A / Feedback

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Overview
3:10-3:45	Creating Simple Tools
3:45-4:00	*Break*
4:00-4:30	Creating Complex Tools
4:30-4:50	DynamoRIO API
4:50-5:15	DynamoRIO on ARM
5:15-5:30	Q & A