UNIVERSITY OF
WISCONSIN
MADISON

Instrumentation Technology Update

Brian J. N. Wylie *et al.*
`wylie@cs.wisc.edu`

Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685
USA

Dyninst vision

Dynamically instrument

- ***anything*** that can be called/coded
- ***anywhere*** in process address space
- ***anytime*** it’s not active, or blocked in syscall
- ***efficiently*** as it’s possible to make it
- ***safely*** if you know what you’re doing

Outline

- Dynamic instrumentation vision
- Tour of selected technology developments
  - Retroactive “catch-up” instrumentation
  - System-call interruption/resume
  - Instrumentation trap handling
  - Function relocation/rewriting/expansion
  - Instrumentation recursion guards
  - Virtual timers

“the stuff that couldn’t be put off any longer”

Dynamic instrumentation 101

When instrumentation of a function requested

- at key “inst-points”, function patched with detours to instrumentation basetramps & minitramp-chains
- subsequent execution includes instrumentation (until it is removed when no longer required)

- Typical profiling instrumentation

@entry: set/increment flag, start timer

@exits: unset/decrement flag, stop timer

@calls: stop timer before call;
re-start after call

Instrumentation points

Location	*When*	Semantic
Entry	***pre***	• First instruction in function
	post	• First instruction in function after activation record created
Call *i*	***pre***	• Last instruction before call
	post	• First instruction after call
Exit *x*	***pre***	• Last instruction in function before activation record destroyed
	post	• Last instruction in function

Instrumentation assumptions

- Instrumentation relations:
  - A.entry < A.pre-call(B) < A.post-call(B) < A.exit
  - A.pre-call(B) < B.entry < B.exit < A.post-call(B)
  - no other relations supported (though definable)
- Instrumentation scenarios:
  - function to be instrumented is not on stack
  - function is within body of stack
  - function is currently top of stack (contains %pc)

Problem case: functions on stack

- Instrumentation will be in an inconsistent state for partially-executed functions, e.g.:
    - @exit stop timer has no matching timer started
    - state flags haven’t been initialized @entry/@pre-call
- Postponing instrumentation until current function instance completes is an option, but
    - effectively lose remainder of current execution
    - may not complete or re-execute soon (e.g., *main*)
    - generally cripples callgraph-based PC search!

