Building Dynamic Tools with DynamoRIO on x86 and ARMv8

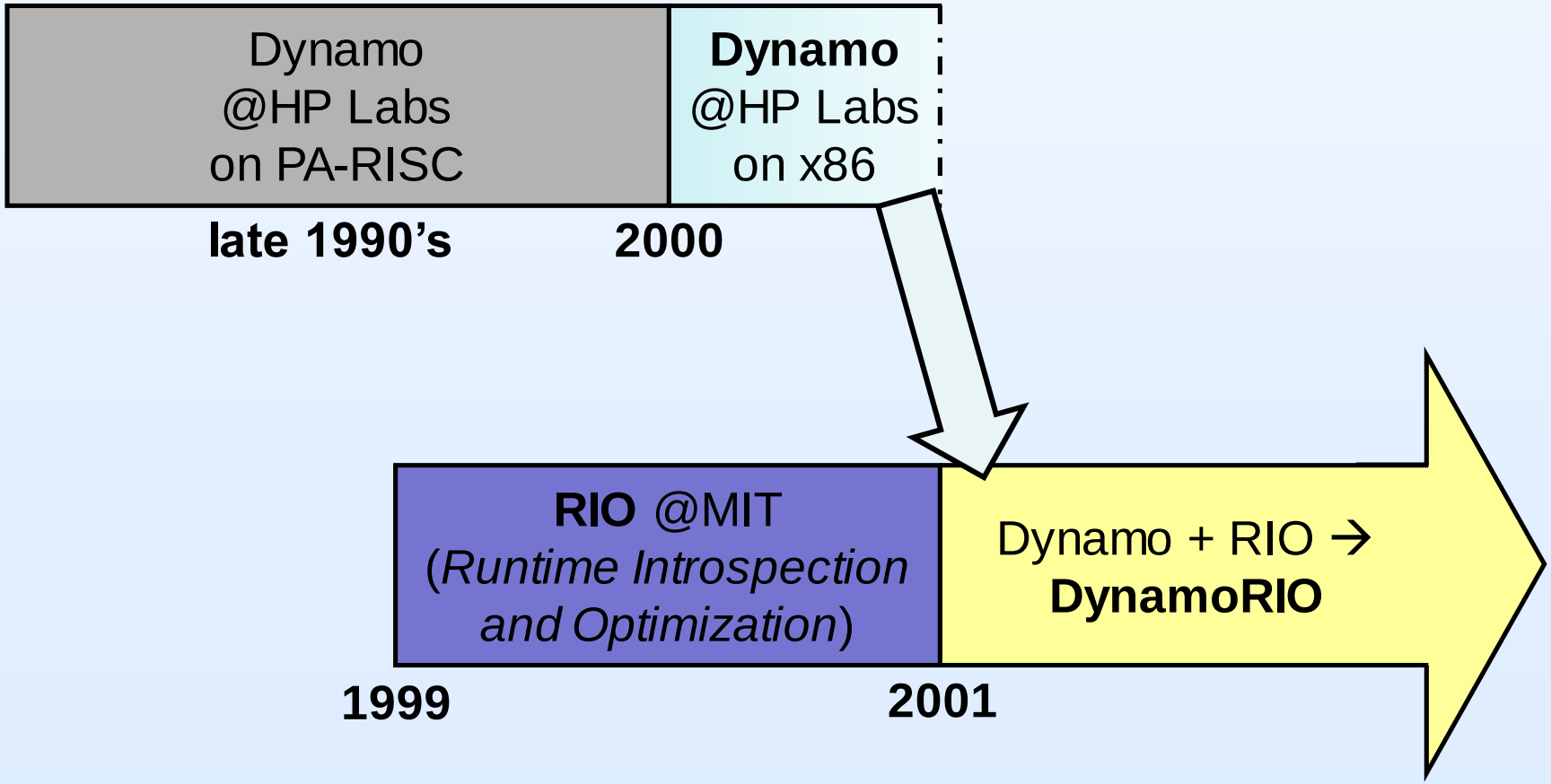
Derek Bruening
Google

Al Grant
Chris Adeniyi-Jones
Edmund Grimley Evans
Kevin Zhou
ARM

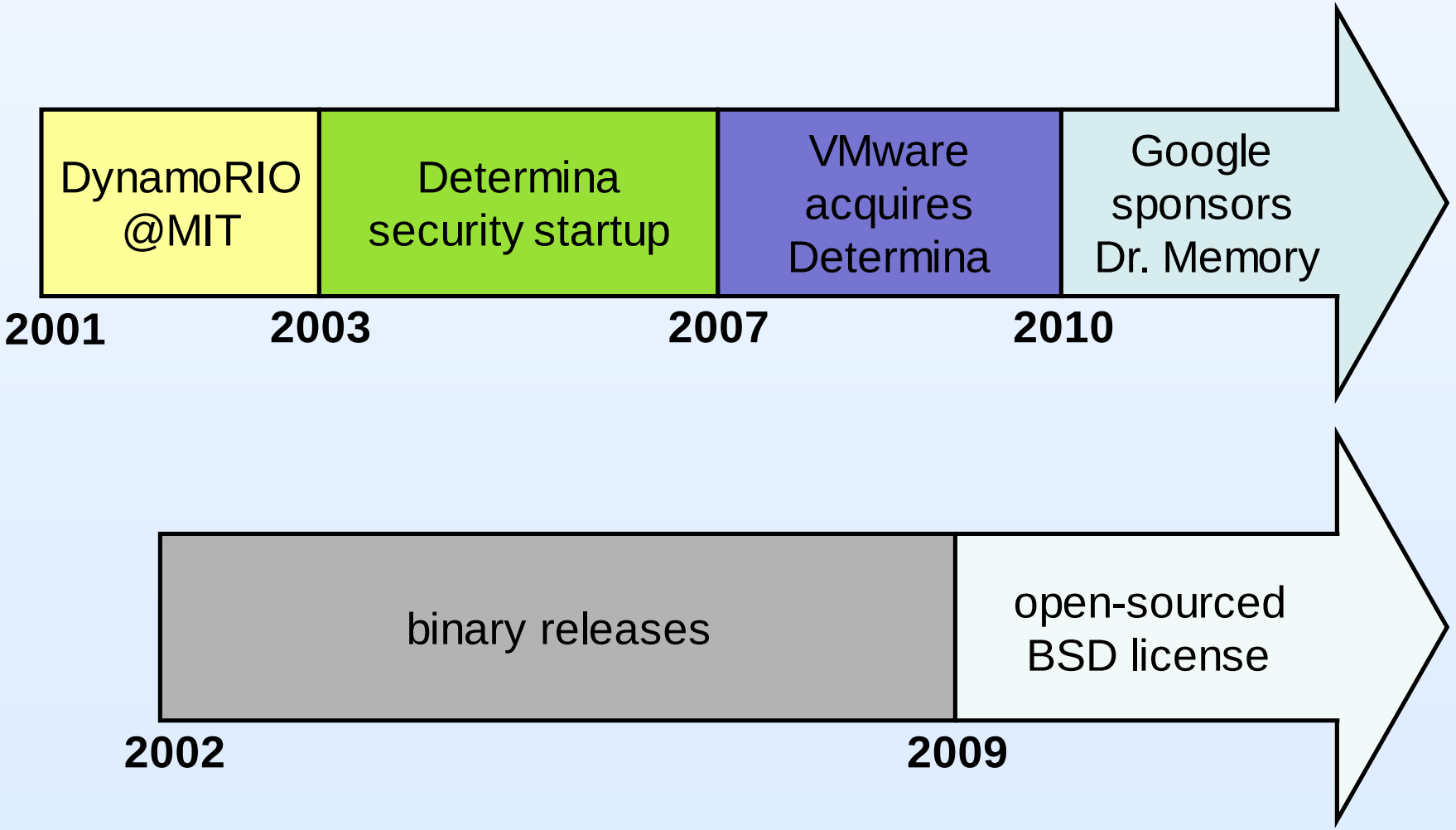
Tutorial Outline

- 8:30- 8:40 Welcome + DynamoRIO History
- 8:40- 9:10 Tool Demonstrations
- 9:10- 9:30 DynamoRIO System Overview
- 9:30- 9:45 DynamoRIO API Part 1
- 9:45-10:00 ARM Status + Demonstrations
- *10:00-10:30 Break*
- 10:30-10:55 DynamoRIO API Part 2
- 10:55-11:20 How to Create Your Own Tool
- 11:20-11:40 Tool Internals
- 11:40-11:45 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



Tool Demonstrations

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

Memory Trace Tool

- Instruments every load and store
- Writes the trace of memory accesses to a file
- Usage:
  - **`drrun -c samples/bin64/libmemtrace_x86_text.so -- <app>`**

Memory Trace Tool Results

```
% bin64/drrun -c samples/bin64/libmemtrace_x86_text.so --
~/multi_threaded_pi_estimator

Format: <instr address>,<(r)ead/(w)rite>,<data size>,<data address>
0x00007fb31a5462d3,w,8,0x00007ffc4febfb18
0x00007fb31a549a74,w,8,0x00007ffc4febfb08
0x00007fb31a549a8f,r,8,0x00007fb31a767e70
0x00007fb3199a54fc,r,4,0x00007ffc4febf438
0x00007fb3199a5526,r,1,0x0000000000400e98
0x00007fb31a549a96,w,8,0x00007fb31a767c98
0x00007fb31a549aa7,r,8,0x00007fb31a768000
0x00007fb3199af4a0,r,8,0x00007ffc4febf9f8
0x0000000000400db3,r,8,0x00007ffc4febfa40
0x0000000000400db4,r,8,0x00007ffc4febfa48
```

Dr. CacheSim

- Multi-process, multi-thread online cache simulator
- Each thread collects and sends its own memory access trace to a single simulator
- Simple static scheduling
- Extensible
- Usage:
  - **`drrun -t drcachesim -- <app>`**

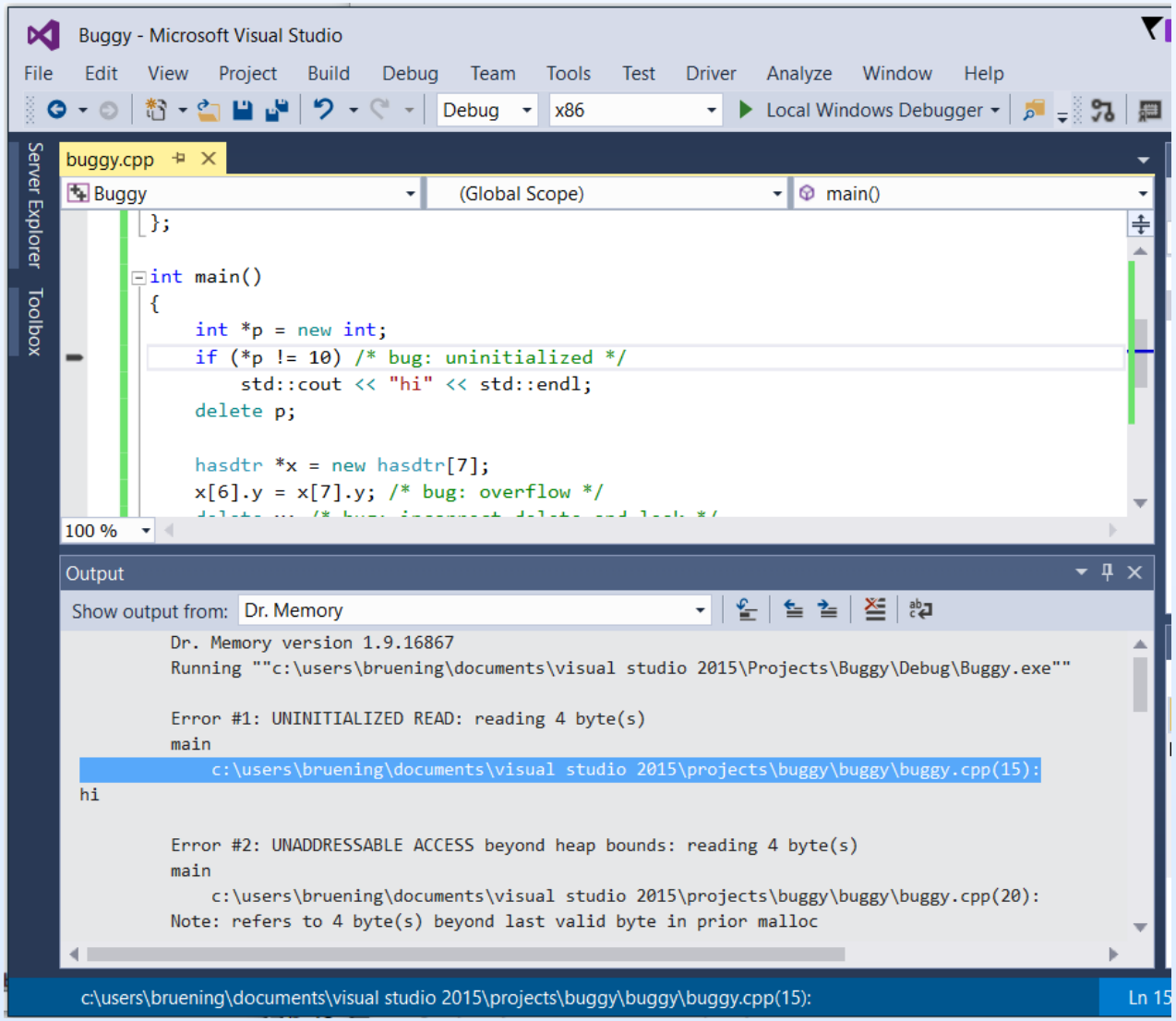
Dr. CacheSim Results

```
% bin64/drrun -t drcachesim -- ./multi_threaded_pi_estimator
Estimation of pi is 3.142425985001098
---- <application exited with code 0> ----
Core #0 (1 thread(s))
  L1I stats:
    Hits:                          243,293
    Misses:                          1,077
    Miss rate:                        0.44%
  L1D stats:
    Hits:                           67,738
    Misses:                          3,471
    Miss rate:                        4.87%
Core #1 (1 thread(s))
  L1I stats:
    Hits:                            8,589
    Misses:                             95
    Miss rate:                        1.09%
  L1D stats:
    Hits:                            3,302
    Misses:                            211
    Miss rate:                        6.01%
Core #2 (1 thread(s))
  L1I stats:
    Hits:                            1,906
    Misses:                             94
    Miss rate:                        4.70%
  L1D stats:
    Hits:                              749
    Misses:                            121
    Miss rate:                       13.91%
Core #3 (0 thread(s))
LL stats:
    Hits:                            1,186
    Misses:                          3,883
    Local miss rate:                76.60%
    Child hits:                    325,577
    Total miss rate:                  1.17%
```

Dr. Memory: Memory Debugger

- Detects reads of uninitialized memory
- Detects heap errors
  - Out-of-bounds accesses (underflow, overflow)
  - Access to freed memory
  - Invalid frees
  - Memory leaks
- Detects other accesses to invalid memory
  - Stack tracking
- Detects Windows handle leaks
- Operates at runtime on unmodified Windows, Linux, MacOS, and Android binaries

Dr. Memory Results

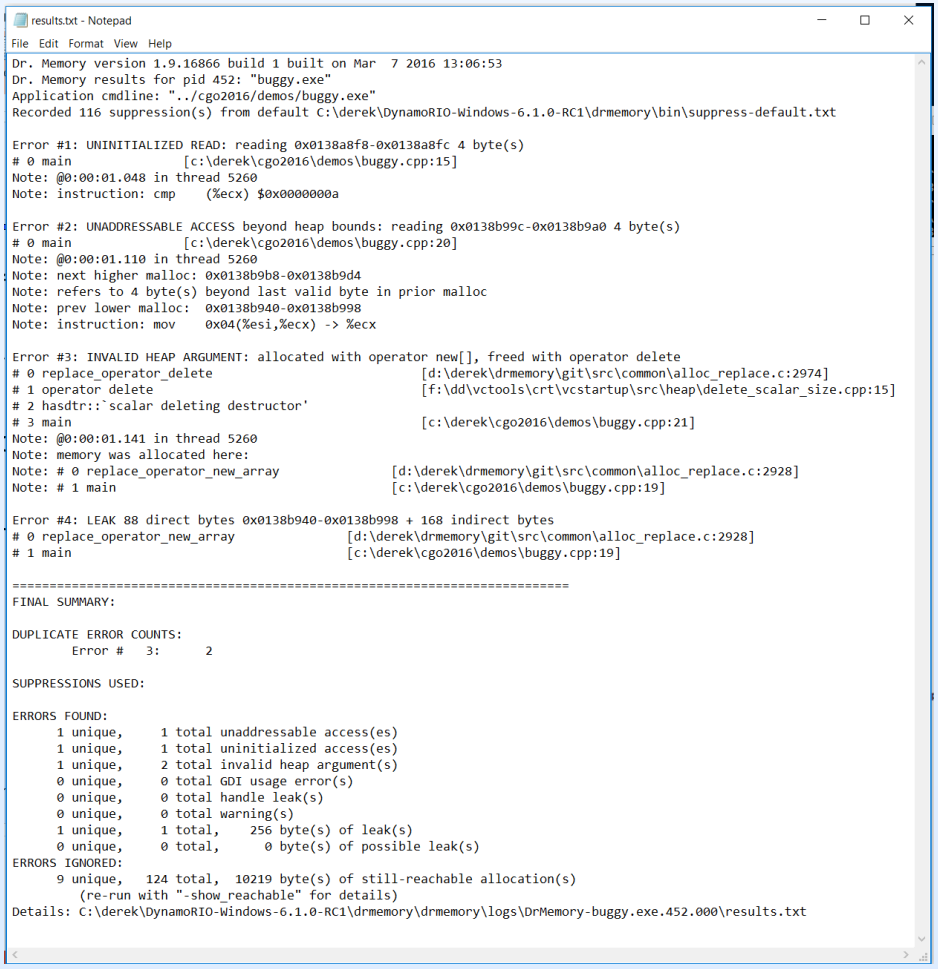


Dr. Memory Command Line Results

- To run:

  `drrun -t drmemory -- <app>`

- On Windows, notepad shows results



results.txt - Notepad

File Edit Format View Help

```
Dr. Memory version 1.9.16866 build 1 built on Mar  7 2016 13:06:53
Dr. Memory results for pid 452: "buggy.exe"
Application cmdline: "../cgo2016/demos/buggy.exe"
Recorded 116 suppression(s) from default C:\derek\DynamoRIO-Windows-6.1.0-RC1\drmemory\bin\suppress-default.txt

Error #1: UNINITIALIZED READ: reading 0x0138a8f8-0x0138a8fc 4 byte(s)
# 0 main                [c:\derek\cgo2016\demos\buggy.cpp:15]
Note: @0:00:01.048 in thread 5260
Note: instruction: cmp    (%ecx) $0x0000000a

Error #2: UNADDRESSABLE ACCESS beyond heap bounds: reading 0x0138b99c-0x0138b9a0 4 byte(s)
# 0 main                [c:\derek\cgo2016\demos\buggy.cpp:20]
Note: @0:00:01.110 in thread 5260
Note: next higher malloc: 0x0138b9b8-0x0138b9d4
Note: refers to 4 byte(s) beyond last valid byte in prior malloc
Note: prev lower malloc:  0x0138b940-0x0138b998
Note: instruction: mov    0x04(%esi,%ecx) -> %ecx

Error #3: INVALID HEAP ARGUMENT: allocated with operator new[], freed with operator delete
# 0 replace_operator_delete                             [d:\derek\drmemory\git\src\common\alloc_replace.c:2974]
# 1 operator delete                                     [f:\dd\vctools\crt\vcstartup\src\heap\delete_scalar_size.cpp:15]
# 2 hasdtr::`scalar deleting destructor'
# 3 main                                                [c:\derek\cgo2016\demos\buggy.cpp:21]
Note: @0:00:01.141 in thread 5260
Note: memory was allocated here:
Note: # 0 replace_operator_new_array              [d:\derek\drmemory\git\src\common\alloc_replace.c:2928]
Note: # 1 main                                    [c:\derek\cgo2016\demos\buggy.cpp:19]

Error #4: LEAK 88 direct bytes 0x0138b940-0x0138b998 + 168 indirect bytes
# 0 replace_operator_new_array              [d:\derek\drmemory\git\src\common\alloc_replace.c:2928]
# 1 main                                    [c:\derek\cgo2016\demos\buggy.cpp:19]

===========================================================================
FINAL SUMMARY:

DUPLICATE ERROR COUNTS:
       Error #   3:      2

SUPPRESSIONS USED:

ERRORS FOUND:
      1 unique,     1 total unaddressable access(es)
      1 unique,     1 total uninitialized access(es)
      1 unique,     2 total invalid heap argument(s)
      0 unique,     0 total GDI usage error(s)
      0 unique,     0 total handle leak(s)
      0 unique,     0 total warning(s)
      1 unique,     1 total,    256 byte(s) of leak(s)
      0 unique,     0 total,      0 byte(s) of possible leak(s)
ERRORS IGNORED:
      9 unique,   124 total,  10219 byte(s) of still-reachable allocation(s)
         (re-run with "-show_reachable" for details)
Details: C:\derek\DynamoRIO-Windows-6.1.0-RC1\drmemory\drmemory\logs\DrMemory-buggy.exe.452.000\results.txt
```

Dr. Fuzz

- In-process function-level fuzzing
- Repeatedly runs a target function, varying the inputs
- Integrated with Dr. Memory for finding errors sooner
  - Also available in a standalone library
- Basic usage:
  - **`drmemory -fuzz_function <func> -- <app>`**
- Includes typical fuzz testing features:
  - Code coverage-guided fuzzing
  - Input corpus
  - Input dictionary
  - Custom mutators

Dr. Fuzz Results

```
% drrun -t drmemory -batch -fuzz_function some_function -fuzz_num_iters 10 -- ./fuzzme.exe
~~Dr.M~~ Dr. Memory version 1.11.0
~~Dr.M~~ Running "./fuzzme.exe"
in some_function
hello!
in some_function
~~Dr.M~~
~~Dr.M~~ Error #1: UNADDRESSABLE ACCESS beyond heap bounds: reading 0x00f58efc-0x00f58f00 4 byte(s)
~~Dr.M~~ # 0 some_function              [c:\derek\cgo2017\demos\fuzzme.c:8]
~~Dr.M~~ # 1 main                      [c:\derek\cgo2017\demos\fuzzme.c:18]
~~Dr.M~~ Note: @0:00:00.609 in thread 4956
~~Dr.M~~ Note: next higher malloc: 0x00f58f20-0x00f59f20
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
~~Dr.M~~ Note: prev lower malloc:  0x00f58ef0-0x00f58efc
~~Dr.M~~ Note: instruction: cmp    (%eax,%edx,4) $0x00000000
in some_function
in some_function
hello!
in some_function
in some_function
in some_function
in some_function
in some_function
in some_function
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       1 unique,     8 total unaddressable access(es)
```

Opcode Mix Tool

- Monitors every instruction executed
- Counts the dynamic executions of each ISA opcode for the target application workload
- Usage:
  - **`drrun -c samples/bin32/opcodes.dll -- <app>`**

Opcode Mix Tool Results

```
% bin64/drrun -c samples/bin64/libopcodes.so -- gzip lib64/libdrinjectlib.a
    Client opcodes is running
    Top 15 opcode execution counts in 64-bit AMD64 mode:
        3932460 : shl
        4607686 : shr
        6736566 : cmovnb
        7110994 : test
       13250403 : mov
       15827664 : xor
       17002535 : lea
       20301339 : add
       23794160 : jnb
       24993299 : jz
       26312571 : movsxd
       28064475 : sub
       29210955 : and
       37652399 : jnz
       69217406 : mov
       71688400 : movzx
       79491451 : cmp
```

Dr. CPUSim: Legacy CPU Checker

- A tool for testing that applications will run correctly on legacy processors
- Looks for instructions that are illegal on the specified CPU model
- Fools **`cpuid`** checks
- Usage:
  - **`drrun -t drcpusim -cpu PentiumPro -- <app>`**

Dr. CPUSim Results

```
% bin32/drrun -t drcpusim -cpu Pentium -continue -- hello.exe
<Invalid Pentium instruction "movd" @ hello.exe+0x85c3.
Continuing.>
<Invalid Pentium instruction "pshufd" @ hello.exe+0x85c7.
Continuing.>
<Invalid Pentium instruction "movups" @ hello.exe+0x85ce.
Continuing.>
<Invalid Pentium instruction "movdqa" @ hello.exe+0x85f0.
Continuing.>
```

Dr. Strace: System Call Tracer

- Monitors all system calls executed by a target application
- Prints a trace of system calls and their arguments
- Operates on Windows, Linux, and Android
- Usage:
  - **`drrun -t drstrace -- <app>`**

Dr. Strace Results

```
NtGdiGetTextFaceW
        arg 0: 0x740122ad (type=HANDLE, size=0x4)
        arg 1: 0x20 (type=int, size=0x4)
        arg 2: 0x001fcd10 (type=<struct>*, size=0x4)
        arg 3: 0x0 (type=bool, size=0x4)
    succeeded =>
        arg 2: <NYI> (type=<struct>*, size=0x4)
        retval: 0x9 (type=int, size=0x4)
NtOpenKeyEx
        arg 0: 0x001fcd0c (type=HANDLE*, size=0x4)
        arg 1: 0x109 (type=unsigned int, size=0x4)
        arg 2: len=0x18, root=0x3c, name=150/152 "SOFTWARE\Microsoft\Windows
           NT\CurrentVersion\LanguagePack\SurrogateFallback", att=0x40, sd=0x00000000,
           sqos=0x00000000 (type=OBJECT_ATTRIBUTES*, size=0x4)
        arg 3: REG_OPTION_RESERVED or REG_OPTION_NON_VOLATILE (type=named constant,
           value=0x0, size=0x4)
    succeeded =>
        arg 0: 0x001fcd0c => 0x134 (type=HANDLE*, size=0x4)
        retval: 0x0 (type=NTSTATUS, size=0x4)
NtQueryKey.KeyCachedInformation
        arg 0: 0x134 (type=HANDLE, size=0x4)
        arg 1: 0x4 (type=named constant, size=0x4)
        arg 2: 0x001fcb5c (type=<struct>*, size=0x4)
        arg 3: 0xb0 (type=unsigned int, size=0x4)
        arg 4: 0x001fca34 (type=unsigned int*, size=0x4)
    succeeded =>
        arg 2: _KEY_CACHED_INFORMATION {_LARGE_INTEGER {0x1ca043f05a7a595}, int=0x0, int=0x4,
           int=0x1a, int=0x1, int=0xc, int=0x18, int=0x22} (type=<struct>*, size=0x4)
        arg 4: 0x001fca34 => 0x28 (type=unsigned int*, size=0x4)
        retval: 0x0 (type=NTSTATUS, size=0x4)
```

Dr. Ltrace: Library Tracer

- Intercepts calls to exported library functions
- Prints a trace of each function called and some argument information
- Usage:
  - **`drrun -t drltrace -- <app>`**

Dr. Ltrace Results

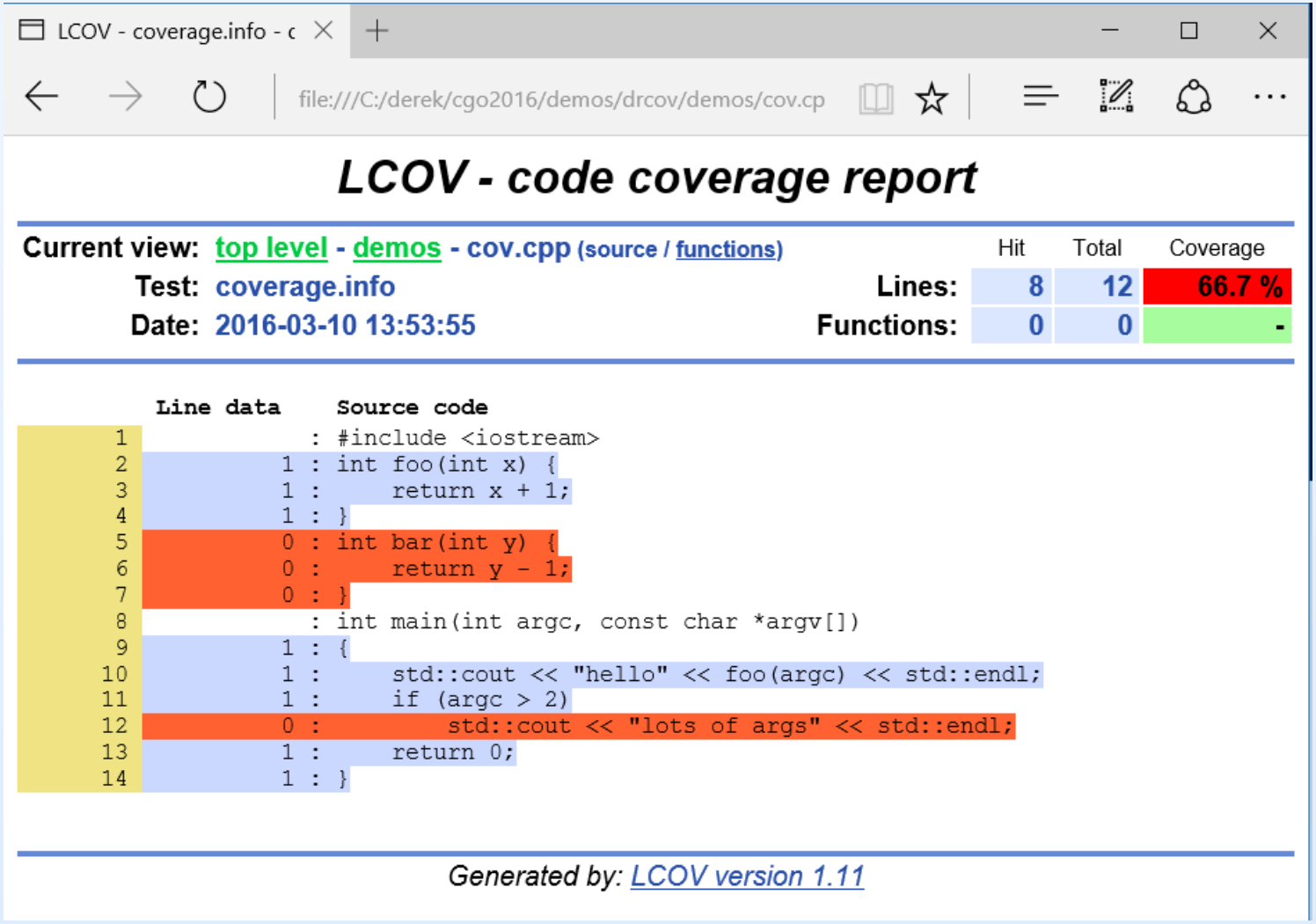
- Snippet from running an app that prints to the screen:

```
~~~~ ntdll.dll!RtlEnterCriticalSection(0x006fc758, 0x0018fe50)
~~~~ KERNEL32.dll!WriteFile(0x00000268, 0x0018e9b0)
~~~~ KERNELBASE.dll!WriteFile(0x00000268, 0x0018e9b0)
Hello, world!
~~~~ ntdll.dll!RtlLeaveCriticalSection(0x006fc758, 0x0018fe50)
```

Dr. Cov

- Collects code coverage information
- Records which basic blocks have been executed
- Operates on unmodified binaries
- Results are post-processed into the standard lcov format
- Usage:
  - **`drrun -t drcov -- <app>`**
  - **`tools/bin32/drcov2lcov -input drcov*.log`**
  - Now pass the resulting data file to lcov's genhtml script

