

# Building Dynamic Instrumentation Tools with DynamoRIO

Derek Bruening  
Qin Zhao

# Tutorial Outline

---

- 2:00-2:10 Welcome + DynamoRIO History
- 2:10-3:10 DynamoRIO Internals
- 3:10-4:00 DynamoRIO API
- 4:00-4:15 *Break*
- 4:15-5:15 Demonstrations
- 5:15-5:45 Lab Session
- 5:45-6:00 Feedback

# DynamoRIO History

---

- Dynamo
  - HP Labs: PA-RISC late 1990's
  - x86 Dynamo: 2000
- RIO → DynamoRIO
  - MIT: 2001-2004
- Prior releases
  - 0.9.1: Jun 2002 (PLDI tutorial)
  - 0.9.2: Oct 2002 (ASPLOS tutorial)
  - 0.9.3: Mar 2003 (CGO tutorial)
  - 0.9.4: Feb 2005
  - 0.9.5: Apr 2008 (CGO tutorial)
  - 1.0 (0.9.6): Sep 2008 (GoVirtual.org launch)

# DynamoRIO History

---

- Determina
  - 2003-2007
  - Security company
- VMware
  - Acquired Determina (and DynamoRIO) in 2007
- Open-source BSD license
  - Feb 2009: 1.3.1 release
  - Dec 2009: 1.5.0 release
  - Apr 2010: 2.0.0 release
  - Apr 2011: 2.2.0 release

# DynamoRIO Internals

2:00-2:10 Welcome + DynamoRIO History

2:10-3:10 DynamoRIO Internals

3:10-4:00 DynamoRIO API

4:00-4:15 *Break*

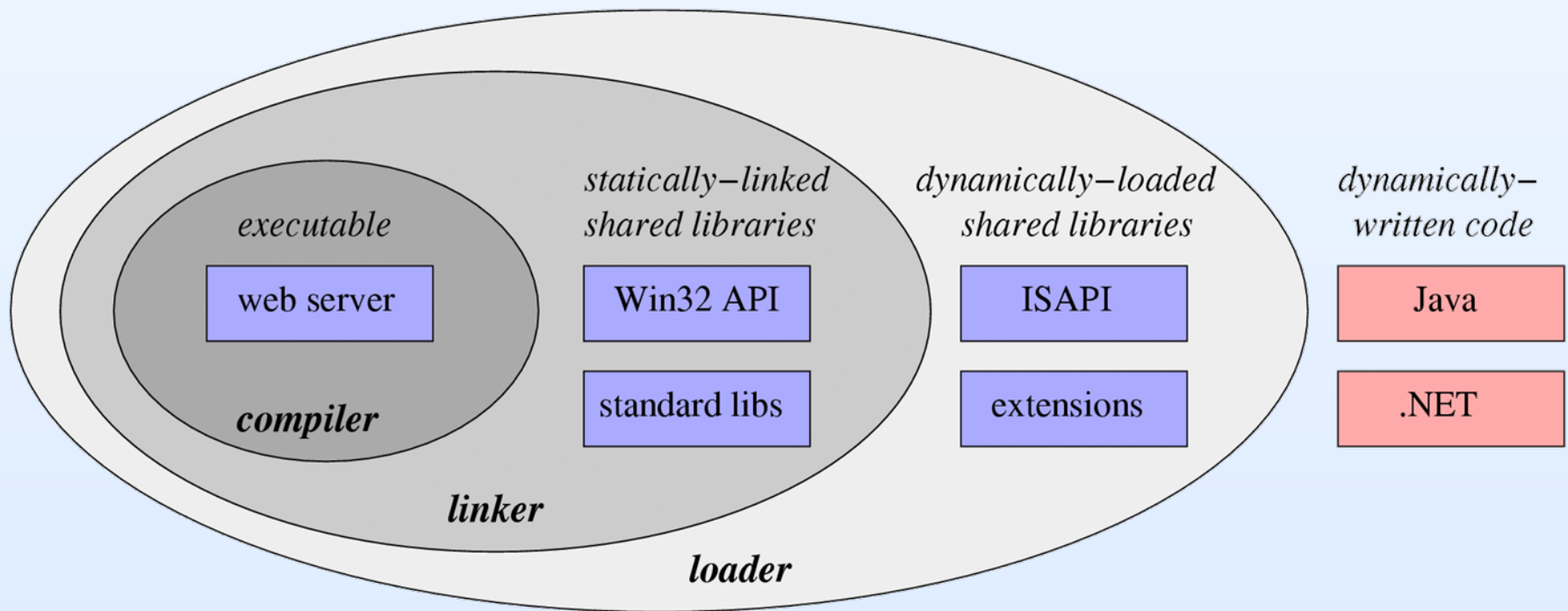
4:15-5:15 Demonstrations

5:15-5:45 Lab Session

5:45-6:00 Feedback

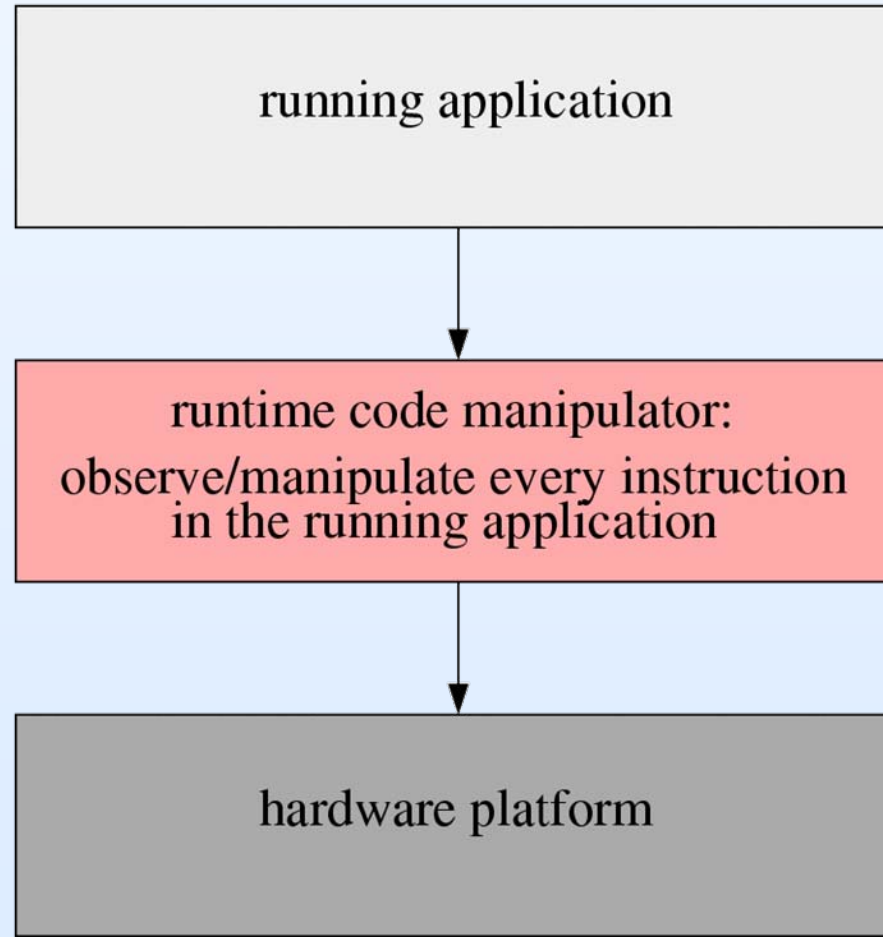
# Typical Modern Application: IIS

---



# Runtime Interposition Layer

---



# 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

# Internals Outline

---


- Efficient
  - Software code cache overview
  - Thread-shared code cache
  - Cache capacity limits
  - Data structures
- Transparent
- Comprehensive
- Customizable

# Direct Code Modification

---

```
e9 37 6f 48 92    jmp <callout>
```

Kernel32!TerminateProcess:



```
7d4d1028 7c 05          jl     7d4d102f
7d4d102a 33 c0          xor    %eax,%eax
7d4d102c 40            inc    %eax
7d4d102d eb 08          jmp    7d4d1037
7d4d102f 50            push   %eax
7d4d1030 e8 ed 7c 00 00 call  7d4d8d22
```

# Debugger Trap Too Expensive

---

cc int3 (breakpoint)

Kernel32!TerminateProcess:

7d4d1028	7c 05	j1	7d4d102f
7d4d102a	33 c0	xor	%eax,%eax
7d4d102c	40	inc	%eax
7d4d102d	eb 08	jmp	7d4d1037
7d4d102f	50	push	%eax
7d4d1030	e8 ed 7c 00 00	call	7d4d8d22

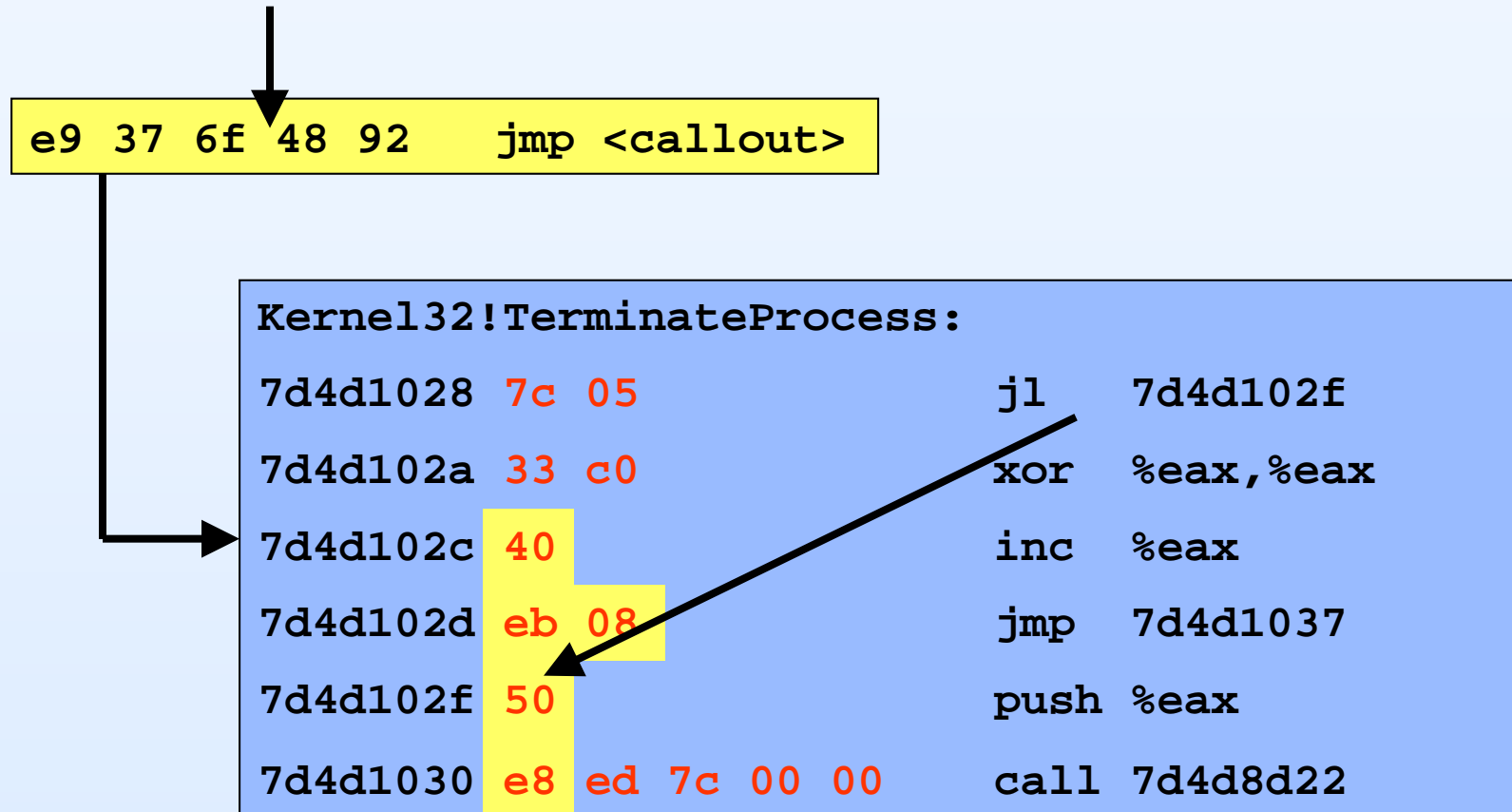
# Variable-Length Instruction Complications

**e9 37 6f 48 92    jmp <callout>**

**Kernel32!TerminateProcess:**

7d4d1028	<b>7c 05</b>	j1	7d4d102f
7d4d102a	<b>33 c0</b>	xor	%eax,%eax
7d4d102c	<b>40</b>	inc	%eax
7d4d102d	<b>eb 08</b>	jmp	7d4d1037
7d4d102f	<b>50</b>	push	%eax
7d4d1030	<b>e8 ed 7c 00 00</b>	call	7d4d8d22

# Entry Point Complications



# Direct Code Modification: Too Limited

---

- Not transparent
  - Cannot write jump atomically if crosses cache line
  - Even if write is atomic, not safe if overwrites part of next instruction
  - Jump may span code entry point
- Too limited
  - Not safe w/o suspending all threads and knowing all entry points
  - Limited to inserting callouts
- Code displaced by jump is a mini code cache
  - All the same consistency challenges of larger cache
- Inter-operation issues with other hooks

# We Need Indirection

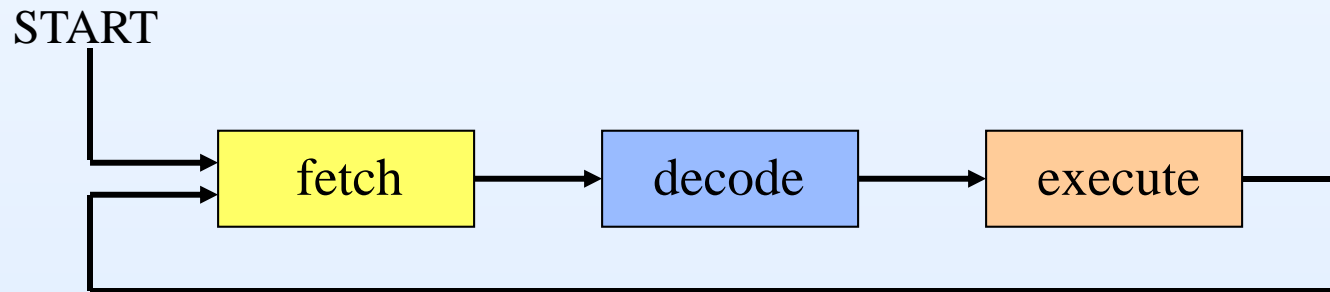
---

- 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

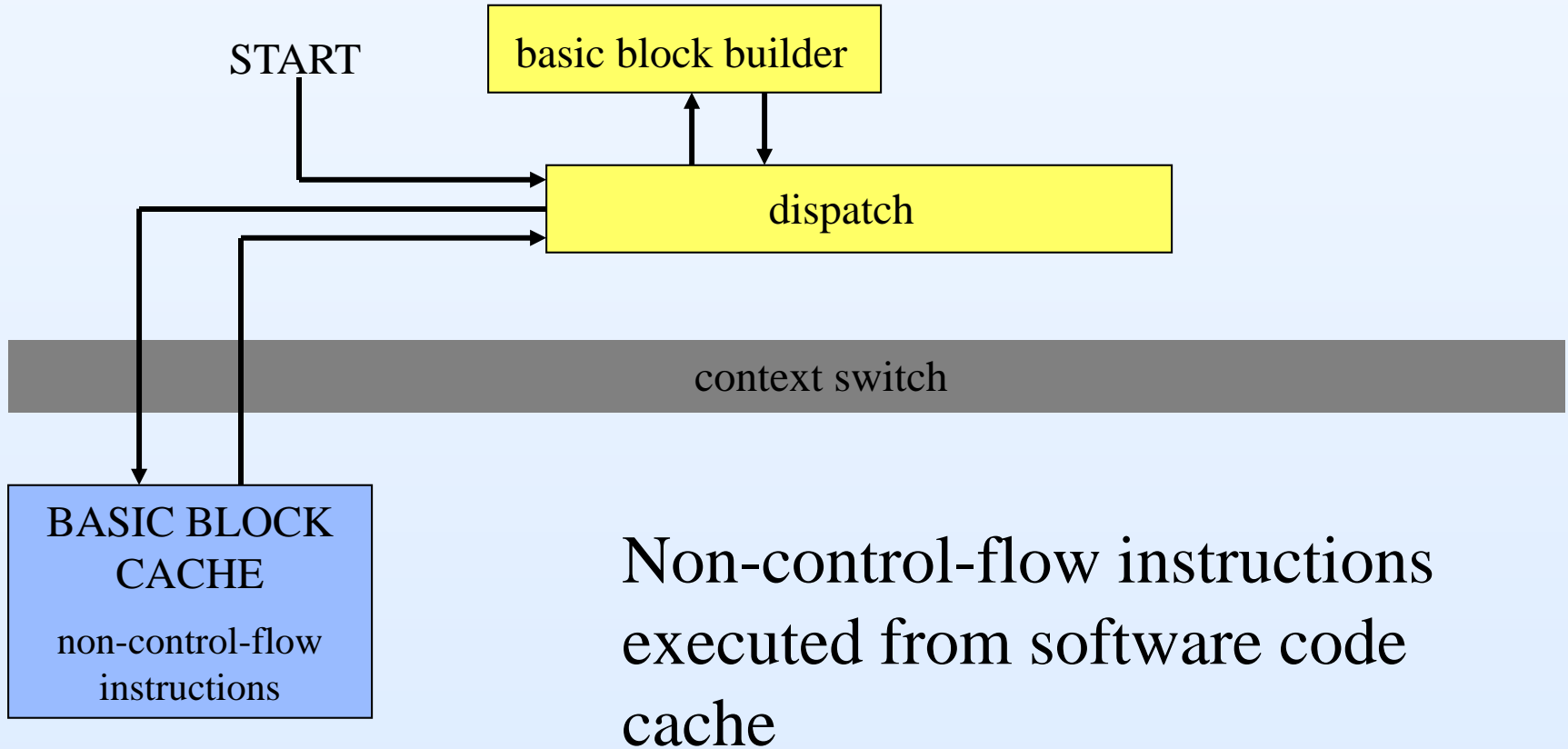
---



**Slowdown: ~300x**

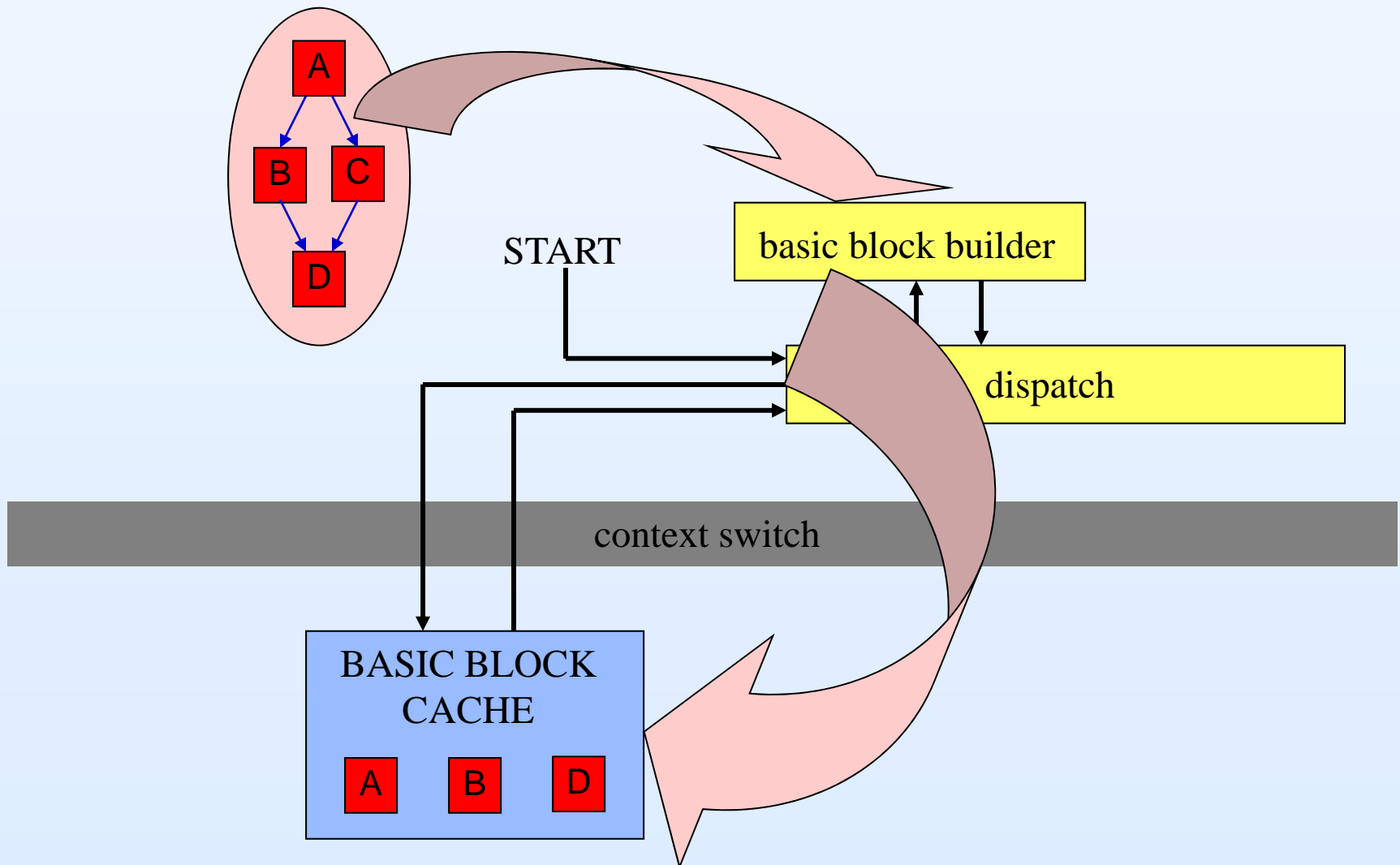
# Improvement #1: Interpreter + Basic Block Cache