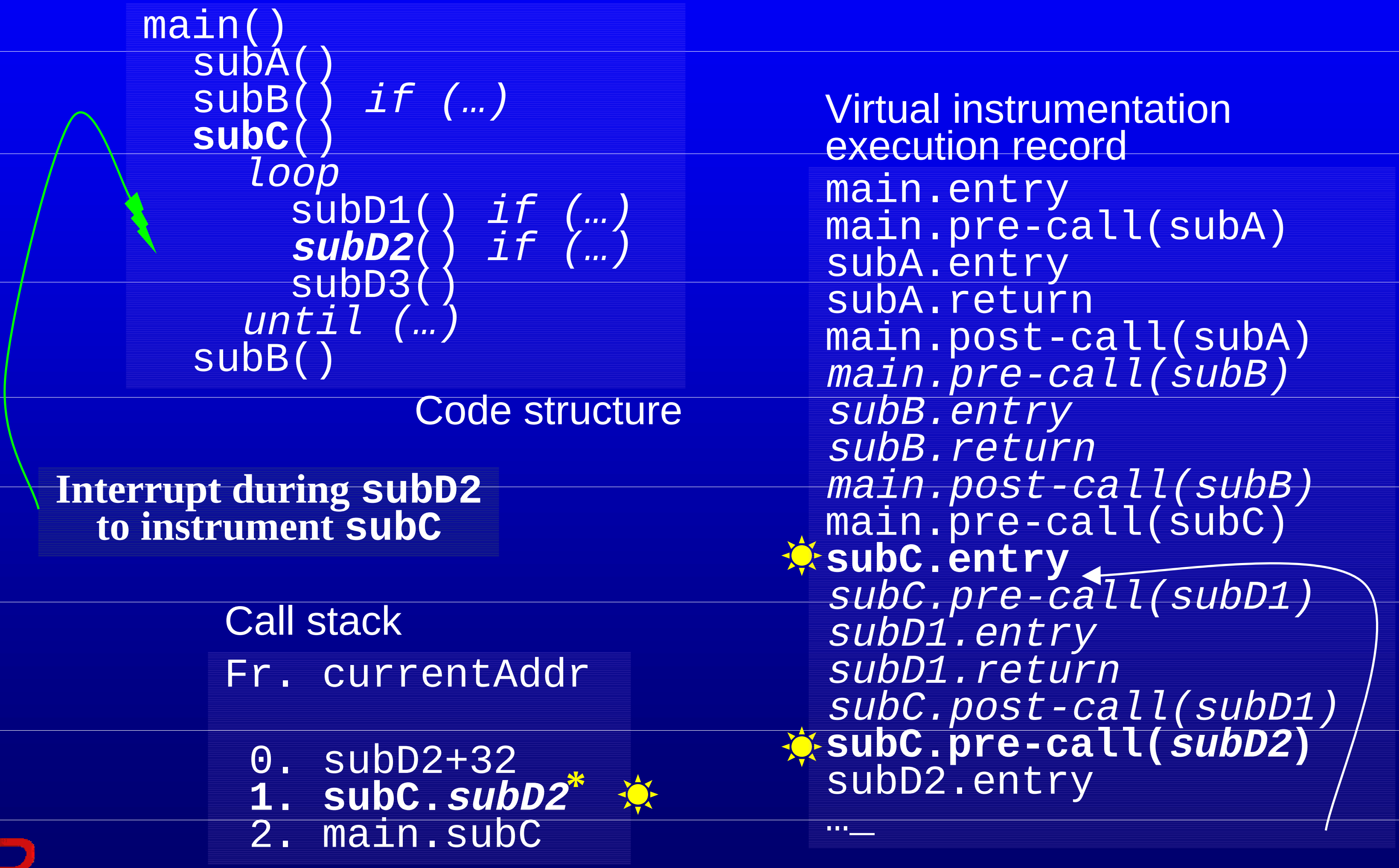
Retroactive instrumentation

- Provides illusion of pre-instrumented function with context set for subsequent execution
- Execute instrumentation which can guarantee would have been executed
  - examine call-stack for residual evidence
- Approximate past times with best estimates available (*i.e.*, current time)

Stack function instrumentation

- Functions currently on the stack need very careful instrumentation
    - function entry and active callee pre-call instrumentation should be executed immediately
        - use one-time-code (*aka* inferiorRPCs)
        - set flags, start timers, etc. (instrumentation context)
    - function return addresses on stack should be updated to return via base trampolines which contain post-call instrumentation
    - other instrumentation can be freely inserted

Retroactive instrumentation example



```
main()
   subA()
   subB() if (…)
   subC()
      loop
         subD1() if (…)
         subD2() if (…)
         subD3()
      until (…)
   subB()
```