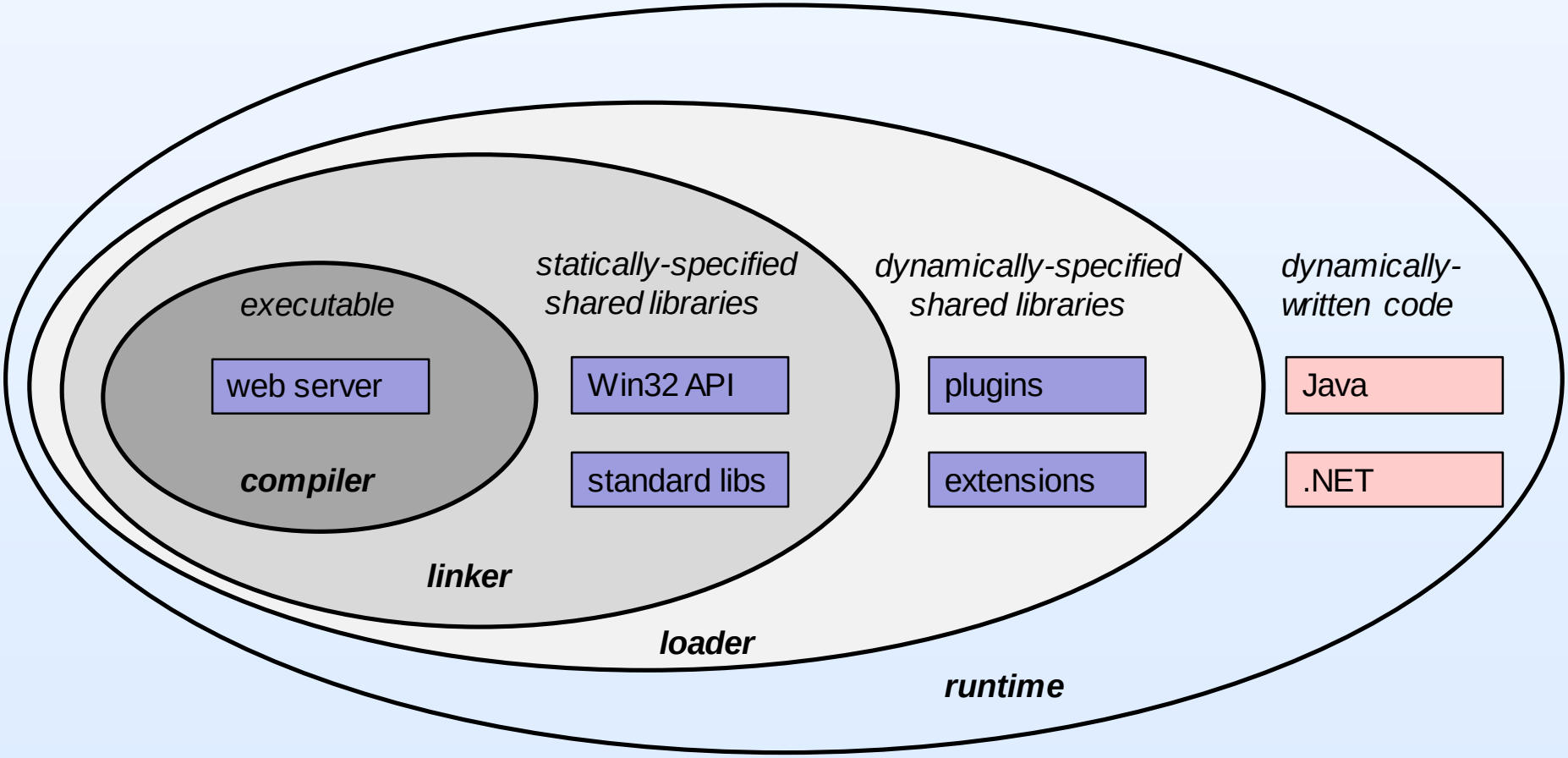
Dr. Cov Results



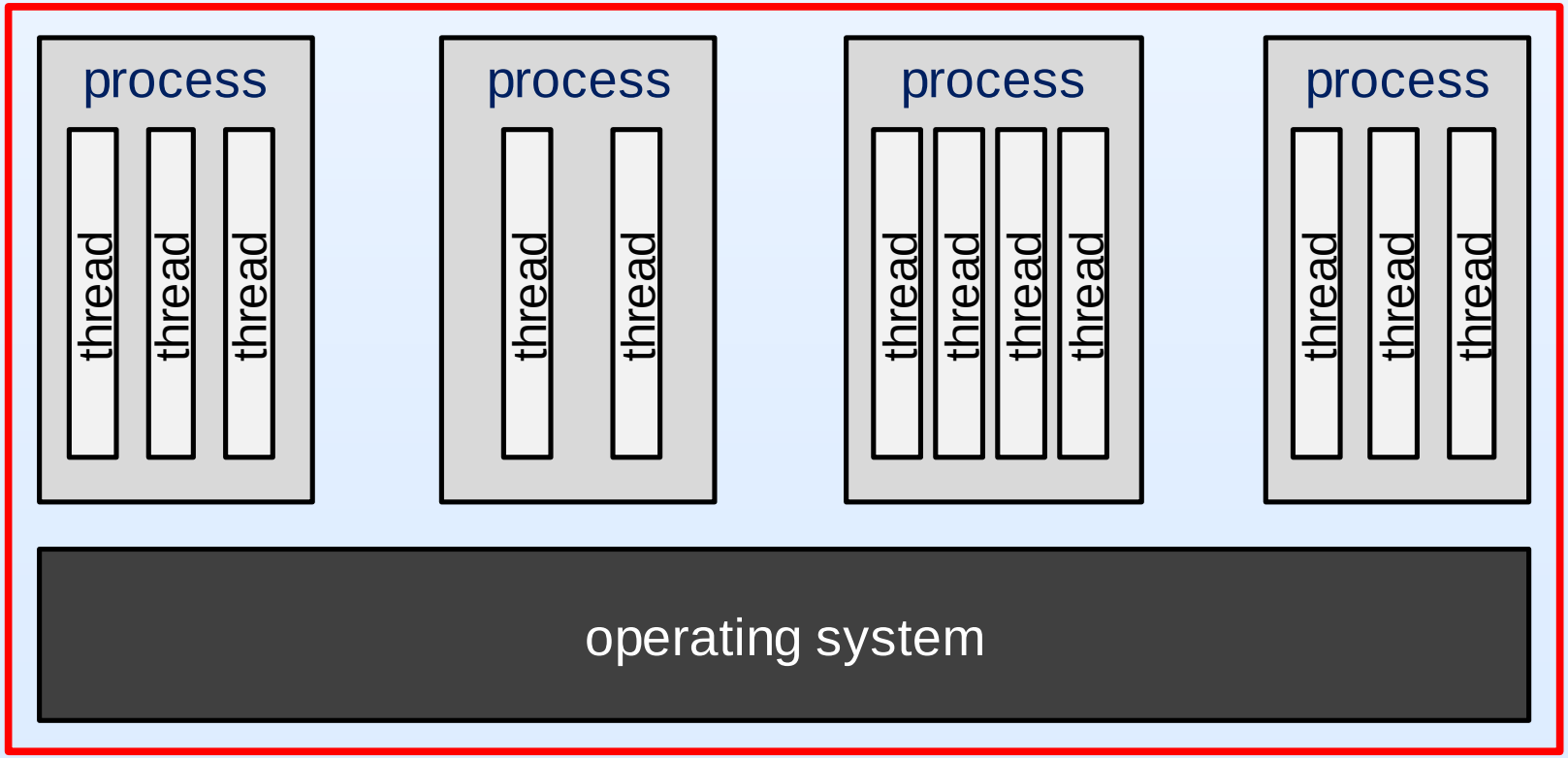
DynamoRIO Overview

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

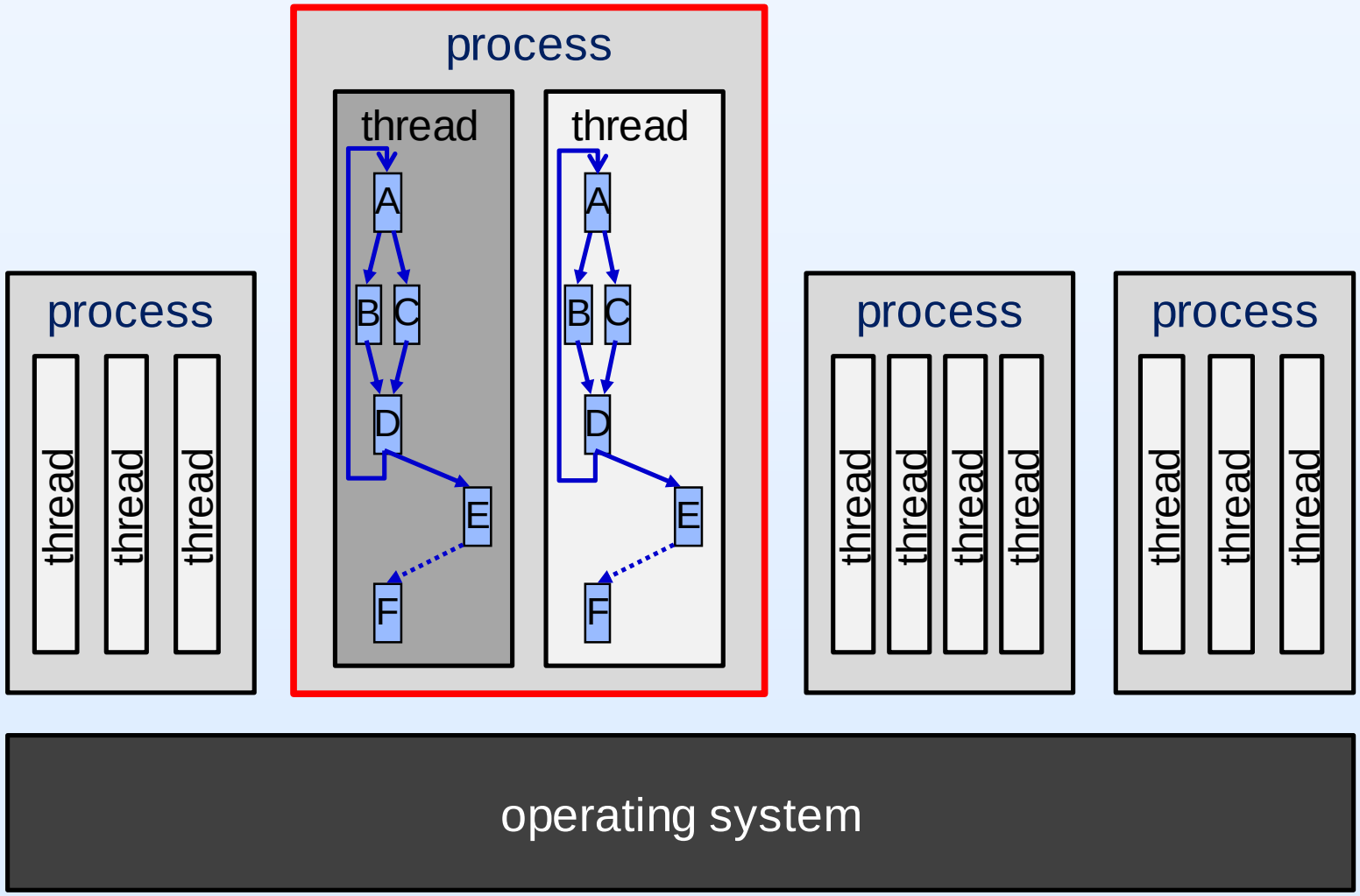
Reach of Toolchain Control Points



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

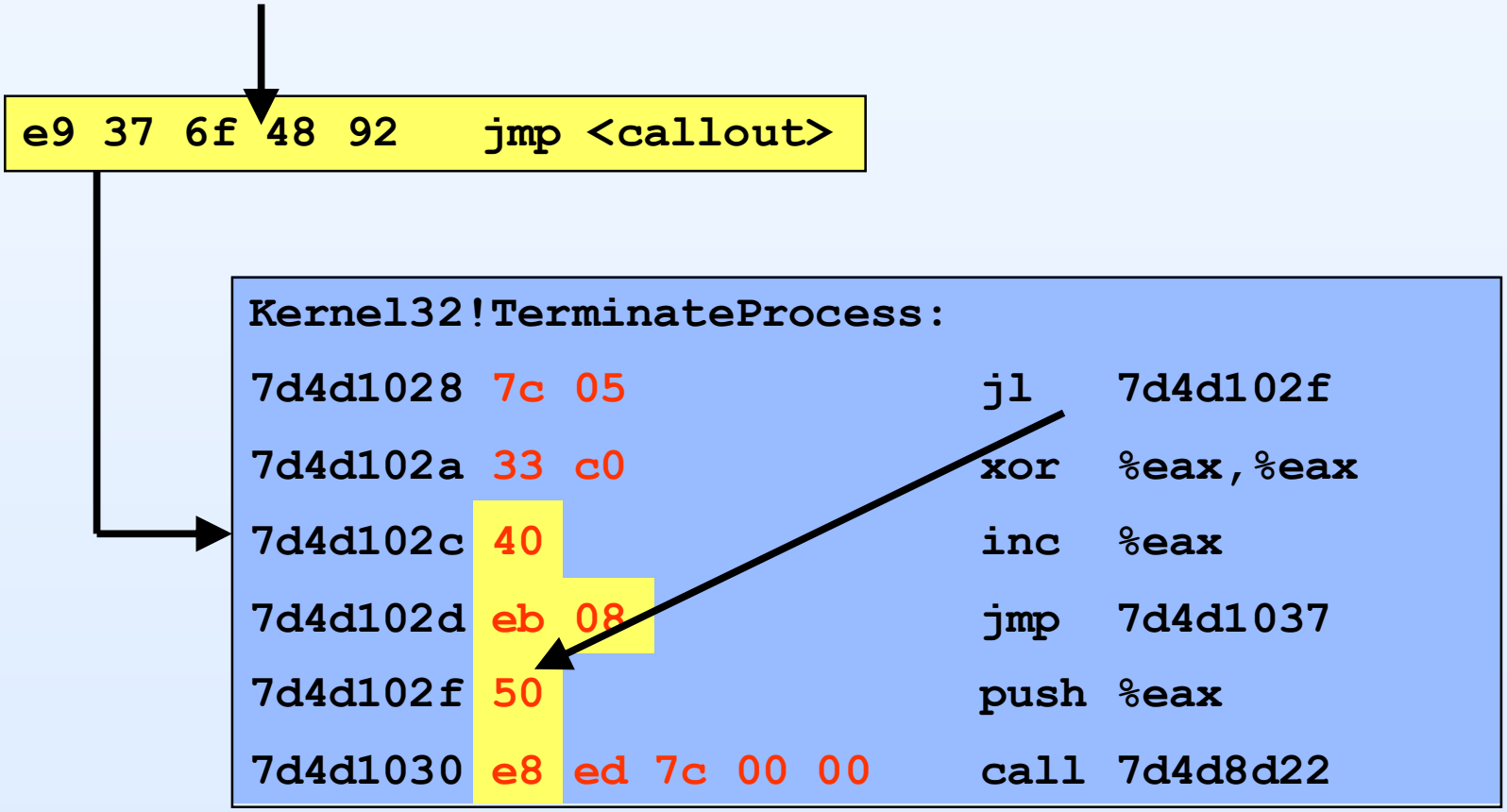
Challenges of Real-World Apps

- 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

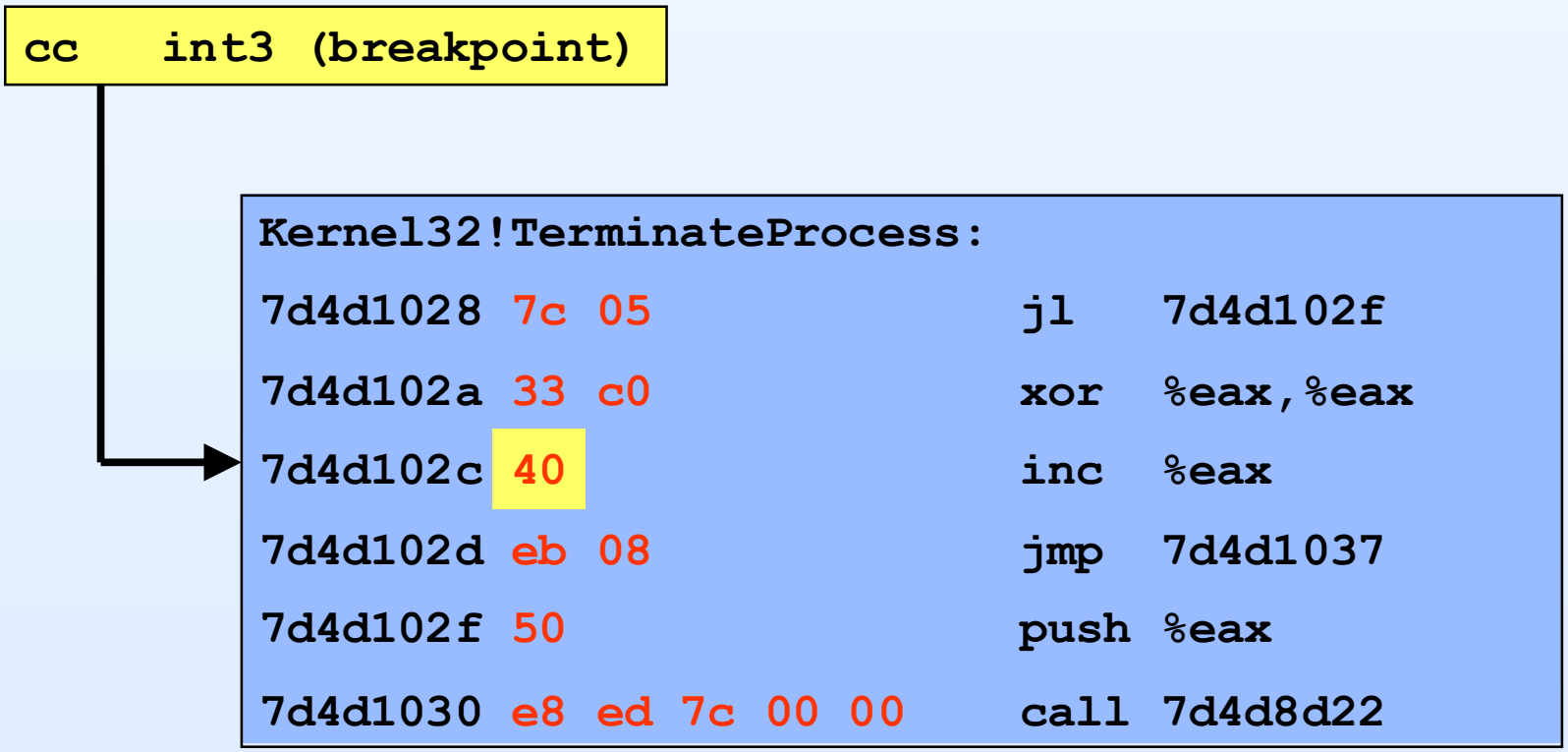
Overview Outline

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

Direct Code Modification



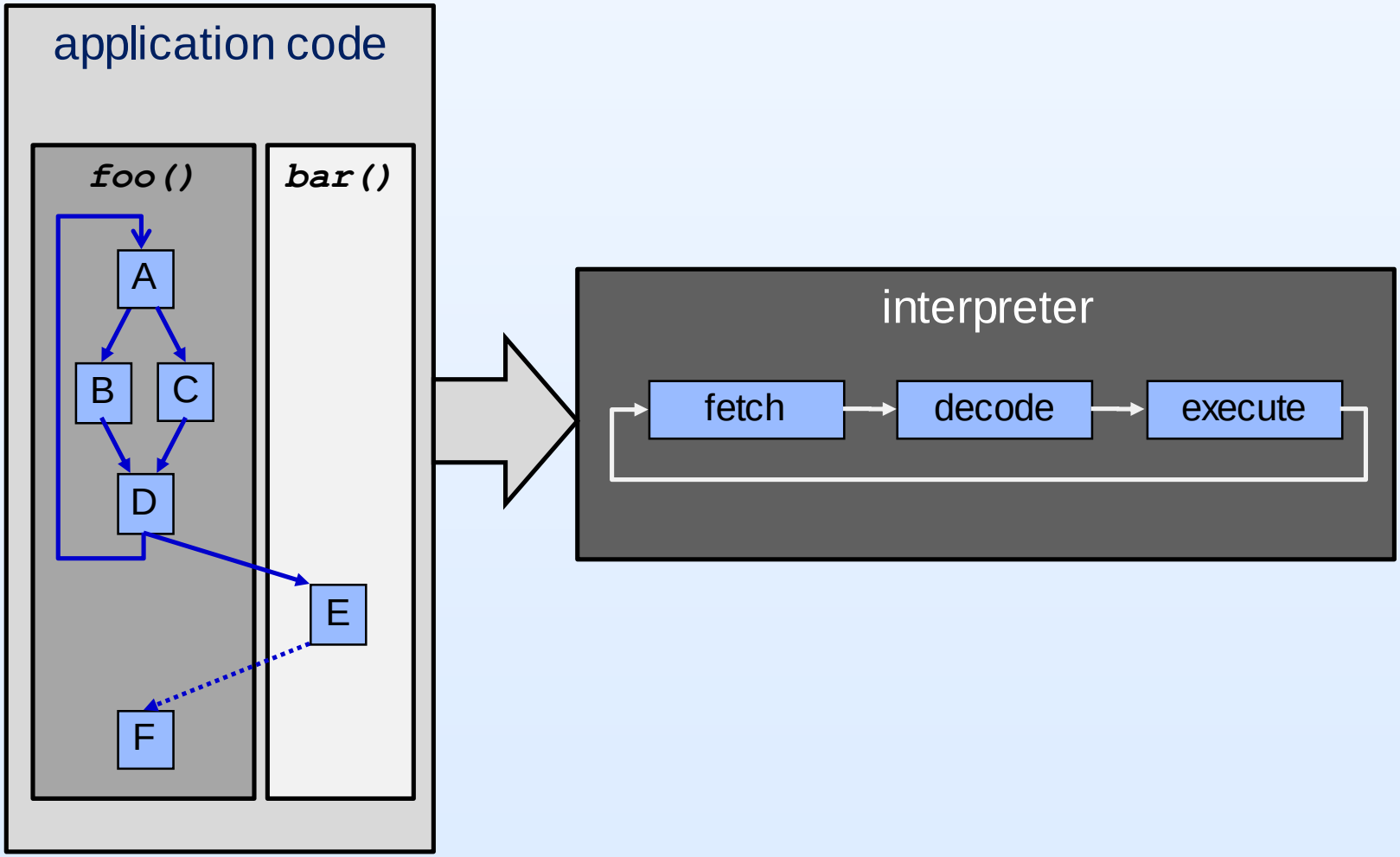
Debugger Trap Too Expensive



We Need Indirection

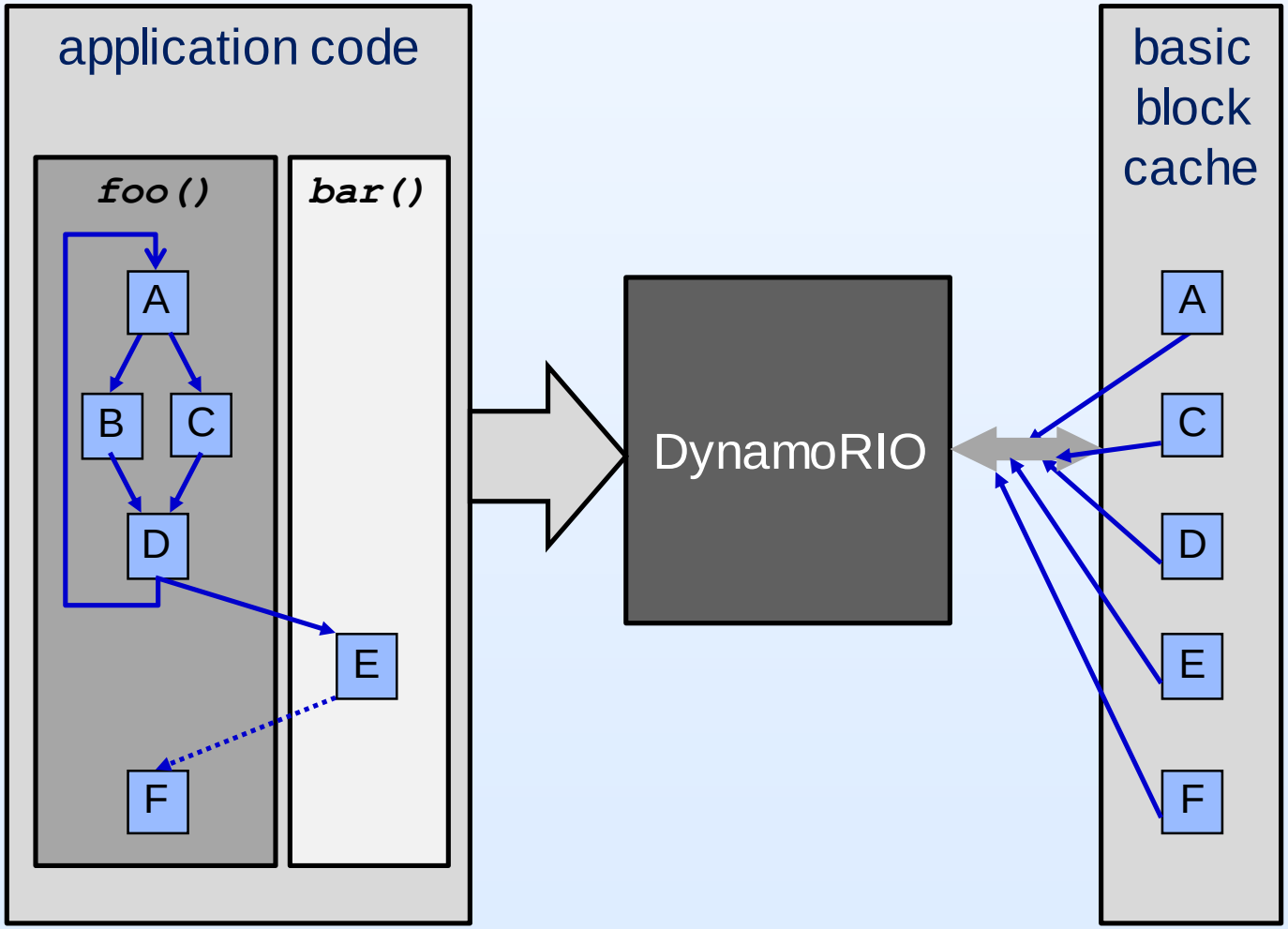
- Avoid transparency and granularity limitations of directly modifying application code
- Allow arbitrary modifications at unrestricted points in code stream
- Allow systematic, fine-grained modifications to code stream
- Guarantee that all code is observed

Basic Interpreter



Slowdown: 300x

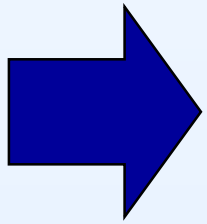
Improvement #1: Basic Block Cache



Slowdown: ~~300x~~ 25x

Example Basic Block Fragment

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

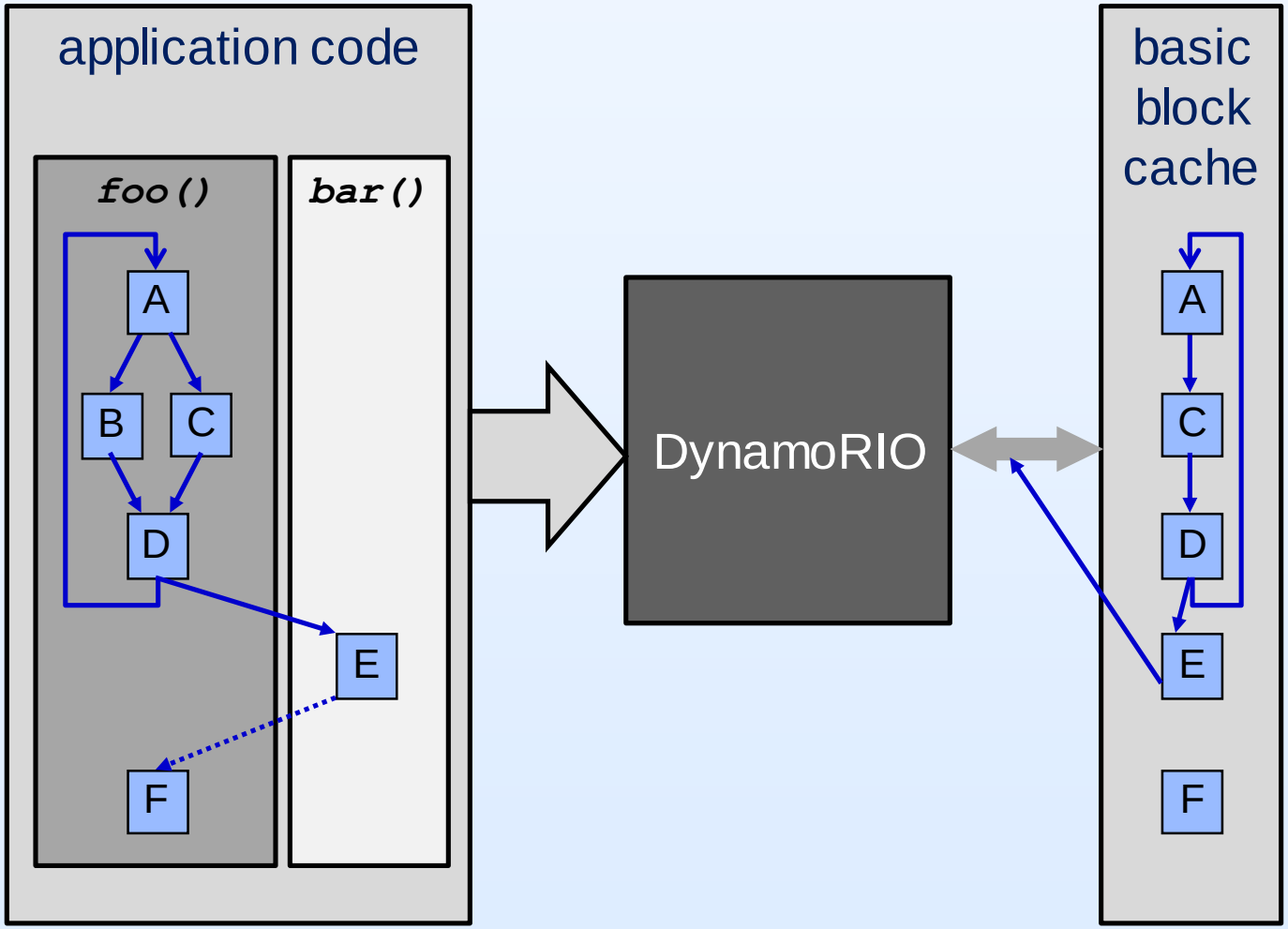


```
frag7: add  %eax, %ecx
       cmp  $4, %eax
       jle  <stub0>
       jmp  <stub1>
stub0: mov  %eax, eax-slot
       mov  &dstub0, %eax
       jmp  context_switch
stub1: mov  %eax, eax-slot
       mov  &dstub1, %eax
       jmp  context_switch
```

dstub0
target: 0x40106f

dstub1
target: fall-thru

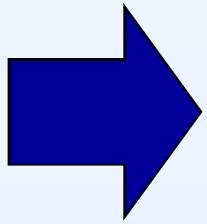
Improvement #2: Linking Direct Branches



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

Direct Linking

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

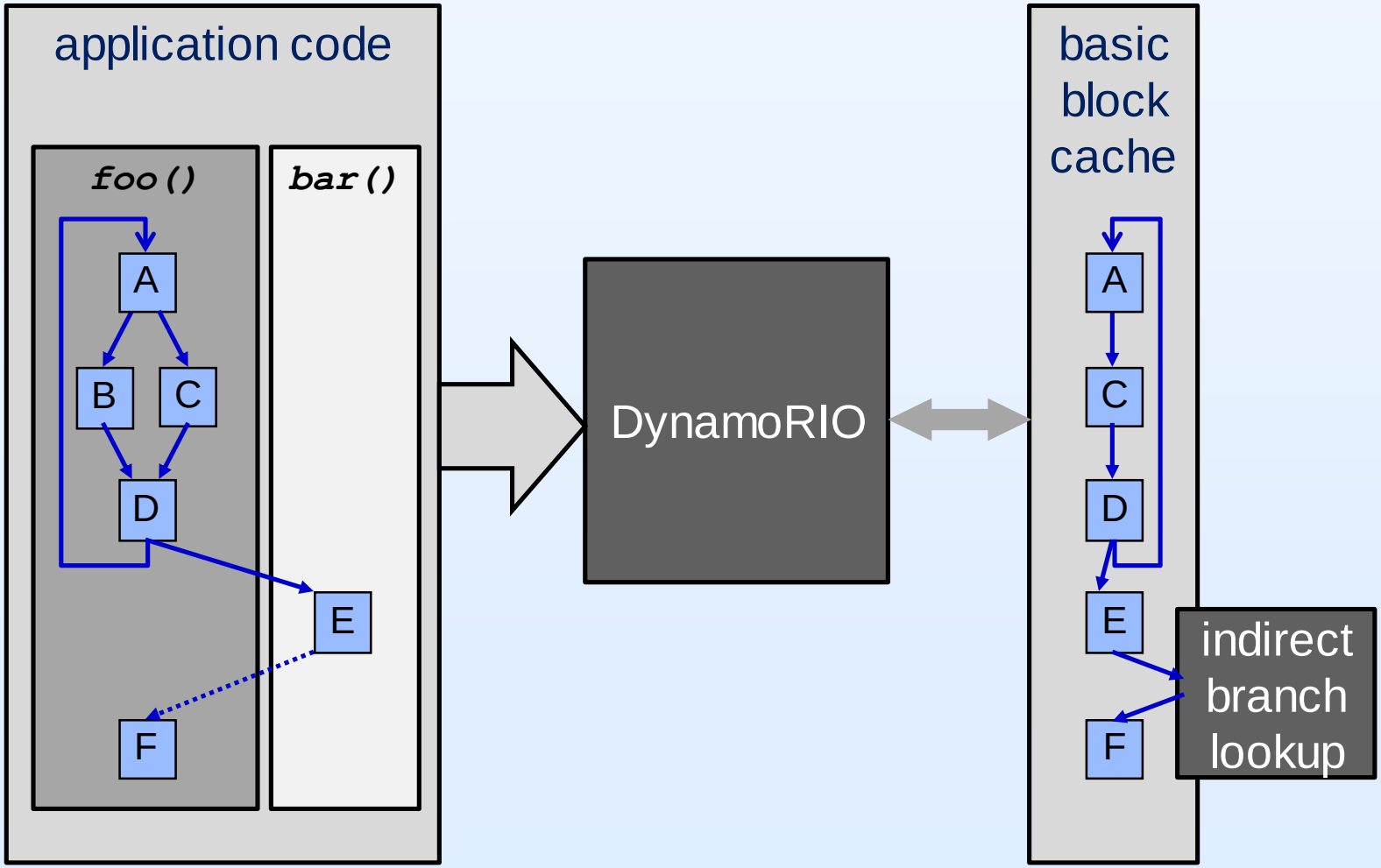


```
frag7: add  %eax, %ecx
       cmp  $4, %eax
       jle  <frag8>
       jmp  <stub1>
stub0: mov  %eax, eax-slot
       mov  &dstub0, %eax
       jmp  context_switch
stub1: mov  %eax, eax-slot
       mov  &dstub1, %eax
       jmp  context_switch
```

dstub0
target: 0x40106f

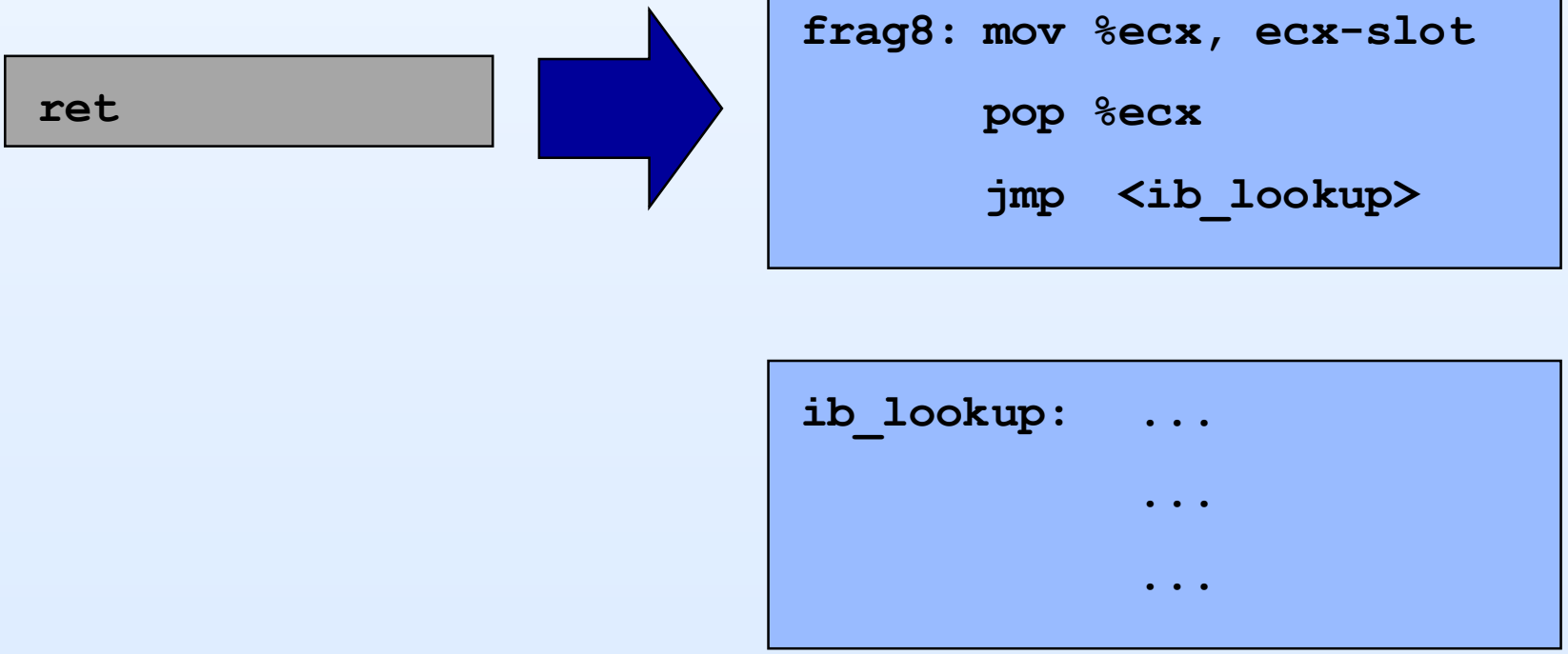
dstub1
target: fall-thru

Improvement #3: Linking Indirect Branches

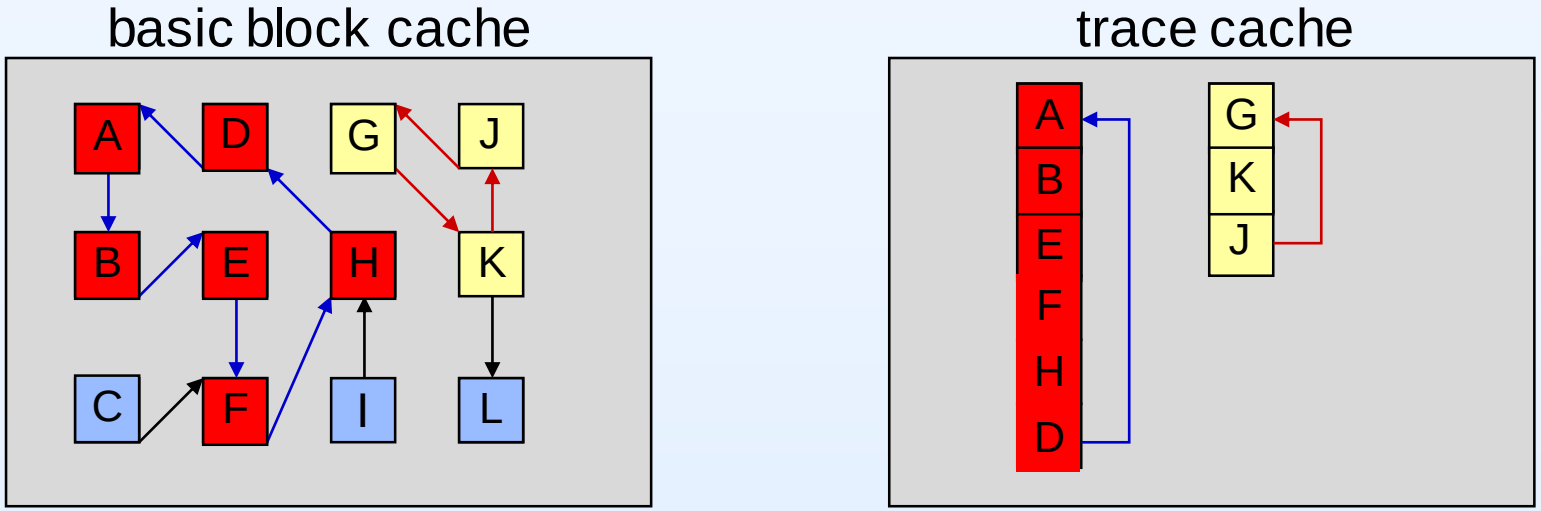


Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ 1.2x

Indirect Branch Transformation



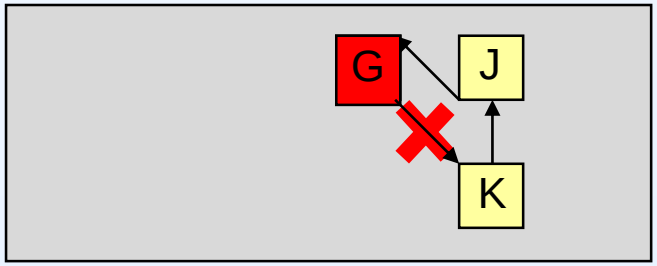
Improvement #4: Trace Building

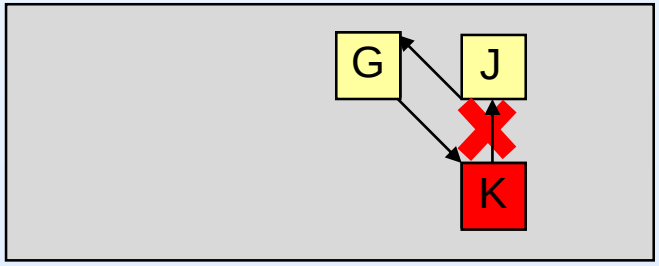


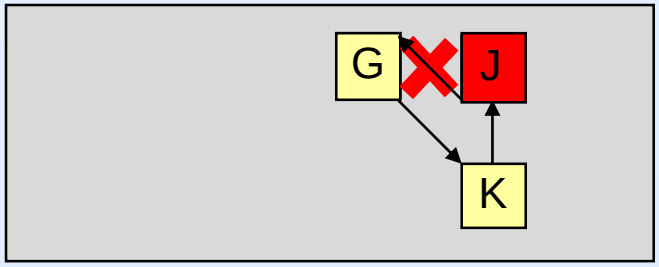
- Traces reduce branching, improve layout and locality, and facilitate optimizations across blocks
  - We avoid indirect branch lookup
- Next Executing Tail (NET) trace building scheme [Duesterwald 2000]