---



Slowdown: ~~300x~~ 25x

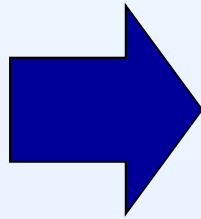
# Improvement #1: Interpreter + Basic Block Cache



# Example Basic Block Fragment

---

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

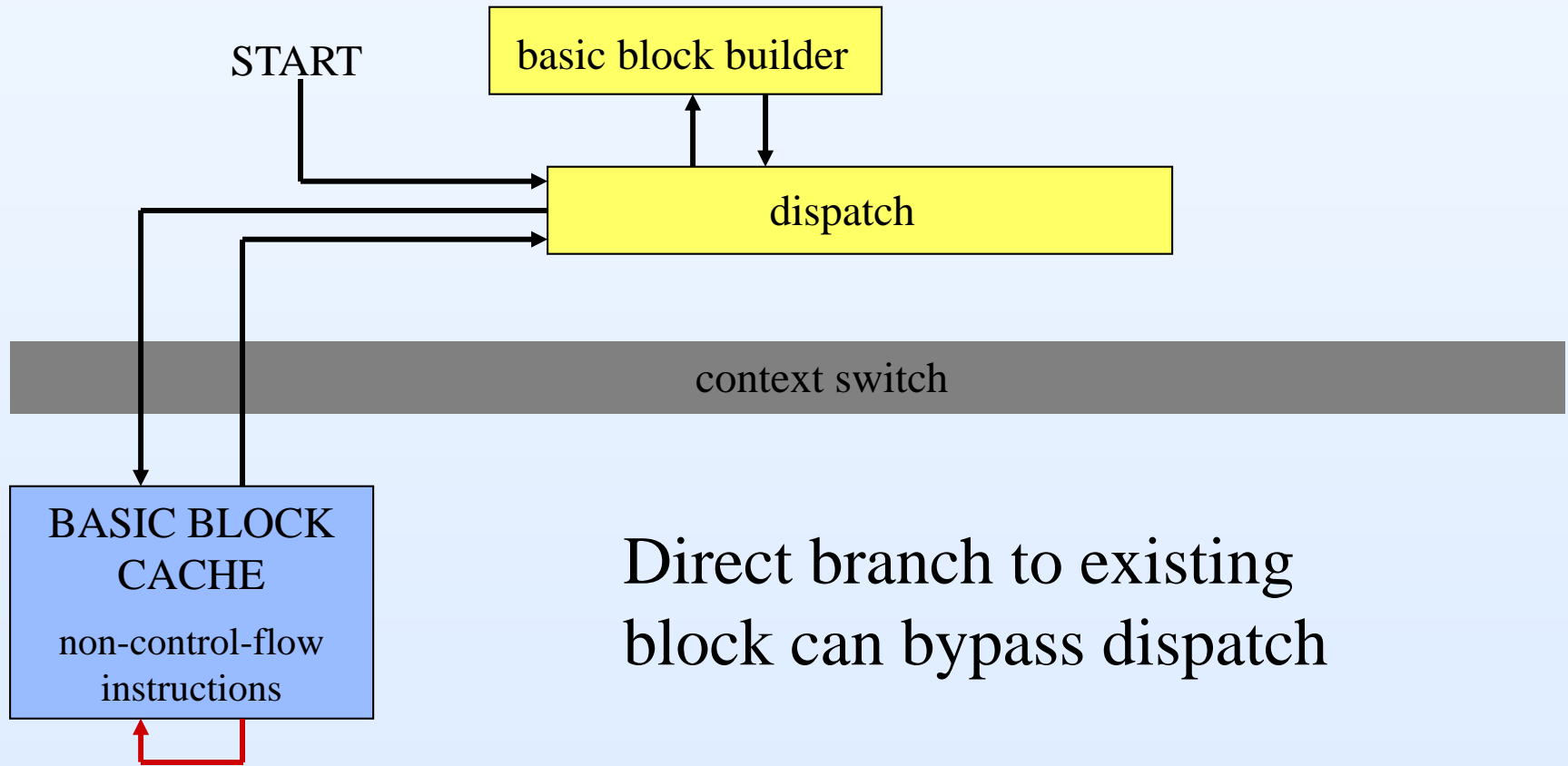


```
frag7: add    %eax, %ecx  
        cmp     $4, %eax  
        jle     <stub0>  
        jmp     <stub1>  
stub0: mov     %eax, eax-slot  
        mov     &dstub0, %eax  
        jmp     context_switch  
stub1: mov     %eax, eax-slot  
        mov     &dstub1, %eax  
        jmp     context_switch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

dstub0  
target: 0x40106f

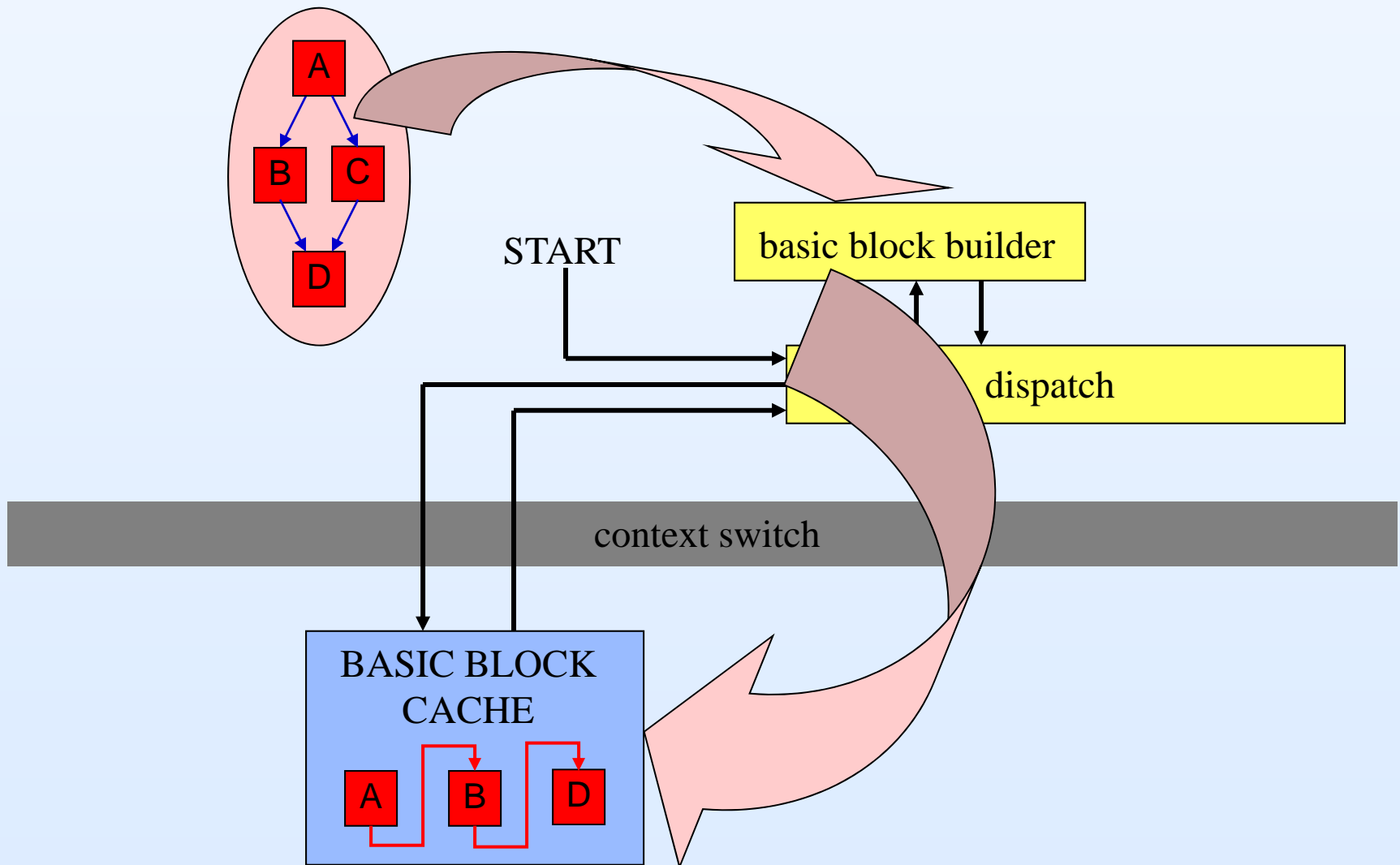
dstub1  
target: fall-thru

# Improvement #2: Linking Direct Branches



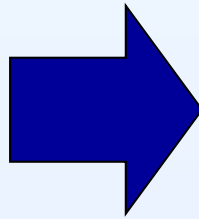
**Slowdown: ~~300x~~ ~~25x~~ 3x**

# Improvement #2: Linking Direct Branches



# Direct Linking

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

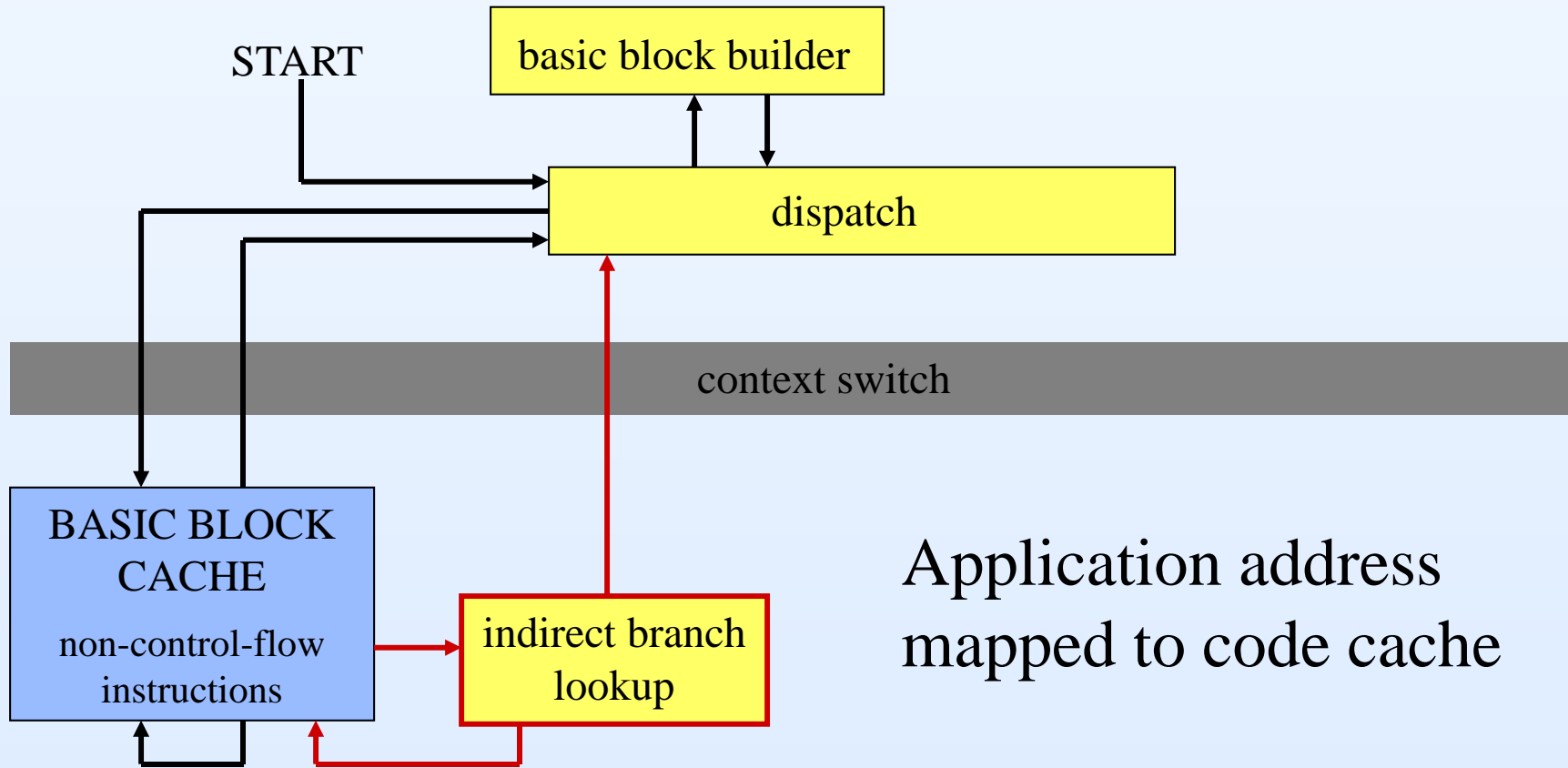


```
frag7: add    %eax, %ecx  
        cmp     $4, %eax  
        jle     <frag8>  
        jmp     <stub1>  
stub0: mov     %eax, eax-slot  
        mov     &dstub0, %eax  
        jmp     context_switch  
stub1: mov     %eax, eax-slot  
        mov     &dstub1, %eax  
        jmp     context_switch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

dstub0  
target: 0x40106f

dstub1  
target: fall-thru

# Improvement #3: Linking Indirect Branches



Application address  
mapped to code cache

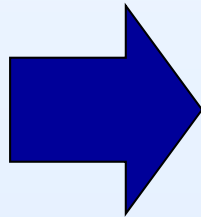
Slowdown: ~~300x~~ ~~25x~~ ~~3x~~ 1.2x



# Indirect Branch Transformation

---

ret

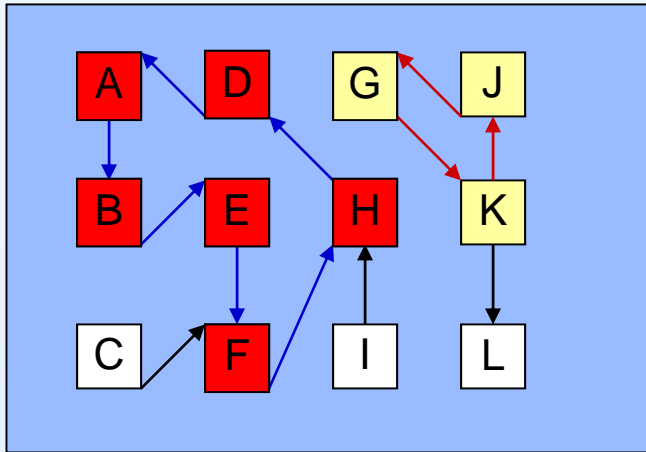


```
frag8: mov %ecx, ecx-slot  
      pop %ecx  
      jmp  <ib_lookup>
```

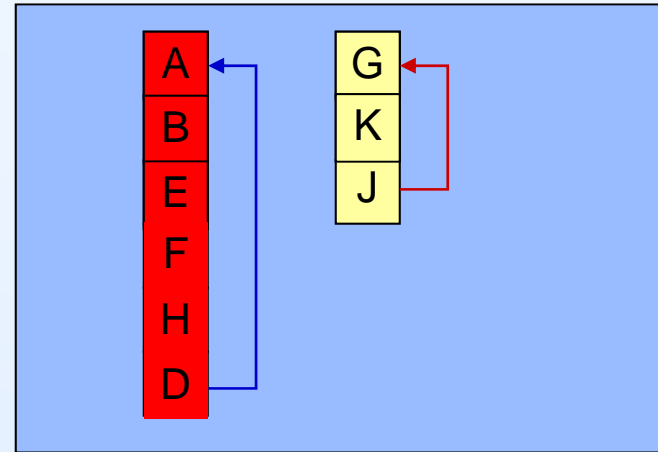
```
ib_lookup:  ...  
           ...  
           ...
```

# Improvement #4: Trace Building

Basic Block Cache



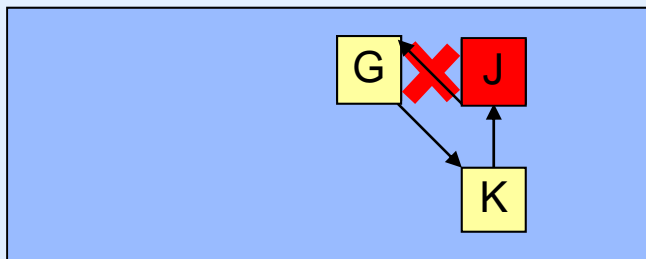
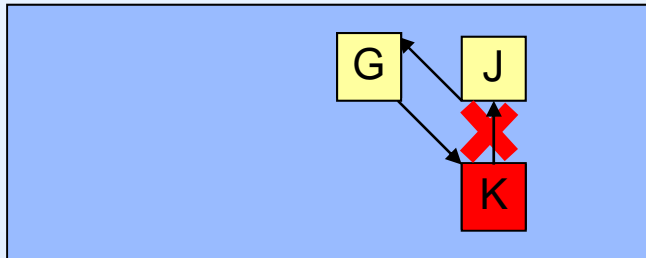
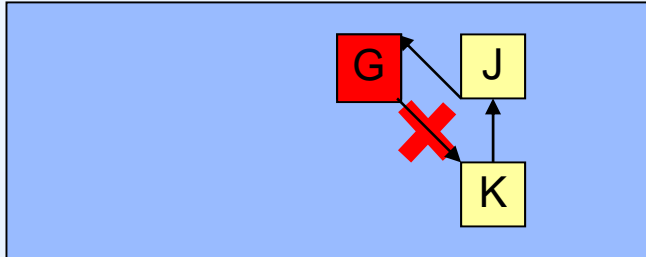
Trace Cache



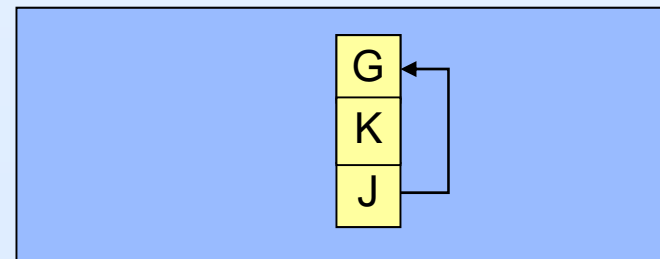
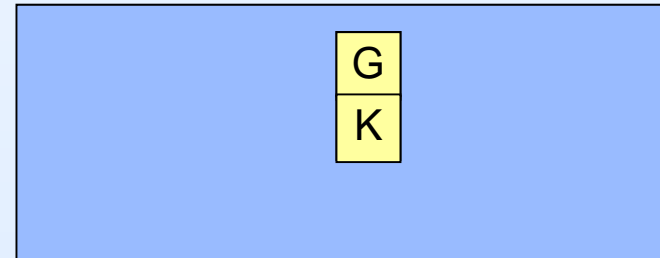
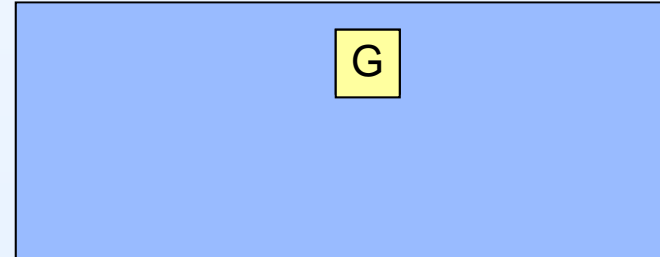
- Traces reduce branching, improve layout and locality, and facilitate optimizations across blocks
  - We avoid indirect branch lookup
- Next Executing Tail (NET) trace building scheme [Duesterwald 2000]