Code structure

Interrupt during subD2 to instrument subC

Call stack

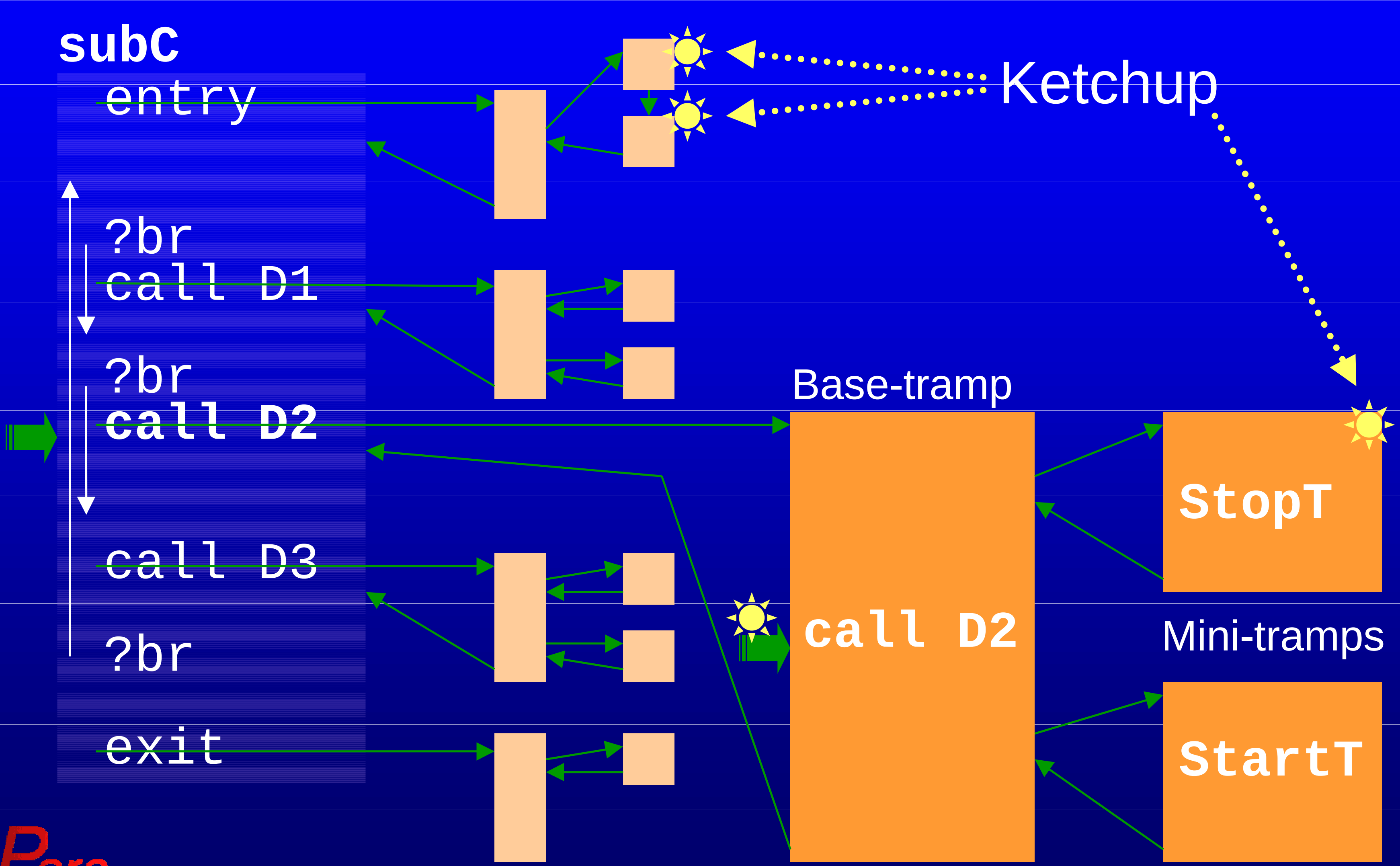
```
Fr. currentAddr

 0. subD2+32
 1. subC.subD2*  ☼
 2. main.subC
```

Virtual instrumentation execution record

```
 main.entry
 main.pre-call(subA)
 subA.entry
 subA.return
 main.post-call(subA)
 main.pre-call(subB)
 subB.entry
 subB.return
 main.post-call(subB)
 main.pre-call(subC)
☼subC.entry
 subC.pre-call(subD1)
 subD1.entry
 subD1.return
 subC.post-call(subD1)
☼subC.pre-call(subD2)
 subD2.entry
 …_
```

Dyninst plumbing diagram



Retroactive instrumentation walk-thru

- Pause/interrupt process execution with request to instrument function `subC` (@entry,@calls,@exits)
- Stack-walk finds `subC` on the stack (in frame 1)
- Function `subC` instrumented as specified
- Return address of `subC`-callee `subD2` updated with `subC`.call(`subD2`) basetramp post-call location
- Retroactive execution of @entry & @pre-call(`subD2`) instrumentation of `subC` determined necessary to construct virtual instrumentation record/state
- Instrumentation complete, continue process execution

Advanced ketchup

- If can’t instrument *all* of a function with requested instrumentation, don’t instrument *any* of it
- If can’t retroactively execute *all* instrumentation for *all* essential points in function, don’t run *any* of it
  - (and don’t instrument function either)
  - execute ketchup instrumentation in “virtual record” order
- If *%pc within* an instrumentation footprint, relocate it to the corresponding instruction in the basetramp
- Update returning destinations of *callee(s)* on stack
  - should return to appropriate post-call basetramp location such that post-call instrumentation will be executed