Incremental NET Trace Building

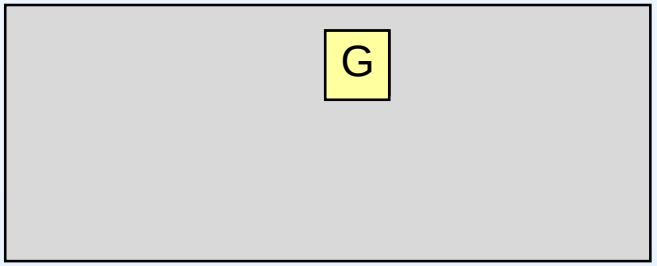
basic block cache

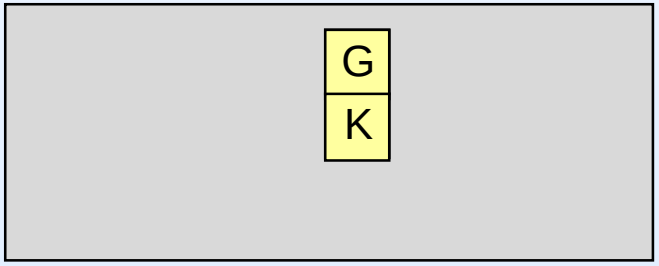


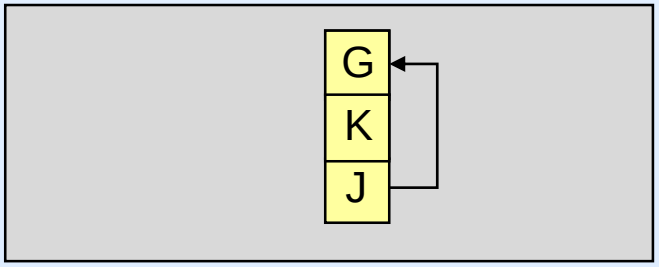




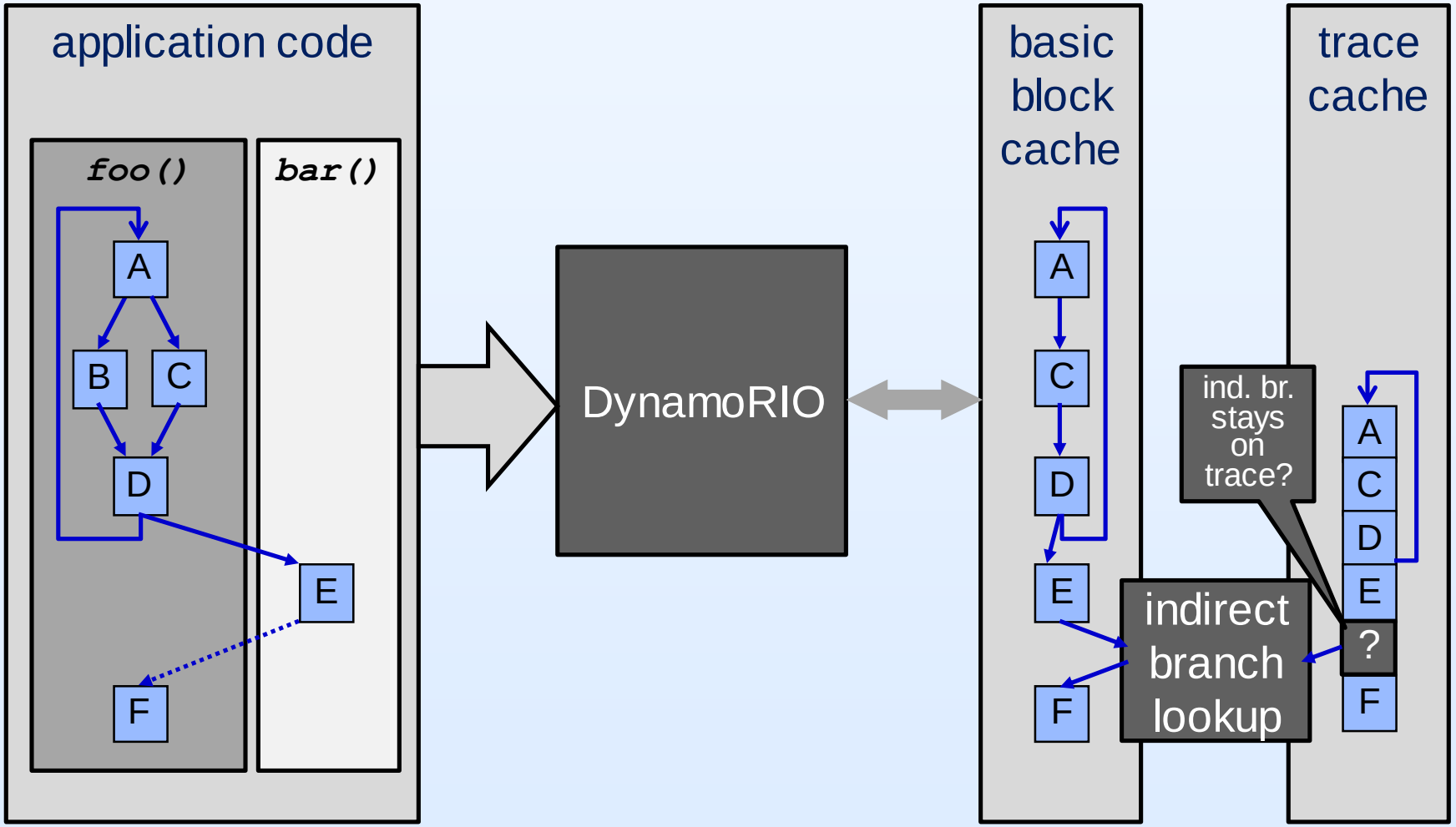
trace cache





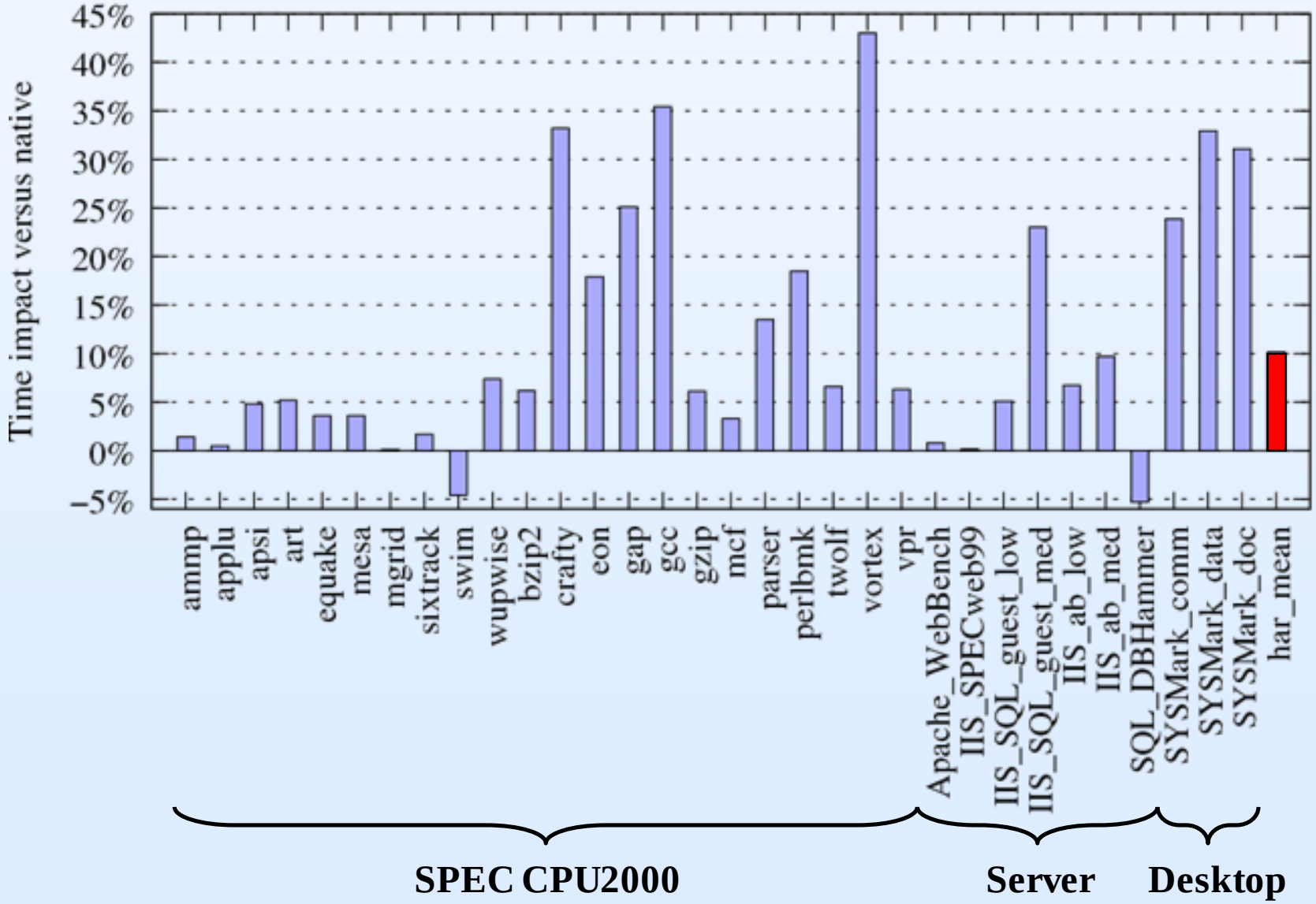


Improvement #4: Trace Building

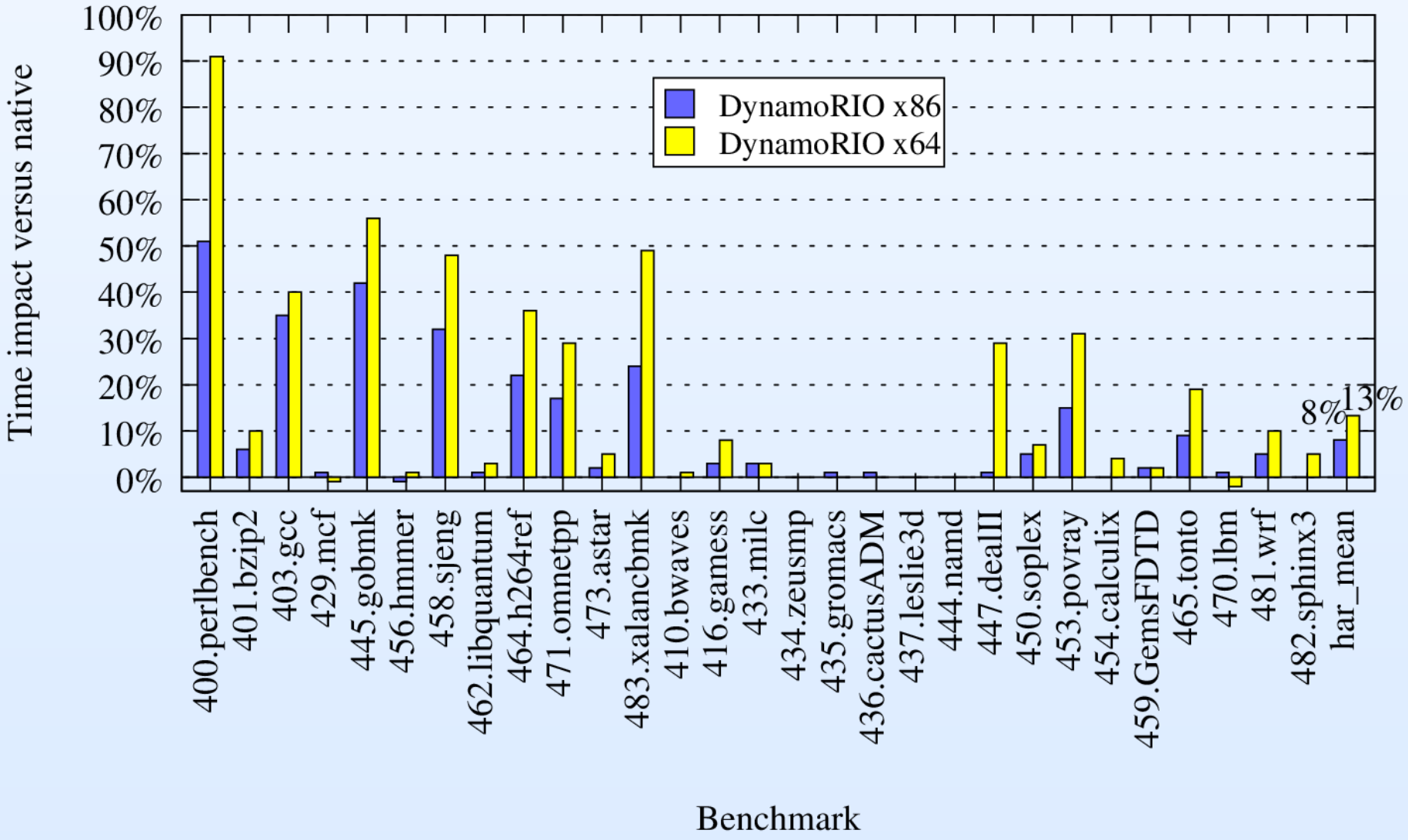


Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ ~~1.2x~~ 1.1x

Base Performance



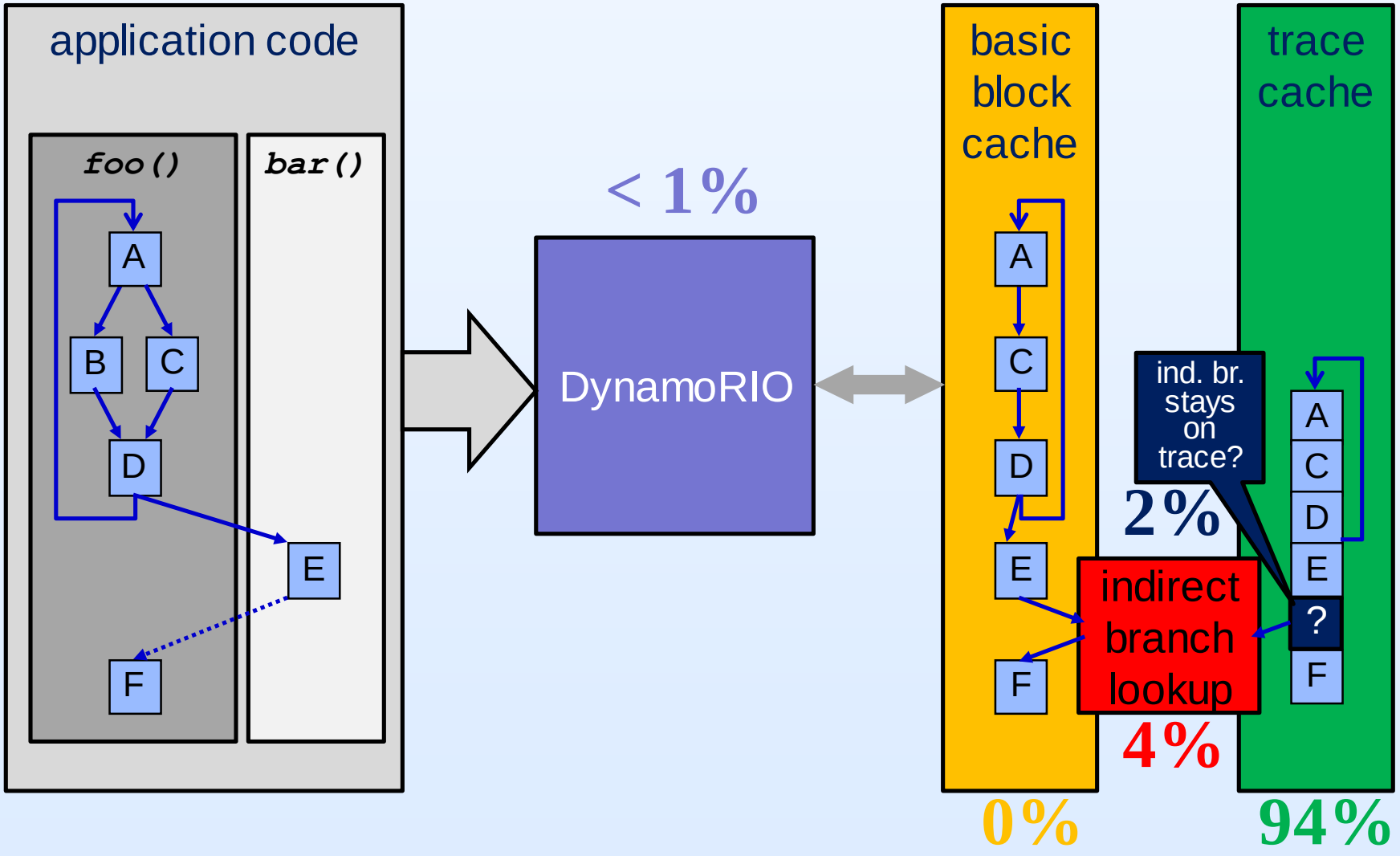
Base Performance: SPEC 2006



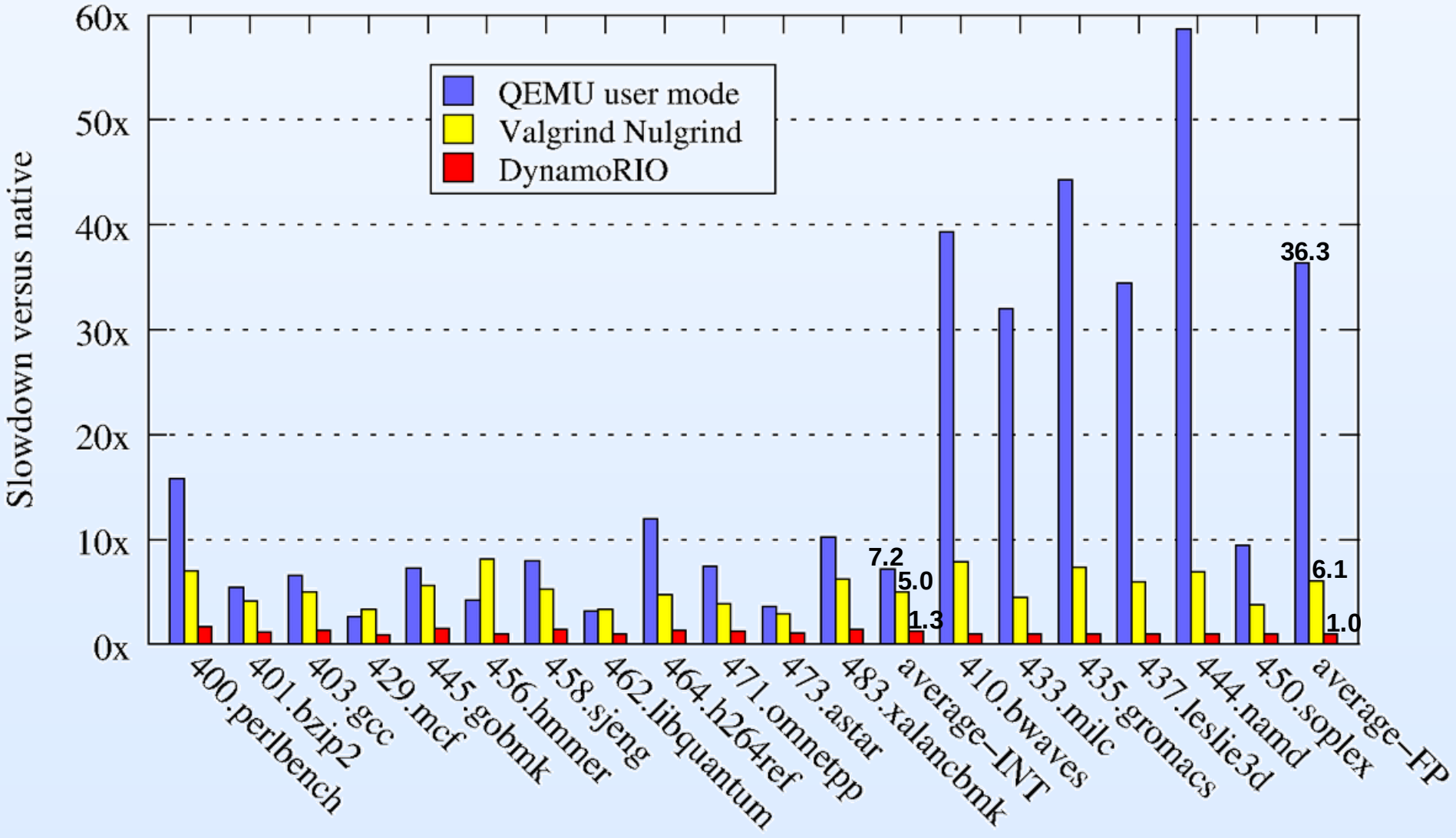
Sources of Overhead

- Extra instructions
  - Indirect branch target comparisons
  - Indirect branch hashtable lookups
- Extra data cache pressure
  - Indirect branch hashtable
- Branch mispredictions
  - Calls and returns turn into jumps on some architectures
- Application code modification

Time Breakdown for SPEC CPU INT



Avoiding Intermediate Layers



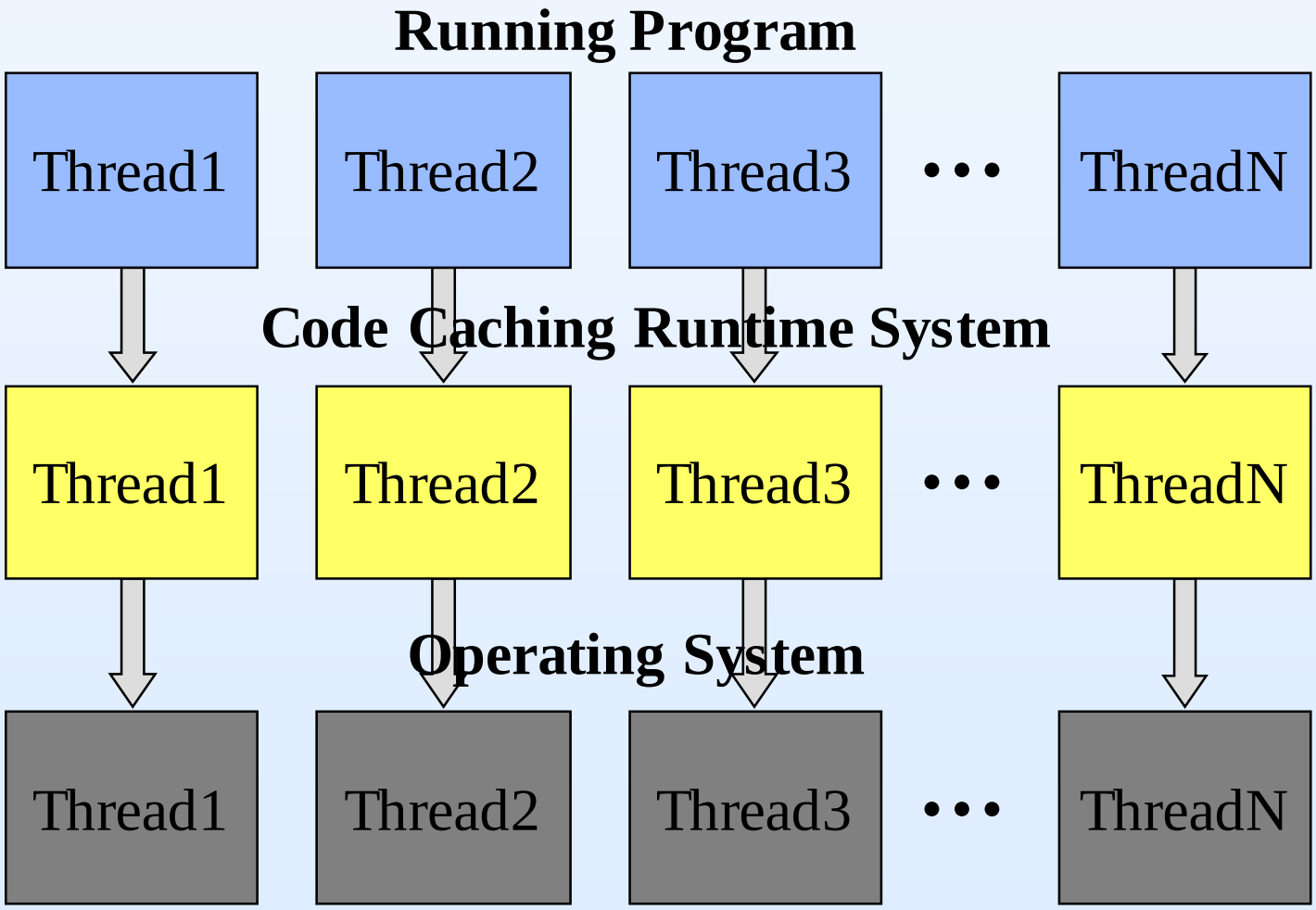
No Intermediate Layers

- IR mirrors underlying ISA
  - Preserves optimized application code
  - Intermediate layers incur significant performance impact
    - QEMU (user-mode) **6x** slower than DR, Valgrind **4x** slower than DR
  - *This is the key to good performance*
- Still have an abstraction layer
  - Block or trace = list of instructions
  - Instruction = lists of source and destination operands
  - Tool code often still cross-platform
    - "Does this instruction read memory?"

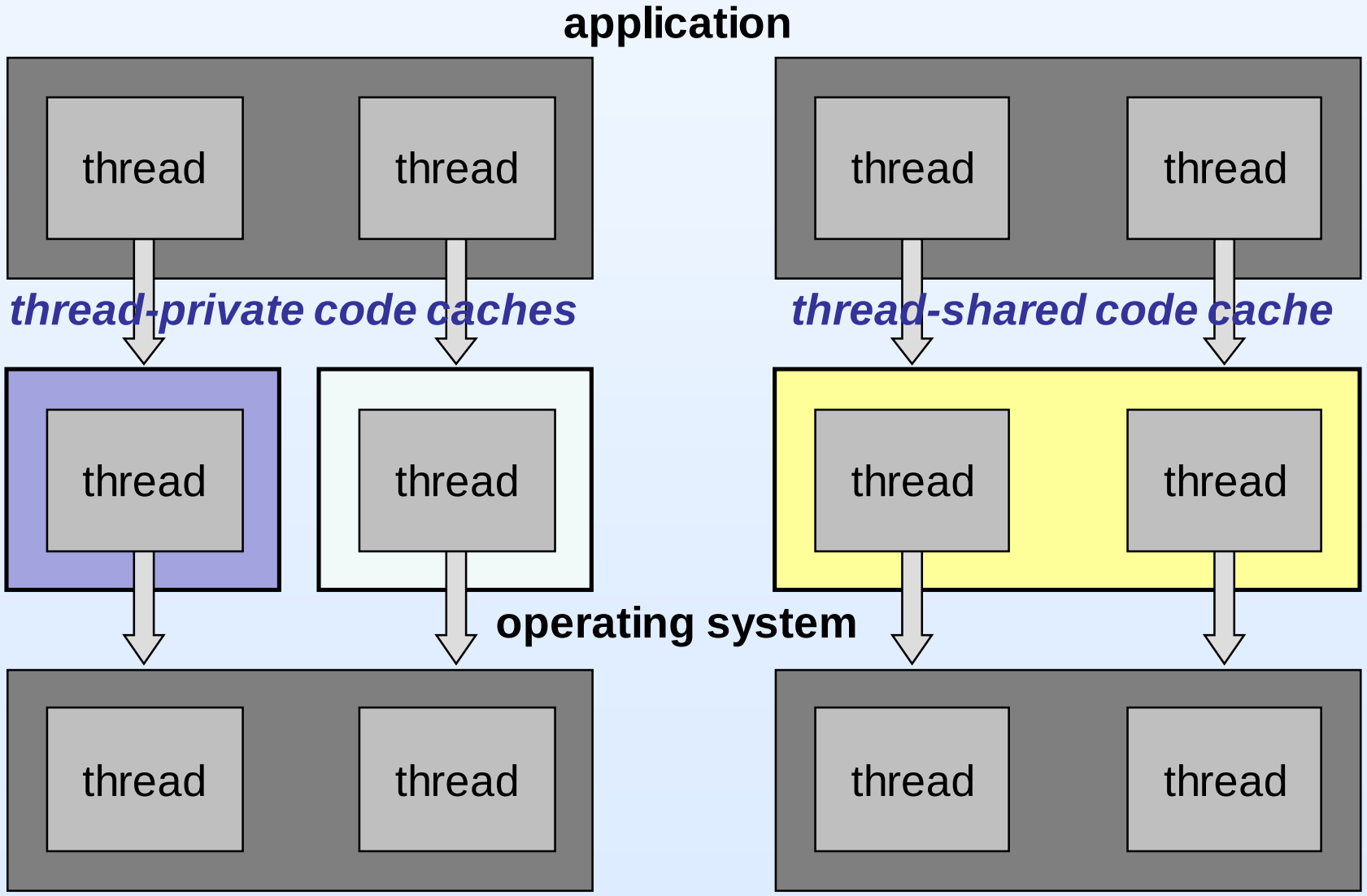
Overview Outline

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

Threading Model



Code Cache Threading Models



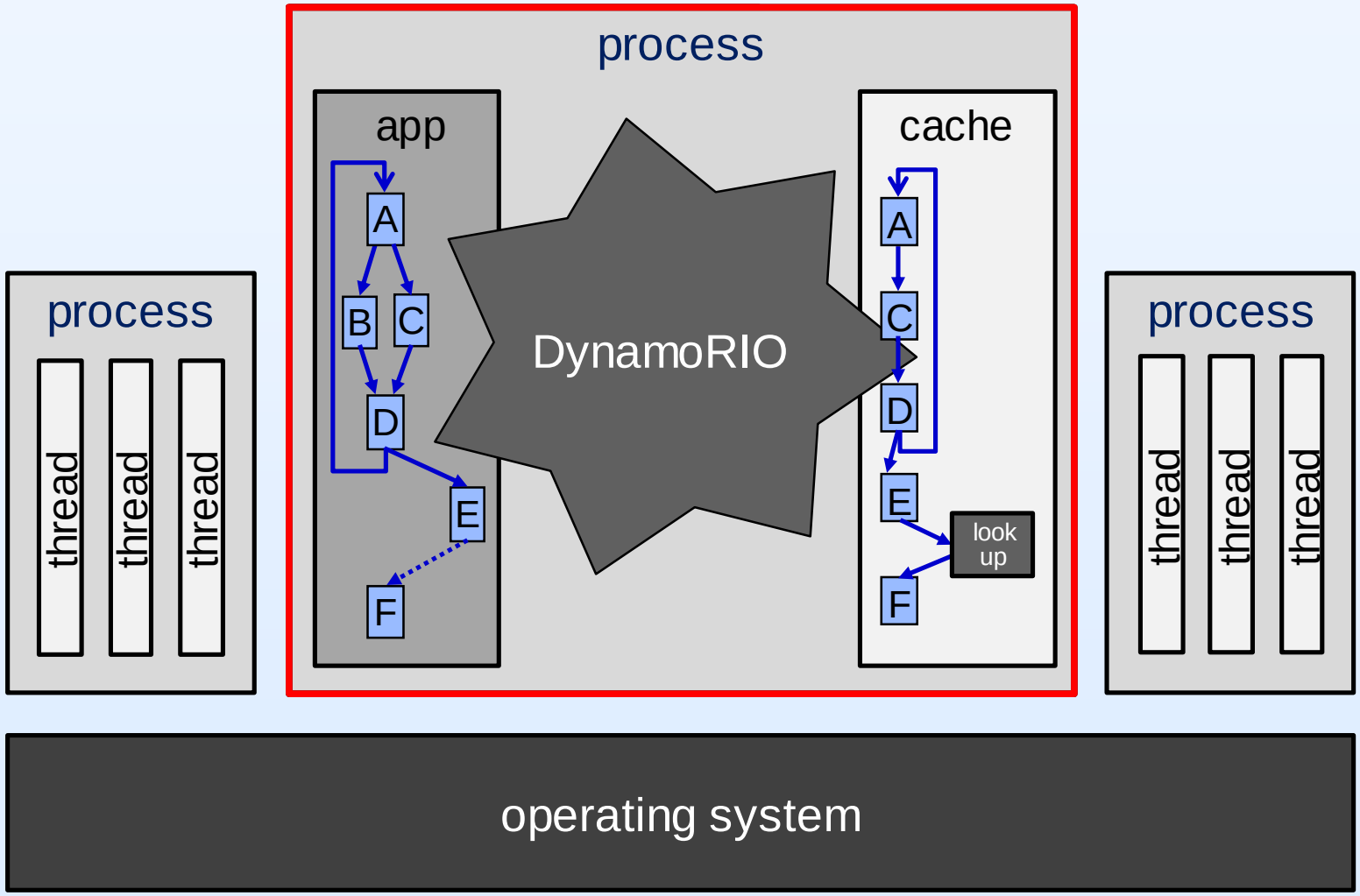
Thread-Private versus Thread-Shared

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

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

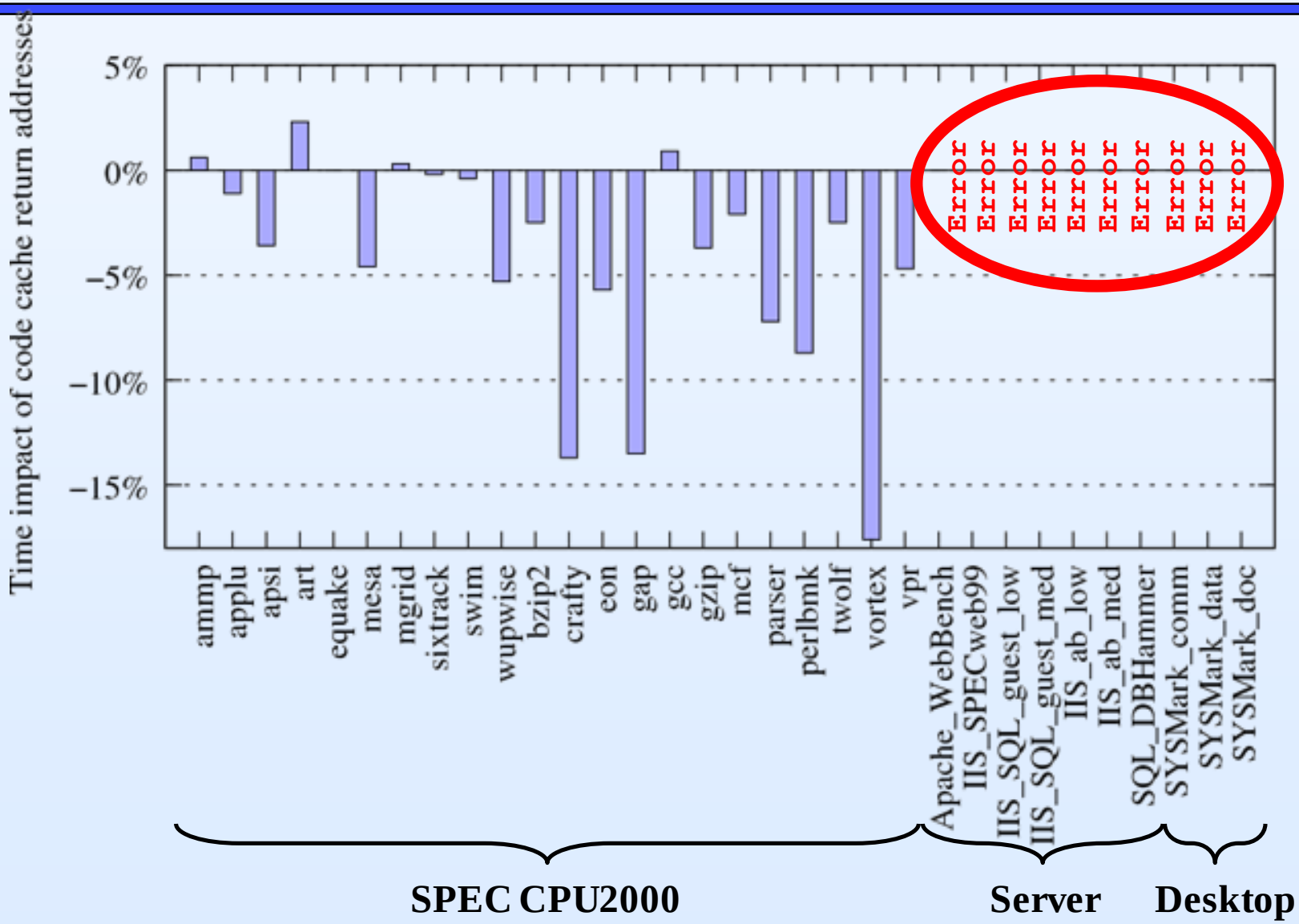
- **Principle 1: *As few changes as possible***
  - Set a high bar for value before changing the native environment
- **Principle 2: *Hide necessary changes***
  - Whatever is valuable enough to change must be hidden
  - Changes that cannot be hidden should not be made
- **Principle 3: *Separate resources***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