# Incremental NET Trace Building

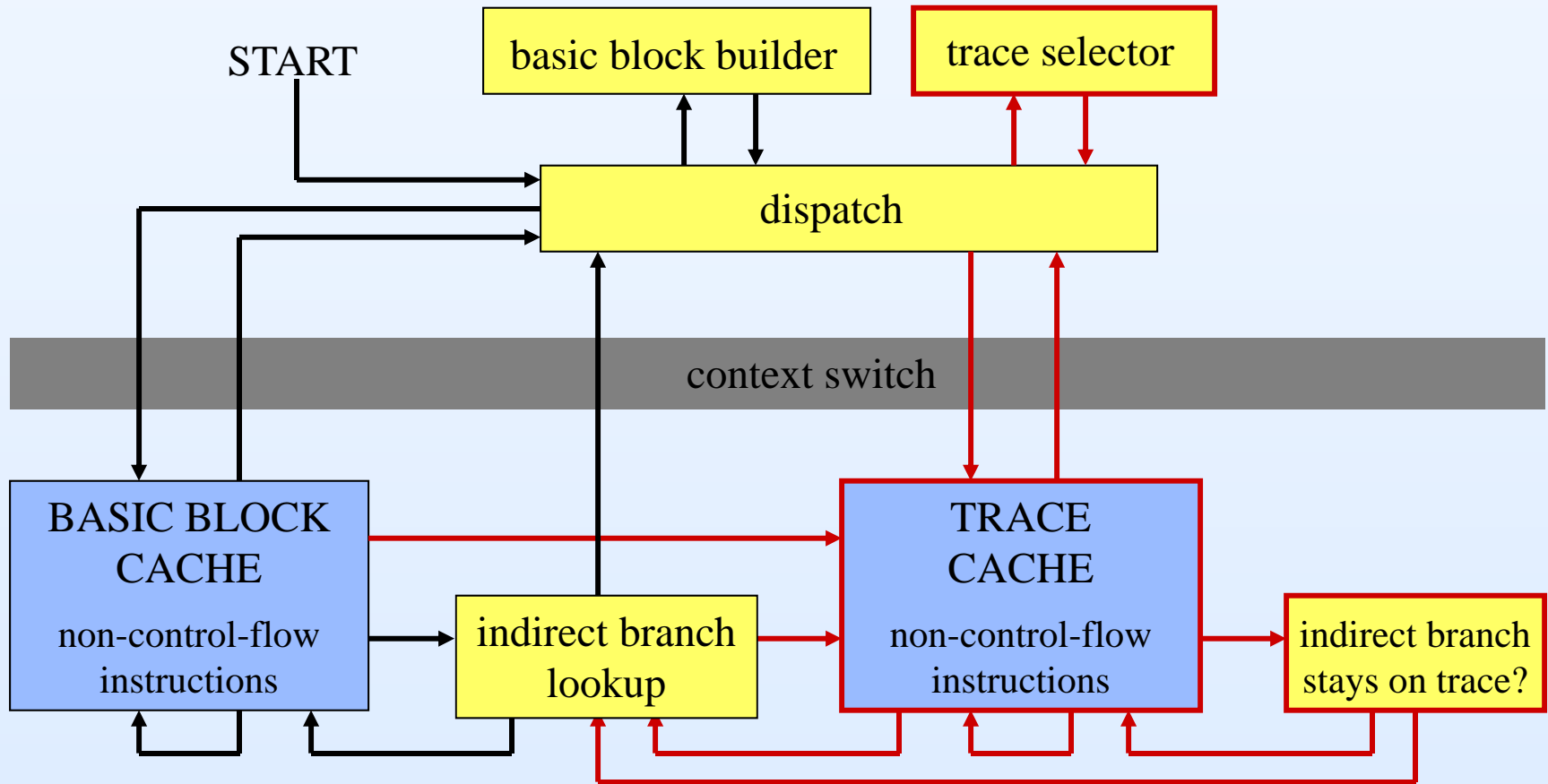
## Basic Block Cache



## Trace Cache

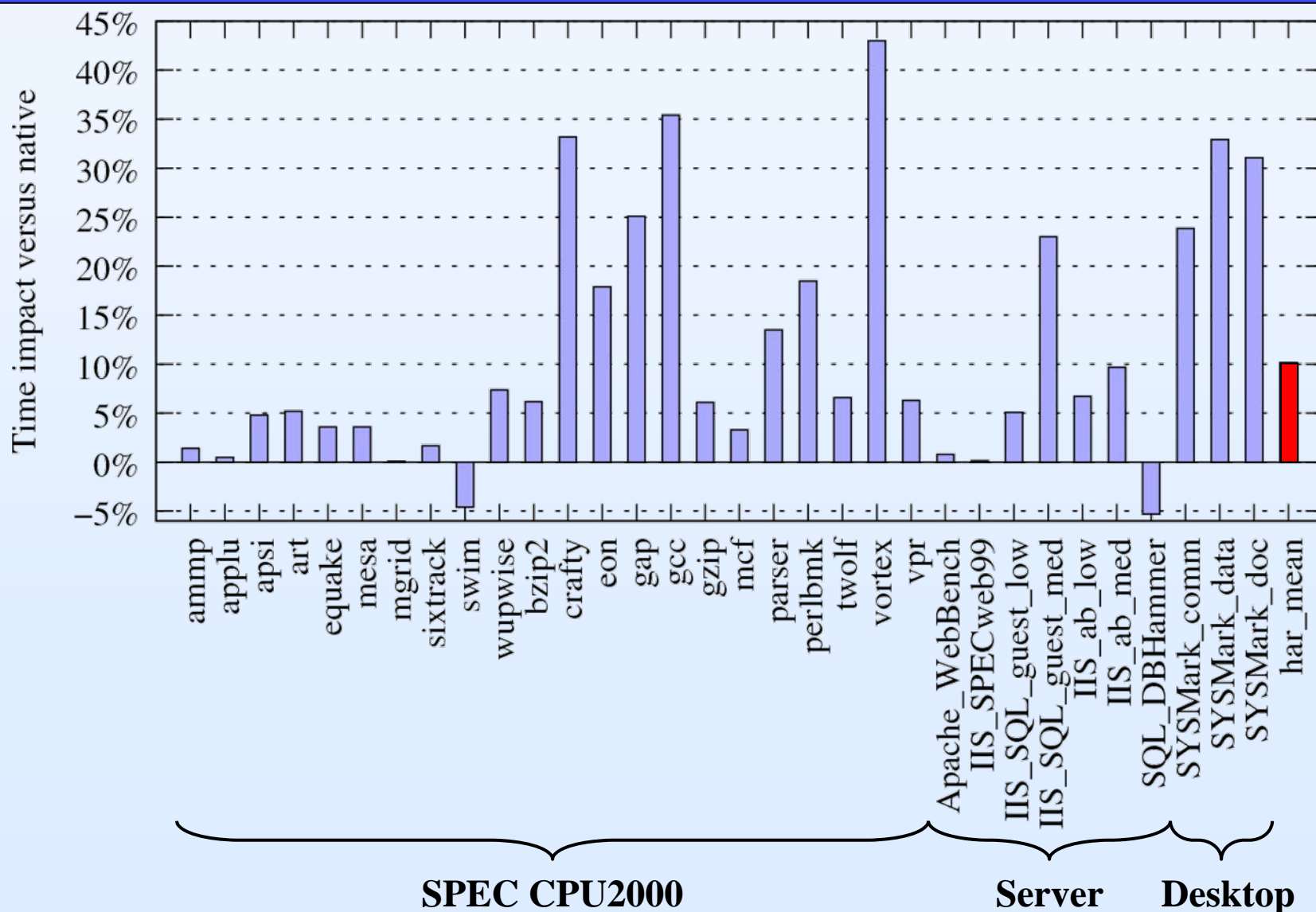


# Improvement #4: Trace Building



Slowdown: ~~300x~~ ~~26x~~ ~~3x~~ ~~1.2x~~ 1.1x

# Base Performance

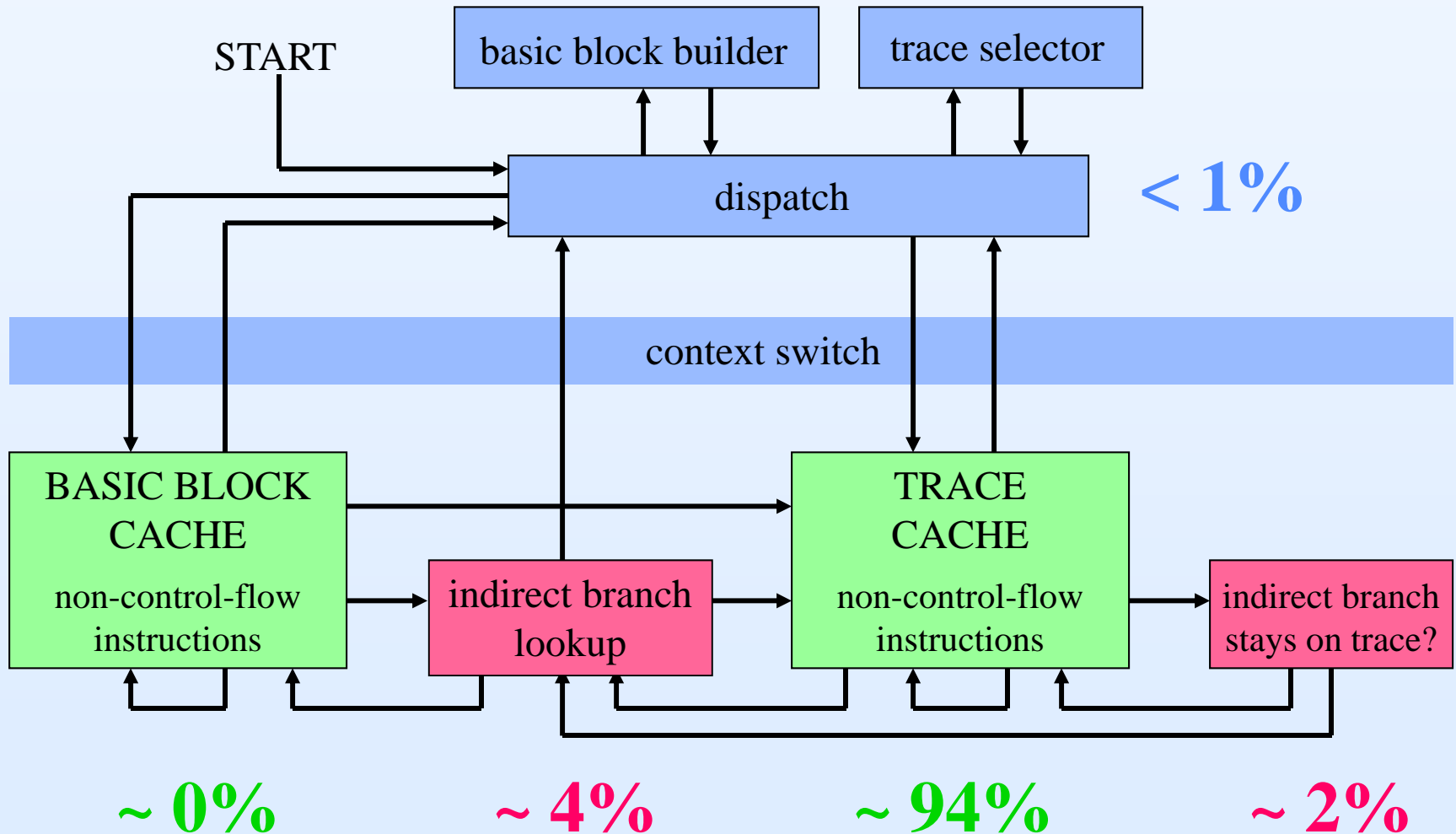


# Sources of Overhead

---

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

# Time Breakdown for SPECINT



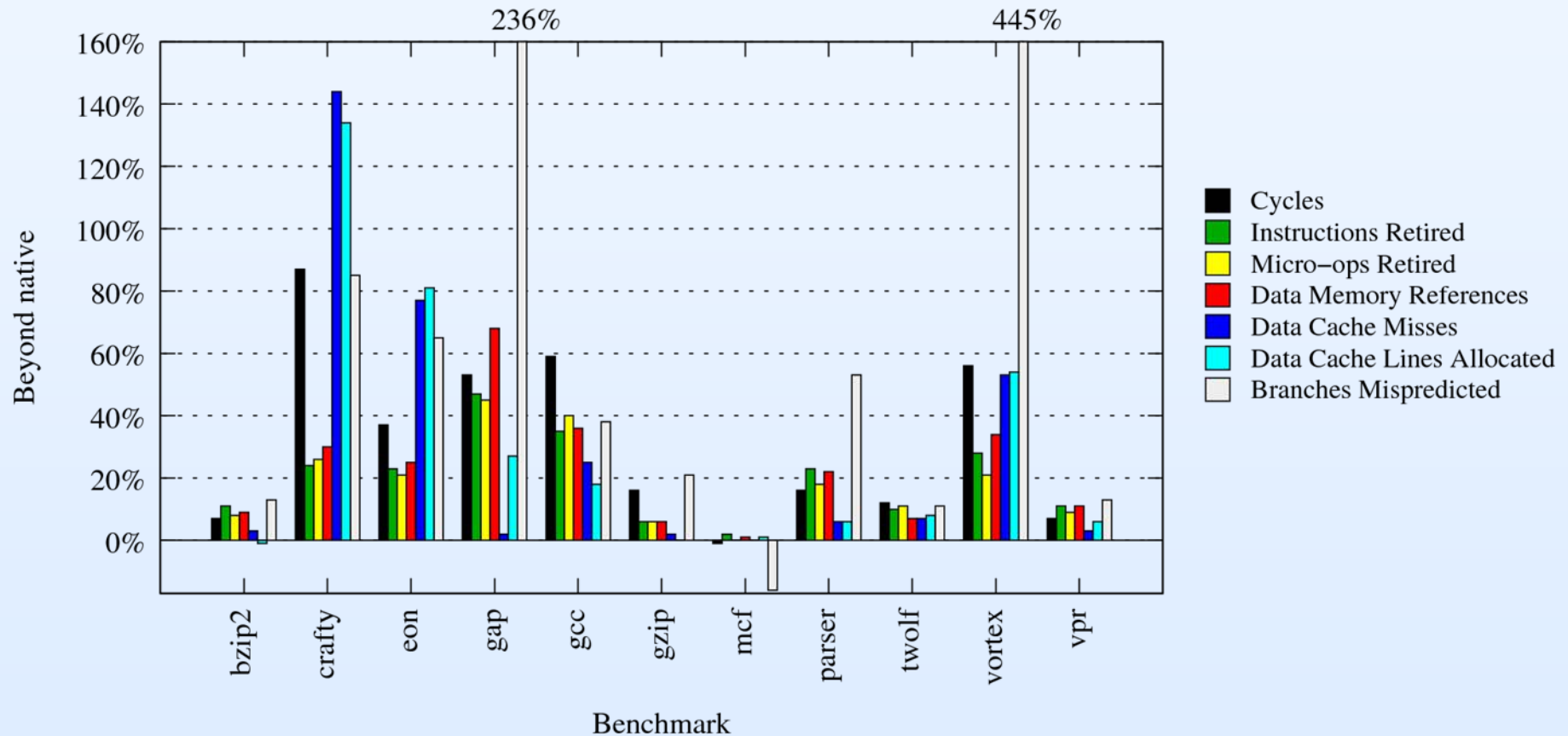
# Not An Ordinary Application

---

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



# Performance Counter Data



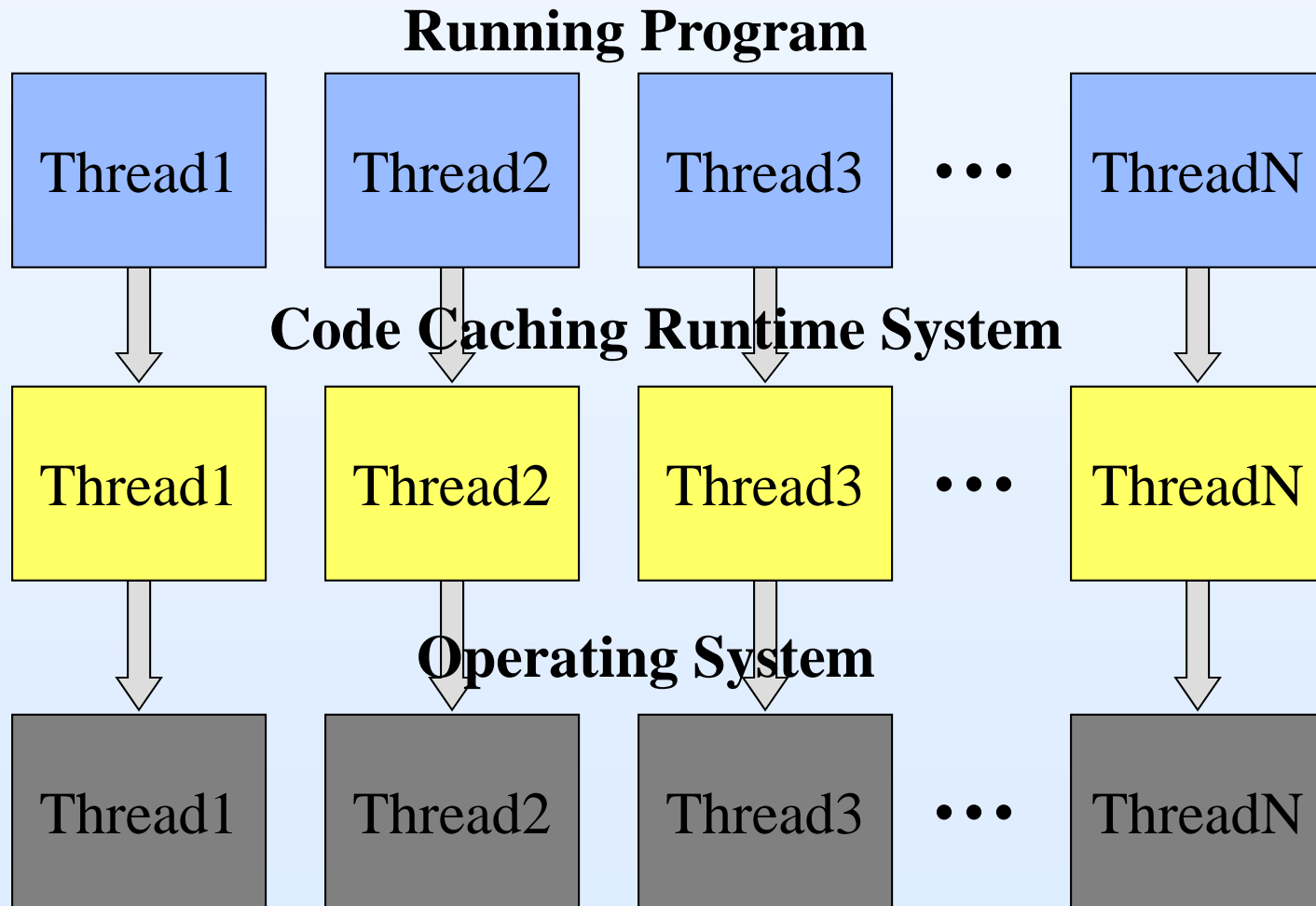
# Internals Outline

---

- Efficient
  - Software code cache overview
  - Thread-shared code cache
  - Cache capacity limits
  - Data structures
- Transparent
- Comprehensive
- Customizable