Advanced ketchup (cont'd)

- Stack-walk must understand already-instrumented functions (with their basetramps & minitramps)
- Don't execute retroactive instrumentation that will be executed in the now-instrumented base function when the process continues
  - check prepend/append conditions vs. current location
- State for context-dependent instrumentation must be reconstructed for its retroactive execution
  - *e.g.*, when a snippet accesses function call arguments or local variables from the stack, these must be restored or appropriate acquisition code incorporated

Instrument anytime

- Reliably instrument active functions (*i.e.*, those on the call-stack)
  - these are generally the most interesting functions for execution/performance analysis
  - requires retroactive instrumentation activation to ensure consistency
- Execute actions promptly
  - Current program execution, including system calls, must be temporarily interrupted

System-call interruption/resume

- Need to interrupt application’s system calls to run inferiorRPCs, during attach, ketchup, etc.
  - select(), sleep(), wait(), ...
- Solution:
  - Solaris & Irix have */proc* interrupt mechanism; syscall resumes/restarts when execution continues
  - Linux, AIX, WindowsNT require investigation
  - Workaround on Linux awaits completion of system-call before execution of inferiorRPC

Basic instrumentation challenges

- Address spaces are too vast for 1-inst jumps
  - fast/compact jumps have insufficient reach
  - multiple instruction jump sequences required
- Some available instrumentation techniques are highly intrusive
  - use of traps (often extremely inefficiently handled)
- Some functions can’t be instrumented in-situ
  - too compact or convoluted (i.e., highly optimized)

Instrumentation-trap handling

- On x86 platforms, single byte trap instructions required for tight instrumentation points
- Signal handler uses address of trap to lookup and jump to destination base-trampoline
- *sigaction* instrumented to register application’s `SIGTRAP` handlers for execution only with non-instrumentation traps
- Interrupt signal delivery varies by platform

Solaris signal-handling

- Use */proc* to mask forwarding trap signals for efficient handling directly in inferior process
  - Signal may be delivered and instrumentation signal handler started at any time
  - Handler needs to defer to started inferiorRPCs (which are ‘registered’ prior to execution)
  - Upon inferiorRPC completion, execution will continue with re-executing & handling the trap

Linux signal-handling

- Signals delivered to attached ‘debugger’ returned to inferior process for handling
  - Round-trip routing and context switches result in low efficiency and high daemon overheads
- Daemon/mutator detaching will allow traps to be efficiently handled in inferior process
  - Daemon/mutator needs to temporarily re-attach to perform instrumentation, etc.

Windows NT signal-handling

- Signals always delivered to debugger/daemon
  - Costly context-switches involved
  - Resulting poor performance
- Debuggee always terminated on detach
  - No hope of efficient instrumentation trap handling
- Function rewriting with expansion required to avoid use of traps to reach instrumentation