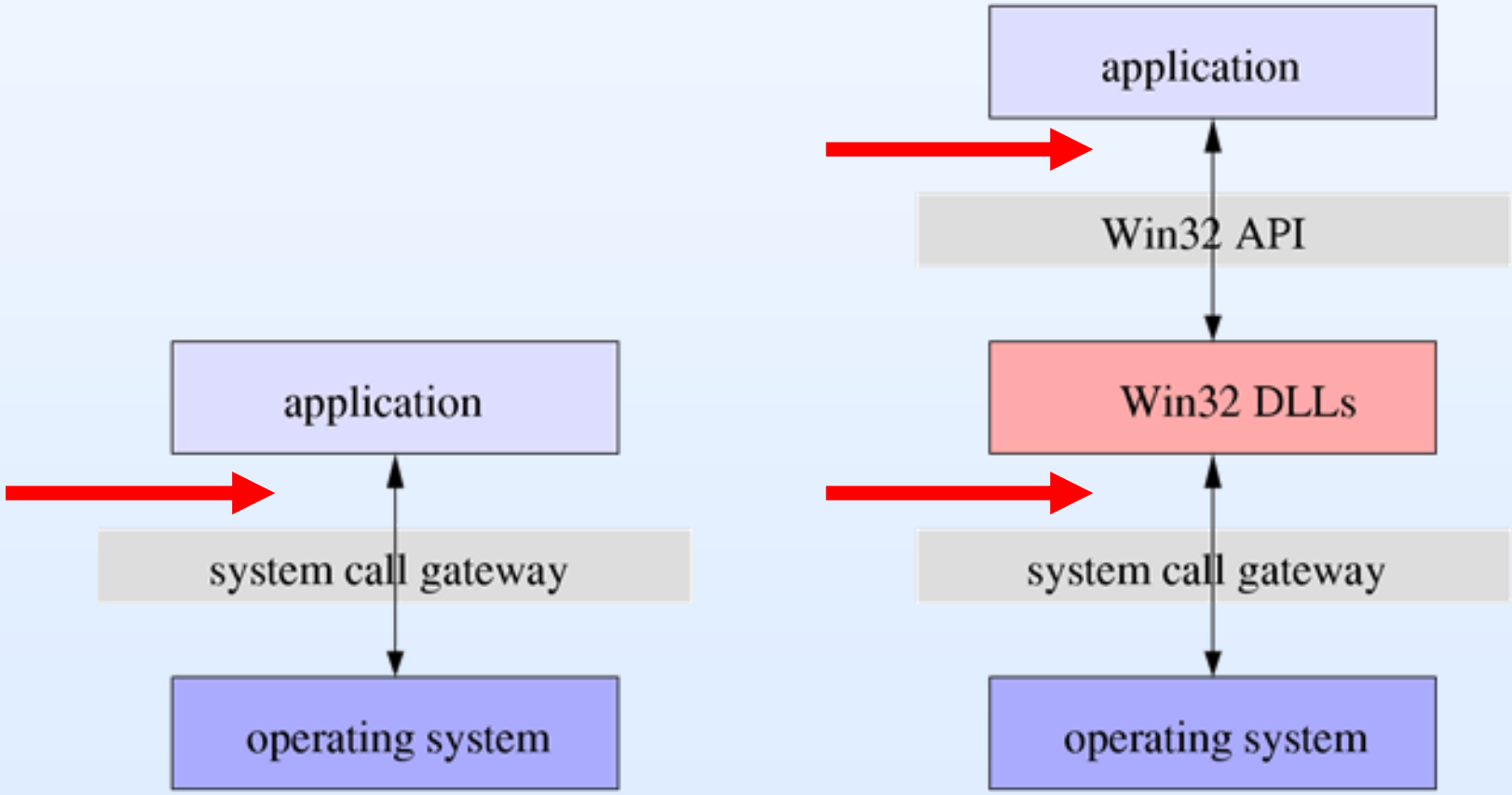
Return Address Transparency



Principle 2: *Hide necessary changes*

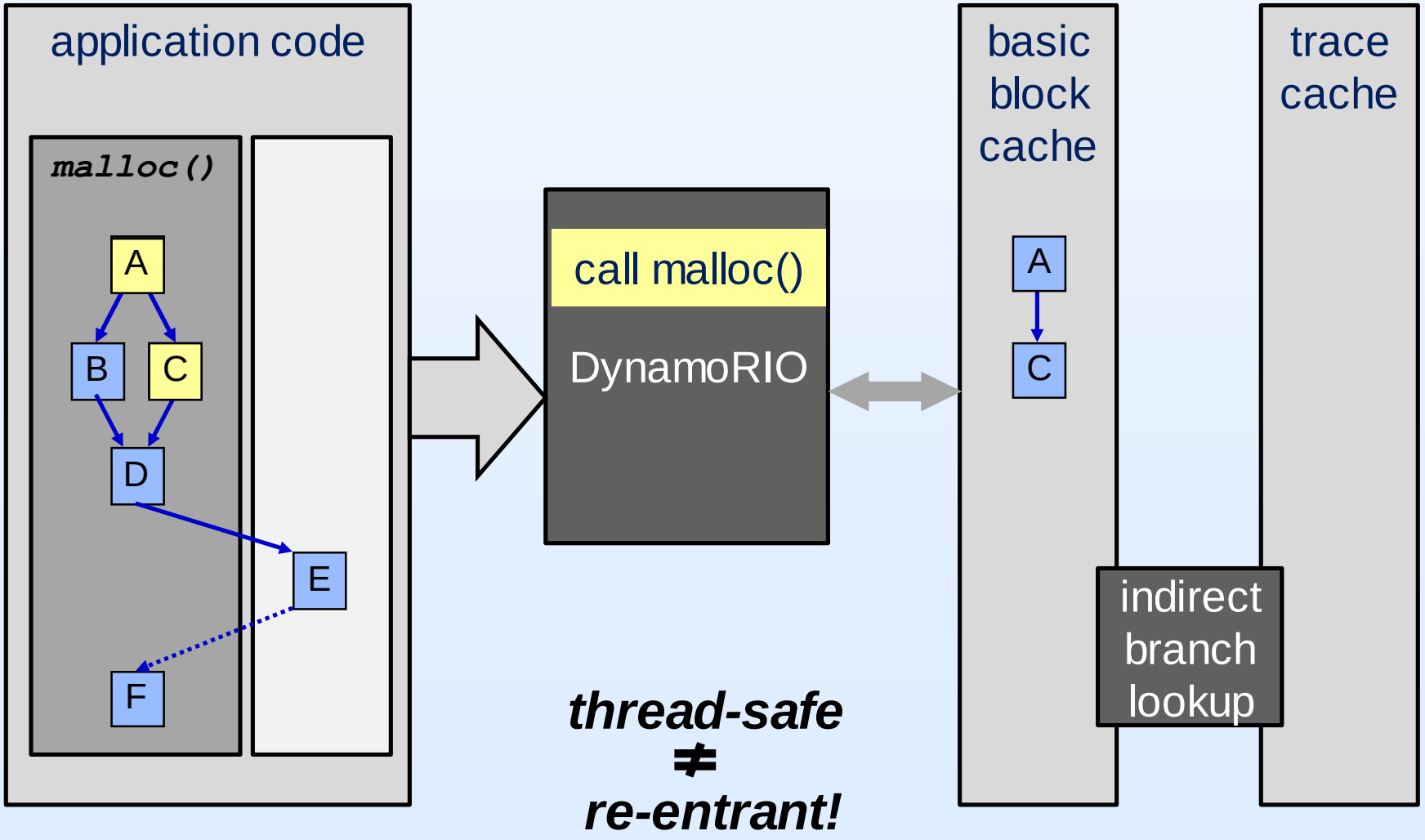
- Application addresses
- Address space
- Error transparency
- Code cache consistency

Principle 3: *Separate resources*

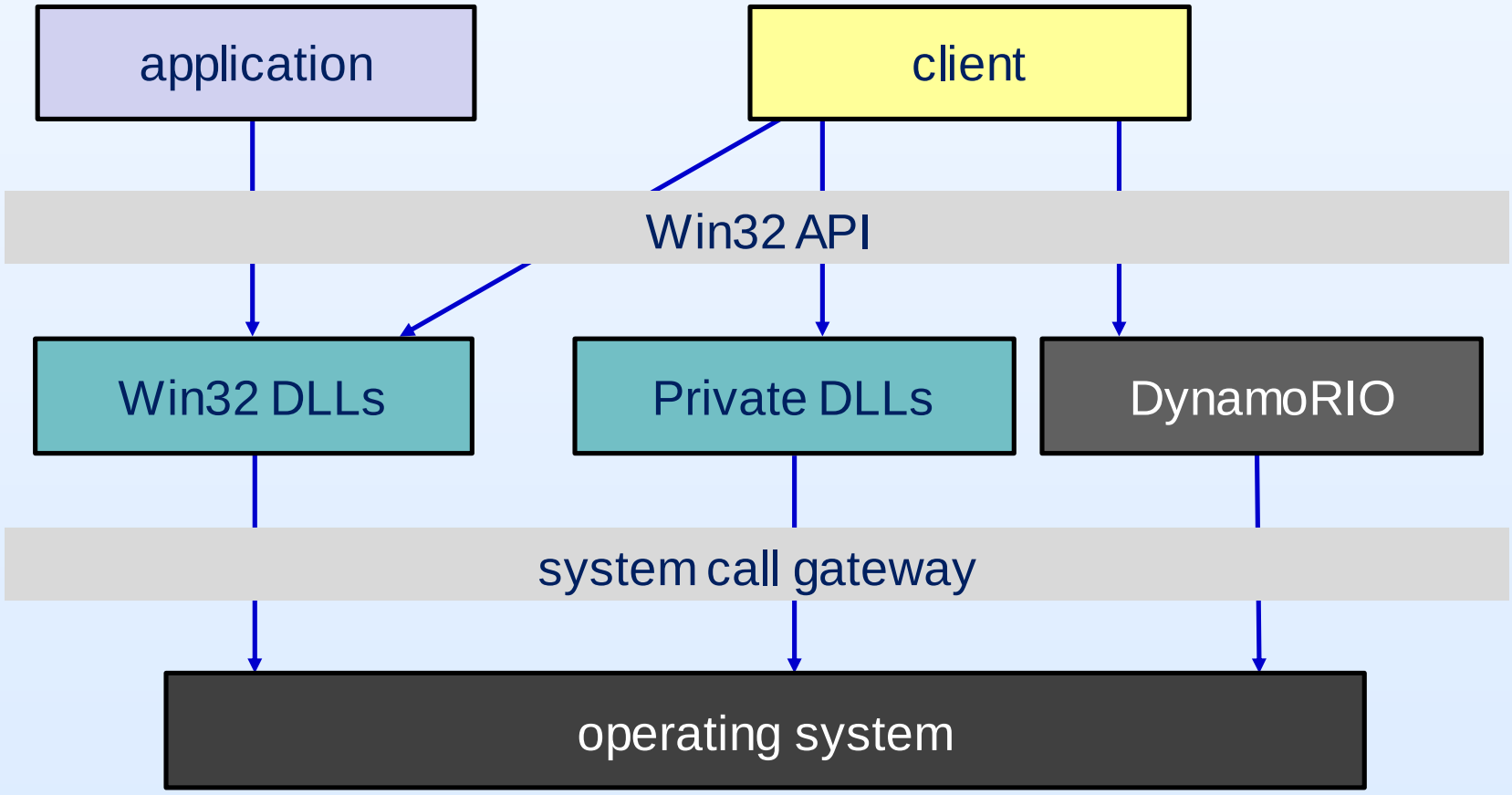


Linux ≠ Windows

Arbitrary Interleaving



Private Libraries



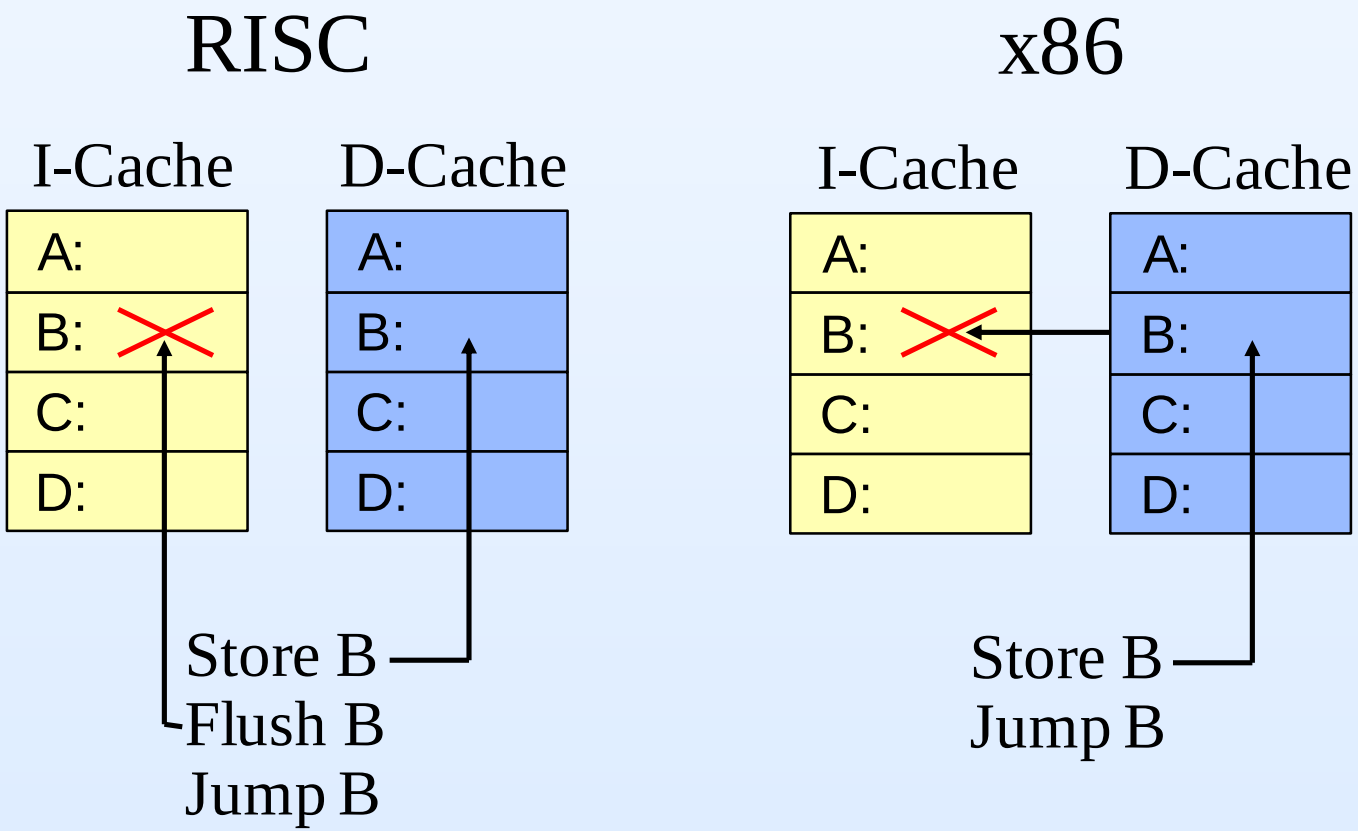
Transparency Landscape

	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

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

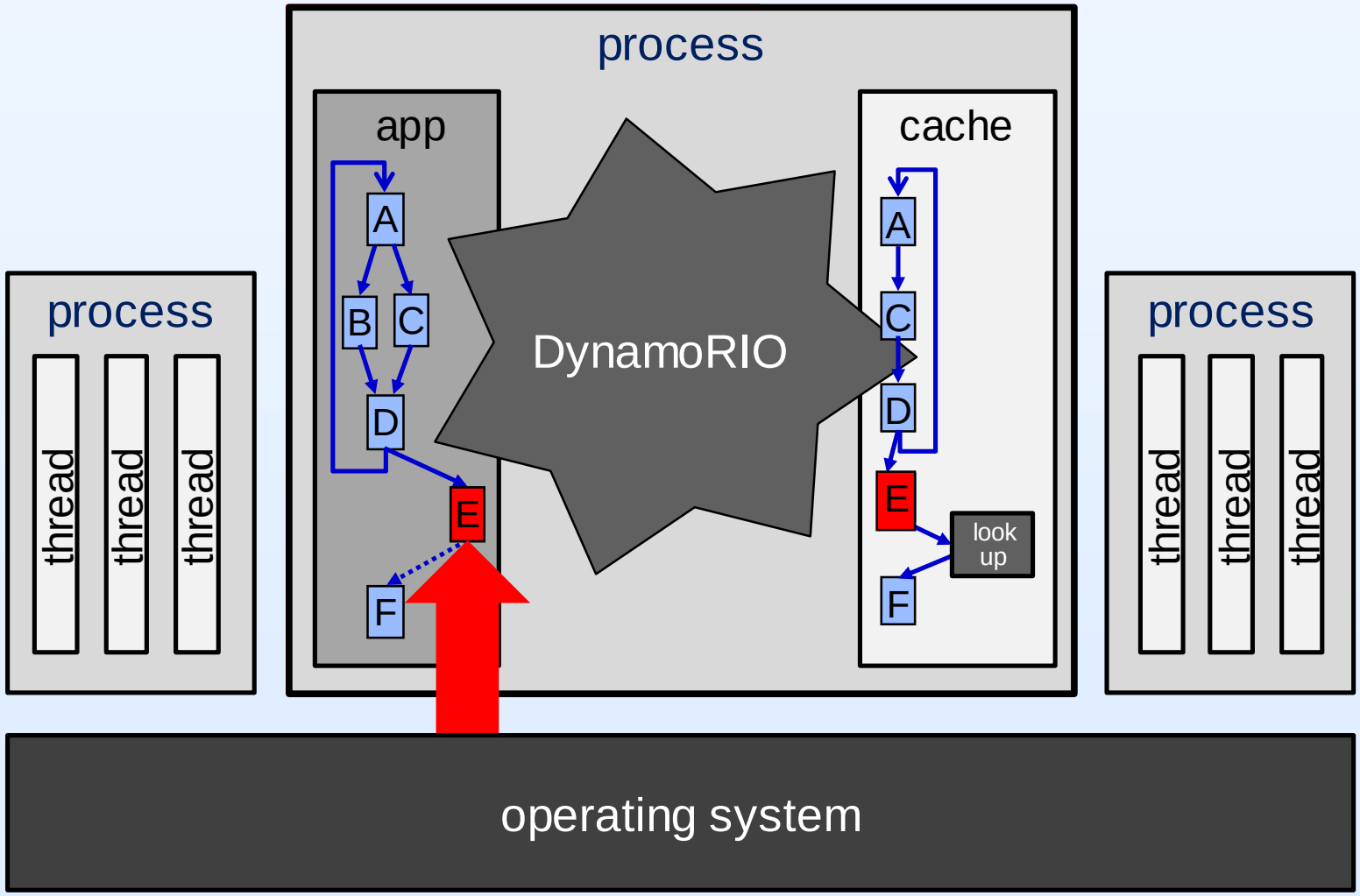
Synchronization Transparency

- 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

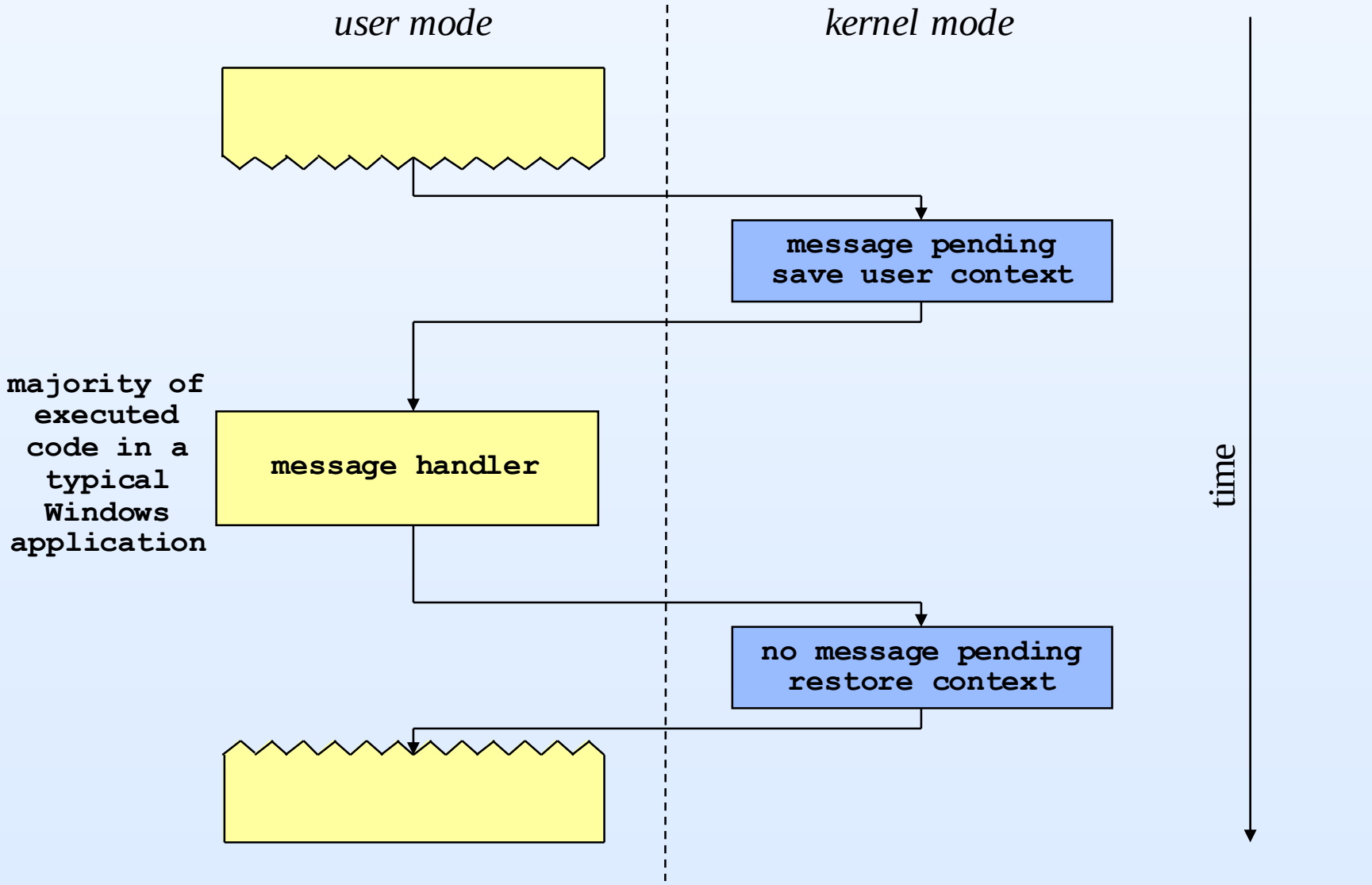
Overview Outline

- Efficient
- Transparent
- Comprehensive
- Customizable

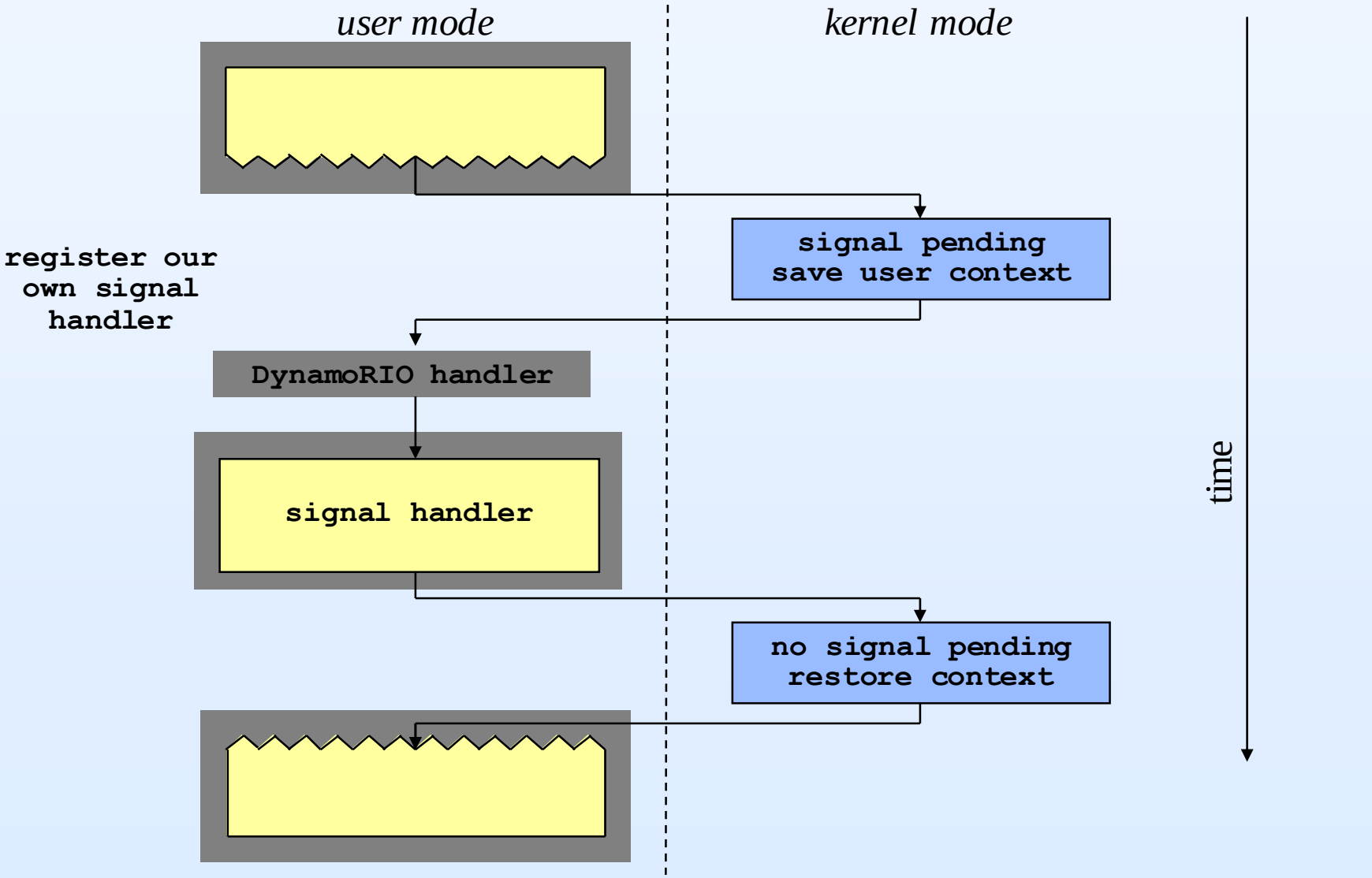
Above the Operating System



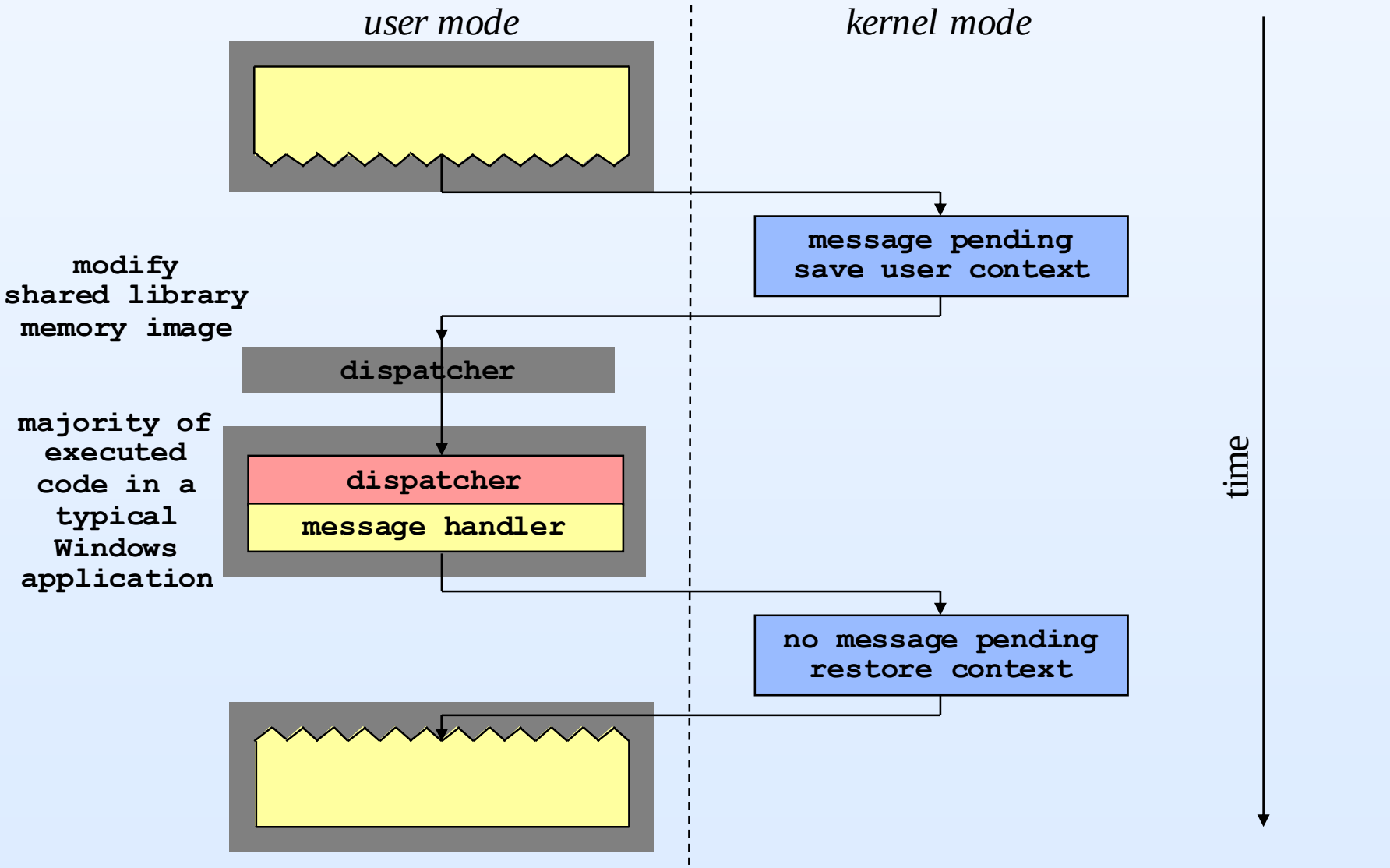
Kernel-Mediated Control Transfers



Intercepting Linux Signals



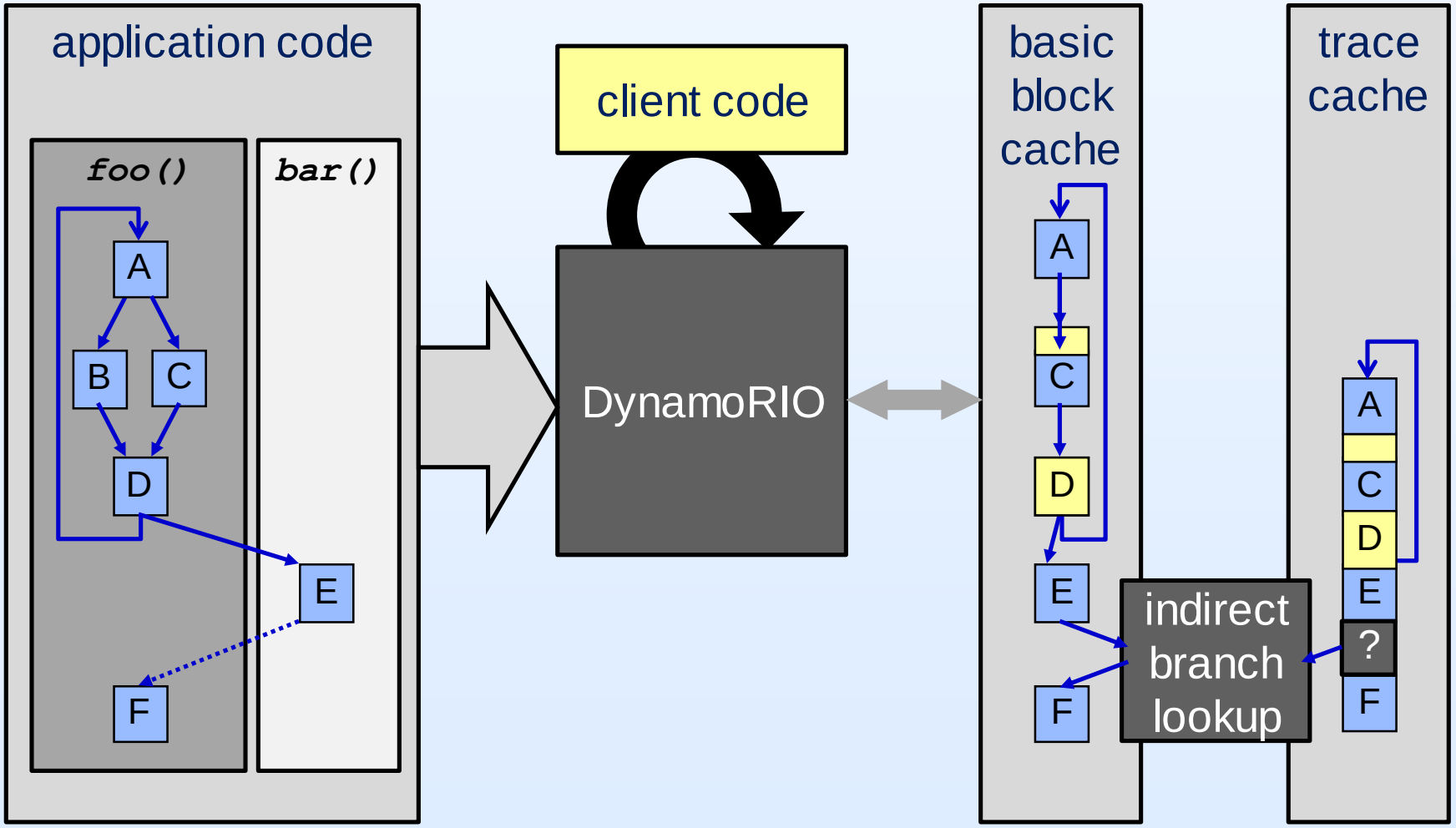
Intercepting Windows Messages



Overview Outline

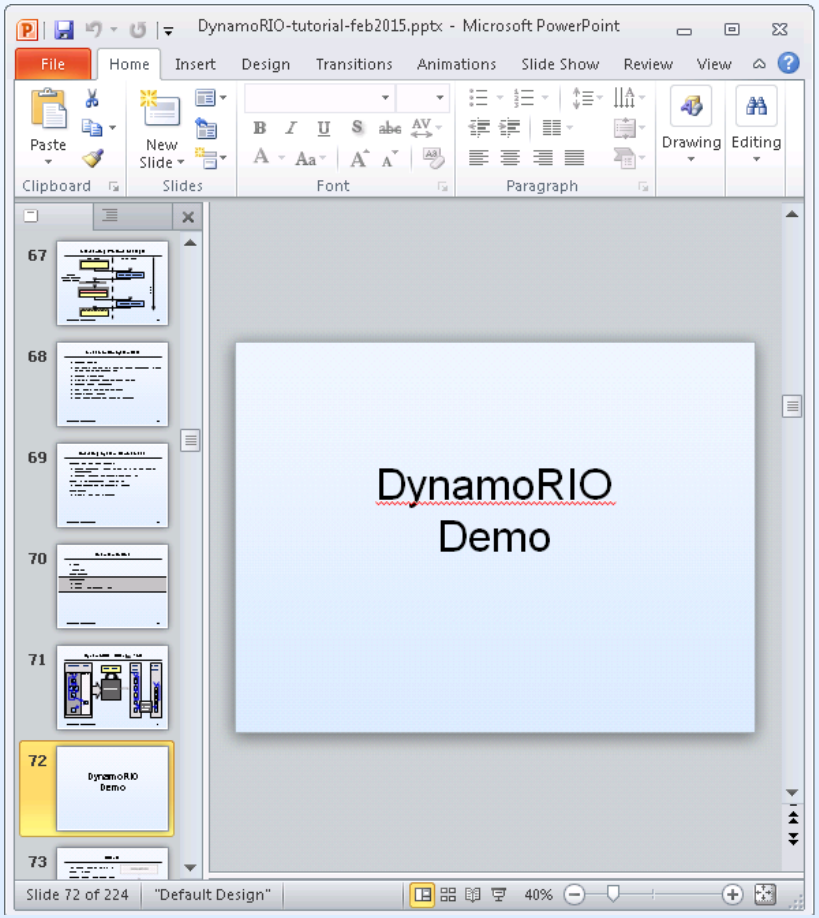
- Efficient
- Transparent
- Comprehensive
- Customizable
  - Clients