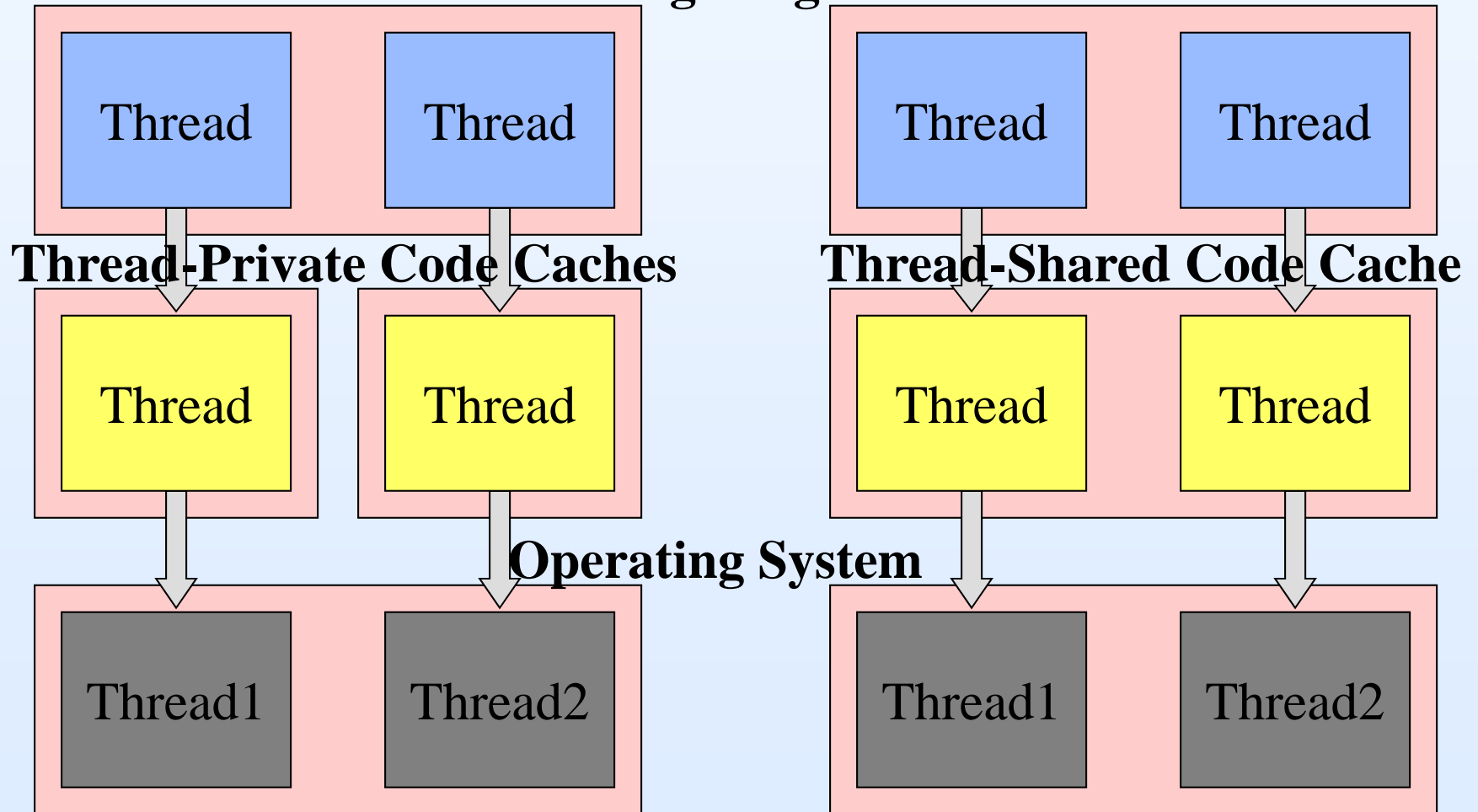
# Threading Model

---



# Code Space

## Running Program



# Thread-Private versus Thread-Shared

---

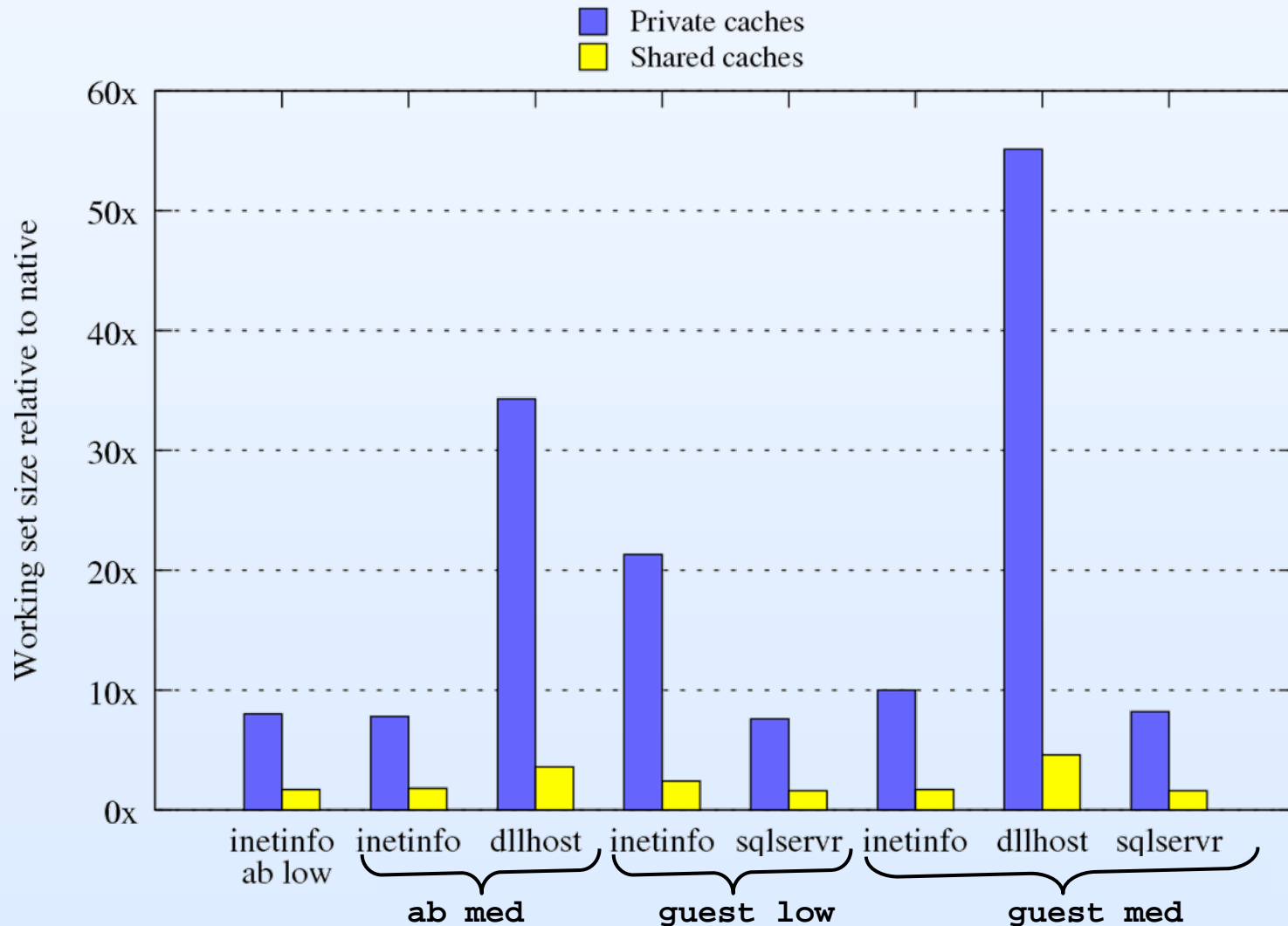
- Thread-private
  - Less synchronization needed
  - Absolute addressing for thread-local storage
  - Thread-specific optimization and instrumentation
- Thread-shared
  - Scales to many-threaded apps

# Database and Web Server Suite

---

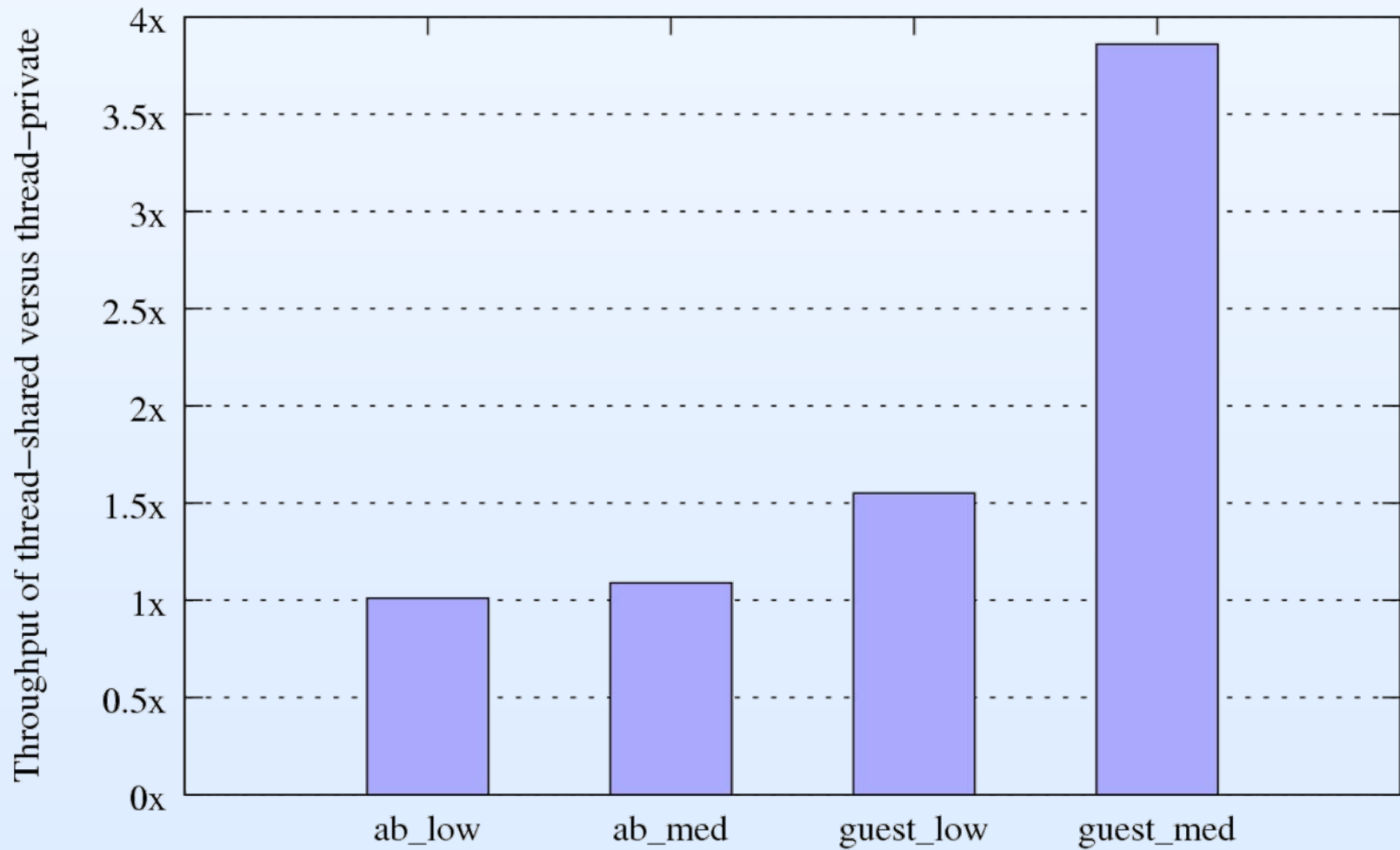
Benchmark	Server	Processes
ab low	IIS low isolation	inetinfo.exe
ab med	IIS medium isolation	inetinfo.exe, dllhost.exe
guest low	IIS low isolation, SQL Server 2000	inetinfo.exe, sqlservr.exe
guest med	IIS medium isolation, SQL Server 2000	inetinfo.exe, dllhost.exe, sqlservr.exe

# Memory Impact



# Performance Impact

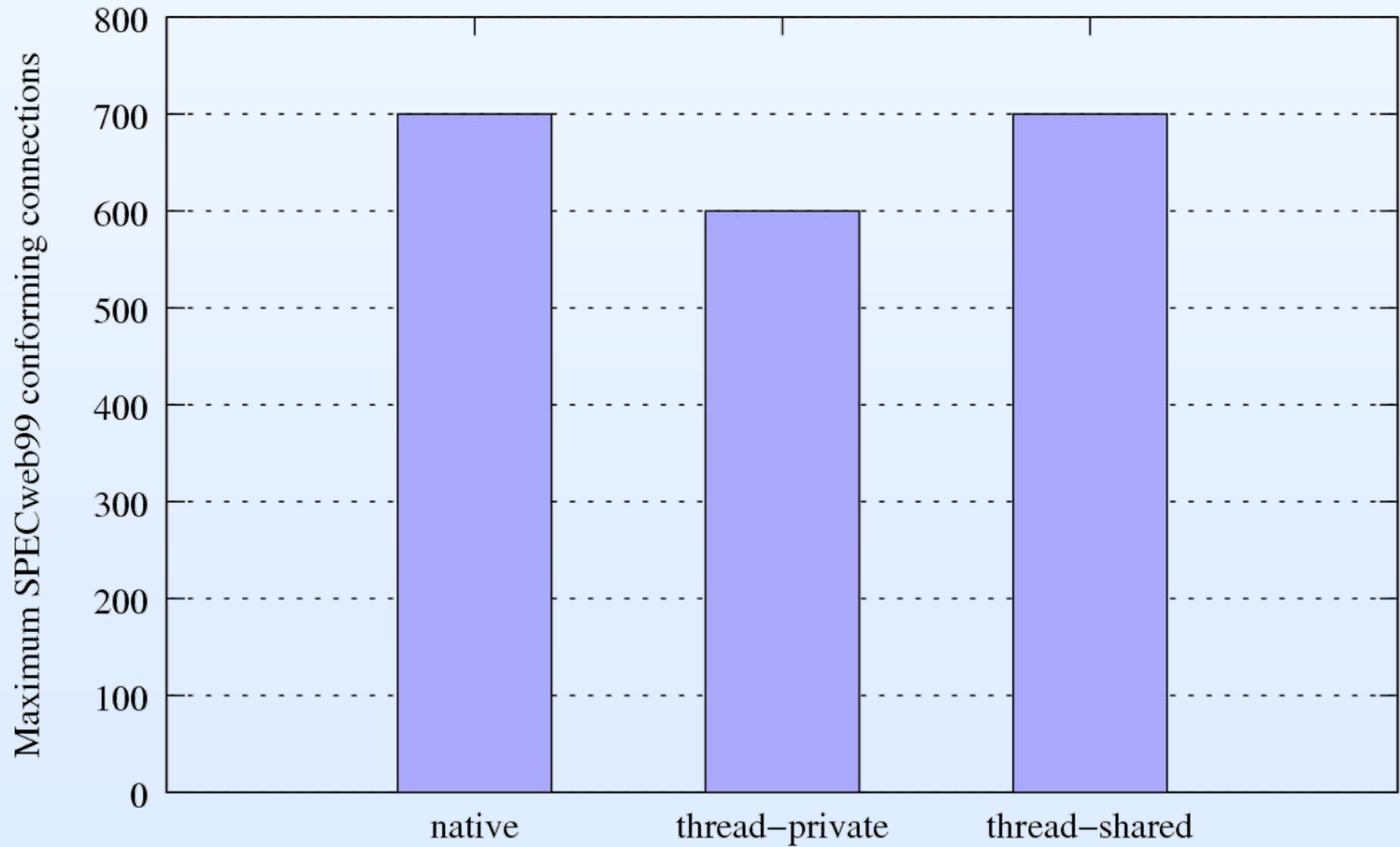
---





# Scalability Limit

---

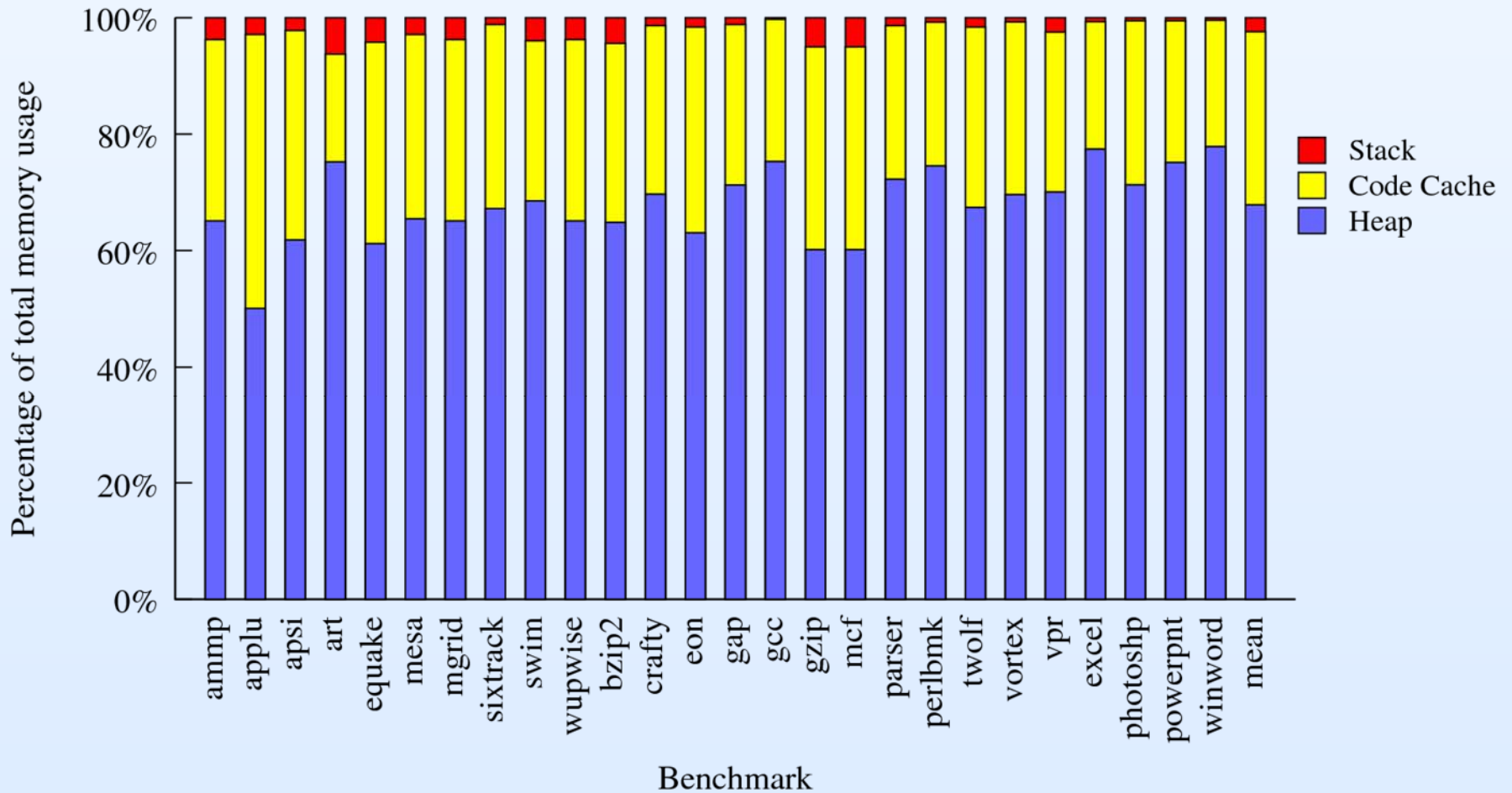


# Internals Outline

---

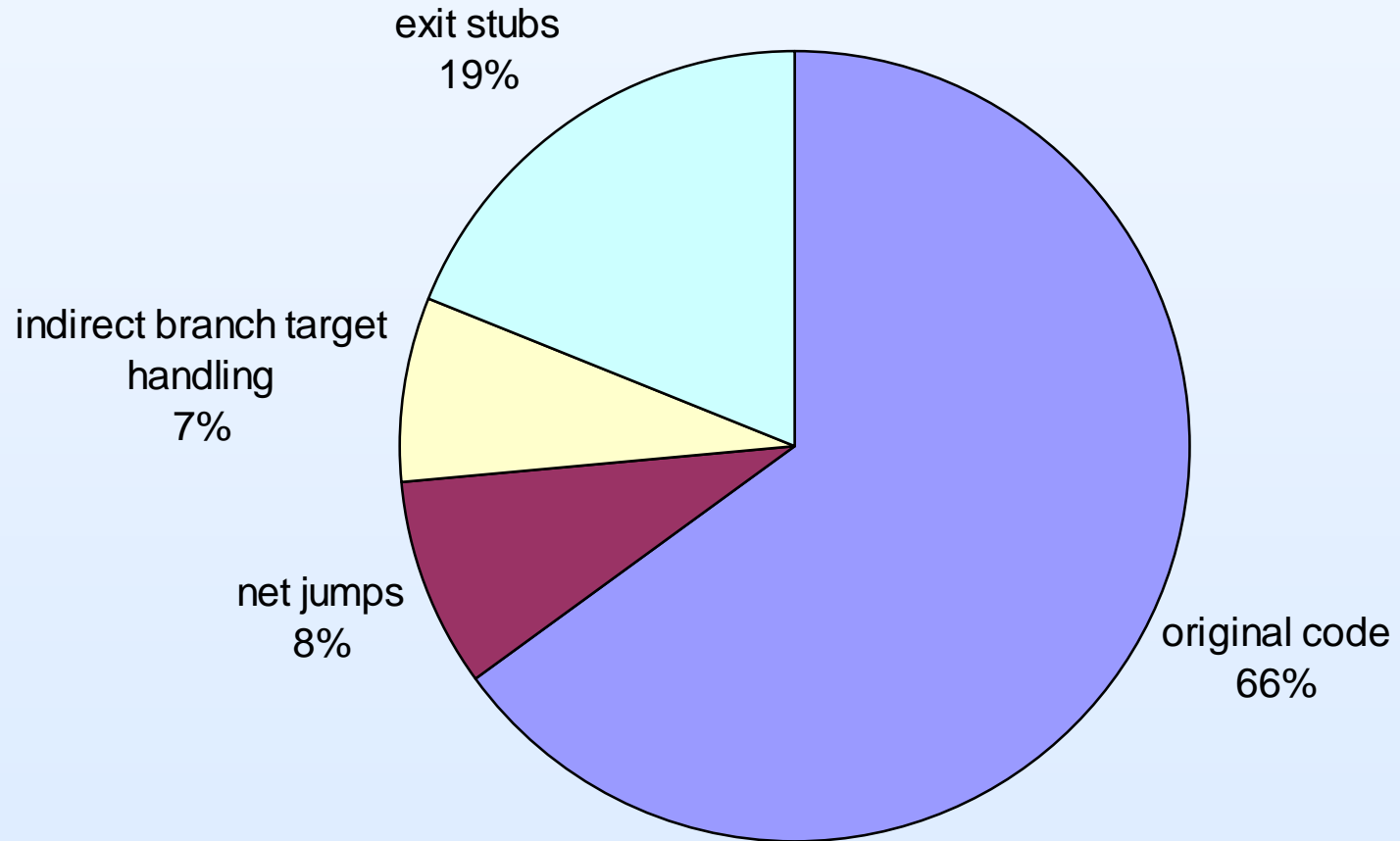
- Efficient
  - Software code cache overview
  - Thread-shared code cache
  - Cache capacity limits
  - Data structures
- Transparent
- Comprehensive
- Customizable