Function relocation & expansion

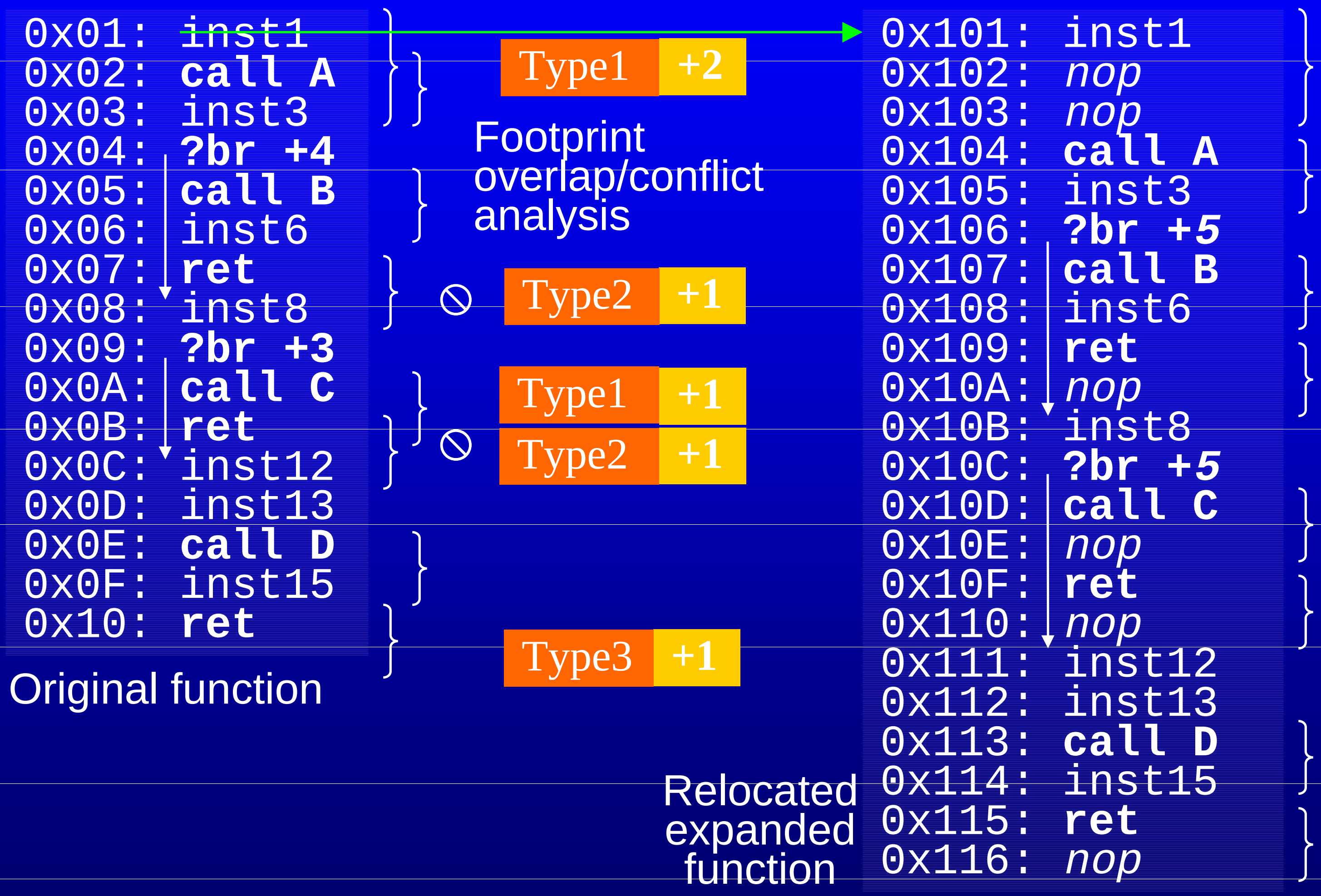
- Copy of original function relocated to heap, selectively de-optimized, and rewritten with extra space provided for instrumentation
  - tease apart optimized call-returns (“tail-calls”) and overlapping instrumentation point footprints to allow each to be individually instrumented
  - provide extra space for footprints which overrun the end of the function or basic block
- Original function rewritten to branch to new

Reasons for relocation/expansion

1. Instrumentation footprints would overlap
2. Instrumentation footprint internally contains a branch target (i.e., crosses a basic block boundary)
3. Instrumentation footprint would extend past the end of function

- Previously, these would all have resulted in functions considered “uninstrumentable”

Relocation/expansion example



```
0x01: inst1
0x02: call A
0x03: inst3
0x04: ?br +4
0x05: call B
0x06: inst6
0x07: ret
0x08: inst8
0x09: ?br +3
0x0A: call C
0x0B: ret
0x0C: inst12
0x0D: inst13
0x0E: call D
0x0F: inst15
0x10: ret
```

Original function

Type1 +2

Footprint overlap/conflict analysis

⊘ Type2 +1

Type1 +1

⊘ Type2 +1

Type3 +1

```
0x101: inst1
0x102: nop
0x103: nop
0x104: call A
0x105: inst3
0x106: ?br +5
0x107: call B
0x108: inst6
0x109: ret
0x10A: nop
0x10B: inst8
0x10C: ?br +5
0x10D: call C
0x10E: nop
0x10F: ret
0x110: nop
0x111: inst12
0x112: inst13
0x113: call D
0x114: inst15
0x115: ret
0x116: nop
```