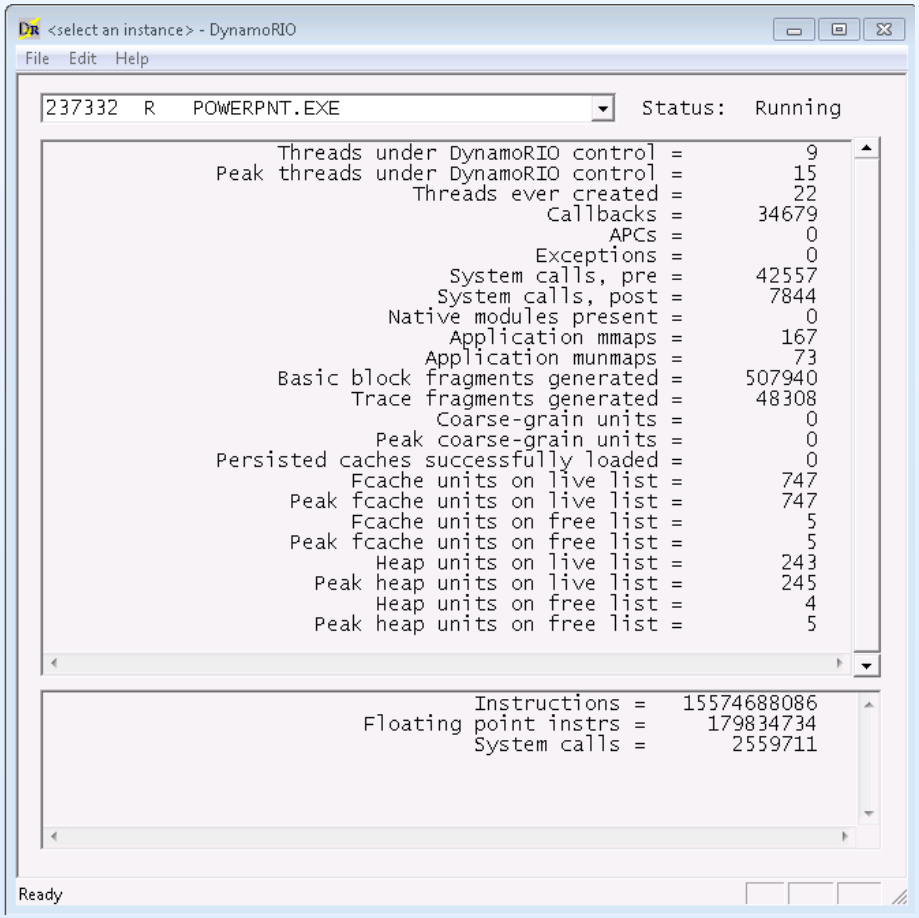
DynamoRIO + Client ⇒ Tool



DynamoRIO Demo

DynamoRIO Demo





DynamoRIO API

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

Cross-Platform Approach

- A consistent interface that works across platforms
  - Windows, Linux, Android, MacOS
  - x86, AMD64, ARM, AArch64
  - Thread-private versus thread-shared
- Same client source code generally works on all combinations of platforms
- Main exception is low-level hand-tuned instruction sequences
  - Even that can be made mostly cross-platform using provided IR utilities explained later

DynamoRIO Clients

- DynamoRIO
  - "*Interpret*" the application execution
  - API: provide a cross-platform interface for clients
    - Event callbacks
    - Utilities
- Client code
  - Extend DynamoRIO with customized operations
    - Event driven: operate on different events
    - Manipulate code stream
- DynamoRIO Extensions
  - Libraries that extend DynamoRIO API
    - *drmgr, drsyms, drwrap, ...*

Events

- Application events
  - Runtime events occur during application execution
    - E.g. pre/post system calls
- DynamoRIO events
  - Code stream
    - Basic block/trace creation
  - Nudge
  - State restore
- Event callbacks registration
  - dr_register_*_event
  - dr_unregister_*_event

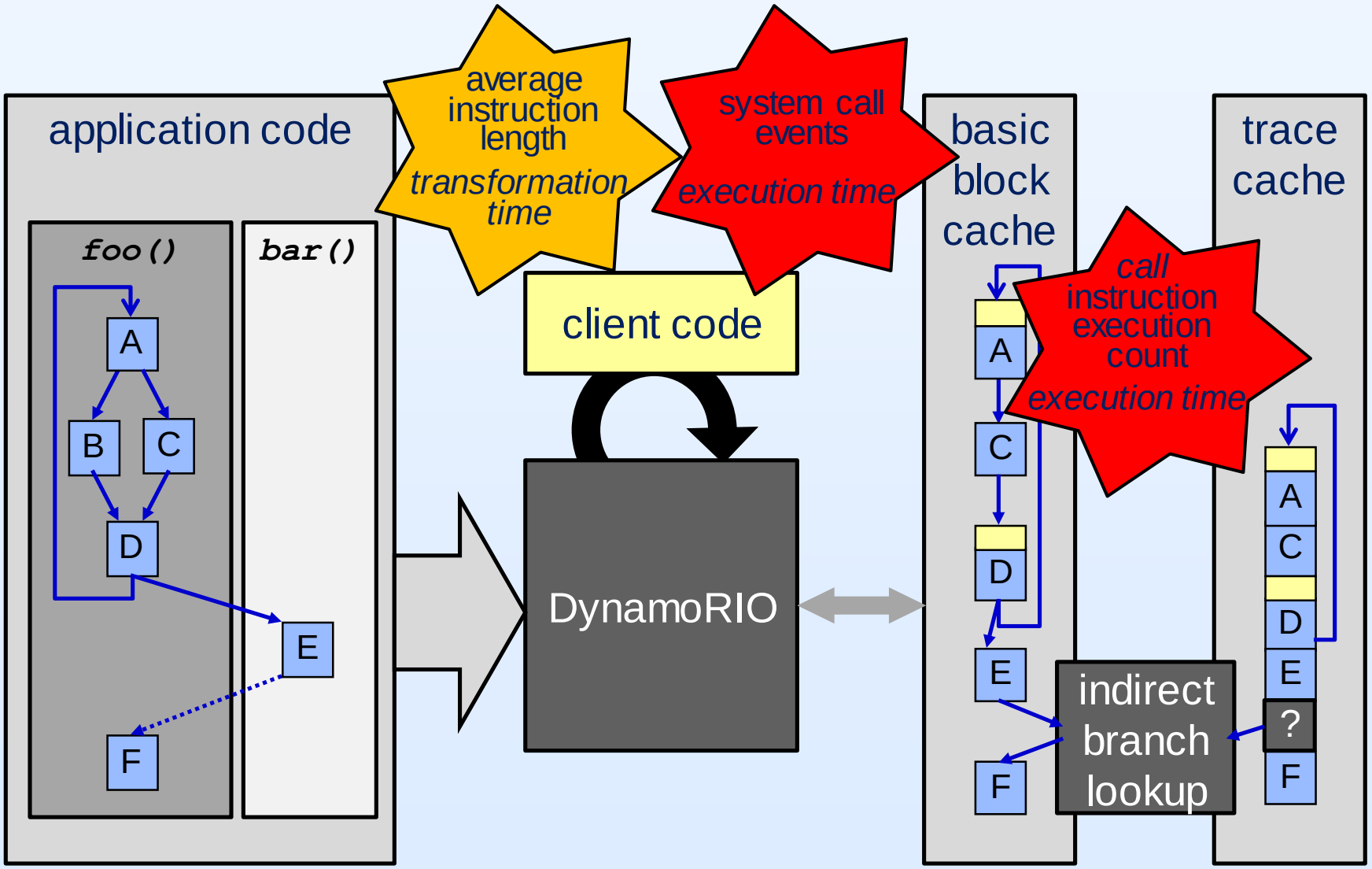
Application Events

- Process start, stop, fork
- Thread creation and deletion
- Library load and unload
- Exception (Windows)
  - Client chooses whether to deliver or suppress
- Signal (Linux)
  - Client chooses whether to deliver, suppress, bypass, or redirect
- Pre- and post- system call
  - Platform-independent system call parameter access
  - Client can modify a system call
  - Invoke additional system call as the app

DynamoRIO 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

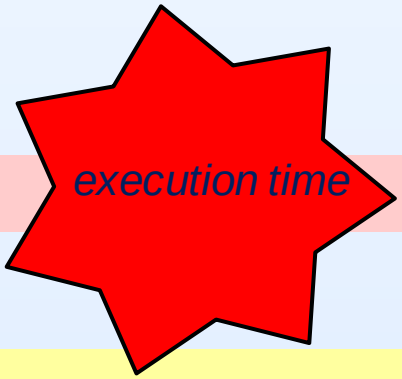
Transformation Time vs Execution Time



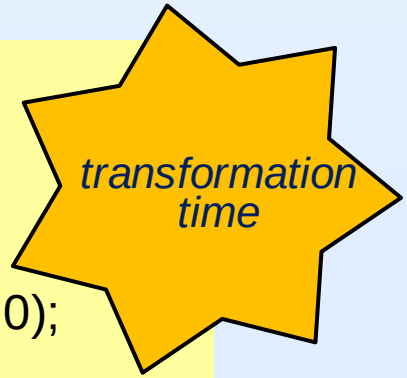
Client Example: Dynamic Basic Block Count

```
#include “dr_api.h”
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[]) {
    dr_register_bb_event(event_bb);
}
```

```
static void bbcount() { global_count++; }
```




```
static dr_emit_flags_t
event_bb(void *drcontext, void *tag, instrlist_t *bb,
         bool for_trace, bool translating) {
    dr_insert_clean_call(drcontext, bb, instrlist_first_app(bb),
                         (void *)bbcount, false /* save fpstate */, 0);
    return DR_EMIT_DEFAULT;
}
```



DynamRIO Extensions

- Challenges (while building Dr. Memory)
  - Multiple components
    - System call database, shadow memory managment, memory allocation tracking, function wrapping, etc.
  - Manipulate the same instruction list
    - Different instrumentation passes may interfere with each other
  - Good performance
    - Avoid unnecessary save/restore
    - Instrumentation across multiple instructions (e.g., shadow xl8)
  - Code reuse
- Solution
  - DynamoRIO Extensions
    - drmgr, drwrap, drsyscall, …

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 with CMake
  - **`use_DynamoRIO_extension(myclient extensionname)`**

DynamoRIO Extensions

- Current Extensions:
  - *drreg*: register stealing and allocating
  - *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/AArch64 decoding/encoding library
  - *umbra*: shadow memory framework

DynamoRIO Extensions: drmgr

- Mediation among multiple components
  - Callbacks invocation orders: priority

```
drmgr_priority_t {
  size_t       struct_size;
  const char   *name;       // relative order
  const char   *before;
  const char   *after;
  int          priority;    // numeric order
};
```

  - Basic block instrumentation: 4 instrumentation phases
    - Application-to-application transformation
    - Application analysis
    - Instrumentation insertion
    - Instrumentation optimization

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)
```

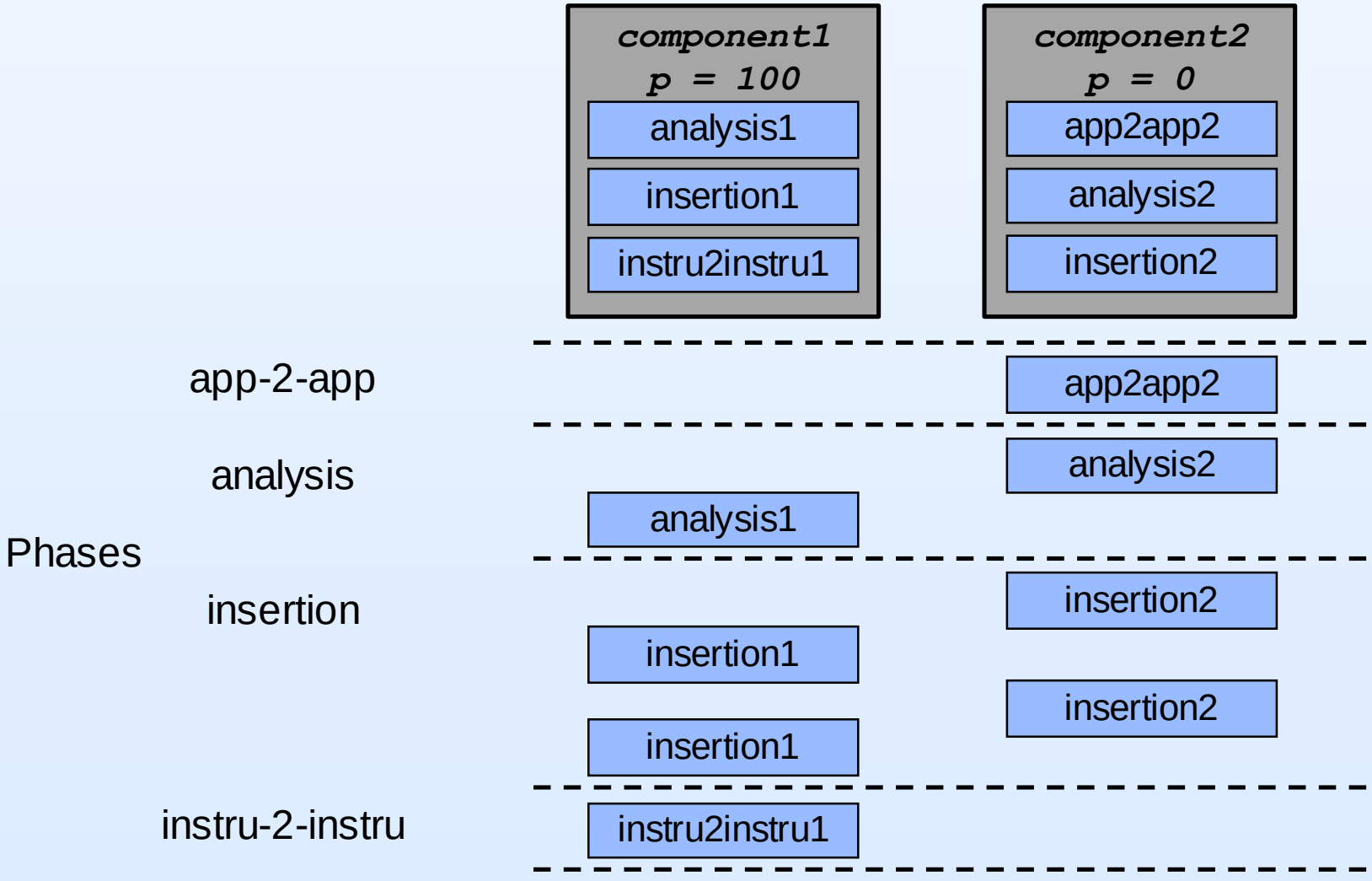
- Application Analysis
- Instrumentation Insertion

```
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: Instrumentation Phases

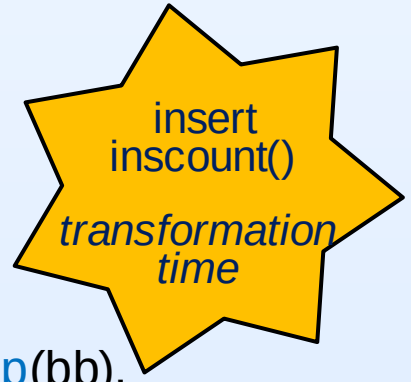


drmgr: instruction counting

```
static dr_emit_flags_t
event_bb_analysis(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating, void **user_data)
{
    instr_t *instr;
    uint num_instrs;

    for (instr = instrlist_first_app(bb), num_instrs = 0;
         instr != NULL;
         instr = instr_get_next_app(instr)) {
        num_instrs++;
    }

    *user_data = (void *)(ptr_uint_t) num_instrs;

    return DR_EMIT_DEFAULT;
}
```

drmgr: instruction counting

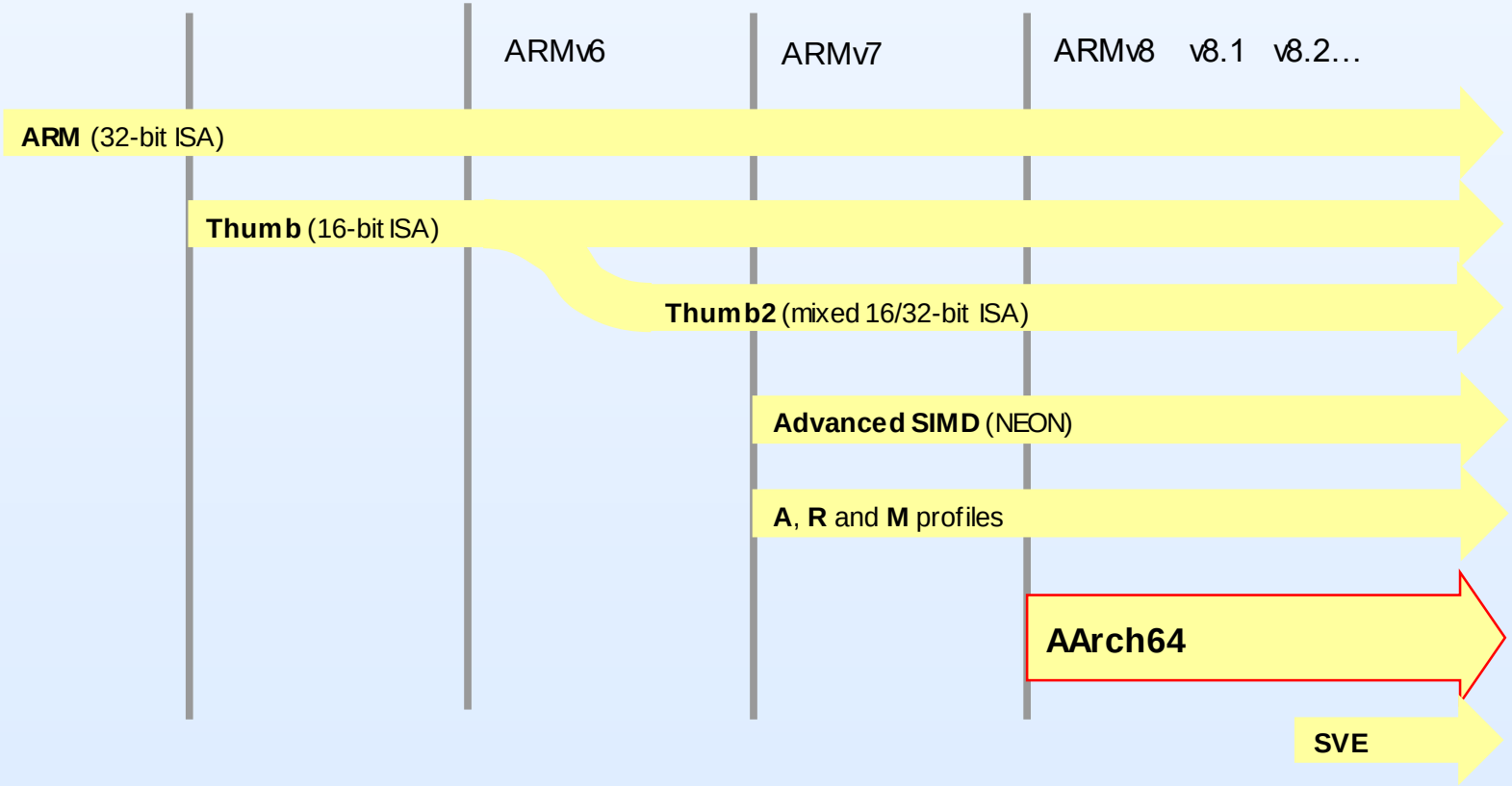
```
static dr_emit_flags_t
event_app_instruction(void *drcontext, void *tag, instrlist_t *bb, instr_t *instr,
                      bool for_trace, bool translating, void *user_data)
{
    uint num_instrs;

    if (!drmgr_is_first_instr(drcontext, instr))
        return DR_EMIT_DEFAULT;

    num_instrs = (uint)(ptr_uint_t) user_data;
    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;
}
```



ARM

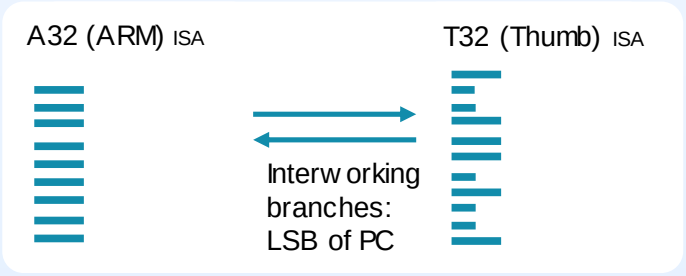
8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

ARM background: architecture timeline



ARM instruction sets and execution states (as of ARMv8)

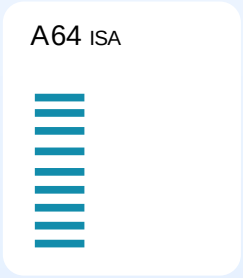
AArch32 execution state



GPR registers (512 bits / 64 bytes total):
16 32-bit registers R0 to R15
R15 is PC – means lots of interesting branches!
R13 is SP
R14 is link register (but may be used as work reg)
Conventions for frame pointers etc. – not architected

VFP/NEON registers (2048 bits / 256 bytes total)
32, 64 and 128 bit registers overlay the register file

AArch64 execution state



GPR registers (1984 bits / 248 bytes total):
31 64-bit registers R0 to R30
Full register accessed as X0 to X30
Low words accessed as W0 to W30
‘31’ in a register operand indicates zero register (XZR or WZR)
Dedicated PC – written by branch instructions only
Dedicated SP

VFP/NEON registers (4096 bits / 512 bytes total)
V0-V31 can be each accessed as 8, 16, 32, 64 or 128 bits

ARM/AArch64 unimplemented features

- Clean call optimisation
- Trace construction
- Thread-private fragment cache
- Some samples: bbcount, cbr, countcalls,...
- AArch64 Android
- AArch64 Dr. Memory
- AArch64 encoder/decoder: FP/SIMD non-memory-access
- Load/store exclusive infinite loop on some hardware (i#1698)
- AArch64 handling of flags

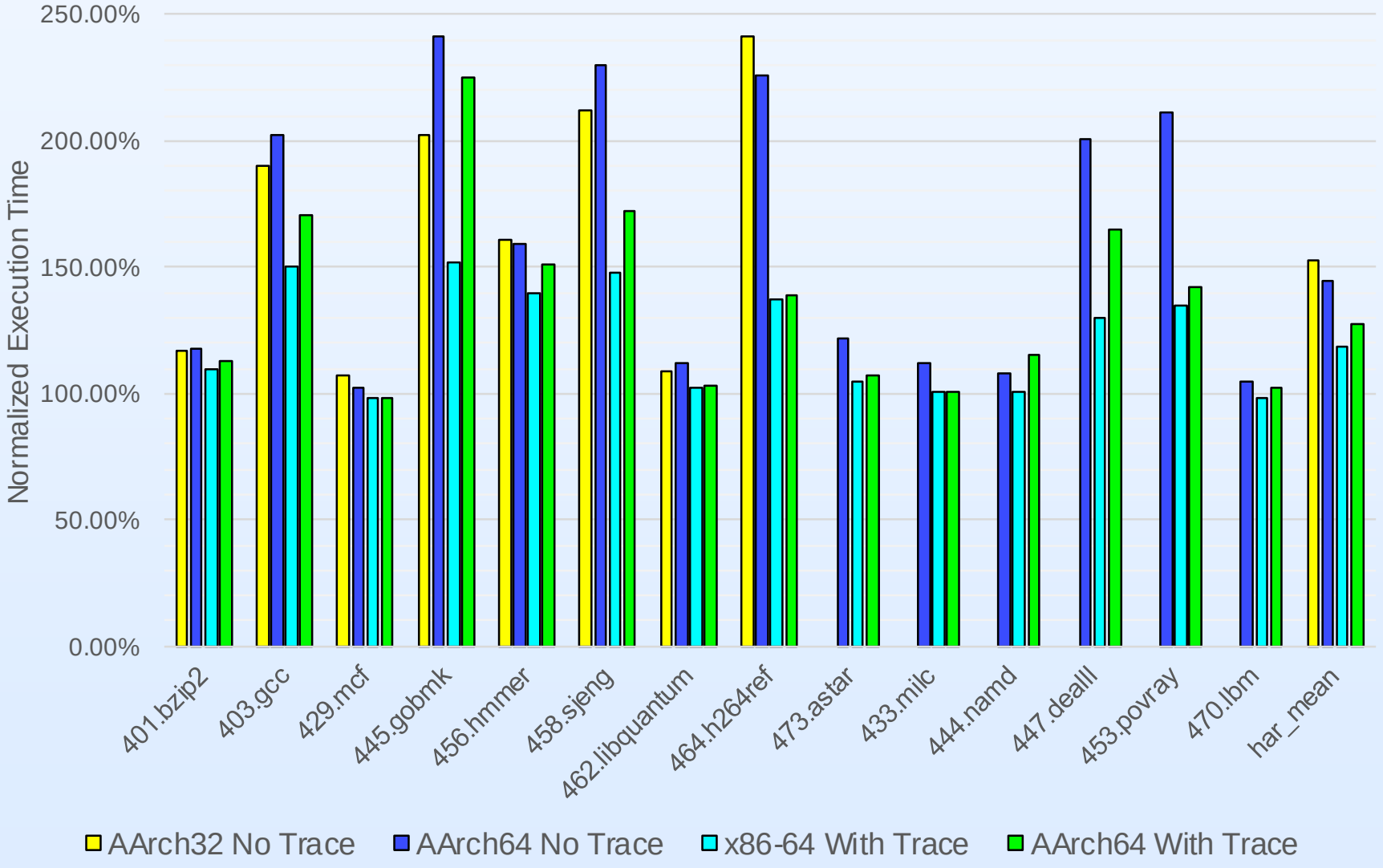
Status of ARM/AArch64

- Number of tests run by "make test" on Linux
  - ARM 131
  - AArch64 139
  - x86_64 251
- AArch64 is not yet as widely tested
  - Firefox runs under DynamoRIO on some OS versions
  - Firefox crashes under DynamoRIO on other OS versions

Writing clients for ARM/AArch64

- Safe to use all client APIs which are covered in the test suites
- Code manipulation API restrictions (read ARMv8)
  - INSTR_CREATE macro covers common opcodes
    - Not yet completed for AArch64
  - Limited offset ranges
    - Direct or conditional branch, pc-relative access, shift offsets, etc.
  - Additional scratch register is required in some cases
    - Saving SP value to memory, etc.
  - Cannot store an immediate directly to memory
    - `INSTR_CREATE_str(dc, OPND_CREATE_INT(imm))` **wrong!**
  - AArch64: SP/ZR is not a general purpose register!
    - `dr_save_reg(dc,…, opnd_create_reg(DR_REG_XSP))` **wrong!**

Preliminary Performance Comparison



AArch64 Demonstrations

(swap to live demo)

DynamoRIO API

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

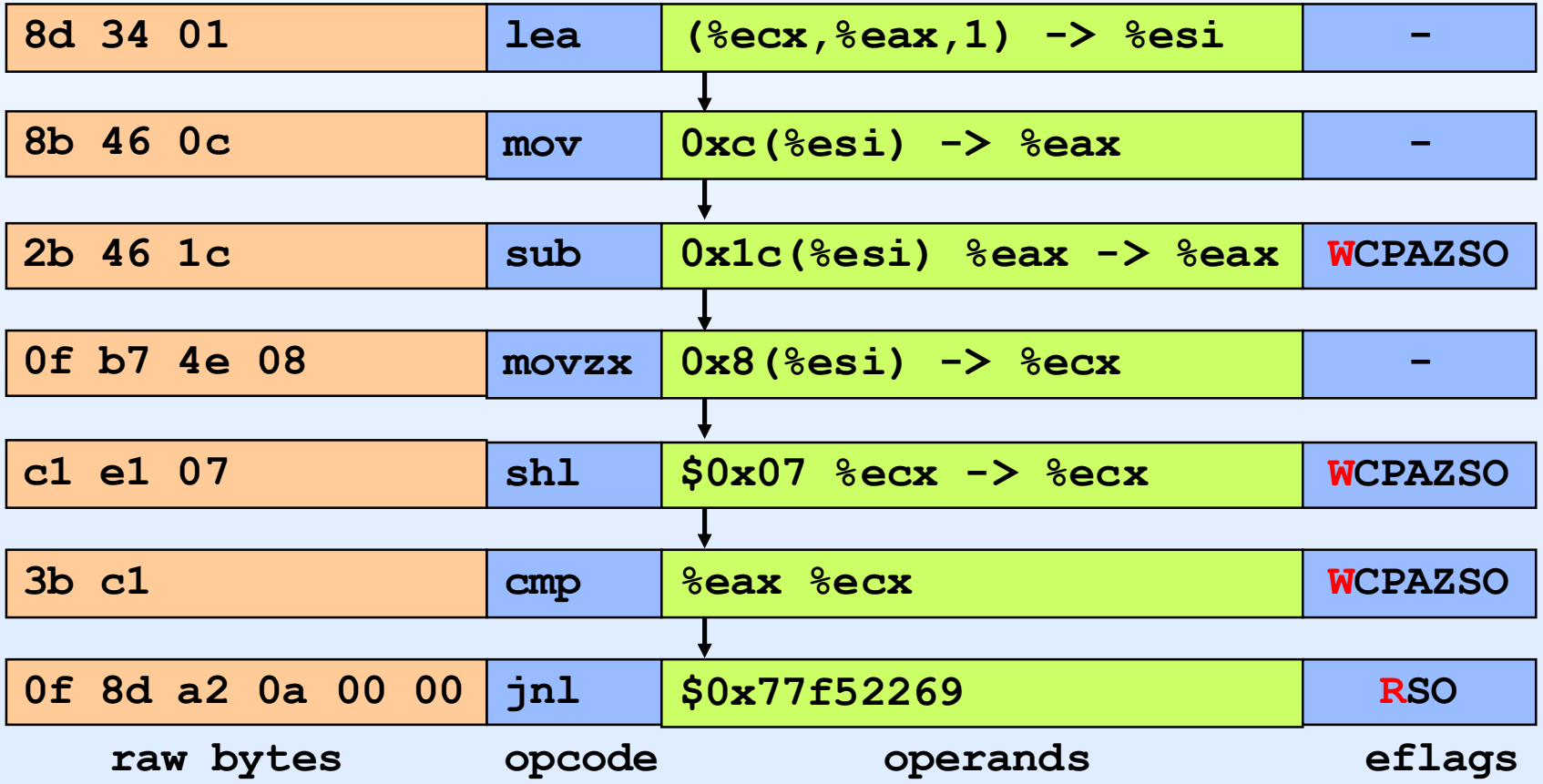
DynamoRIO API: Instruction Representation

- Full IA-32/AMD64/AArch32/AArch64 instruction representation
- Instruction creation with auto-implicit-operands
- Operand iteration
- Instruction lists with iteration, insertion, removal
- Decoding at various levels of detail
- Encoding

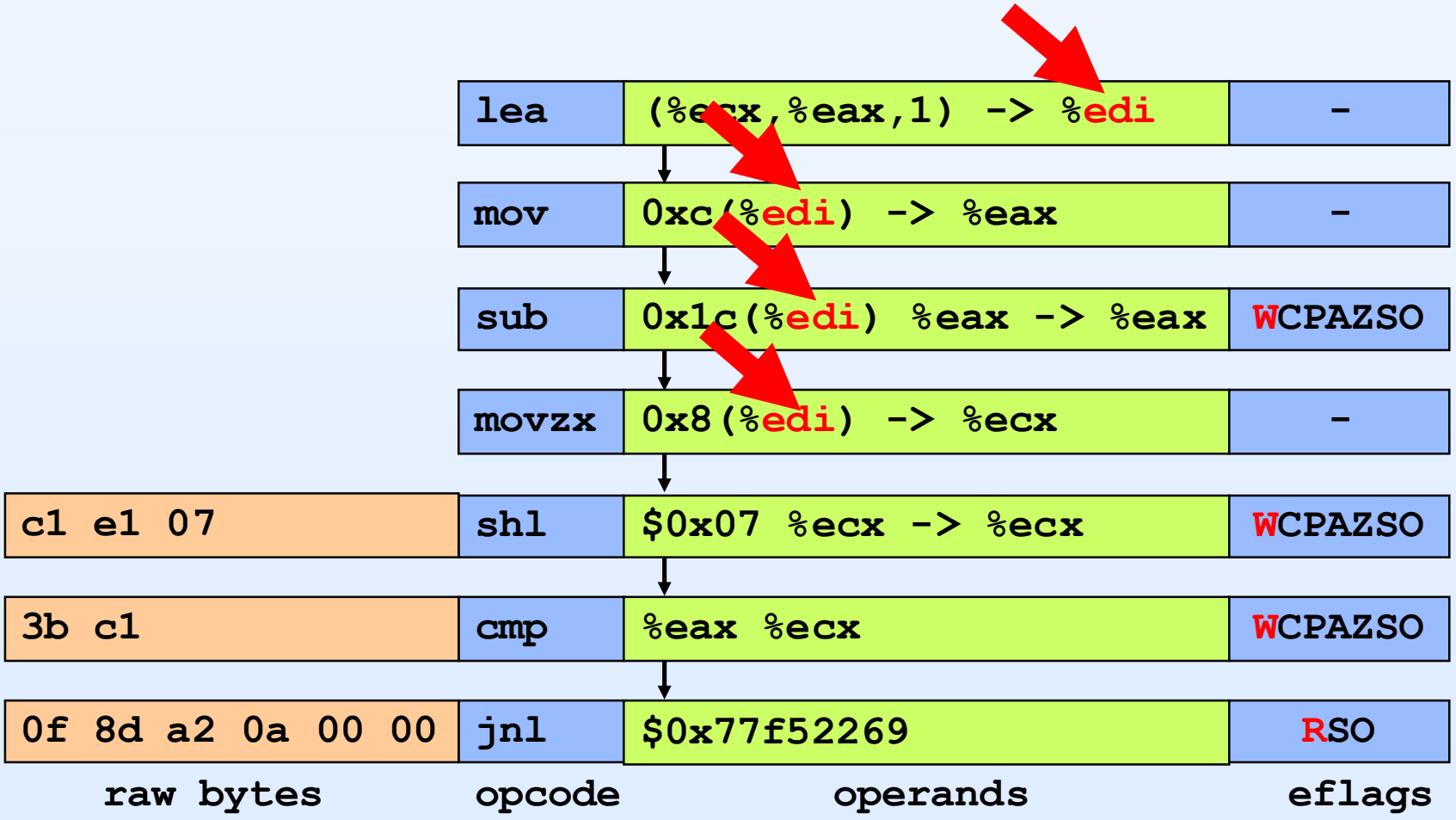
Reminder: No Intermediate Layers

- IR mirrors underlying ISA
  - Preserves optimized application code
  - Intermediate layers incur significant performance impact
    - QEMU (user-mode) **6x** slower than DR, Valgrind **4x** slower than DR
  - *This is the key to good performance*
- Still have an abstraction layer
  - Block or trace = list of instructions
  - Instruction = lists of source and destination operands
  - Tool code often still cross-platform
    - "Does this instruction read memory?"