# Added Memory Breakdown



# Code Expansion

---



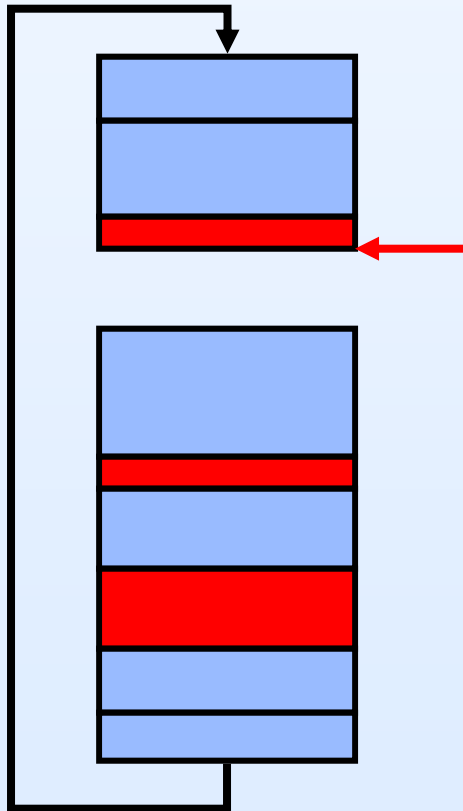
# Cache Capacity Challenges

---

- How to set an upper limit on the cache size
  - Different applications have different working sets and different total code sizes
- Which fragments to evict when that limit is reached
  - Without expensive profiling or extensive fragmentation

# Adaptive Sizing Algorithm

---



- Enlarge cache if warranted by percentage of new fragments that are *regenerated*
- Target *working set* of application: don't enlarge for once-only code
- Low-overhead, incremental, and reactive

# Cache Capacity Settings

---

- Thread-private:
  - Working set size matching is on by default
  - Client may see blocks or traces being deleted in the absence of any cache consistency event
  - Can disable capacity management via
    - -no\_finite\_bb\_cache
    - -no\_finite\_trace\_cache
- Thread-shared:
  - Set to infinite size by default
  - Can enable capacity management via
    - -finite\_shared\_bb\_cache
    - -finite\_shared\_trace\_cache

# Internals Outline

---

- Efficient
  - Software code cache overview
  - Thread-shared code cache
  - Cache capacity limits
  - Data structures
- Transparent
- Comprehensive
- Customizable



# Two Modes of Code Cache Operation

---

- Fine-grained scheme
  - Supports individual code fragment unlink and removal
  - Separate data structure per code fragment and each of its exits, memory regions spanned, and incoming links
- Coarse-grained scheme
  - No individual code fragment control
  - Permanent intra-cache links
  - No per-fragment data structures at all
  - Treat entire cache as a unit for consistency

# Data Structures

---

- Fine-grained scheme
  - Data structures are highly tuned and compact
- Coarse-grained scheme
  - There are no data structures
  - Savings on applications with large amounts of code are typically 15%-25% of committed memory and 5%-15% of working set

# Status in Current Release

---

- Fine-grained scheme
  - Current default
- Coarse-grained scheme
  - Select with `-opt_memory` runtime option
  - Possible performance hit on certain benchmarks
  - In the future will be the default option
  - Required for persisted and process-shared caches

# Adaptive Level of Granularity

---

- Start with coarse-grain caches
  - Plus freezing and sharing/persisting
- Switch to fine-grain for individual modules or sub-regions of modules after significant consistency events, to avoid expensive entire-module flushes
  - Support simultaneous fine-grain fragments within coarse-grain regions for corner cases
- Match amount of bookkeeping to amount of code change
  - Majority of application code does not need fine-grain

# Many Varieties of Code Caches

---

- Coarse-grained versus fine-grained
- Thread-shared versus thread-private
- Basic blocks versus traces

# Internals Outline

---

- Efficient
- Transparent
  - Rules of transparency
  - Cache consistency
  - Synchronization
- Comprehensive
- Customizable

# 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.



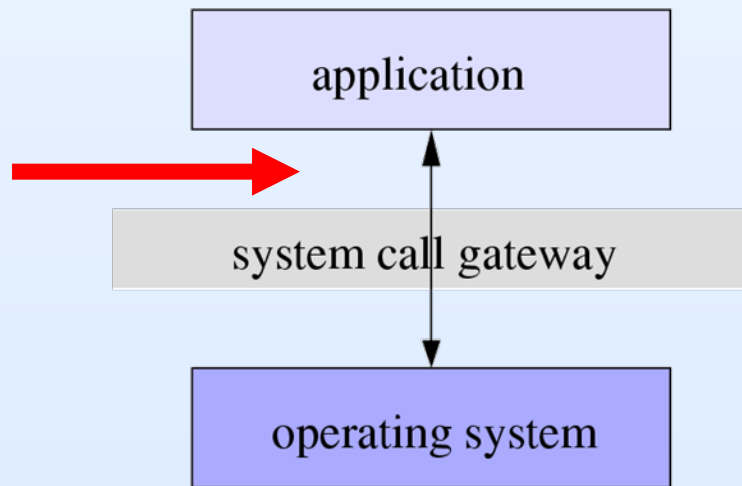
# Rule 1: Avoid Resource Conflicts

---

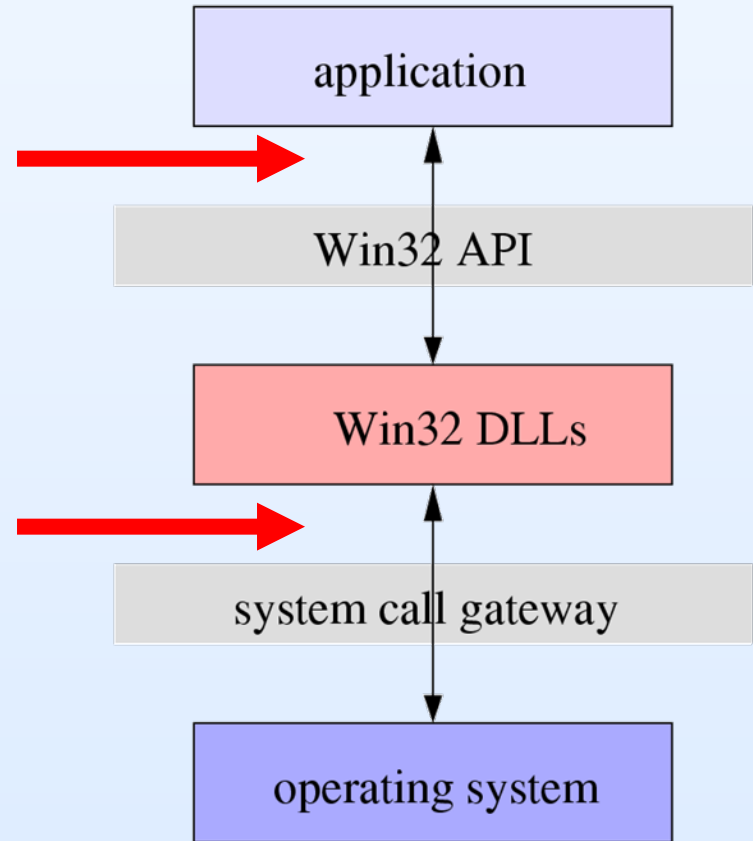
- DynamoRIO system code executes at arbitrary points during application execution
- If DynamoRIO uses the same library routine as the application, it may call that routine in the middle of the same routine being called by the application
- Most library routines are not re-entrant!
  - Many are thread-safe, but that does not help us

# Rule 1: Avoid Resource Conflicts

---



Linux



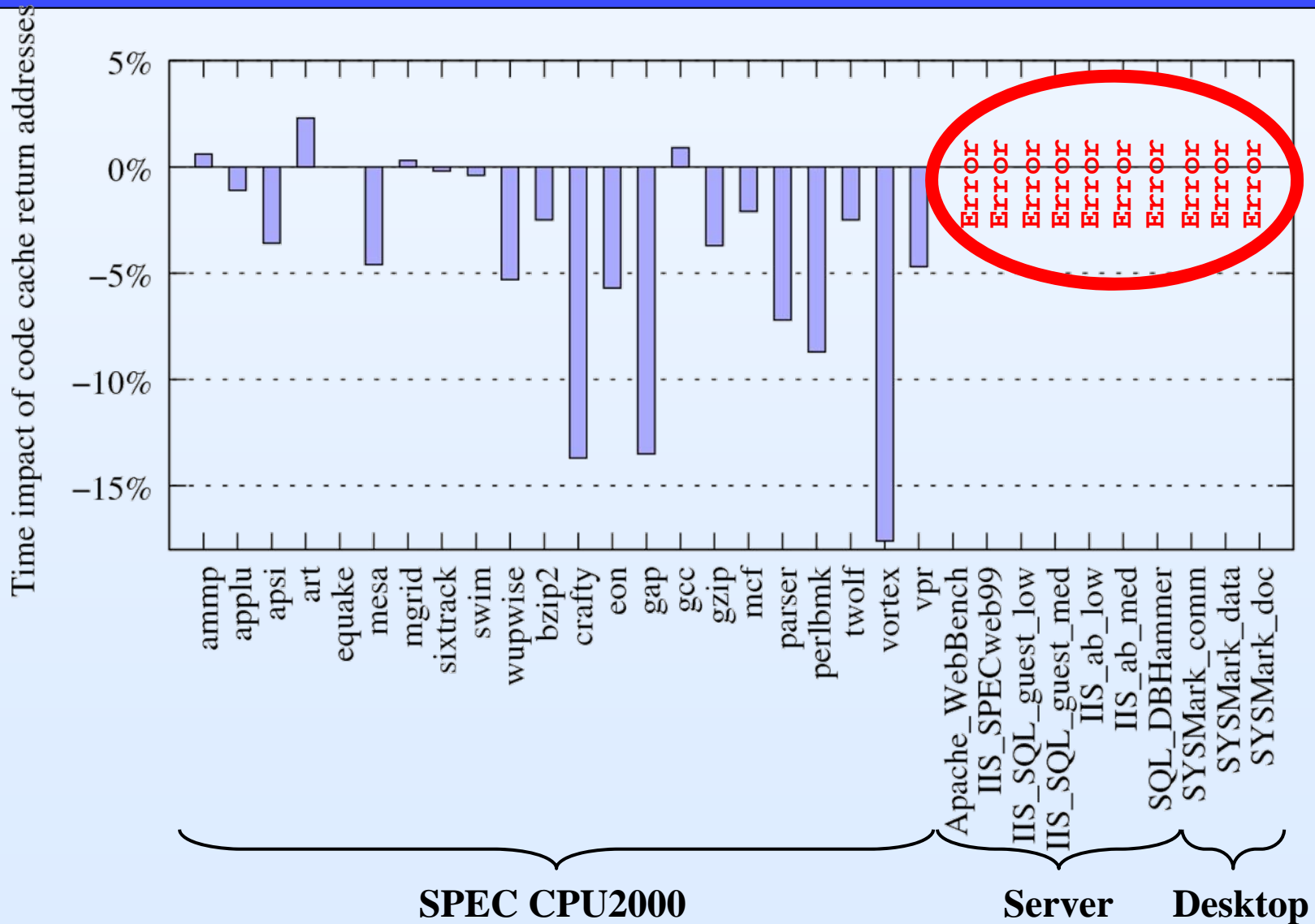
Windows

# Rule 2: If It's Not Broken, Don't Change It

---

- Threads
- Executable on disk
- Application data
  - Including the stack!

# Example Transparency Violation



# Rule 3: If You Change It, Emulate Original Behavior's Visible Effects

---

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

# Internals Outline

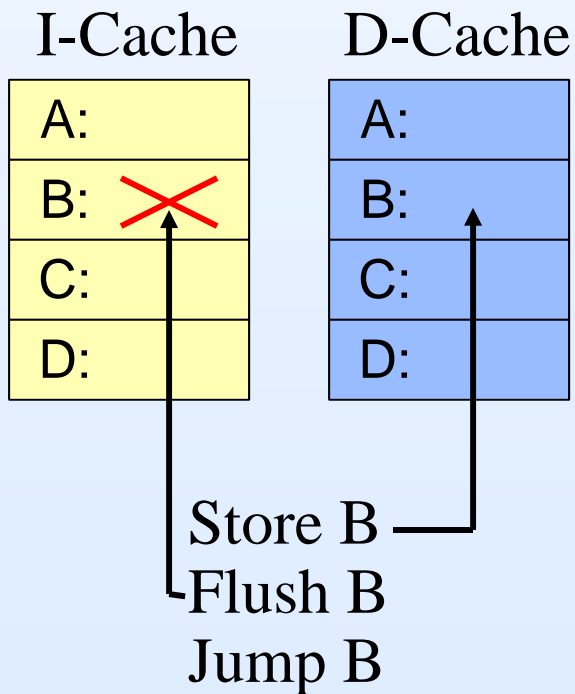
---

- Efficient
- Transparent
  - Rules of transparency
  - Cache consistency
  - Synchronization
- Comprehensive
- Customizable

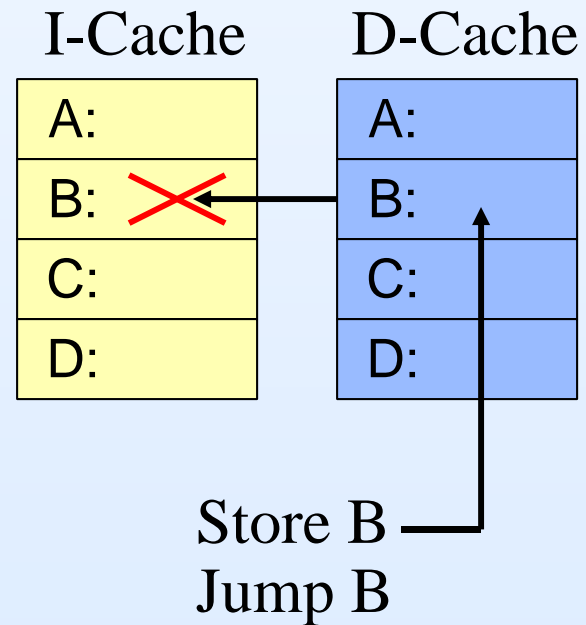
# Code Change Mechanisms

---

## RISC



## x86



# How Often Does Code Change?

---

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



# Code Change Events

---

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

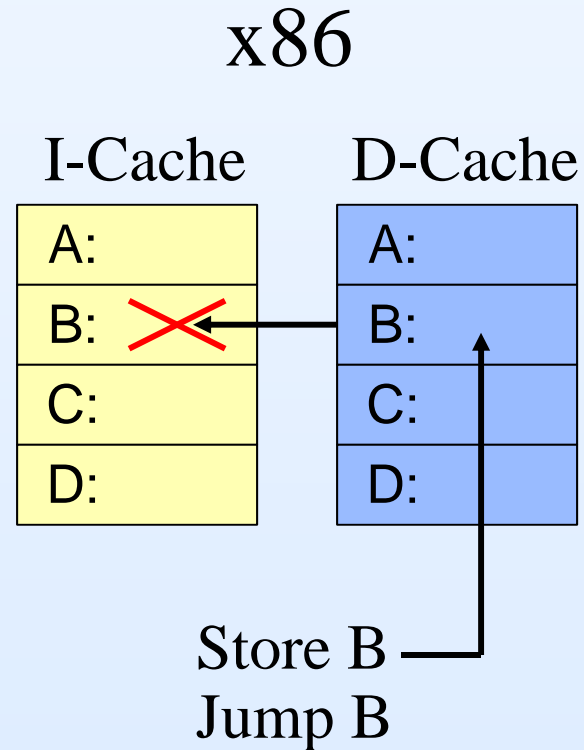
# Detecting Code Removal

---

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

# Detecting Code Modification

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



# Page Protection Plus Instrumentation

---

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

# Adaptive Consistency Algorithm

---

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

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

# Internals Outline

---

- Efficient
- Transparent
  - Rules of transparency
  - 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