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));
```

Cross-Platform IR Support

- Instruction creation: XINST_CREATE_* macros
  - XINST_CREATE_load(), XINST_CREATE_jump(), etc.
- Generic instruction and operand queries
  - E.g.: instr_writes_memory(), instr_reads_from_reg(), instr_compute_address(), instr_is_return()
  - E.g.: opnd_is_memory_reference(), opnd_uses_reg()
- Generic instrumentation creation helpers
  - E.g.: instrlist_insert_mov_immed_ptrsz(), instrlist_insert_mov_instr_addr()
- ISA-specific concepts applied to all ISA's
  - E.g., predication
- *drreg* Extension for using registers without naming them

ISA Modes

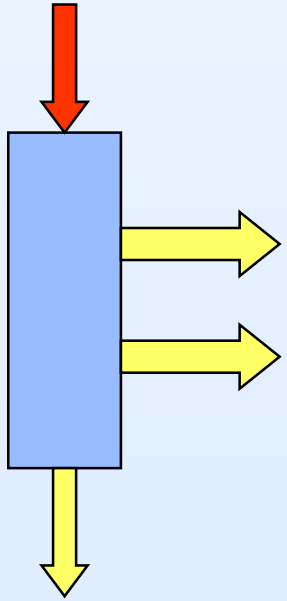
- IR covers all modes
  - Abstracts away underlying mode
- ARM and Thumb
  - ARM build fully supports both modes
- AMD64 and IA-32
  - 32-bit x86 build of DynamoRIO only handles 32-bit code
  - 64-bit AMD64 build of DynamoRIO decodes/encodes both 32-bit and 64-bit code
  - Current release does support executing applications that mix the two modes on Windows
  - "X" cross-mode registers
    - DR_REG_XAX is DR_REG_EAX when compiled 32-bit, and DR_REG_RAX when compiled 64-bit

ISA Modes: Per-Thread and Per-Instruction

- When going to or from the IR, the thread mode and instruction mode determine how instructions are interpreted
- When decoding, current thread's mode is used
  - Can be changed with dr_set_isa_mode()
- When encoding, that instruction's mode is used
  - When created, set to mode of current thread
  - Can be changed with instr_set_isa_mode()

Linear Control Flow

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



DynamoRIO API: Code Manipulation

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

State Preservation

- Application context is the native context in the code cache: it must be explicitly preserved
  - Another key design choice for performance
- Client must spill registers before using them
  - Must also save and restore condition codes
  - *drreg* Extension makes this easy
- In C/C++ code, client must preserve floating-point state
  - Not saved by DR context switches (unless request on clean call)
  - dr_insert_save_fpstate(), dr_insert_restore_fpstate()

drreg Extension: Register Preservation

- *drreg* mediates general-purpose scratch register and condition code spilling and restoring
- Reservation model: request exclusive access
- Lazily restores to avoid superfluous adjacent spills
- Most library routines take in a scratch register
- Example:

```
/* acquire a register, spilling it if necessary */
drreg_reserve_register(drcontext, bb, inst, NULL, &reg);
/* pass register to library routine */
drx_buf_insert_load_buf_ptr(drcontext, buf, bb, inst, reg);
/* release the register, restoring from memory if necessary */
drreg_unreserve_register(drcontext, bb, inst, reg);
```

Where to Spill?

- Absolute addressing
  - Thread-private code caches only
- Application stack
  - Not reliable or transparent
- Stolen register
  - Used by DynamoRIO on ARM and AArch64
- Segment
  - Used by DynamoRIO on AMD64 and IA-32

Thread-Local Storage

- Spill slots for registers
  - DR provides several fast slots and slower slots
  - *drreg* provides a configurable number of fast slots
- General thread-local storage
  - drmgr_register_tls_field()
  - drmgr_insert_read_tls_field(), drmgr_insert_write_tls_field()
  - From C code: drmgr_get_tls_field(), drmgr_set_tls_field()
  - Parallel routines for CLS fields (advanced Windows topic)

Clean Calls

```
if (instr_is_mbr(instr)) {
  app_pc address = instr_get_app_pc(instr);
  uint opcode = instr_get_opcode(instr);
  dr_insert_clean_call(drcontext, ilist, instr, (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

Flavors of Cache Flushing

- 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

Memory Tracing

- *drutil* Extension provides utilities for memory address tracing:
  - Address acquisition
    - Converts segment-based addresses into linear addresses
  - String loop expansion
    - Turns x86 string loops into regular loops with explicit control flow for simpler instrumentation

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

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

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

DynamoRIO Heap

- Three flavors:
  - Thread-private: no synchronization; thread lifetime
  - Global: synchronized, process lifetime
    - malloc(), HeapAlloc(), etc. redirect here
  - "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 and Synchronization 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
  - “Stop-the-world”

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 identification
    - Cache line size, ISA features, etc.
  - Symbol lookup
  - Function replacing and wrapping

drsyms Extension: 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

drwrap Extension: Func Replacing/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

Create Your Own Tool

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

How to Create Your Own Tool

- Event driven
  - Application, DynamoRIO, client, and Extensions
  - Events
- Write your DynamoRIO client
  - Part 1: Register event callbacks
  - Part 2: Implement event callbacks
  - Part 3: Instrumentation
- Config, build, and run
  - CMake

How to Create Your Own Tool

- Event driven
  - Application, DynamoRIO, client, and Extensions
  - Events
- Write your DynamoRIO client
  - Part 1: Register event callbacks
  - Part 2: Implement event callbacks
  - Part 3: Instrumentation
- Config, build, and run
  - CMake

Event Driven

- Application, DynamoRIO, clients, and Extensions
  - DynamoRIO "*interprets*" the application execution
  - A client takes actions on interested events
  - Client interacts with DynamoRIO and app via API + Extensions
    - Event callbacks
    - Utility functions
- Events
  - Application events
    - E.g., pre/post system calls
  - DynamoRIO events
    - E.g., basic block creation
  - Multiple components mediated by drmgr

Write Your DynamoRIO Client

- Part 1: Register event callbacks
  - dr_client_main
  - drmgr_register_*_event()
    - App events: pre_syscall, …
    - DR events: bb, …
- Part 2: Implement event callbacks
  - Strace
- Part 3: Instrumentation
  - Dynamic instruction counting

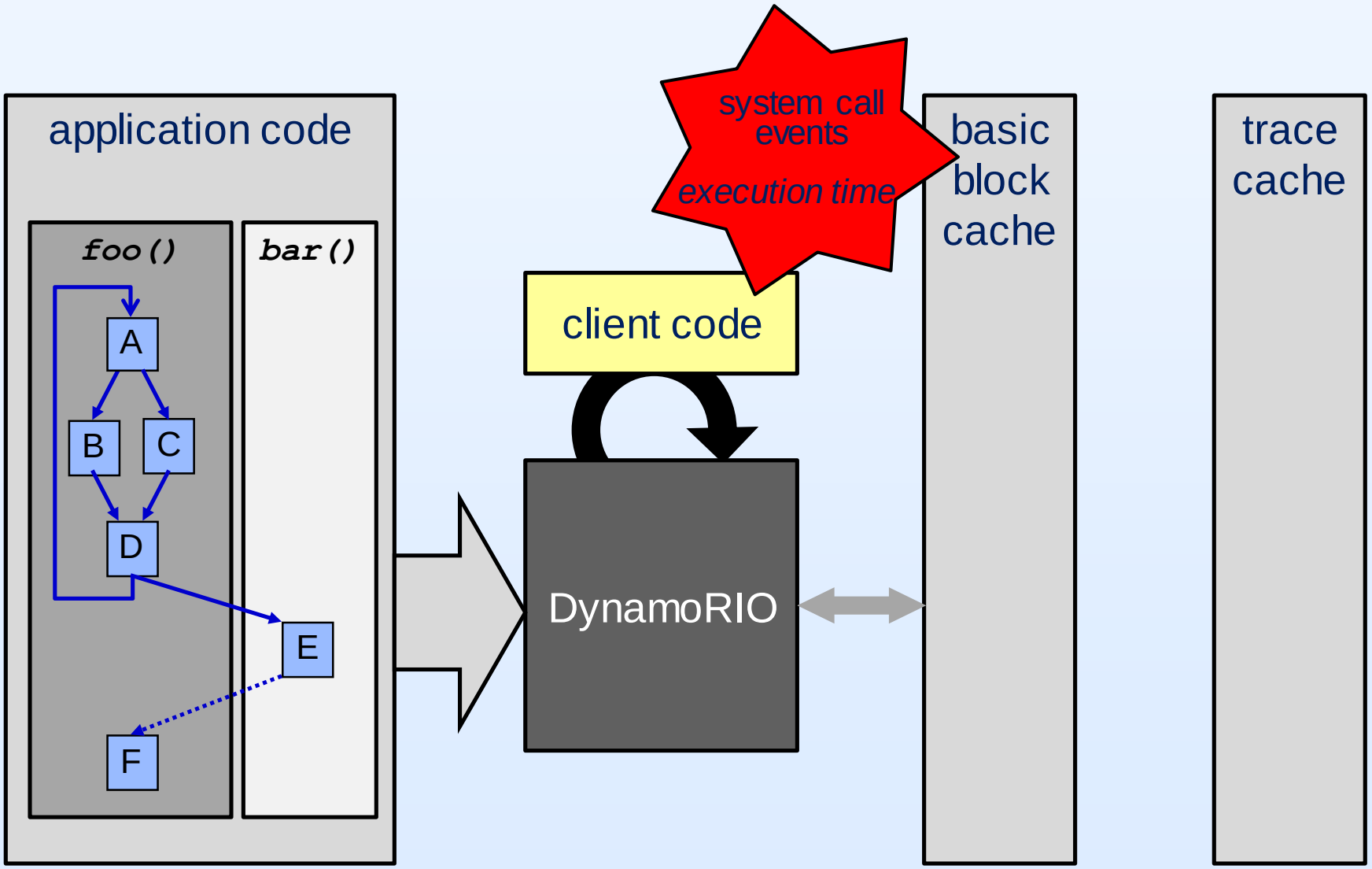
Part 1: Register Event Callbacks - strace

```
#include “dr_api.h”
#include “drmgr.h”

DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[]) {
    drmgr_init();
    dr_register_exit_event(event_exit);
    drmgr_register_post_syscall_event(event_post_syscall);
}


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

Write Your DynamoRIO Client

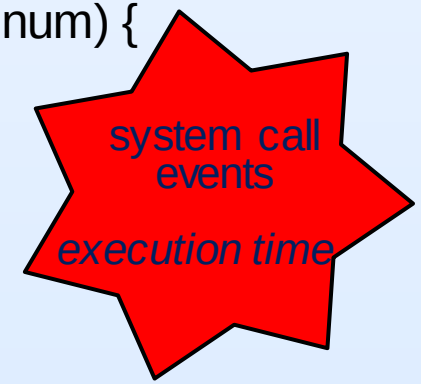
- Part 1: Register event callbacks
  - dr_client_main
  - drmgr_register_*_event()
    - App events: pre_syscall, …
    - DR events: bb, …
- Part 2: Implement event callbacks
  - Strace
- Part 3: Instrumentation
  - Dynamic instruction counting

Part 2: Implement Event Callbacks - strace



Part 2: Implement Event Callbacks - strace

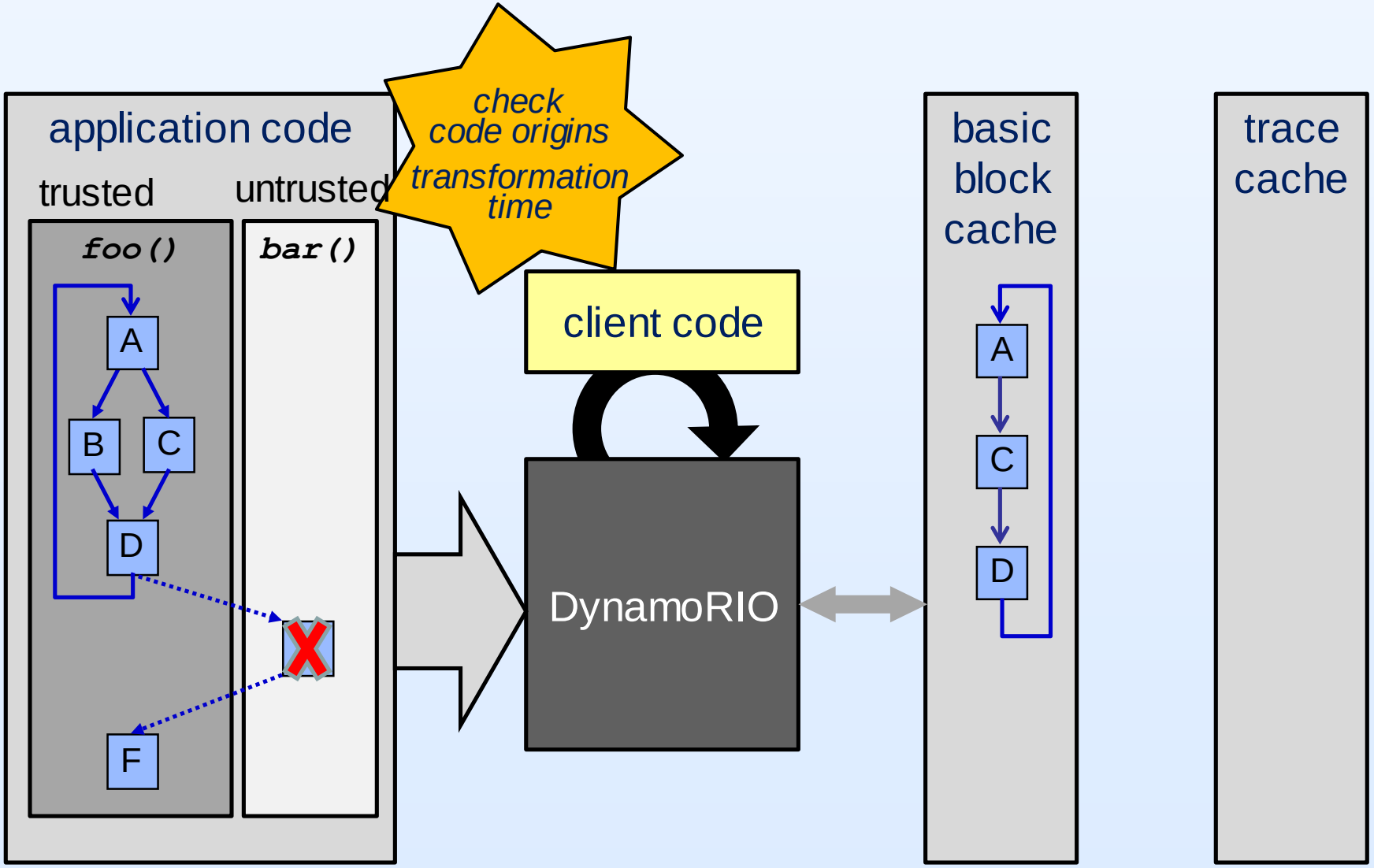
```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[]) {
    …
    drmgr_register_post_syscall_event(event_post_syscall);
}

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 Callbacks - strace

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[]) {
    …
    drmgr_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;
}
```

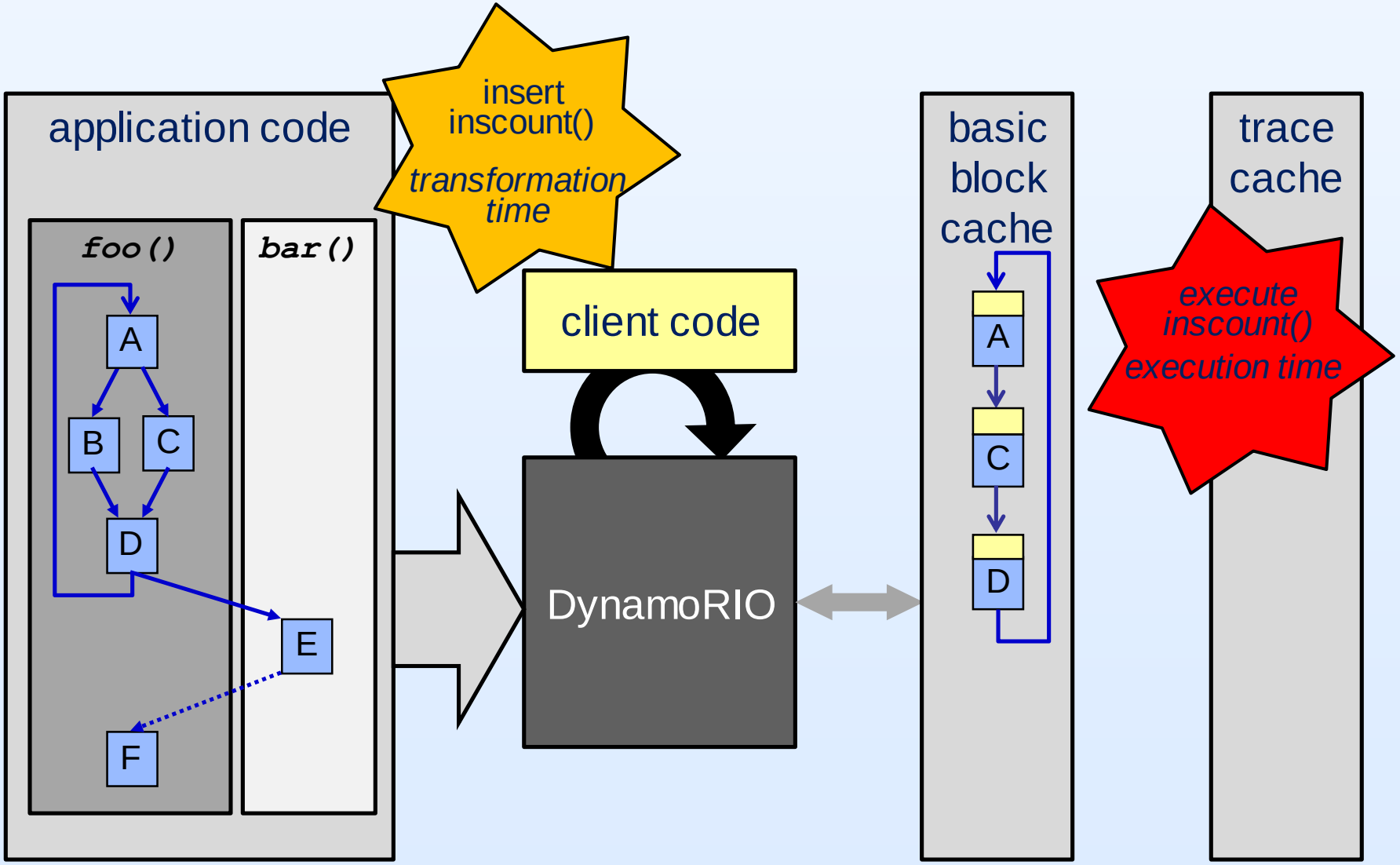
Part 2: Implement Event Callbacks - security



Write Your DynamoRIO Client

- Part 1: Register event callbacks
  - dr_client_main
  - drmgr_register_*_event()
    - App events: pre_syscall, …
    - DR events: bb, …
- Part 2: Implement event callbacks
  - Strace
- Part 3: Instrumentation
  - Dynamic instruction counting

Part 3: Instrumentation: instruction counting



Part 3: Instrumentation: instruction counting

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[]) {
    …
    drmgr_register_bb_instrumentation_event
        (event_bb_analysis, event_app_instruction, NULL);
}
```

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

Part 3: Instrumentation: instruction counting

```
static dr_emit_flags_t
event_bb_analysis(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating, void **user_data)
{
    instr_t *instr;
    uint num_instrs;

    for (instr = instrlist_first_app(bb), num_instrs = 0;
         instr != NULL;
         instr = instr_get_next_app(instr)) {
        num_instrs++;
    }

    *user_data = (void *)(ptr_uint_t) num_instrs;

    return DR_EMIT_DEFAULT;
}
```

Part 3: Instrumentation: instruction counting

```
static dr_emit_flags_t
event_app_instruction(void *drcontext, void *tag, instrlist_t *bb, instr_t *instr,
                      bool for_trace, bool translating, void *user_data)
{
    uint num_instrs;

    if (!drmgr_is_first_instr(drcontext, instr))
        return DR_EMIT_DEFAULT;

    num_instrs = (uint)(ptr_uint_t) user_data;
    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;
}
```

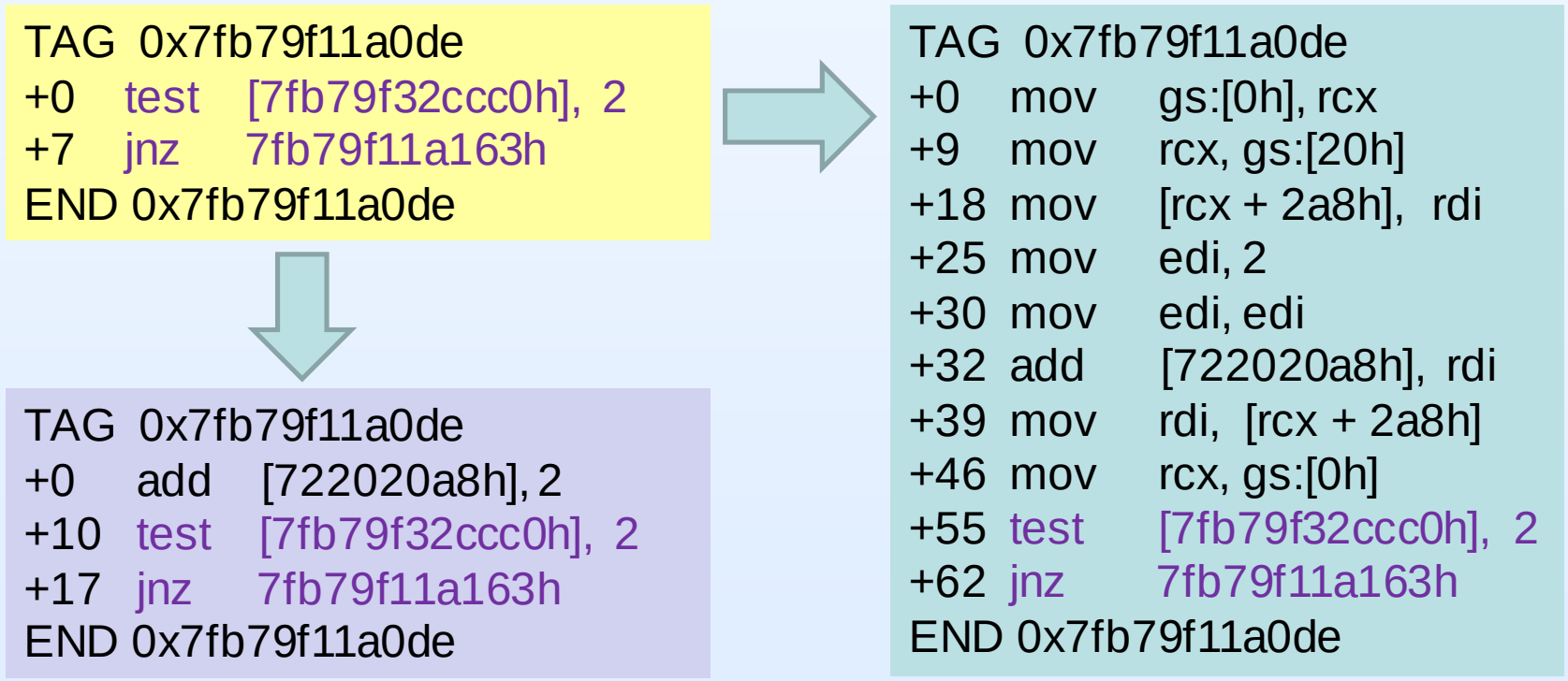
Clean Calls

- 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
  - -opt_cleancall runtime option controls aggressiveness

Counter Incrementing

- Instruction counting example uses a clean call for illustration purposes only
  - Additional actions can be added to the callee
  - If only a counter increment is needed, use drx_insert_counter_update() instead of the clean call
- drx_insert_counter_update()
  - Convenience routine for inserting inlined counter updates
  - Supports 32-bit and 64-bit counters
  - Supports atomic updates
  - Automatically saves the condition codes if necessary
  - Merges adjacent counter updates

Part 3: Instrumentation: instruction counting



How to Create Your Own Tool

- Event driven
  - Application, DynamoRIO, client, and Extensions
  - Events
- Write your DynamoRIO client
  - Part 1: Register event callbacks
  - Part 2: Implement event callbacks
  - Part 3: Instrumentation
- Config, build, and run
  - CMake

Config, Build, and Run

- CMake
  - http://www.cmake.org/
  - Generates 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)
use_DynamoRIO_extension(myclient drmgr)
```

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, for better child process blacklisting, etc.)
    - drconfig -reg <appname> -c <client> <client options>
    - drinject <app cmdline>

Runtime Options

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

Non-Standard Deployment

- drdecode
  - Static IA-32/AMD64/ARM/AArch64 decoding/encoding/instruction manipulation library
- Standalone API
  - Use DynamoRIO as a library of IA-32/AMD64/ARM/AArch64 manipulation routines plus cross-platform file i/o, locks, etc.
- Start/Stop API
  - Can instrument source code with where DynamoRIO should control the application

Debugging Your DynamoRIO Client

- Run with the debug build of DynamoRIO
  - drrun -debug …
  - -stderr_mask 0xN
  - -ignore_assert_list '*'
  - -disable_traces
- Run with logging
  - drrun -debug -loglevel 3
- Run under a debugger
  - drrun -debug -loglevel 3 -msgbox_mask 0xf …
  - -no_hide (windows)
  - Attach debugger to the process

Using a Debugger on a 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)

Obtaining Help

- Read the documentation
  - http://dynamorio.org/docs/
- Look at the provided tools and 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

Reporting Bugs

- Search the issue tracker linked from http://dynamorio.org first
  - https://github.com/DynamoRIO/dynamorio/issues
- File a new issue if not found
- Follow conventions on wiki
  - https://github.com/DynamoRIO/dynamorio/wiki/Bug-Reporting
  - 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

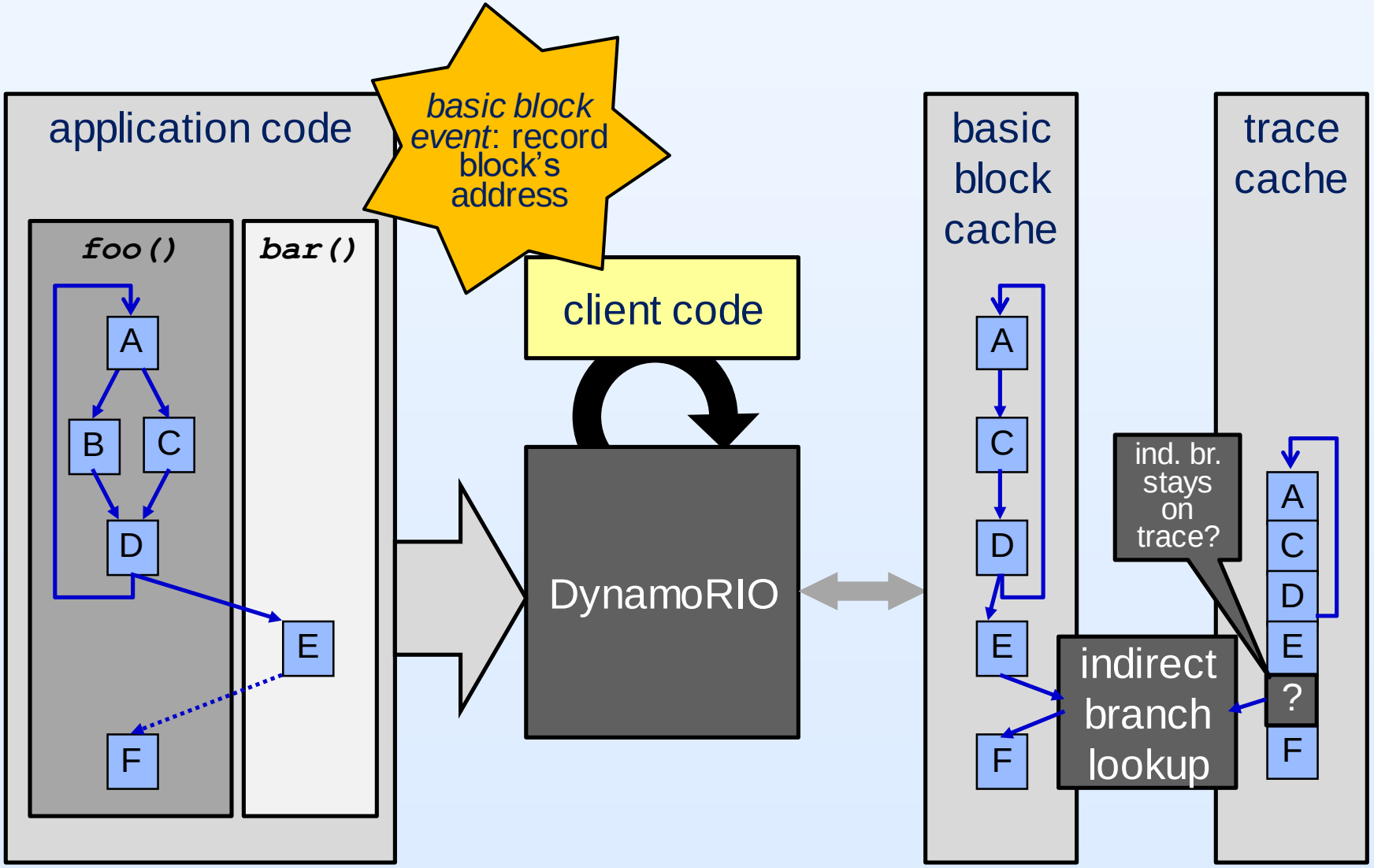
Tool Internals

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

Dr. Cov

- Collects code coverage information
- Records which basic blocks have been executed
- Operates on unmodified binaries
- Results are post-processed into the standard lcov format
- Usage:
  - **`drrun -t drcov -- <app>`**
  - **`tools/bin32/drcov2lcov -input drcov*.log`**
  - Now pass the resulting data file to lcov's genhtml script

Dr. Cov Implementation

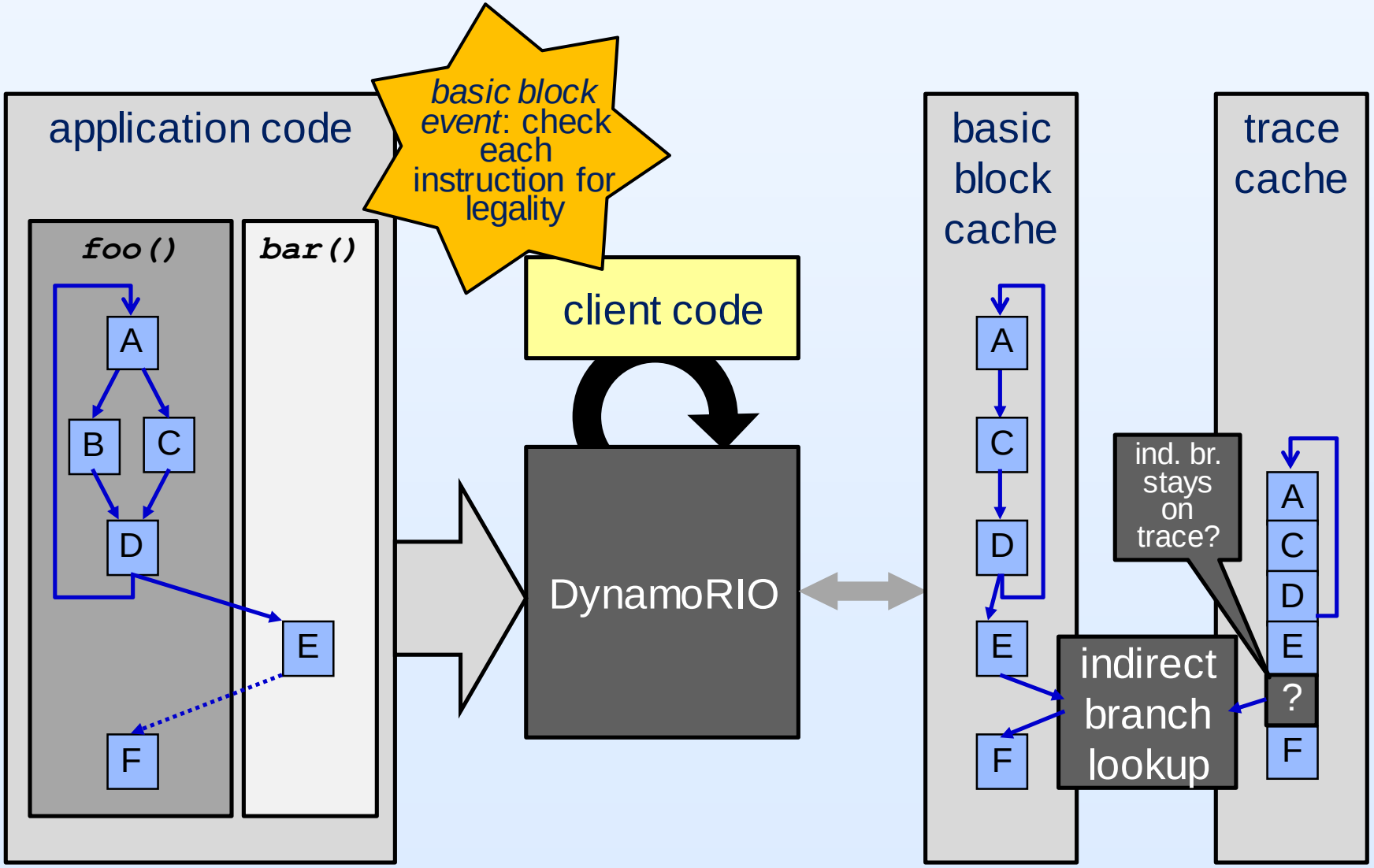


Dr. CPUSim: Legacy CPU Checker

- A tool for testing that applications will run correctly on legacy processors
- Looks for instructions that are illegal on the specified CPU model
- Fools **`cpuid`** checks
- Usage:
  - **`drrun -t drcpusim -cpu PentiumPro -- <app>`**
- Example:

```
% bin32/drrun -t drcpusim -cpu Pentium -continue -- hello.exe
<Invalid Pentium instruction "movd" @ hello.exe+0x85c3.
Continuing.>
```

Dr. CPUSim Implementation

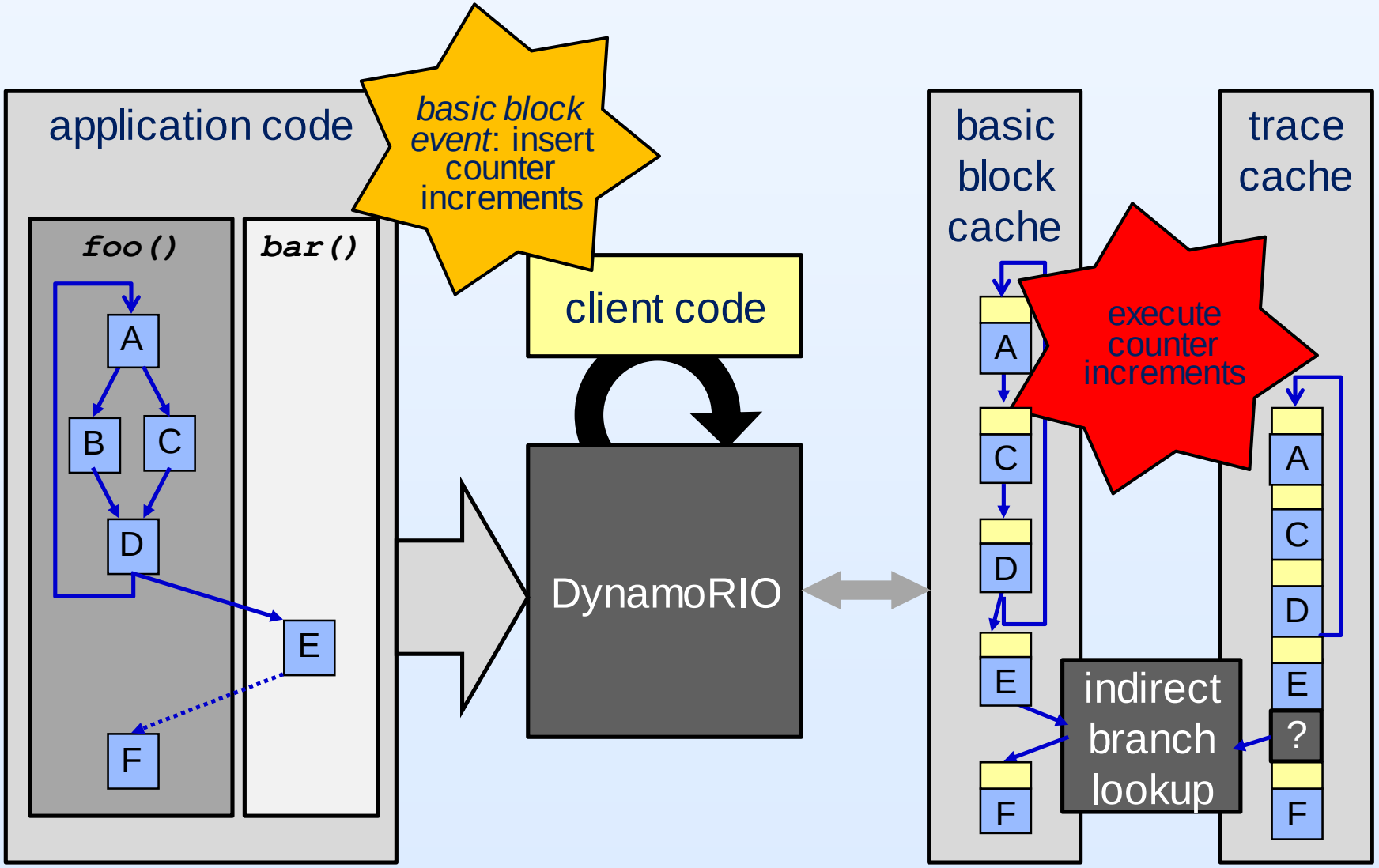


Opcode Mix Tool

- Monitors every instruction executed
- Counts the dynamic execution of each ISA opcode for the target application workload
- Usage:
  - **drrun -c samples/bin32/opcodes.dll -- <app>**
- Example:

```
% bin64/drrun -c samples/bin64/libopcodes.so -- gzip
lib64/libdrinjectlib.a
    Client opcodes is running
    Top 15 opcode execution counts in 64-bit AMD64 mode:
        3932460 : shl
        4607686 : shr
        6736566 : cmovnb
        <..>
```

Opcode Mix Tool Implementation

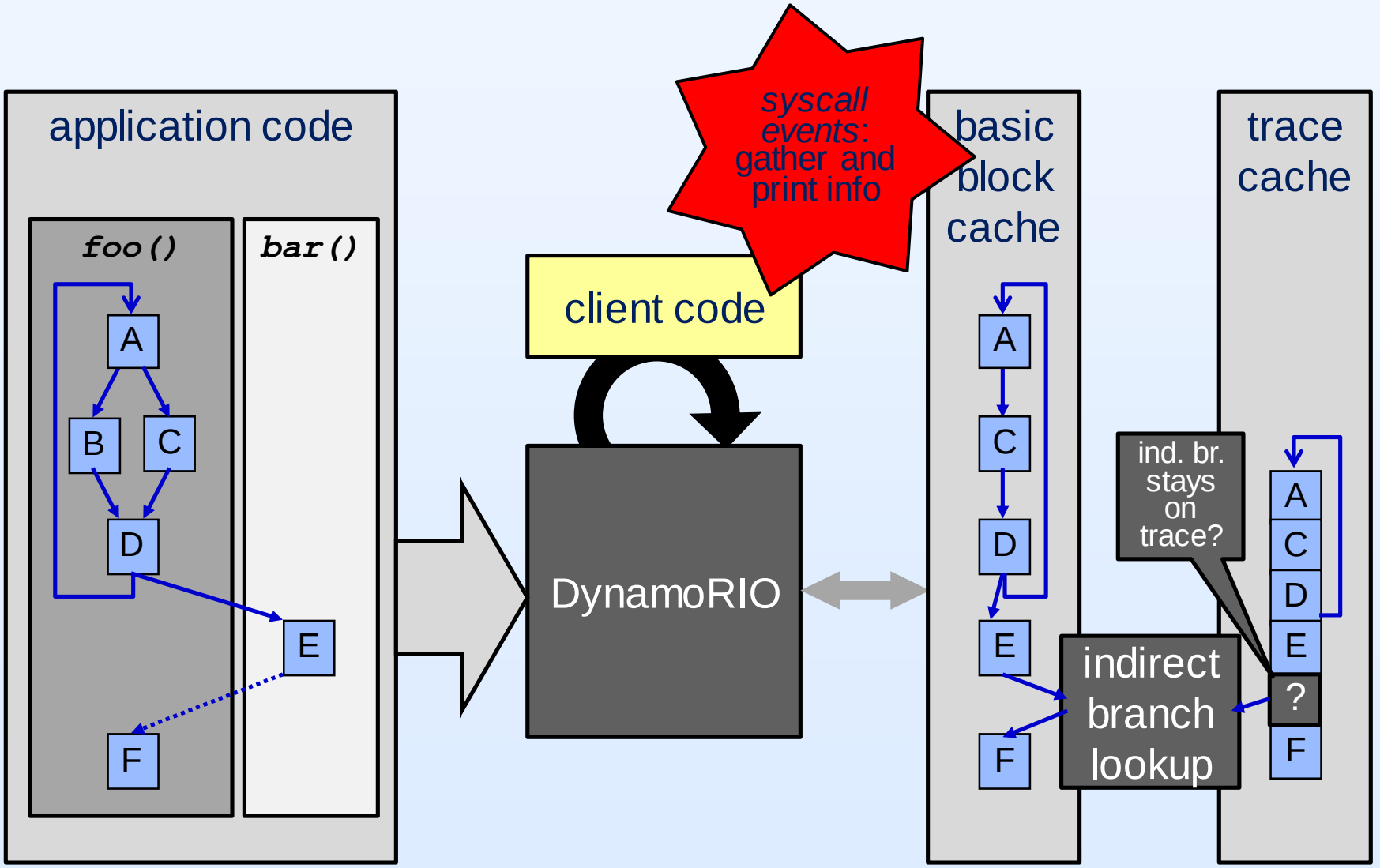


Dr. Strace: System Call Tracer

- Monitors all system calls executed by a target application
- Prints a trace of system calls and their arguments
- Operates on Windows, Linux, and Android
- Usage:
  - **`drrun -t drstrace -- <app>`**
- Example:

```
NtOpenKeyEx
        arg 0: 0x001fcd0c (type=HANDLE*, size=0x4)
        arg 1: 0x109 (type=unsigned int, size=0x4)
        arg 2: len=0x18, root=0x3c, name=150/152 "SOFTWARE\Microsoft\Windows
               NT\CurrentVersion\LanguagePack\SurrogateFallback", att=0x40,
               sd=0x00000000, sqos=0x00000000 (type=OBJECT_ATTRIBUTES*, size=0x4)
        arg 3: REG_OPTION_RESERVED or REG_OPTION_NON_VOLATILE (type=named constant,
               value=0x0, size=0x4)
    succeeded =>
        arg 0: 0x001fcd0c => 0x134 (type=HANDLE*, size=0x4)
        retval: 0x0 (type=NTSTATUS, size=0x4)
```

Dr. Strace Implementation



Dr. Syscall System Call Database

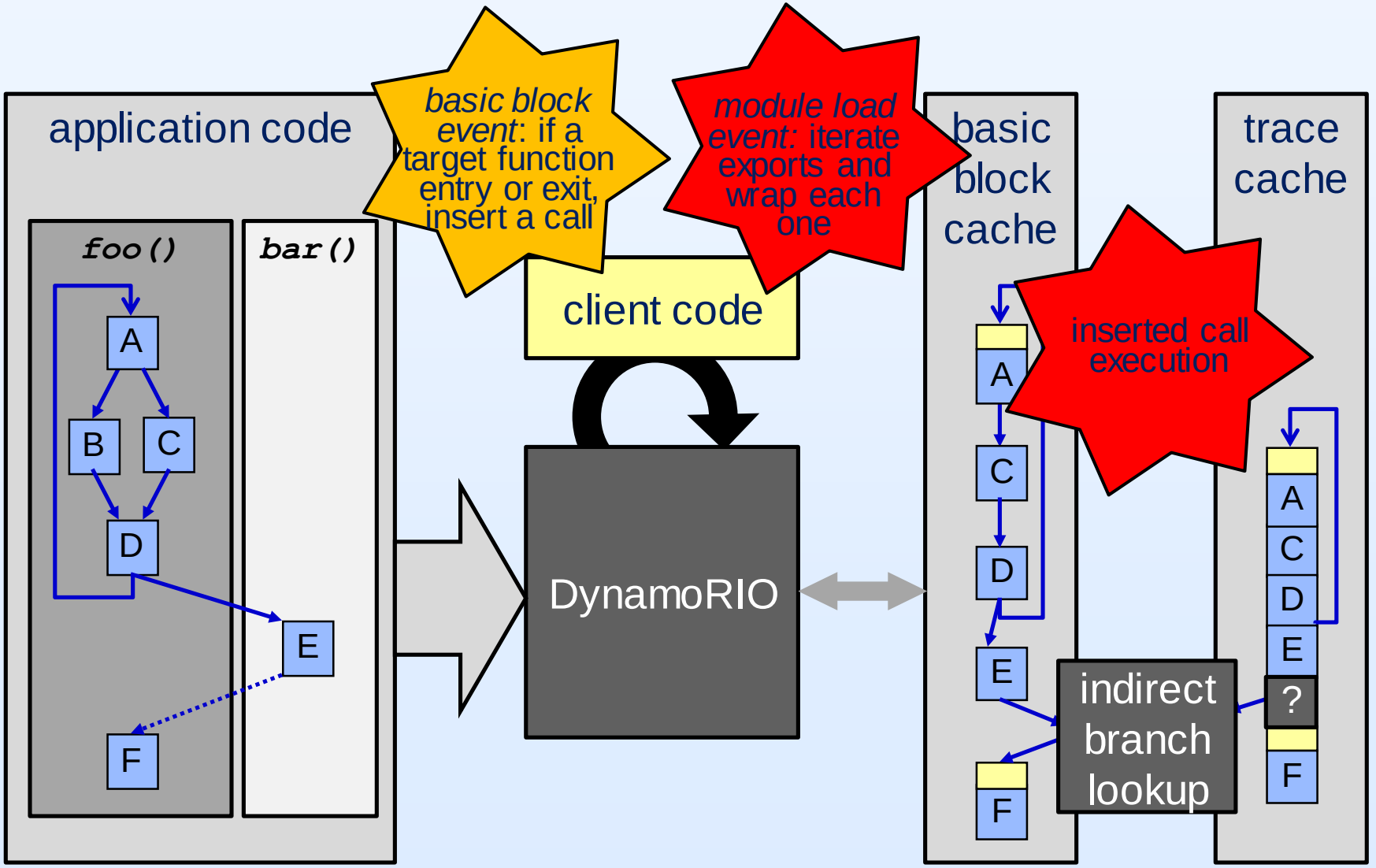
- *drsyscall* Extension
- Windows, Linux, and MacOS system call data
- Library interface to convert numbers to names or vice versa
- Static argument type iteration
- Dynamic argument value iteration
- Dynamic success querying
- Dynamic iteration over each memory region read or written

Dr. Ltrace: Library Tracer

- Intercepts calls to exported library functions
- Prints a trace of each function called and some argument information
- Usage:
  - **`drrun -t drltrace -- <app>`**
- Example:

```
~~~~ ntdll.dll!RtlEnterCriticalSection(0x006fc758, 0x0018fe50)
~~~~ KERNEL32.dll!WriteFile(0x00000268, 0x0018e9b0)
~~~~ KERNELBASE.dll!WriteFile(0x00000268, 0x0018e9b0)
Hello, world!
~~~~ ntdll.dll!RtlLeaveCriticalSection(0x006fc758, 0x0018fe50)
```

Dr. Ltrace Implementation

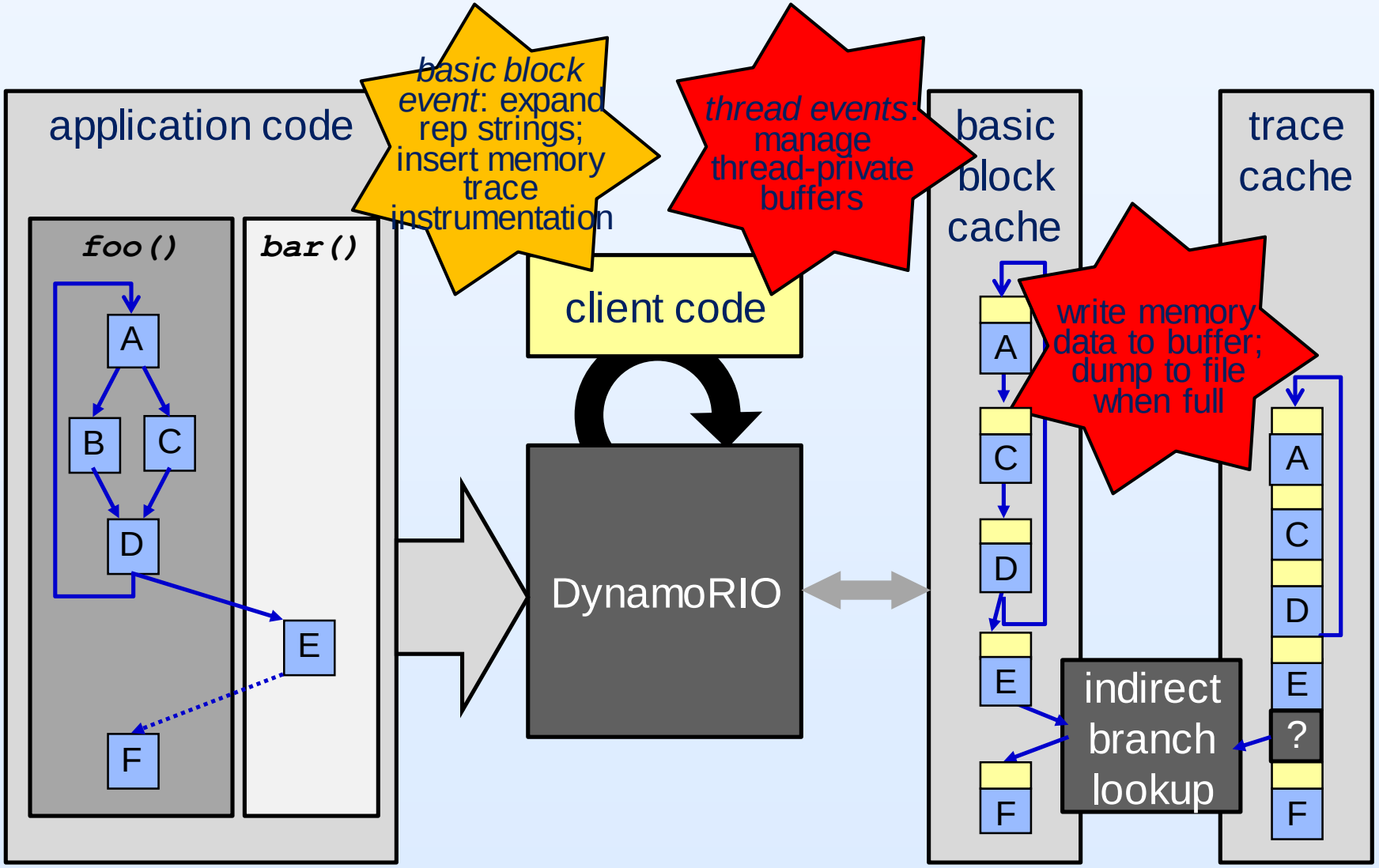


Memory Trace Tool

- Instruments every load and store
- Writes the trace of memory accesses to a file
- Usage:
  - **drrun -c samples/bin64/libmemtrace_x86_text.so -- <app>**
- Results:

```
Format: <instr address>,<(r)ead/(w)rite>,<data size>,<data address>
0x00007fb31a5462d3,w,8,0x00007ffc4febfb18
0x00007fb31a549a8f,r,8,0x00007fb31a767e70
0x00007fb3199a54fc,r,4,0x00007ffc4febf438
0x00007fb3199a5526,r,1,0x0000000000400e98
0x00007fb31a549a96,w,8,0x00007fb31a767c98
0x00007fb3199af4a0,r,8,0x00007ffc4febf9f8
0x0000000000400db4,r,8,0x00007ffc4febfa48
```

Memory Trace Tool Implementation

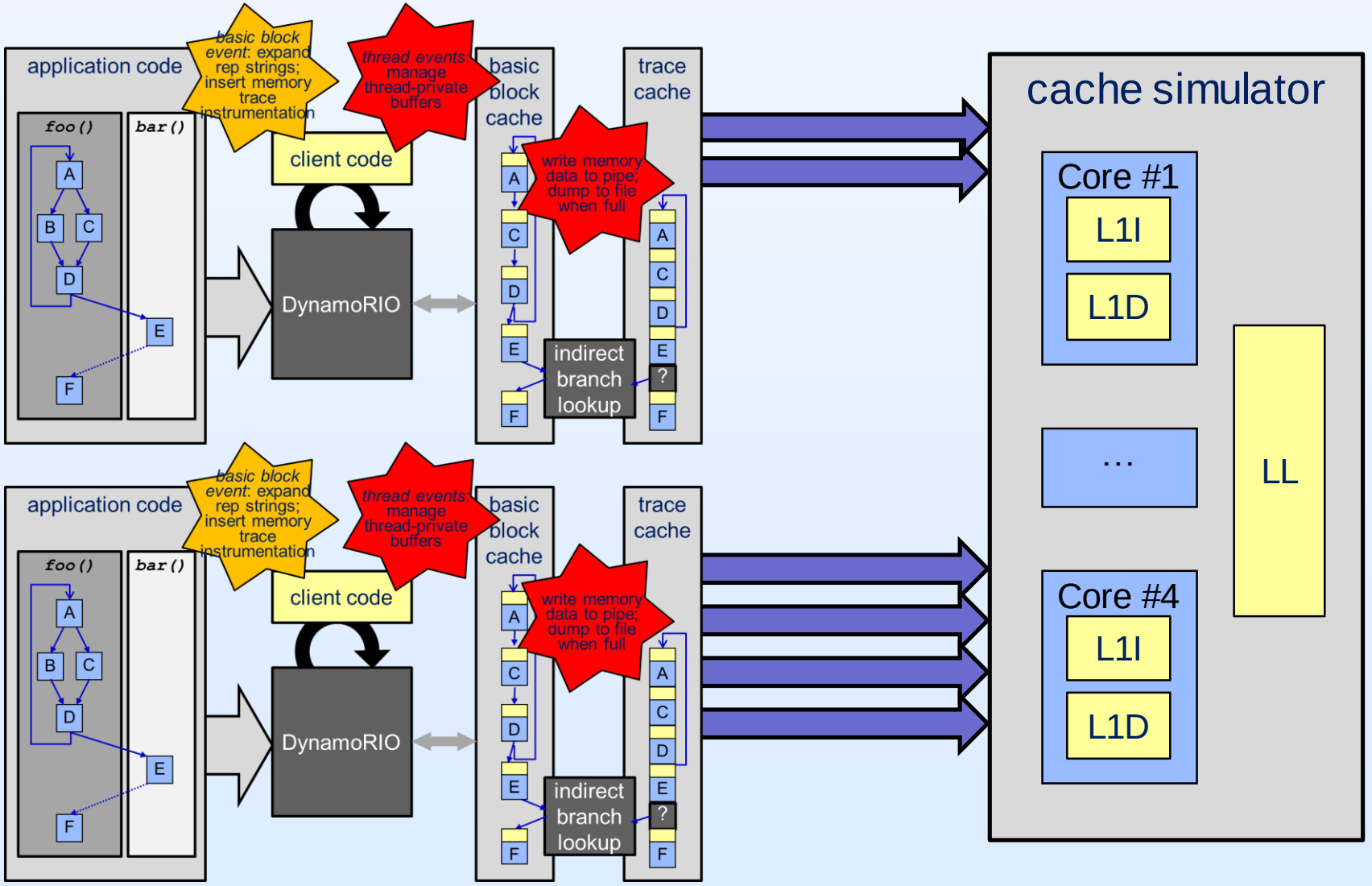


Dr. CacheSim

- Multi-process, multi-thread online cache simulator
- Each thread collects and sends its own memory access trace to a single simulator
- Usage:
  - **`drrun -t drcachesim -- <app>`**
- Results:

```
% bin64/drrun -t drcachesim -- ./multi_threaded_pi_estimator
Estimation of pi is 3.142425985001098
---- <application exited with code 0> ----
Core #0 (1 thread(s))
  L1I stats:
    Hits:                          243,293
    Misses:                          1,077
    Miss rate:                        0.44%
  L1D stats:
    Hits:                           67,738
    Misses:                          3,471
    Miss rate:                        4.87%
Core #1 (1 thread(s))
  L1I stats:
    Hits:                            8,589
    ...
```

Dr. CacheSim Implementation

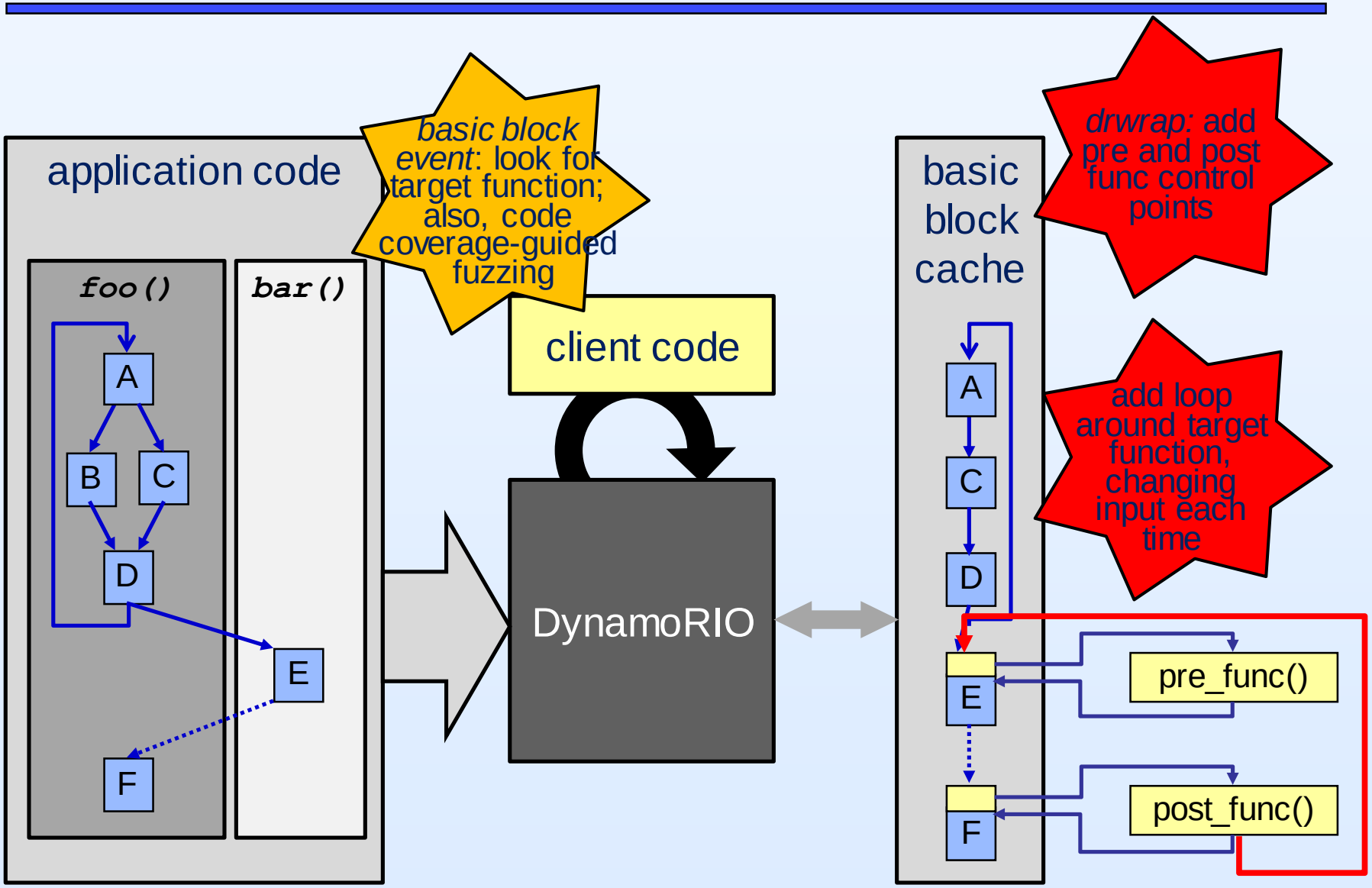


Dr. Fuzz

- In-process function-level fuzzing
- Repeatedly runs a target function, varying the inputs
- Basic usage:
  - **drrun -t drmemory -fuzz_function <func> -- <app>**
- Results:

```
% drrun -t drmemory -batch -fuzz_function some_function -fuzz_num_iters 10 -- ./fuzzme.exe
~~Dr.M~~ Dr. Memory version 1.11.0
~~Dr.M~~ Running "./fuzzme.exe"
in some_function
hello!
in some_function
~~Dr.M~~
~~Dr.M~~ Error #1: UNADDRESSABLE ACCESS beyond heap bounds: reading 0x00f58efc-0x00f58f00 4 byte(s)
~~Dr.M~~ # 0 some_function              [c:\derek\cgo2017\demos\fuzzme.c:8]
~~Dr.M~~ # 1 main                      [c:\derek\cgo2017\demos\fuzzme.c:18]
in some_function
in some_function
hello!
...
```

Dr. Fuzz Implementation



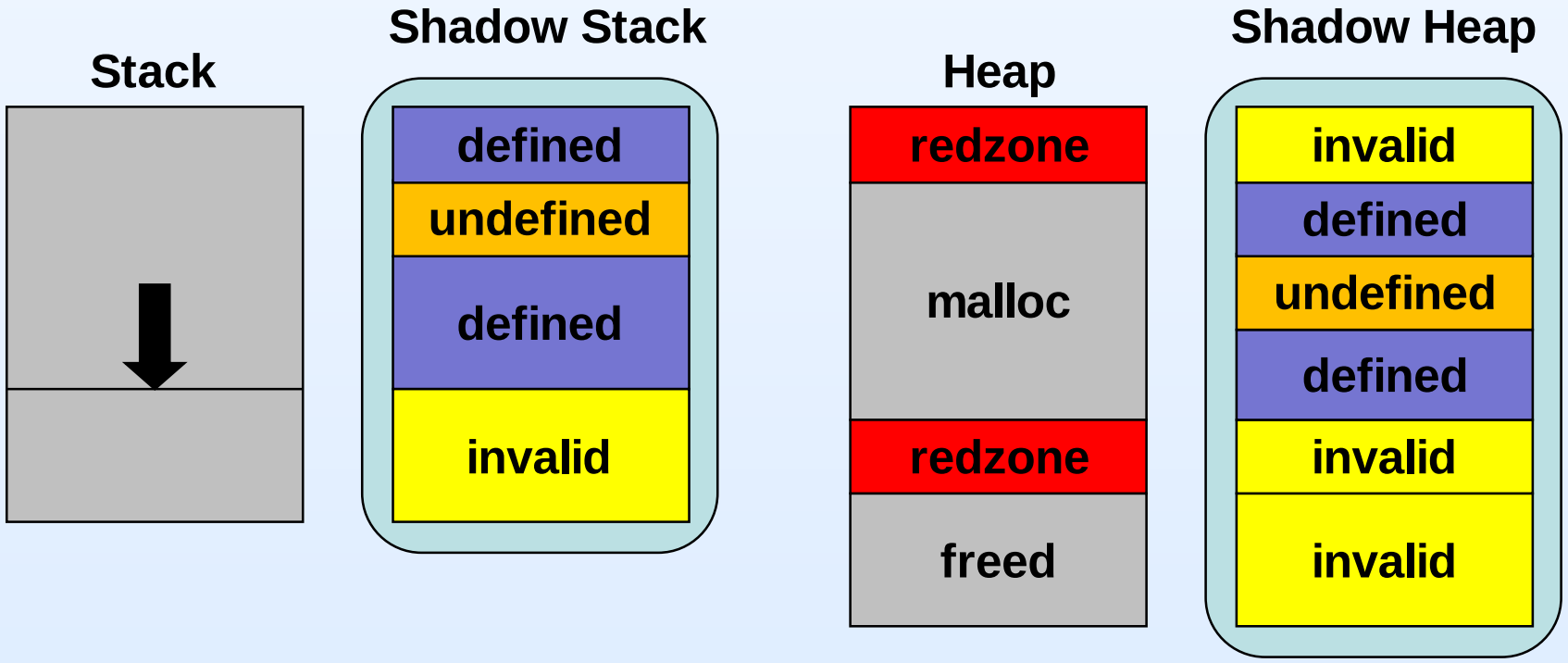
Dr. Memory: Memory Debugger

- Detects reads of uninitialized memory
- Detects heap errors
  - Out-of-bounds accesses (underflow, overflow)
  - Access to freed memory
  - Invalid frees
  - Memory leaks
- Detects other accesses to invalid memory
  - Stack tracking
- Detects Windows handle leaks
- Operates at runtime on unmodified Windows, Linux, MacOS, and Android binaries

Dr. Memory Instrumentation

- Monitor all memory accesses, stack adjustments, and heap allocations
- Shadow each byte of app memory
- Each byte's shadow stores one of 3 values:
  - Unaddressable
  - Uninitialized
  - Defined

Dr. Memory Shadow Memory



Monitoring Stack Changes

- As stack is extended and contracts again, must update stack shadow as unaddressable vs uninitialized
- Push, pop, or any write to stack pointer
- Try to distinguish large alloc/dealloc from stack swap

Partial-Word Defines But Whole-Word Transfers

- Sub-dword variables are moved around as whole dwords
- Cannot raise error when a move reads uninitialized bits
- Must propagate on moves and thus must shadow registers
  - Propagate shadow values by mirroring app data flow
- Check system call reads and propagate system call writes
  - Else, false negatives (reads) or positives (writes)
- Raise errors instead of propagating at certain points
  - Report errors only on “significant” reads

Shadowing Registers

- Use multiple TLS slots
  - dr_raw_tls_calloc()
  - Alternative: steal register
- Can read and write w/o spilling
- Bring into spilled register to combine w/ other args
  - Defined=0, uninitialized=1
  - Combine via bitwise or

Types Of Instrumentation

- Clean call
  - Simplest, but expensive in both time and space: full context switch from application state to tool state with separate stack to execute C code
- Shared clean call (now auto-provided by DR)
  - Saves space
- Lean procedure
  - Shared routine with smaller context switch than full clean call
  - Jump-and-link rather than swapping stack
  - Array of routines, one per pair of dead registers
- Inlined
  - Smallest context switch, but should limit to small sequences of instrumentation

Non-Code-Cache Code

- Use dr_nonheap_alloc() to allocate space to store code
- Generate code using DR's IR and emit to target space
- Mark read-only once emitted via dr_memory_protect()

Jump-and-Link

- Rather than using call+return, avoid stack swap cost by using jump-and-link
  - Store return address in a register or TLS slot
  - Direct jump to target
  - Indirect jump back to source

```
PRE(bb, inst, INSTR_CREATE_mov_st(drcontext,
    spill_slot_opnd(drcontext, SPILL_SLOT_2),
    opnd_create_instr(appinst)));
PRE(bb, inst, INSTR_CREATE_jmp(drcontext,
    opnd_create_pc(shared_slowpath_region)));
...
PRE(ilist, NULL, INSTR_CREATE_jmp_ind(drcontext,
    spill_slot_opnd(SPILL_SLOT_2)));
```

Using Faults For Faster Common Case Code

- Instead of explicitly checking for rare cases, use faults to handle them and keep common case code path fast
- Signal and exception event and restore state extended event all provide pre- and post-translation contexts and containing fragment information
- Client can return failure for extended restore state event
  - When can support re-execution of faulting cache instr, but not re-start translation for relocation

Suspending The World

- Use case: Dr. Memory leak check
  - GC-like memory scan
- Use dr_suspend_all_other_threads() and dr_resume_all_other_threads()
- Cannot hold locks while suspending

Tool Packaging

- DynamoRIO is redistributable, so you can include a copy with your tool
- drrun supports the –t option via a tool configuration file
  - drrun –t drcov -- <app cmdline>
- Custom front end to configure and launch app
  - We provide several libraries for building tool front ends:
    - drconfiglib
    - drinjectlib
    - drfrontendlib

Q & A / Feedback

8:30- 8:40	Welcome + DynamoRIO History
8:40- 9:10	Tool Demonstrations
9:10- 9:30	DynamoRIO System Overview
9:30- 9:45	DynamoRIO API Part 1
9:45-10:00	ARM Status + Demonstrations
10:00-10:30	*Break*
10:30-10:55	DynamoRIO API Part 2
10:55-11:20	How to Create Your Own Tool
11:20-11:40	Tool Internals
11:40-11:45	Q & A

Seeking Contributors

- Many potential projects
  - Contribute new tools
  - Finish feature to attach to a running process
  - Debugger integration
    - Transparent debugging and debugger extensions
  - ARM performance
    - Optimizing assembly code and generated code
  - Android
    - AArch64 support
    - Support running larger apps
  - MacOS
    - Private loader and 64-bit support need owners
  - Search “GoodContrib” label in issue tracker

Optional Slides: Versus Pin

DynamoRIO versus Pin

- DynamoRIO is open-source while Pin is closed
  - If something goes wrong, can look under the hood
  - Can adapt to particular usage scenario
    - E.g., DynamoRIO as static library linked into application
- DynamoRIO supports both Intel (x86, AMD64) and ARM (ARM, AArch64) platforms while Pin supports only Intel

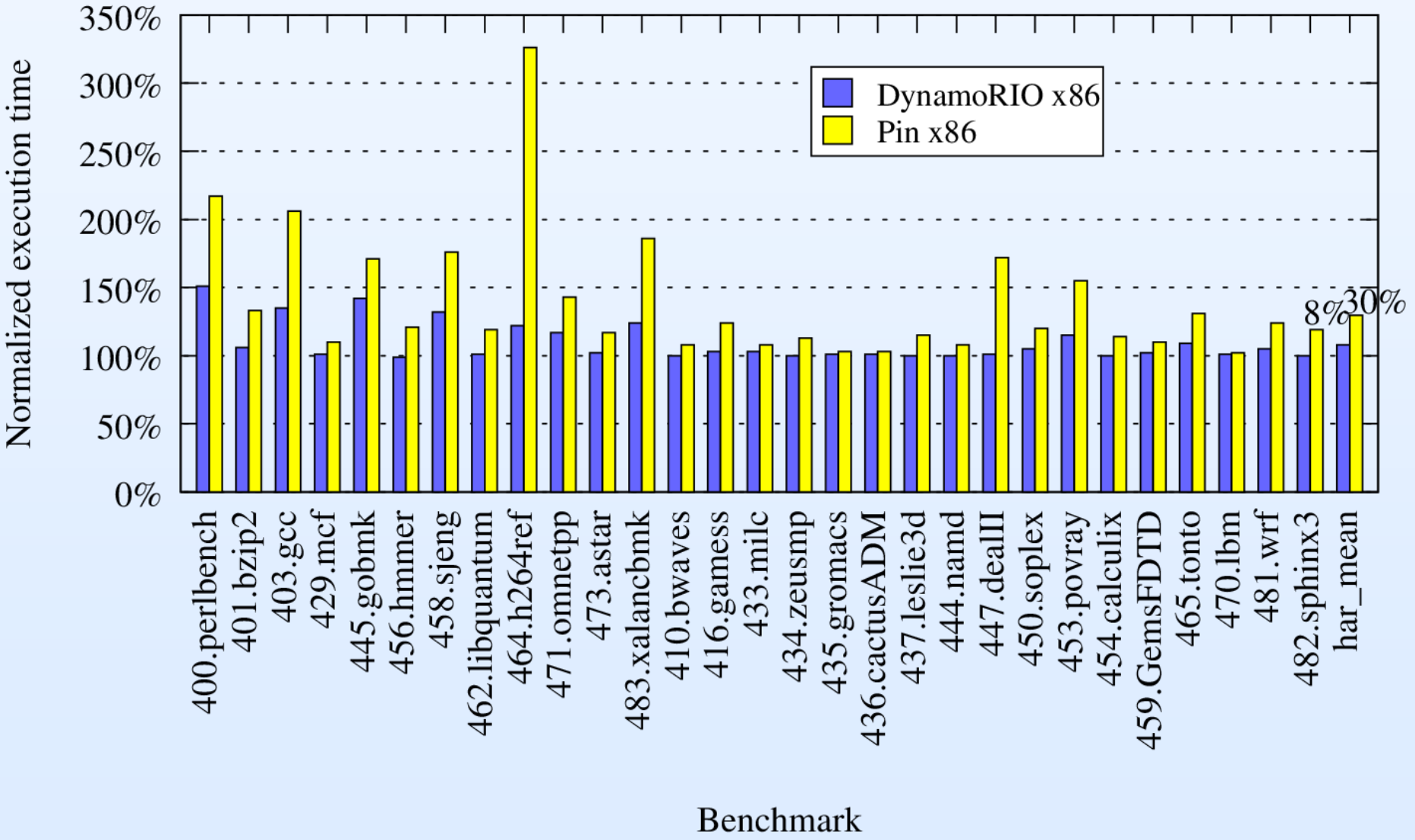
DynamoRIO versus Pin: Interface

- 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
- DynamoRIO = arbitrary code stream modifications
  - Only feasible with a code cache
  - Takes full advantage of power of code cache
  - General IA-32/AMD64/ARM/AArch64 decode/encode/IR support

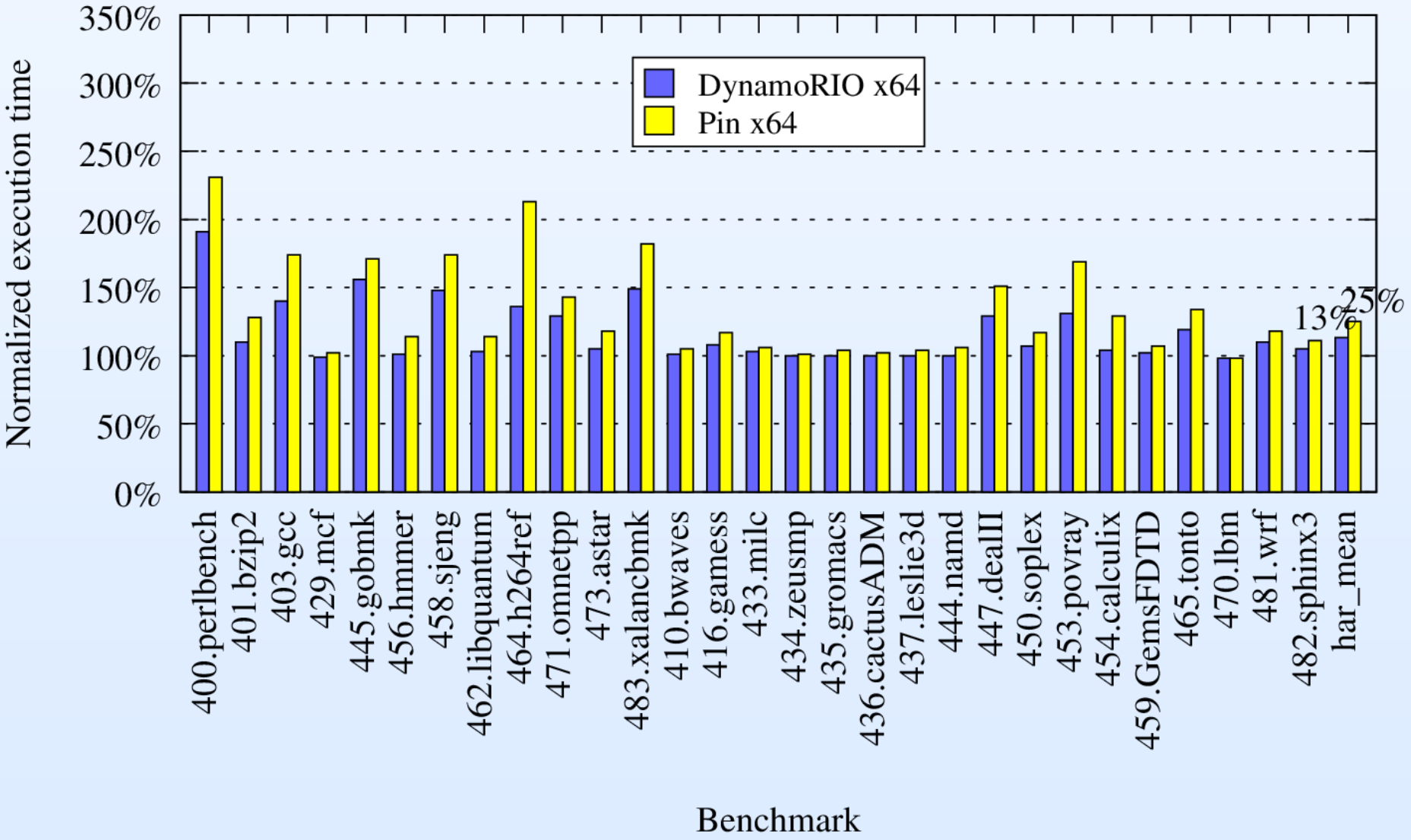
DynamoRIO versus Pin: Consequences

- 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

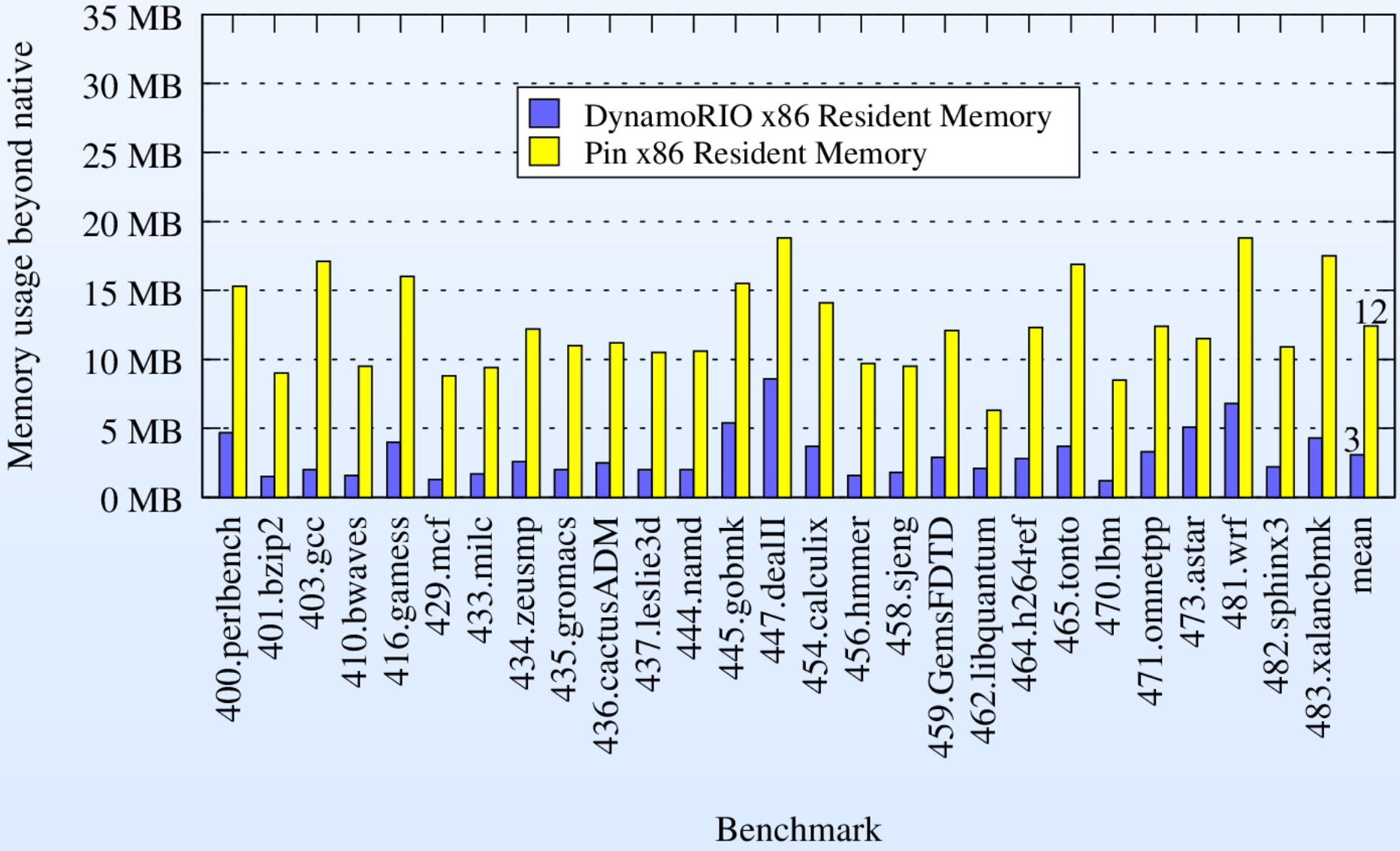
Base Performance Comparison (No Tool)



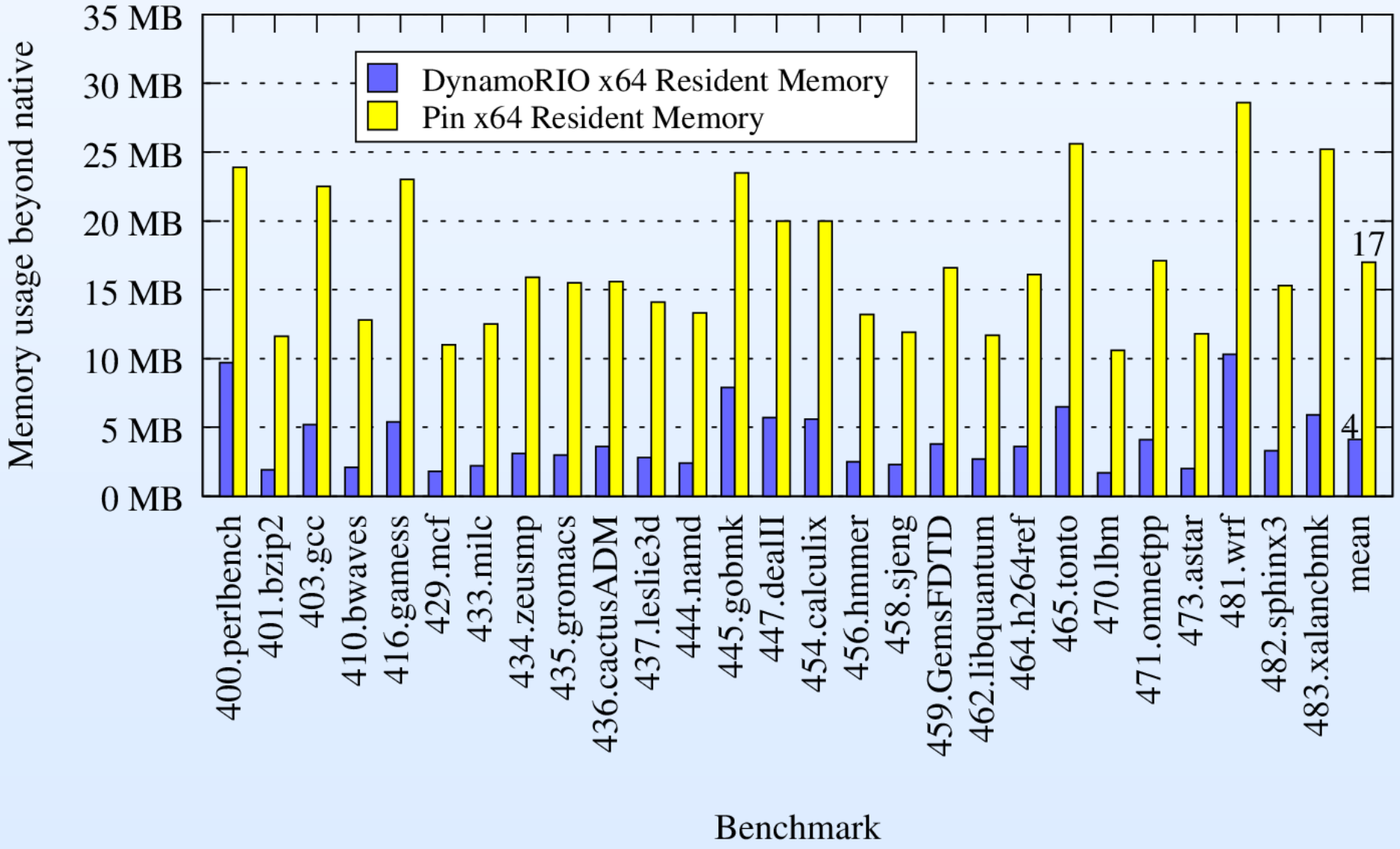
Base Performance Comparison (No Tool)



Base Memory Comparison (No Tool)



Base Memory Comparison (No Tool)



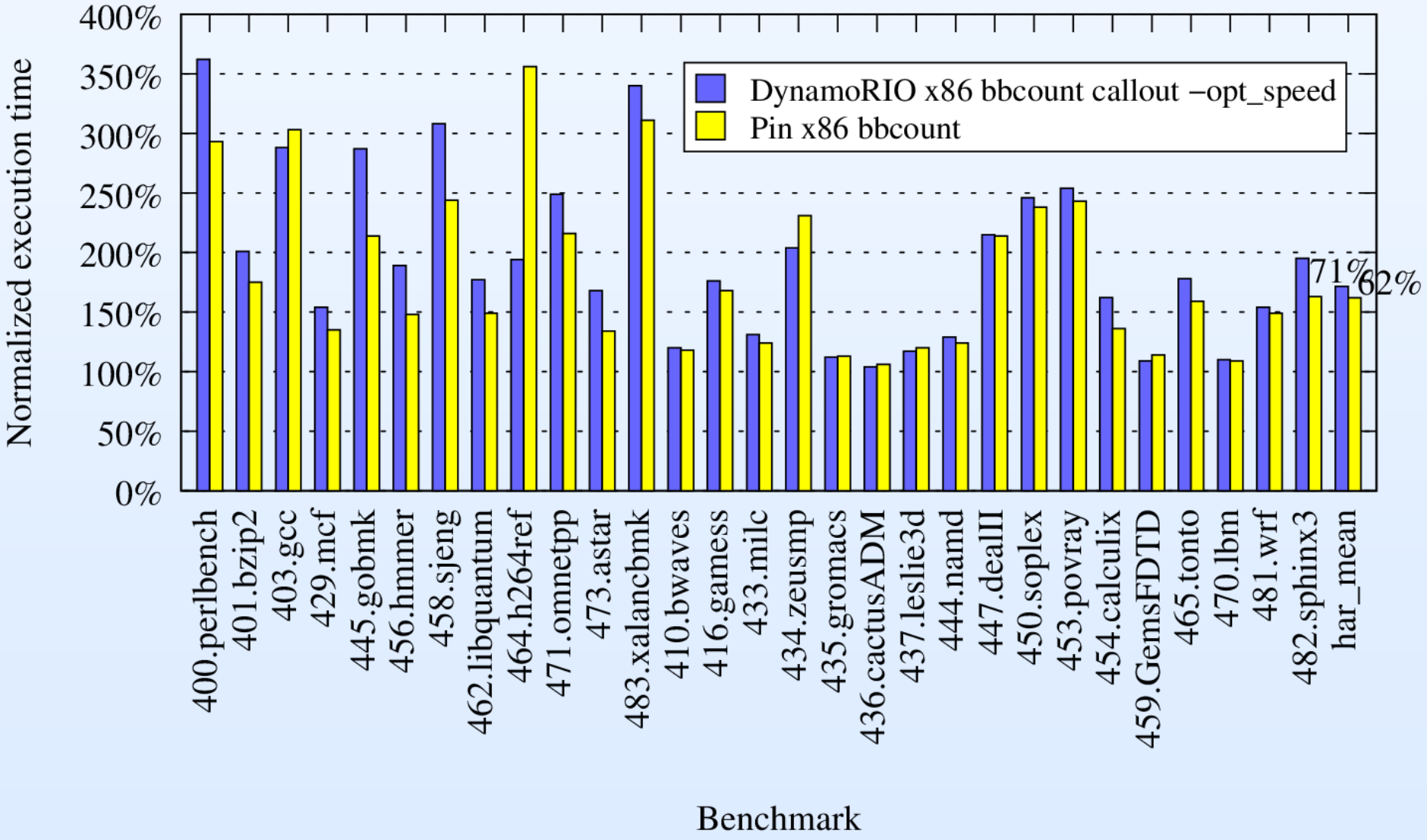
BBCount Pin Tool

```
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;
}
```

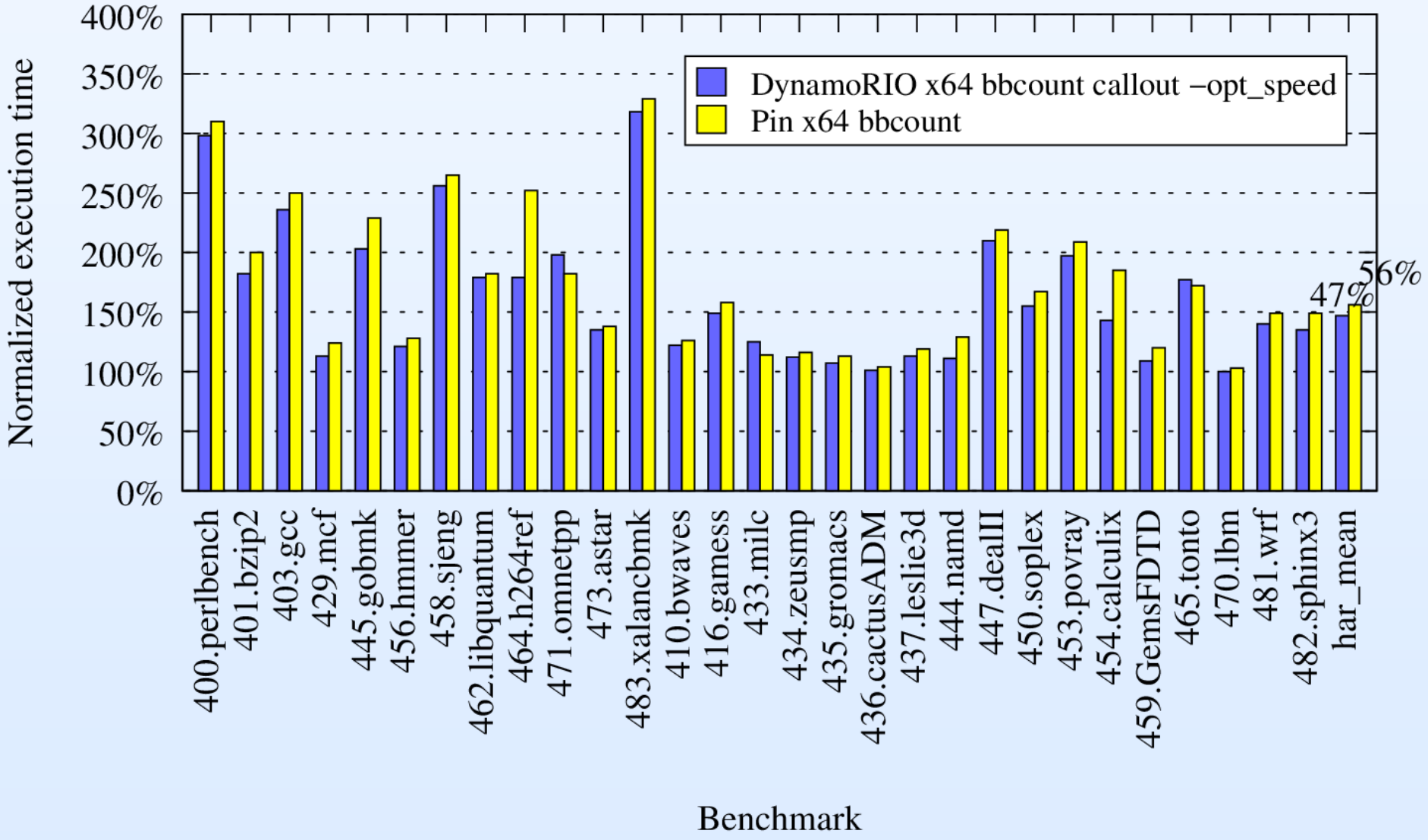
Simple BBCount DynamoRIO Tool

```
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



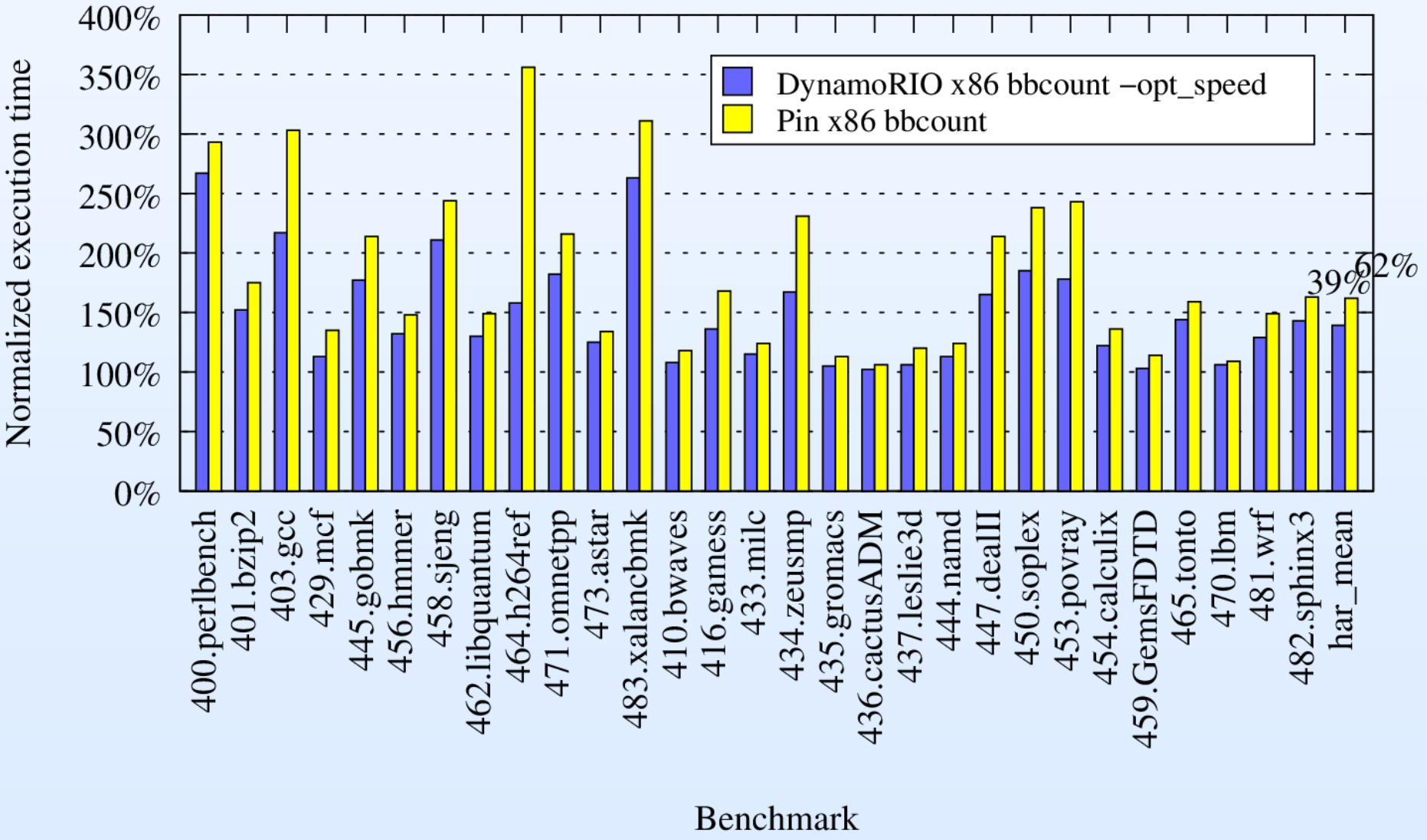
Optimized BBCount DR Tool: Raw Version

```
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);
}
```

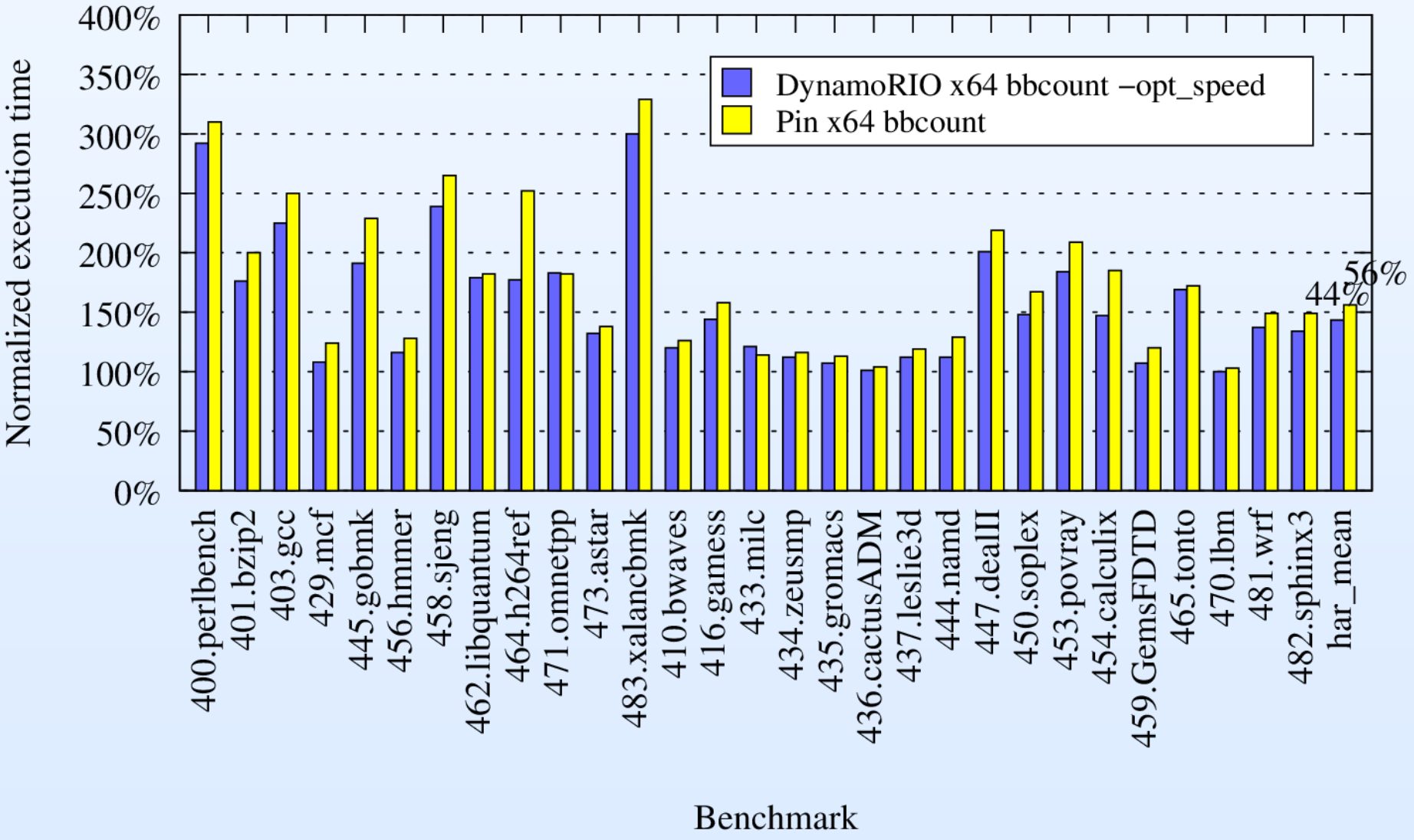
Optimized BBCount DR Tool: Leverage API

```
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 *first = instrlist_first(bb);
    drx_insert_counter_update(drcontext, bb, first, SPILL_SLOT_MAX+1,
                              (void *)&global_count, 1, 0);
    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

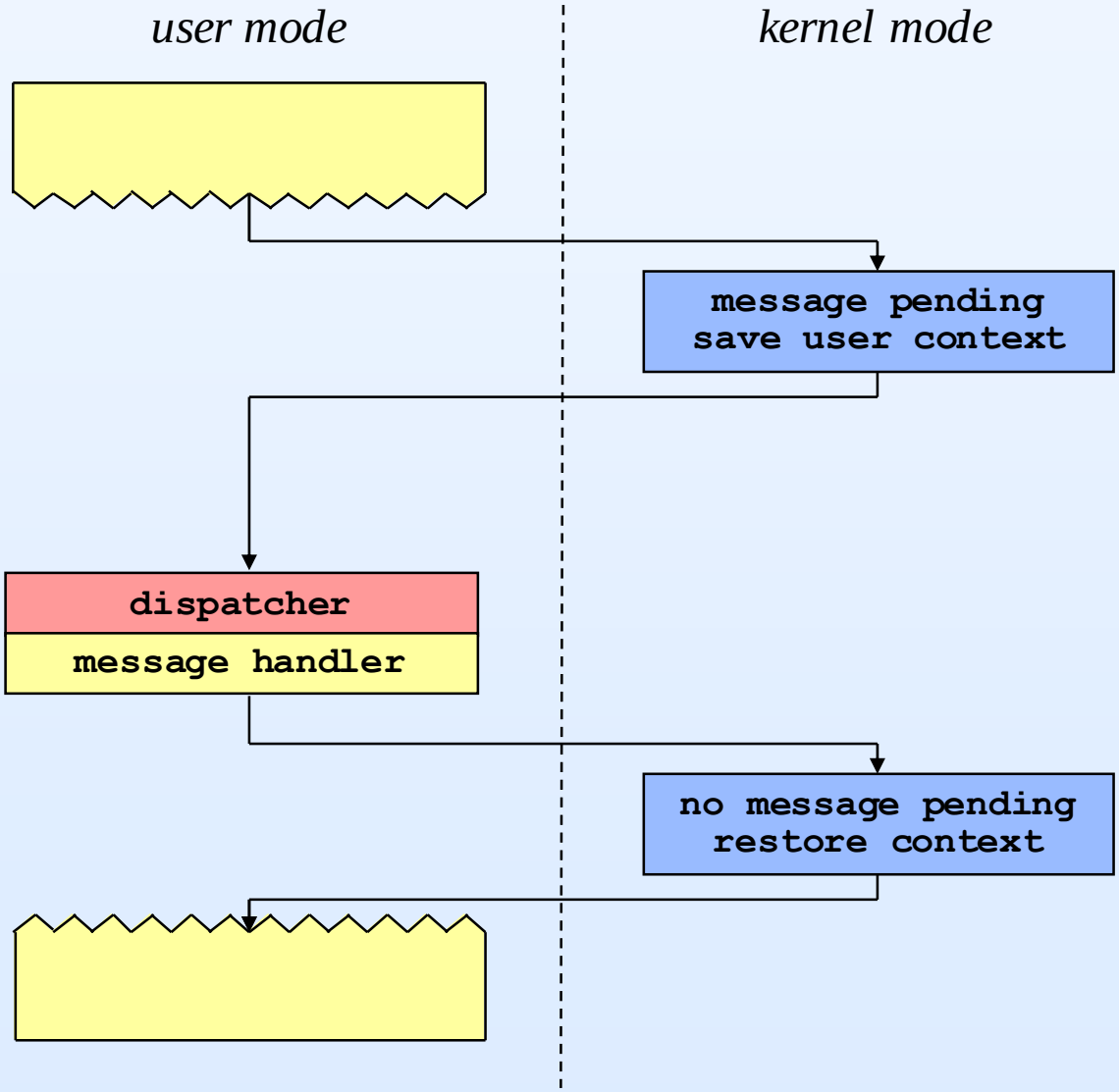


BBCount Performance Comparison: Opt Tool



Optional Slides: Advanced API

Callback-Local Storage (CLS)



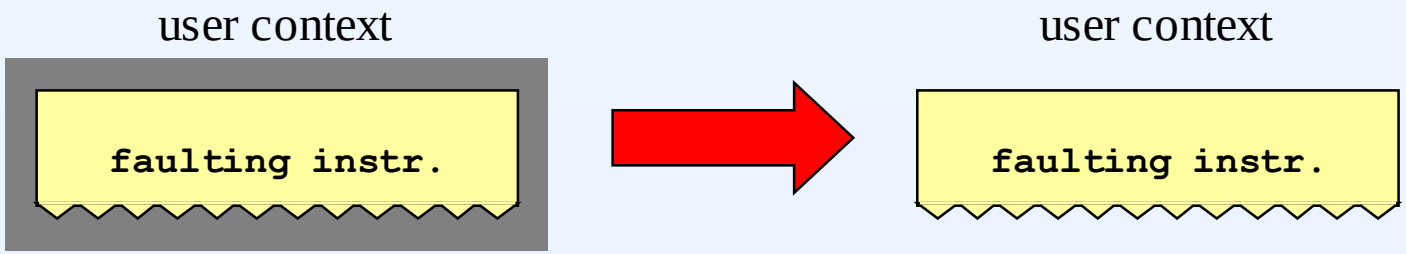
Callback-Local Storage (CLS)

- 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: 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

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

Optional Slides: ARM-Specific

IT Blocks

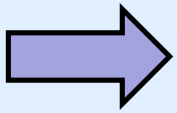
- Thumb allows predication only inside IT blocks, where an IT instruction specifies the predicate for 1-4 subsequent instrs
- Arithmetic instrs have different semantics inside IT vs outside
  - Outside they write condition codes while inside they do not
- Example:

```
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

Optional Slides: Dr. Memory

Inter-Instruction Storage

- Spill slots provided by DR are only guaranteed to be live during a single app instr
  - In practice, live until next selfmod instr
- Allocate own TLS for spill slots
  - dr_raw_tls_calloc()
  - *drreg* does this for you
- Steal registers across whole bb
  - Restore before each app read
  - Update spill slot after each app write
  - Restore on fault
  - *drreg* does all of this for you too

Address Space Iteration

- Repeated calls to dr_query_memory_ex()
- Check dr_memory_is_in_client() and dr_memory_is_dr_internal()
- Heap walk
  - API on Windows
- Initial structures on Windows
  - TEB, TLS, etc.
  - PEB, ProcessParameters, etc.

Delayed Fragment Deletion

- Due to non-precise flushing we can have a flushed bb made inaccessible but not actually freed for some time
- When keeping state per bb, if a duplicate bb is seen, replace the state and increment a counter ignore_next_delete
- On a deletion event, decrement and ignore unless below 0
- Can't tell apart from duplication due to thread-private copies: but this mechanism handles that if saved info is deterministic and identical for each copy

Callstack Walking

- Use case: error reporting
- Technique:
  - Start with xbp as frame pointr (fp)
  - Look for <fp,retaddr> pairs where retaddr = inside a module
- Interesting issues:
  - When scanning for frame pointer (in frameless func, or at bottom of stack), querying whether in a module dominates performance
  - msvcr80!malloc pushes ebx and then ebp, requiring special handling
  - When displaying, use retaddr-1 for symbol lookup
  - More sophisticated techniques needed in presence of FPO

Callstack Walking

- Record on every allocation so must be very fast
  - Cannot afford to do symbol lookup, so just check whether a potential return address is inside a module
- Problem: stale return addresses
- Solutions:
  - For full mode, use definedness info
  - For other modes, zero out the return address upon returning from a function

Using Nudges

- Daemon apps do not exit
- Request results mid-run
- Cross-platform
  - Signal on Linux
  - Remote thread on Windows