# Code Cache Invariant

---

- 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



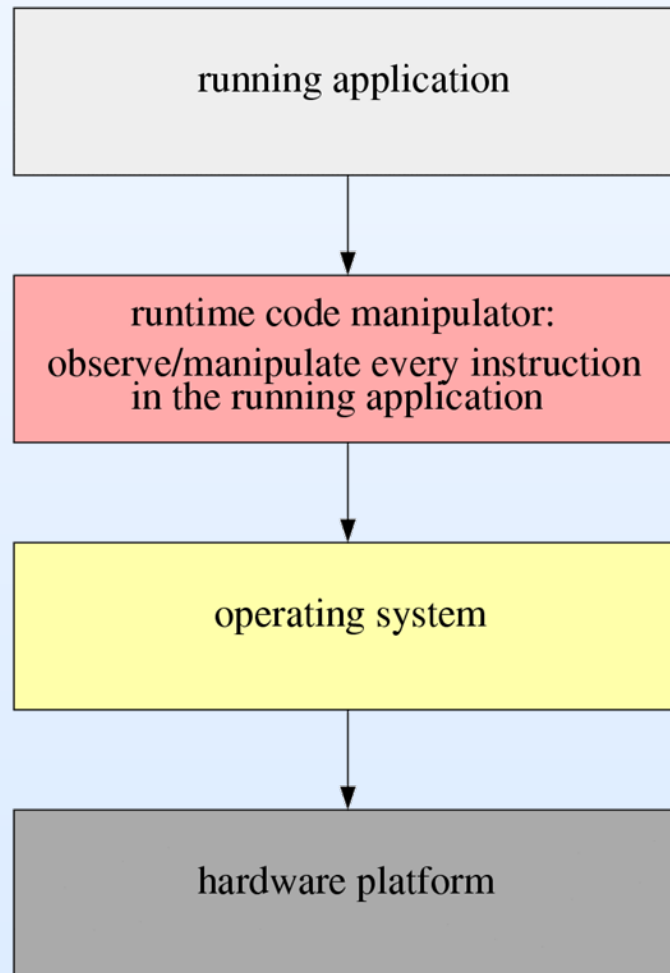
# Internals Outline

---

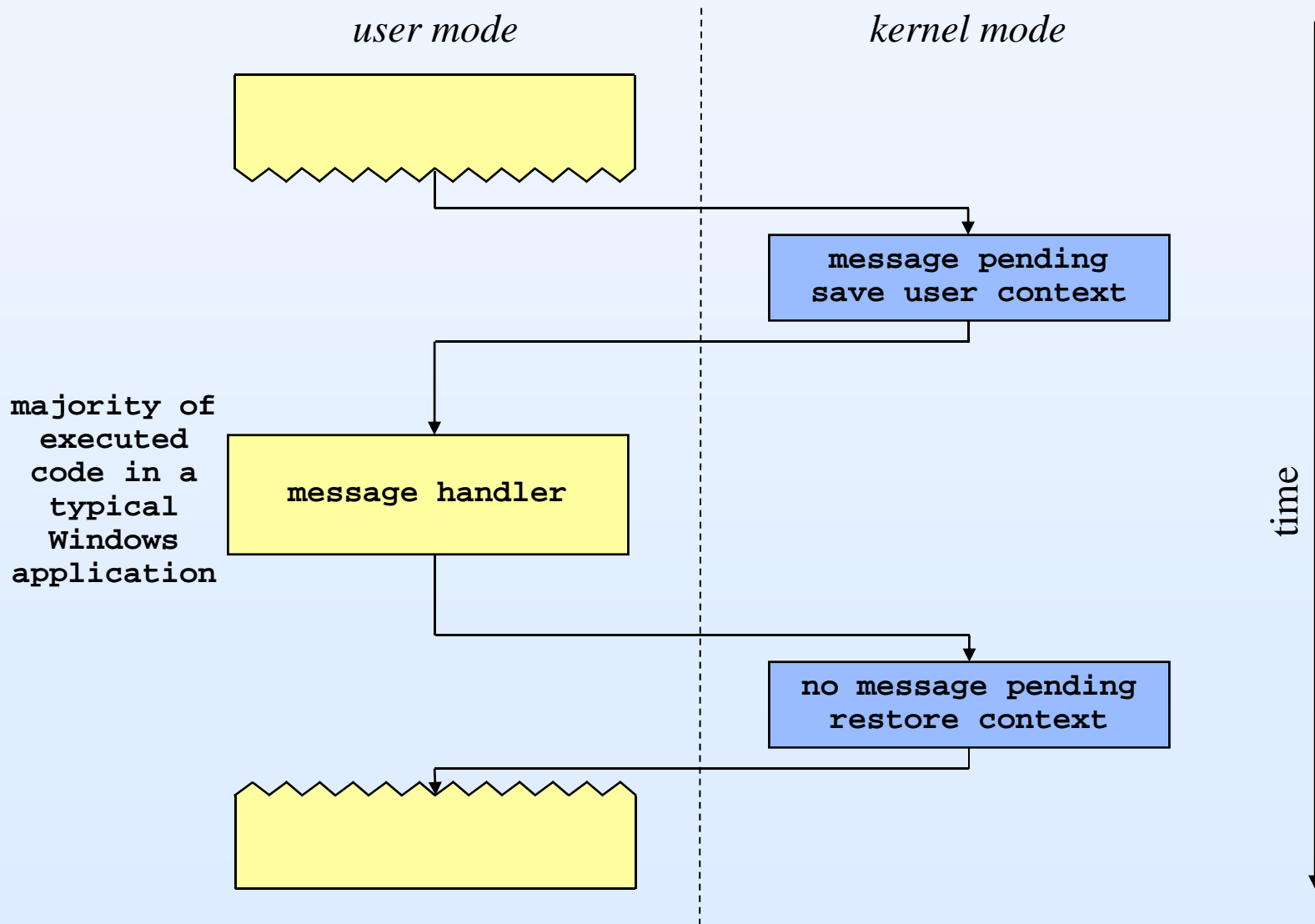
- Efficient
- Transparent
- Comprehensive
- Customizable

# Above the Operating System

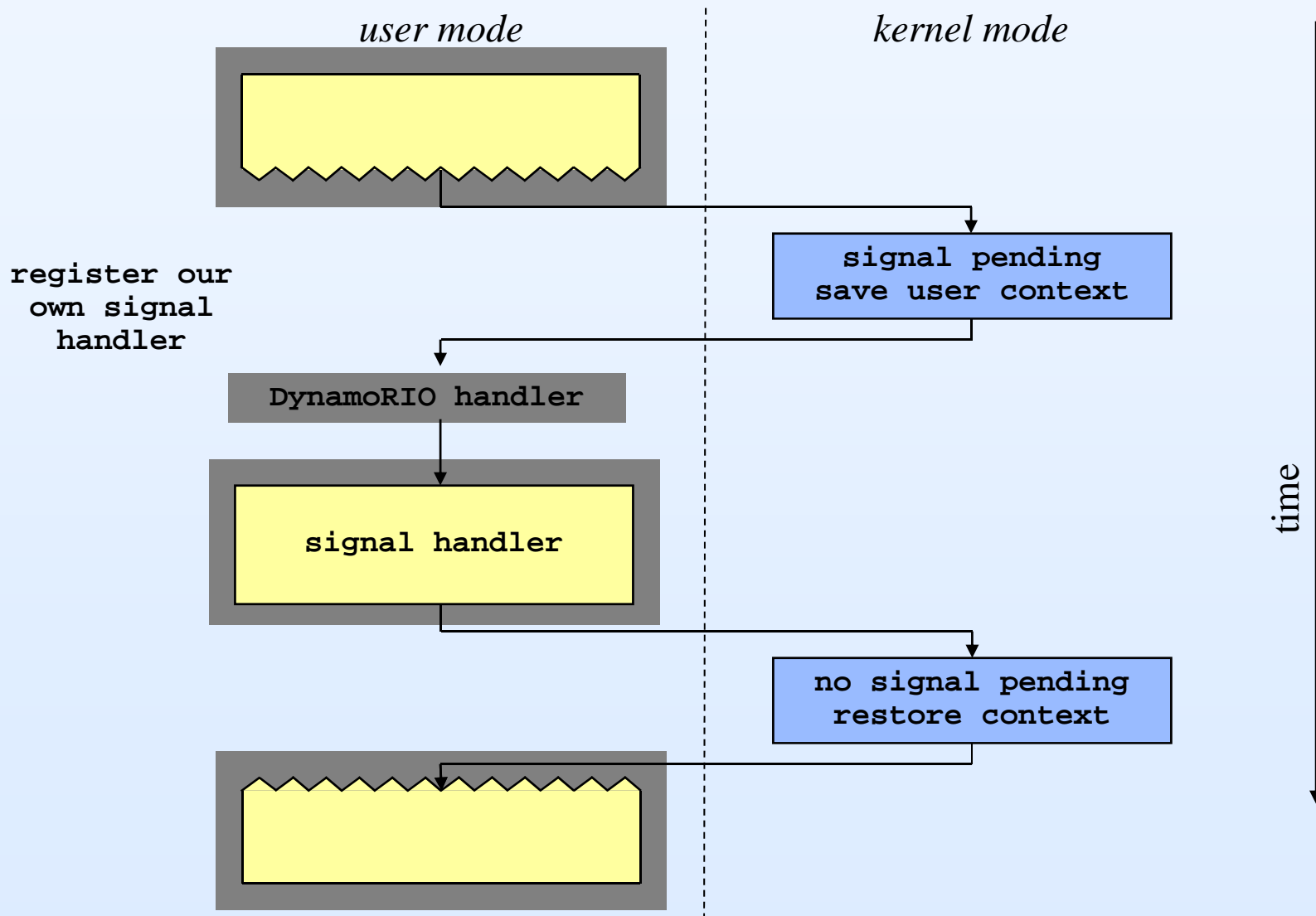
---



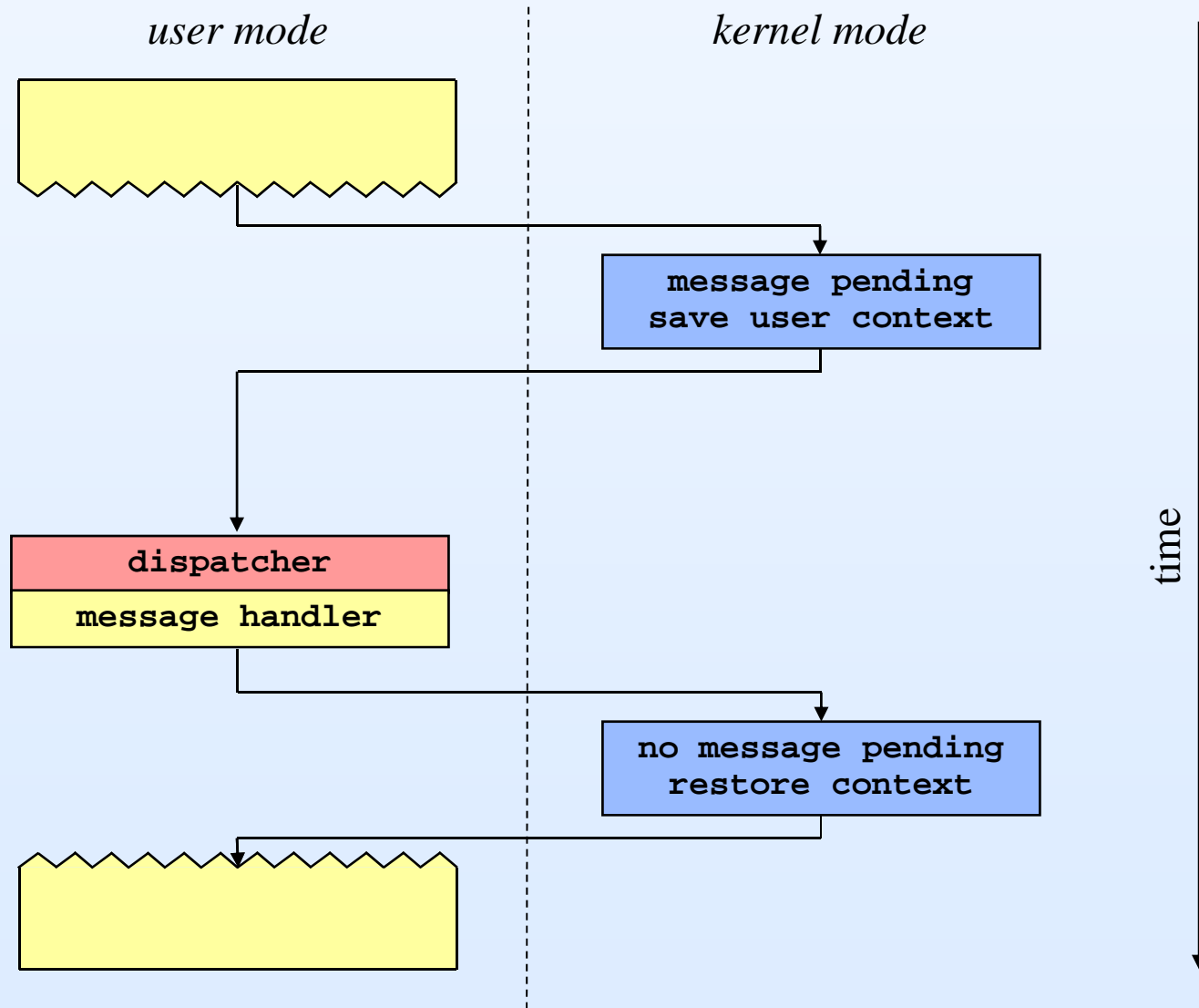
# Kernel-Mediated Control Transfers



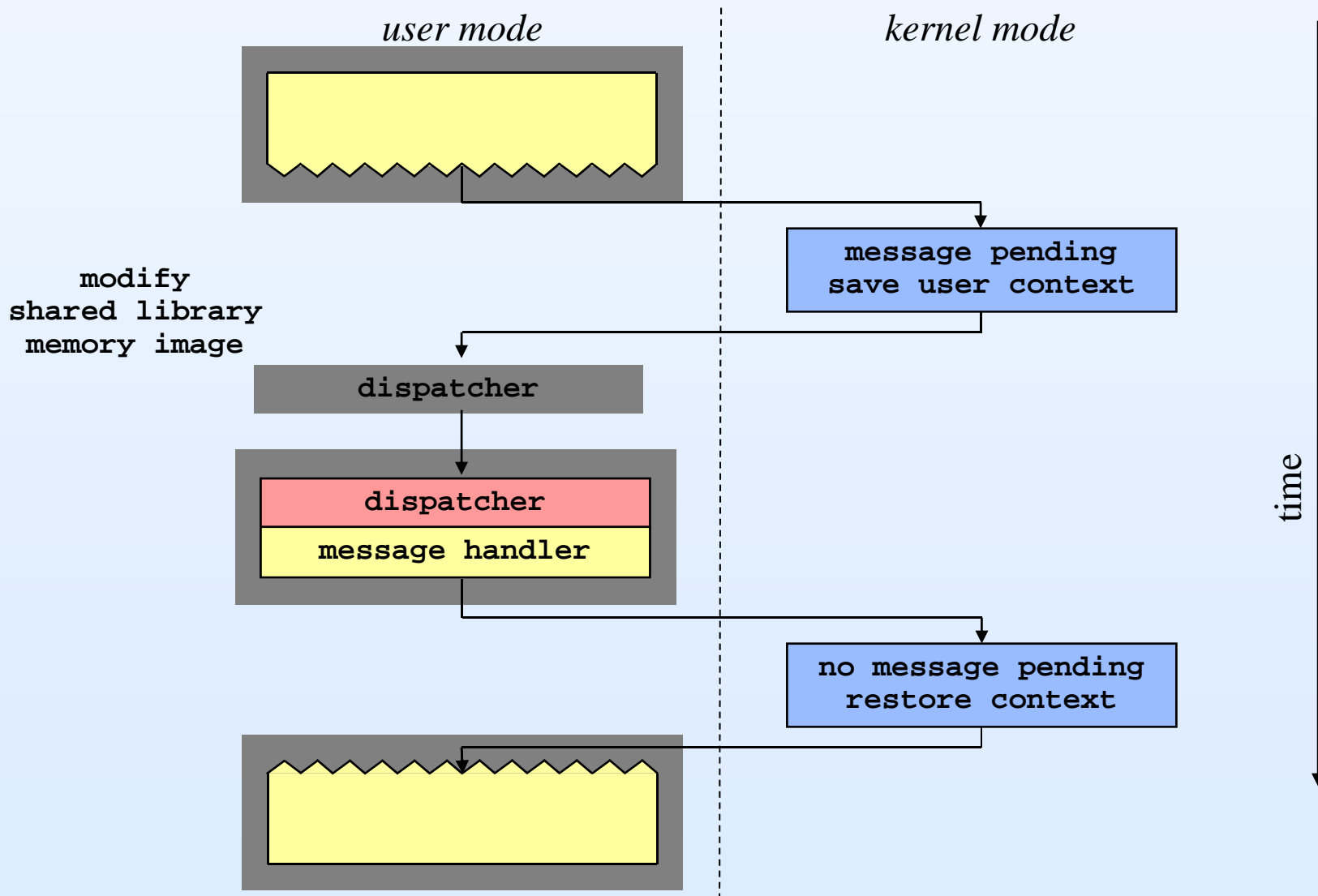
# Intercepting Linux Signals



# Windows Messages



# Intercepting Windows Messages



# Must Monitor System Calls

---

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

# Operating System Dependencies

---

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



# Internals Outline

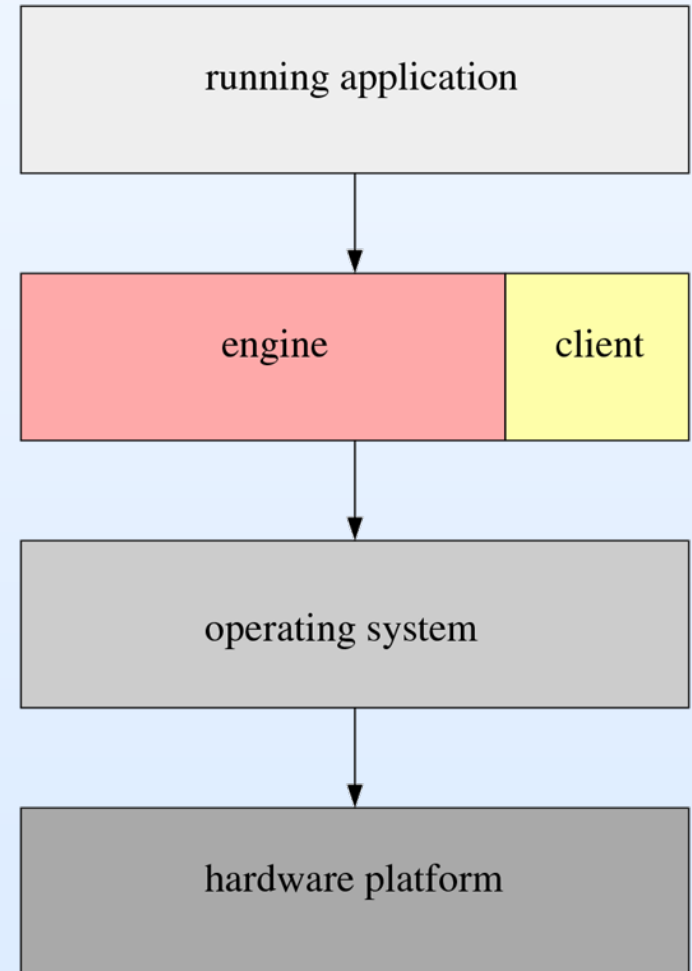
---

- Efficient
- Transparent
- Comprehensive
- Customizable

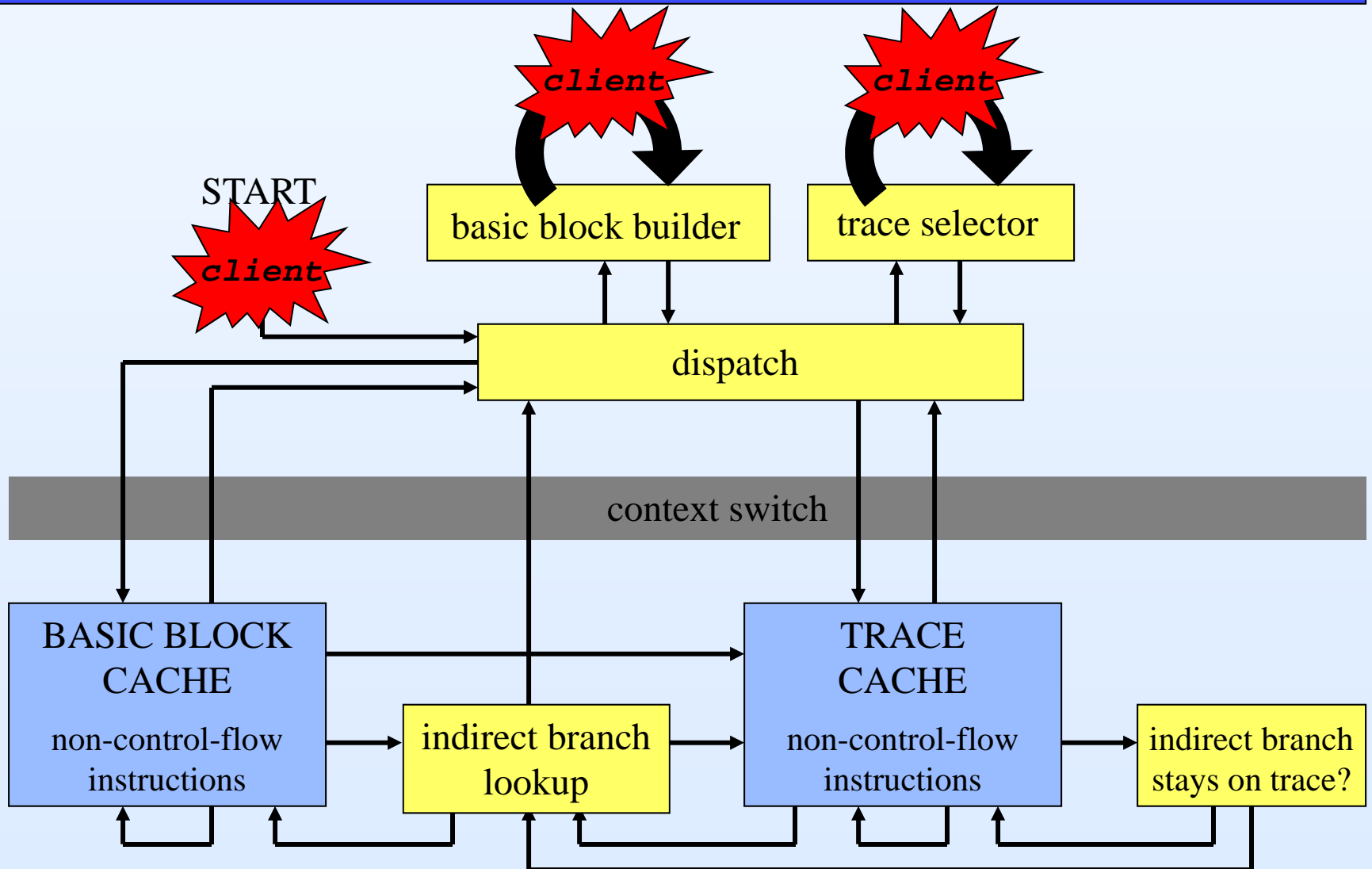
# Clients

---

- The engine exports an API for building a client
- System details abstracted away: client focuses on manipulating the code stream



# Client Events



# DynamoRIO API

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Internals
3:10-4:00	DynamoRIO API
4:00-4:15	<i>Break</i>
4:15-5:15	Demonstrations
5:15-5:45	Lab Session
5:45-6:00	Feedback

# DynamoRIO API Outline

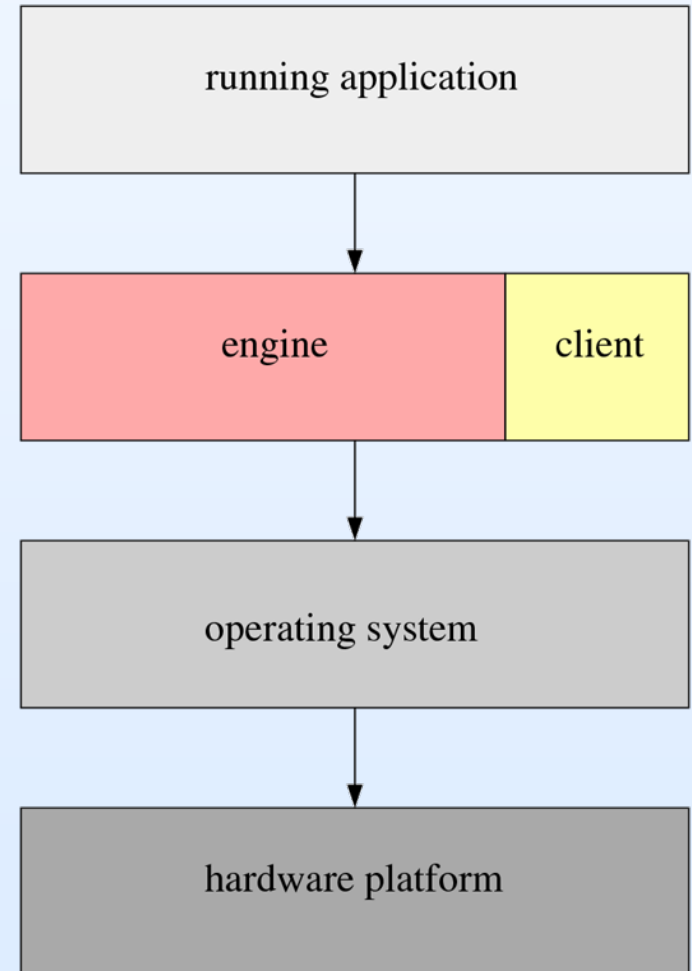
---

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

# Clients

---

- The engine exports an API for building a client
- System details abstracted away: client focuses on manipulating the code stream



# Cross-Platform Clients

---

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

# Building a Client

---

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



# Auto-Configure Using CMake

---

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

# CMake

---

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

# Library Usage and Transparency

---

- For best transparency: completely self-contained client
  - Imports only from DynamoRIO API
  - `-nodefaultlibs` or `/nodefaultlib`
- On Windows:
  - String and utility routines provided by forwards to `ntdll`
  - `Cl.exe` /MT static copy of C/C++ libraries
  - Custom loader loads private copy of client dependences
- On Linux:
  - For static C/C++ lib, use `ld -wrap` to redirect `malloc` to DR's heap
    - Newer distributions don't ship suitable static C/C++ lib
  - Custom loader for private copies of shared libs

# DynamoRIO Extensions

---

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

# DynamoRIO Extensions, Cont'd

---

- Current Extensions:
  - *drsyms*: symbol lookup (currently Windows-only)
  - *drcontainers*: hashtable
  - *drmgr*: multi-instrumentation mediation
  - *drwrap*: function wrapping and replacing
- Coming soon:
  - *drreg*: register stealing and allocating
  - *Umbra*: shadow memory framework
  - *Your utility library or framework contribution!*

# Application Configuration

---

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

# Deploying Clients

---

- One-step configure-and-run usage model:
  - **dr**run <client> <options> <app cmdline>
  - Uses an invisible temporary one-time configuration file
  - Overrides any regular config file
- Two-step usage model giving more control over children:
  - **dr**config –reg <appname> <client> <options>
  - **dr**inject <app cmdline>
- Systemwide injection:
  - **dr**config –syswide\_on –reg <appname> <client> <options>
  - <run app normally>

# Deploying Clients On Linux

---

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



# Deploying Clients On Windows

---

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

# Following Child Processes

---

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

# Non-Standard Deployment

---

- Standalone API
  - Use DynamoRIO as a library of IA-32/AMD64 manipulation routines
- Start/Stop API
  - Can instrument source code with where DynamoRIO should control the application

# Runtime Options

---

- Pass options to drconfig/drrun
- A large number of options; the most relevant are:
  - -code\_api
  - -client <client lib> <client ops> <client id>
  - -thread\_private
  - -follow\_children
  - -opt\_cleancall
  - -tracedump\_text and -tracedump\_binary
  - -prof\_pcs

# Runtime Options For Debugging

---

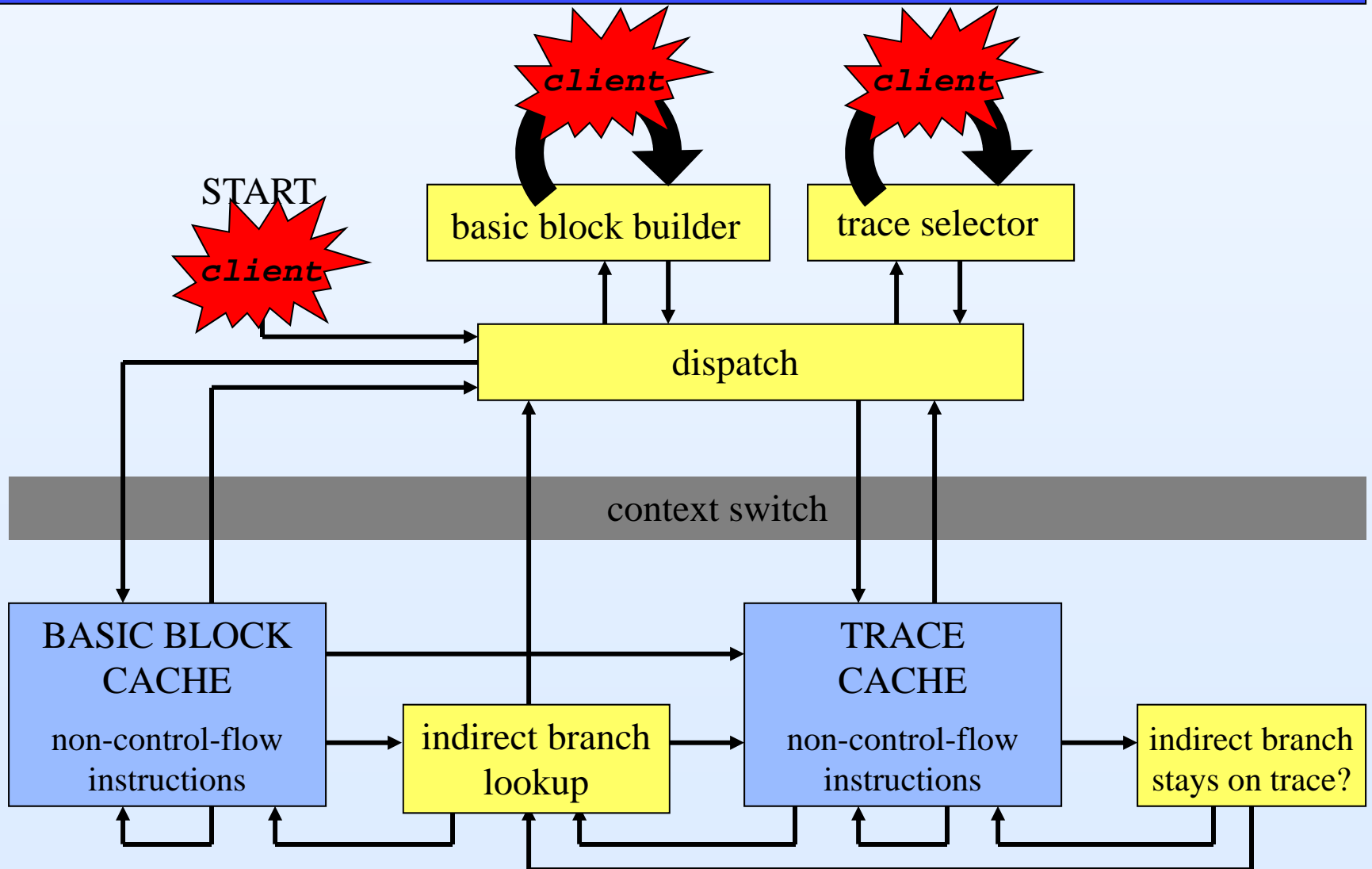
- Notifications:
  - -stderr\_mask 0xN
  - -msgbox\_mask 0xN
- Windows:
  - -no\_hide
- Debug-build-only:
  - -loglevel N
  - -ignore\_assert\_list '\*'

# DynamoRIO API Outline

---

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

# Client Events



# Client Events: Code Stream

---

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



# Simplifying Client View

---

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

# Basic Block Event

---

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

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

# Trace Event

---

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

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

# Client Events: Application Actions

---

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

# Client Events: Application System Calls

---

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

# Client Events: Bookkeeping

---

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

# Multiple Clients

---

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

# DynamoRIO API Outline

---

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



# DynamoRIO API: General Utilities

---

- DynamoRIO provides safe utilities for transparency support
  - Separate stack
  - Separate memory allocation
  - Separate file I/O
- Utility options
  - Use DynamoRIO-provided utilities
  - Use static libraries with dependencies redirected
  - Use shared libraries via DynamoRIO private loader

# DynamoRIO Heap

---

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

# Thread Support

---

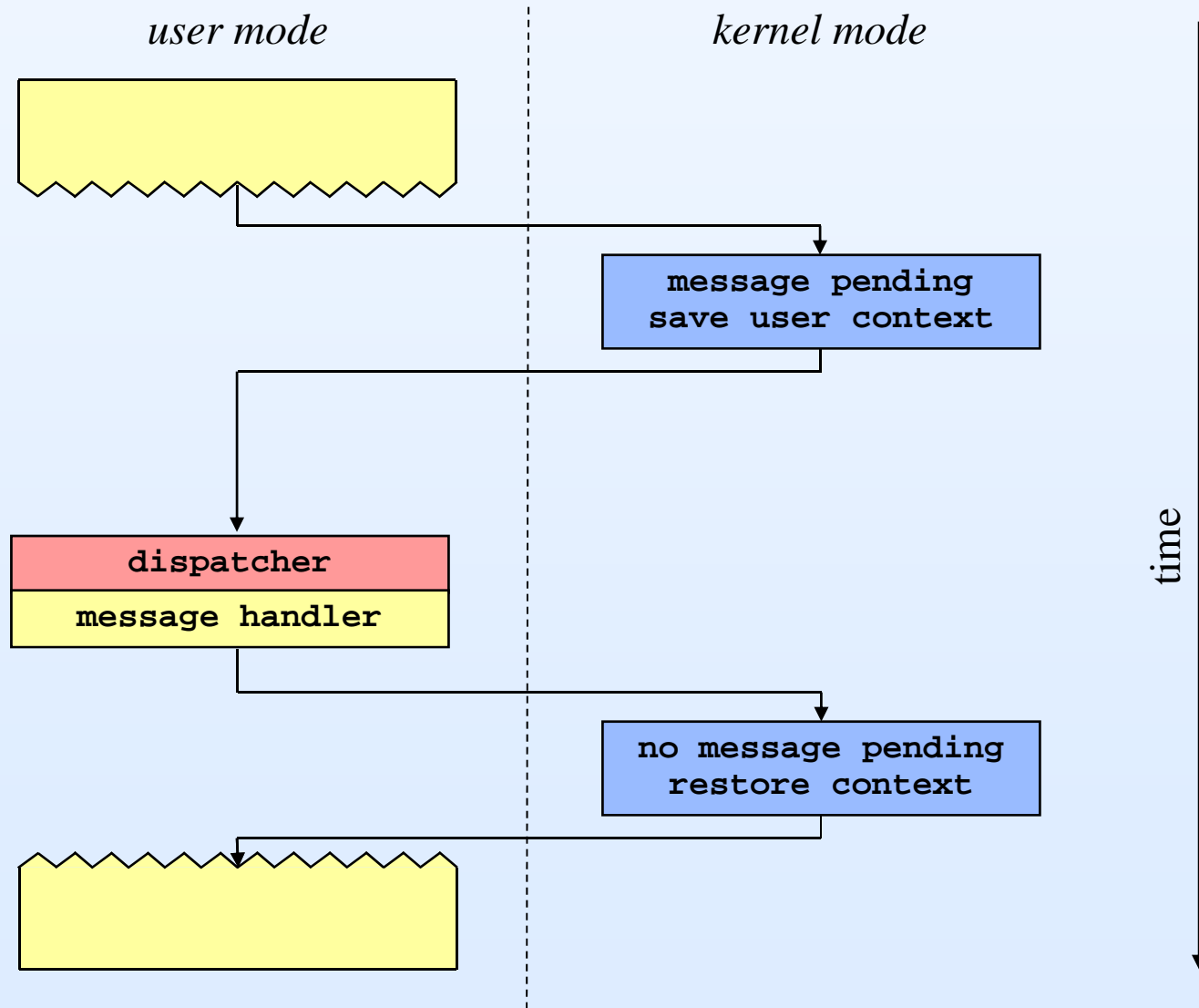
- Thread support
  - Thread-local storage
  - Callback-local storage
  - Simple mutexes
  - Read-write locks
  - Thread-private code caches, if requested
- Sideline support
  - Create new client-only thread
  - Thread-private itimer (Linux-only)

# Thread-Local Storage (TLS)

---

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

# Callback-Local Storage (CLS)



# Callback-Local Storage (CLS)

---

- Windows callbacks interrupt execution to process an event and later resume the suspended context
- TLS data from the suspended context will be overwritten during callback execution
- CLS data is saved at the interruption point and restored at the resumption point
- Whenever keeping persistent data across application execution, in particular system calls, use CLS instead of TLS
- Can be used for Linux signals as well
- Provided by the drmgr Extension

# DynamoRIO API: General Utilities, Cont'd

---

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

# DynamoRIO API: General Utilities, Cont'd

---

- Application inspection
  - Address space querying
  - Module iterator
  - Processor feature identification
  - Symbol lookup (currently Windows-only)



# Third-Party Libraries

---

- Key issue: maintain transparency
- Static libraries with -wrap support for heap
- ntdll.dll link support on Windows
  - Contains “mini-libc”
- Custom loader for private shared library copies

# Private Libraries

---

- Private loader on Windows since release 1.5.0
  - 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 is new in 2.2.0
  - Isolation is simpler and more complete

# DynamoRIO API Outline

---

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

# DynamoRIO API: Instruction Representation

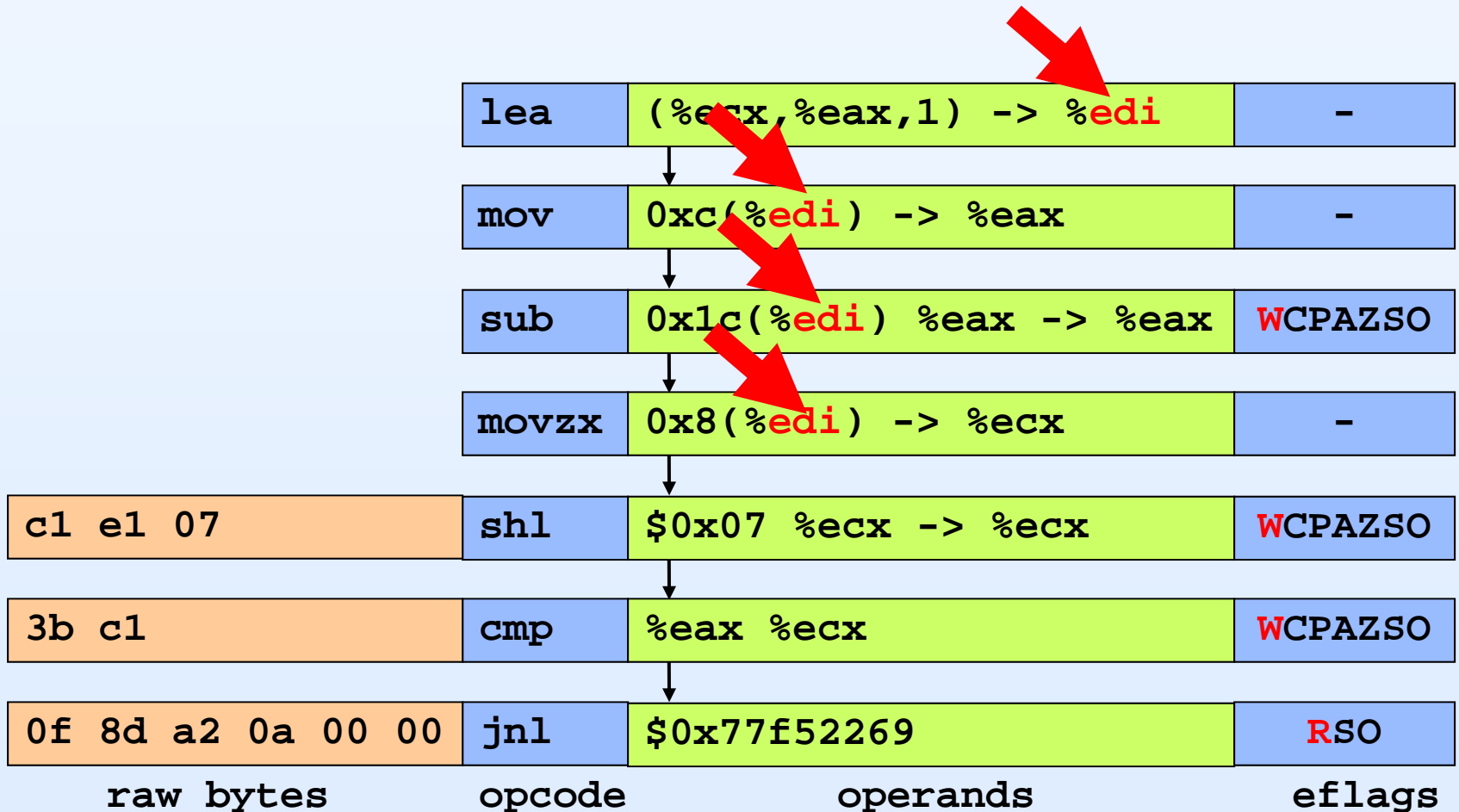
---

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

# Instruction Representation

8d 34 01	lea	(%ecx,%eax,1) -> %esi	-
8b 46 0c	mov	0xc(%esi) -> %eax	-
2b 46 1c	sub	0x1c(%esi) %eax -> %eax	WCPAZSO
0f b7 4e 08	movzx	0x8(%esi) -> %ecx	-
c1 e1 07	shl	\$0x07 %ecx -> %ecx	WCPAZSO
3b c1	cmp	%eax %ecx	WCPAZSO
0f 8d a2 0a 00 00	jnl	\$0x77f52269	RSO
raw bytes	opcode	operands	eflags

# 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(REG_EDX));
```

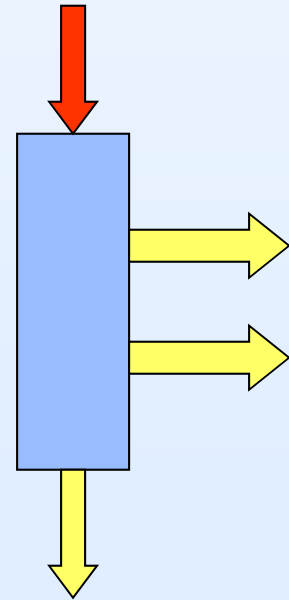
- Method 2: specify opcode + all operands (including implicit operands):

```
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(REG_EDX));  
instr_set_src(instr, 0, opnd_create_reg(REG_EDX));
```

# Linear Control Flow

---

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





# 64-Bit Versus 32-Bit

---

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

# 64-Bit Thread and Instruction Modes

---

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

# 64-Bit Clients

---

- Define X86\_64 before including header files when building a 64-bit client
- Convenience macros for printf formats, etc. are provided
  - E.g.:
    - `printf("Pointer is "PFX"\n", p);`
- Use "X" macros for cross-platform registers
  - REG\_XAX is REG\_EAX when compiled 32-bit, and REG\_RAX when compiled 64-bit

# DynamoRIO API: Code Manipulation

---

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

# Processor Information

---

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

# State Preservation

---

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

# Clean Calls

---

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

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

# Clean Call Inlining

---

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



# Dynamic Instrumentation

---

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

# Flushing the Cache

---

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

# Flushing the Cache

---

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

# Multi-Instrumentation Mediation

---

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

# Function Replacing and Wrapping

---

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

# DynamoRIO API Outline

---

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

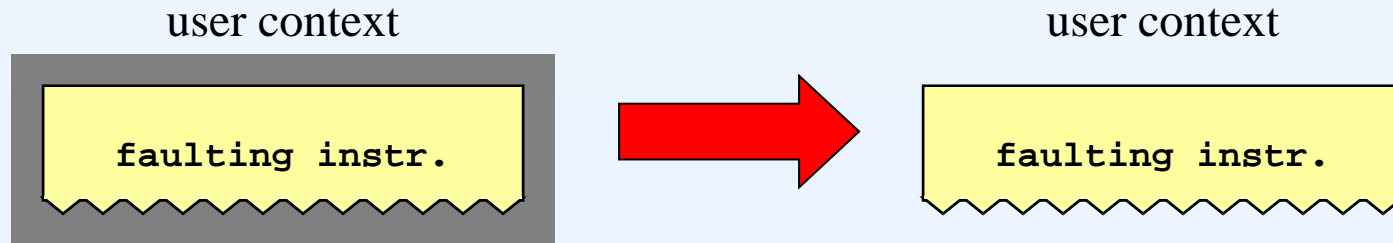
# DynamoRIO API: Translation

---

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

# Translation Case 1: Fault

---



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



# Translation Case 2: Relocation

---

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

# Translation Approaches

---

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

# Instruction Translation Field

---

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

# Context Translation Via Re-Creation

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

```
C1: mov  %ebx, %ecx
C2: add  %eax, (%ecx)
C3: cmp  $4, (%ax)
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>
```

# Meta vs. Non-Meta Instructions

---

- Non-Meta instructions are treated as application instructions
  - They must have translations
  - Control flow changing 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)
  - Cannot fault, so translations not needed
- Meta-may-fault instructions
  - Can fault, but should not be “interpreted”: won’t modify app code
  - Fault typically deliberate and handled by client
- `Xref instr_set_ok_to_mangle()` and `instr_set_meta_may_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 meta and non-meta 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 non-meta instructions and all meta-may-fault instructions
  - 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 non-meta instructions and all meta-may-fault instructions
- Client bb or trace hook will not be re-called with `translating==true`



# Register State Translation

---

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

# Client Register State Translation

---

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

# DynamoRIO API Outline

---

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

# DynamoRIO versus Pin

---

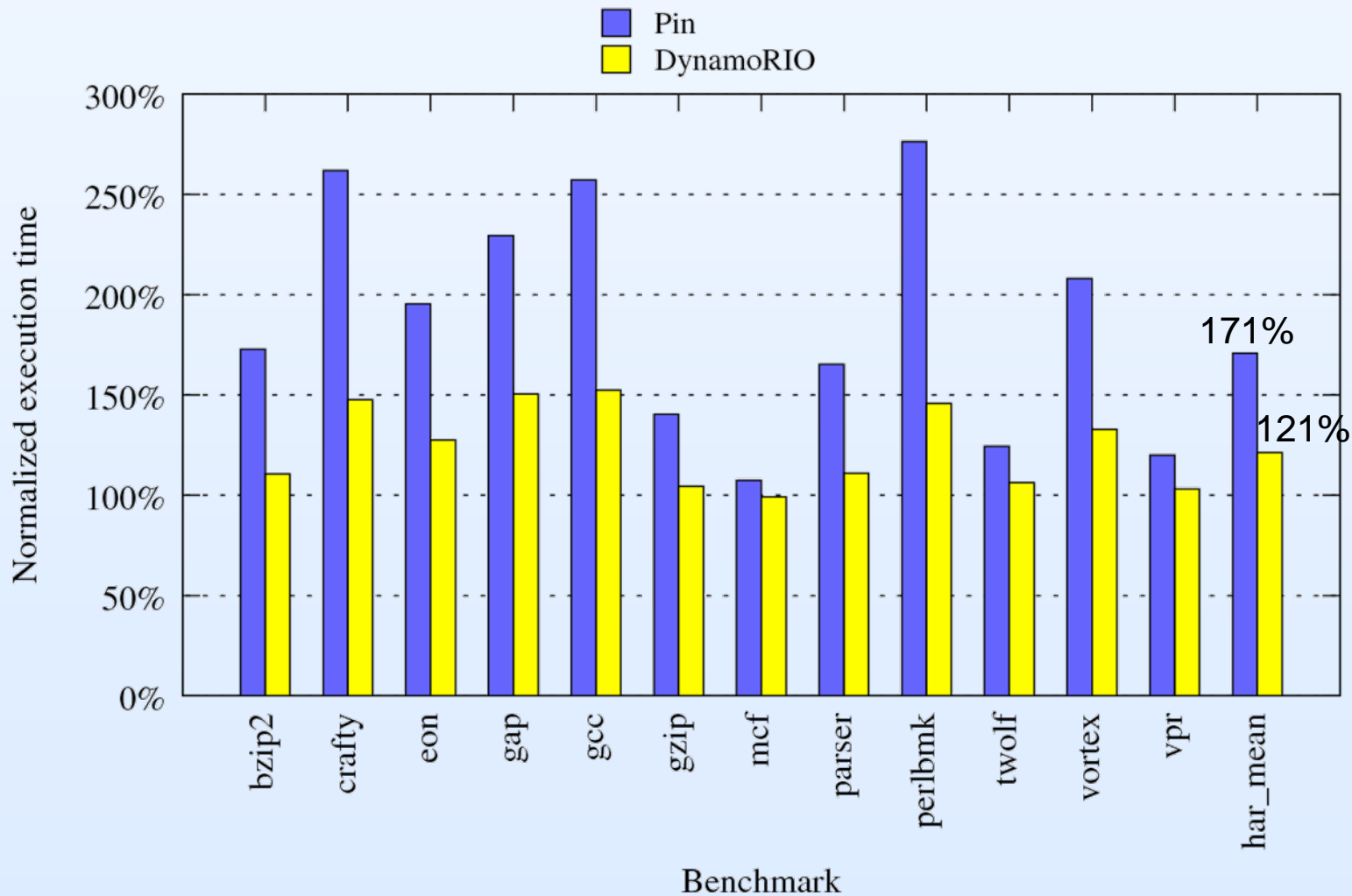
- Basic interface is fundamentally different
- Pin = insert callout/trampoline only
  - Not so different from tools that modify the original code: Dyninst, Vulcan, Detours
  - Uses code cache only for transparency
- DynamoRIO = arbitrary code stream modifications
  - Only feasible with a code cache
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

# DynamoRIO versus Pin

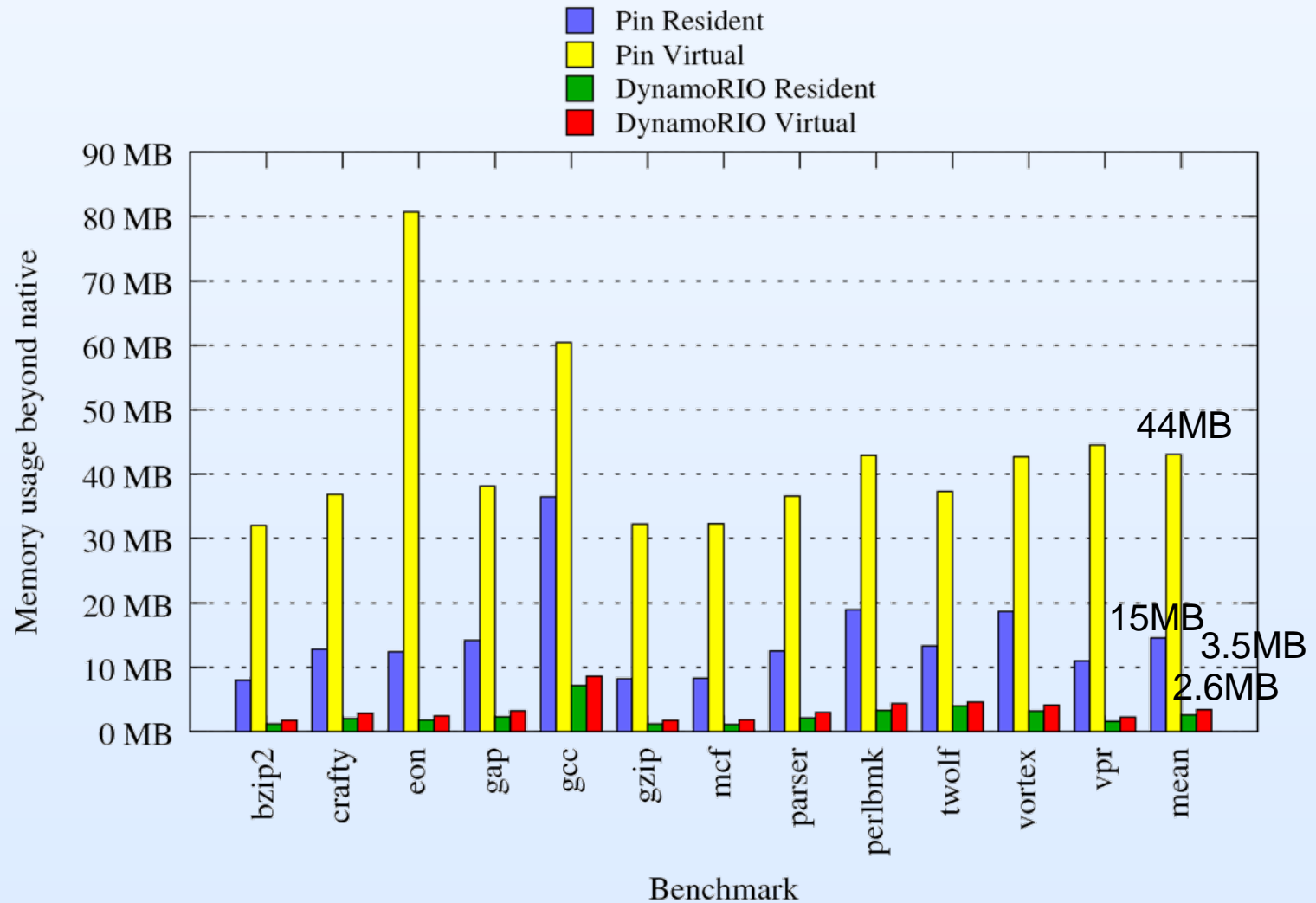
---

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

# Base Performance Comparison (No Tool)



# Base Memory Comparison



# 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;
}
```



# BBCount DynamoRIO Tool

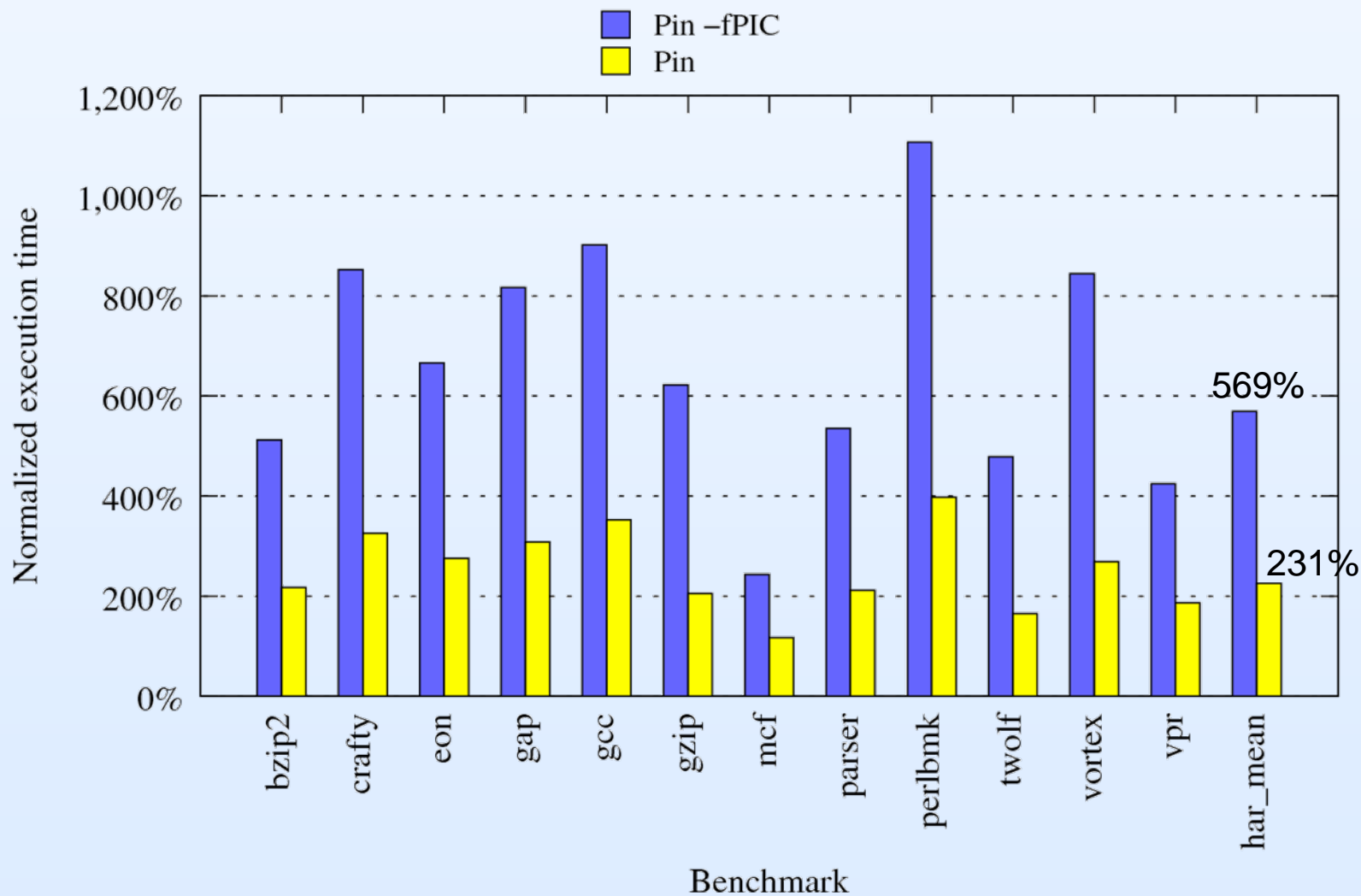
---

```
static int global_count;

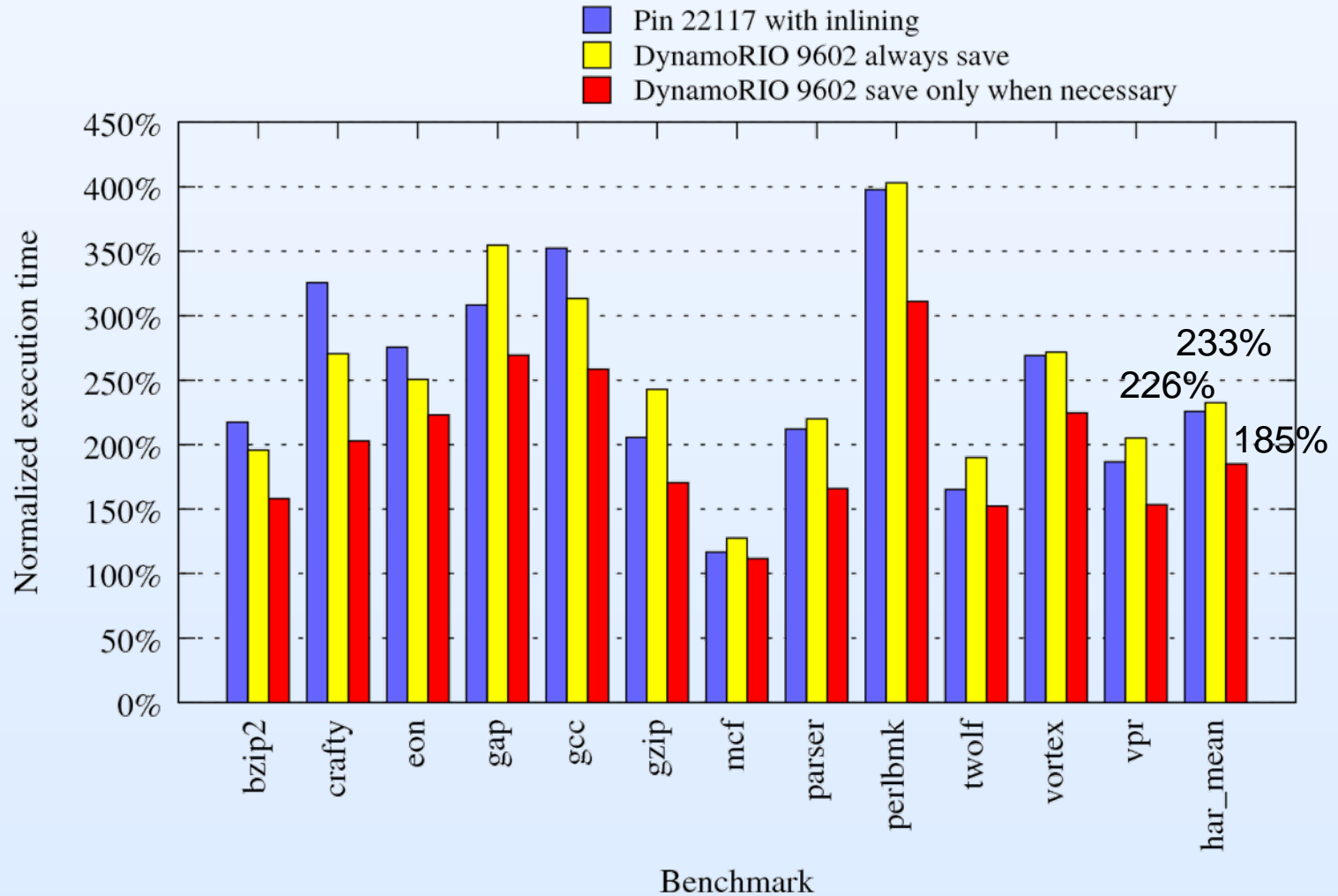
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating) {
    instr_t *instr, *first = instrlist_first(bb);
    uint flags;
    /* Our inc can go anywhere, so find a spot where flags are dead.
     * Technically this can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

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

# BBCount: Pin Inlining Importance



# BBCount Performance Comparison



# DynamoRIO API Outline

---

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

# Obtaining Help

---

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

# Debugging Clients

---

- Use the DynamoRIO debug build for asserts
  - Often point out the problem
- Use logging
  - -loglevel N
  - stored in logs/ subdir of DR install dir
- Attach a debugger
  - gdb or windbg
  - -msgbox\_mask 0xN
  - -no\_hide
  - windbg: .reload myclient.dll=0xN
- More tips:
  - <http://code.google.com/p/dynamorio/wiki/Debugging>

# Reporting Bugs

---

- Search the Issue Tracker off <http://dynamorio.org> first
  - <http://code.google.com/p/dynamorio/issues/list>
- File a new Issue if not found
- Follow conventions on wiki
  - <http://code.google.com/p/dynamorio/wiki/BugReporting>
  - CRASH, APP CRASH, HANG, ASSERT
- Example titles:
  - CRASH (1.3.1 calc.exe)  
vm\_area\_add\_fragment:vmareas.c(4466)
  - ASSERT (1.3.0 suite/tests/common/segfault)  
study\_hashtable:fragment.c:1745 ASSERT\_NOT\_REACHED

# Changes From Prior Releases

---

- Backward compatible with 1.0 (0.9.6) and above
  - Except configuration and deployment scheme and tools: switched to file-based scheme to support unprivileged and parallel execution on Windows
- Not backward compatible with 0.9.1-0.9.5



# Demonstrations

2:00-2:10	Welcome + DynamoRIO History
2:10-3:10	DynamoRIO Internals
3:10-4:00	DynamoRIO API
4:00-4:15	<i>Break</i>
4:15-5:15	Demonstrations
5:15-5:45	Lab Session
5:45-6:00	Feedback