Relocated expanded function

Relocation/expansion benefits

- New function can be (safely) instrumented more thoroughly
  - more points (and entire functions!) become instrumentable, potentially even every instruction
- New function can be (safely) instrumented more efficiently
  - more space for larger instrumentation footprints avoids the need to use costly traps

Rewriting requirements

- Function expansion/rewriting must preserve execution semantics
  - retain expected order of execution
  - set context for de-optimized sequences
  - adjust branches/jumps affected by expansion and relocation of targets
- Allocate sufficient heap space for expanded function (near function or instrumentation)

Trampoline recursion guards

- Dyninst supports arbitrary instrumentation
  - Instrumentation can call other functions or make system calls
- Instrumentation can therefore end up calling itself (directly or more usually indirectly)
- The results are usually unintended
  - infinite loops, resource exhaustion, ...

Simple motivating example

```
void instrumentation ()
{
  // Do something ...
  if (error) printf(“There was an error\n”);
}
```

- If *printf()* is instrumented with a call to this function, then any circumstance which sets error will cause an infinite loop
- If any function *printf()* calls, such as *write()*, is similarly instrumented, same net result
- These errors can be extremely subtle

Guard implementation

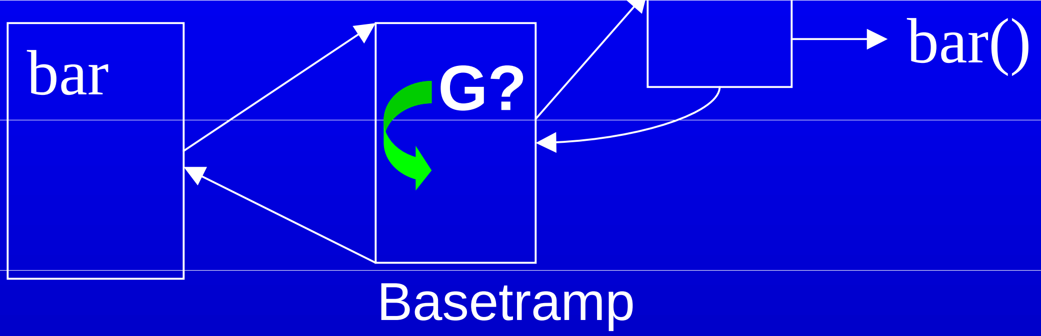
- Guard uses a flag in the inferior heap and extra instructions in the base-tramp

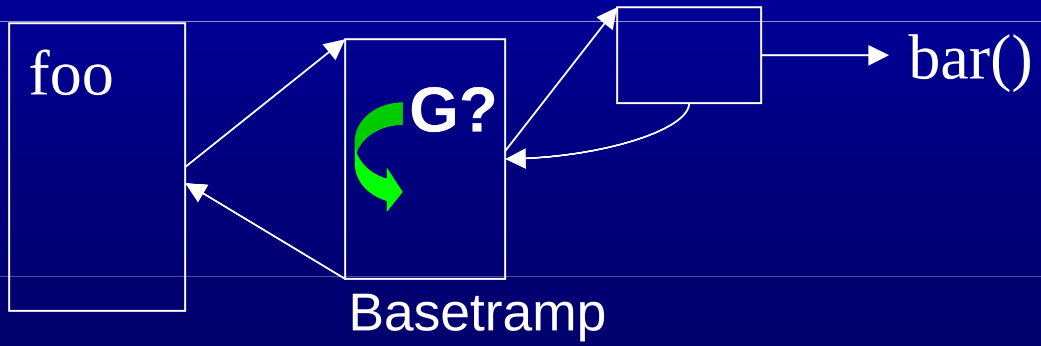
Fragment of guarded base-tramp

```
<save registers>
**<if guard flag is set, jump to POST>**
**<set guard flag>**
<execute mini-tramp>
**<unset guard flag>**
**POST:**
<restore registers>
```

Recursion guard example

- Function *foo()* is instrumented with a call to *bar*
- Function *bar()* is also instrumented with a call to *bar*
- With the tramp guards, inner instrumentation will not be executed, i.e., *foo()* calls *bar*, but guard prevents recursive call to *bar*





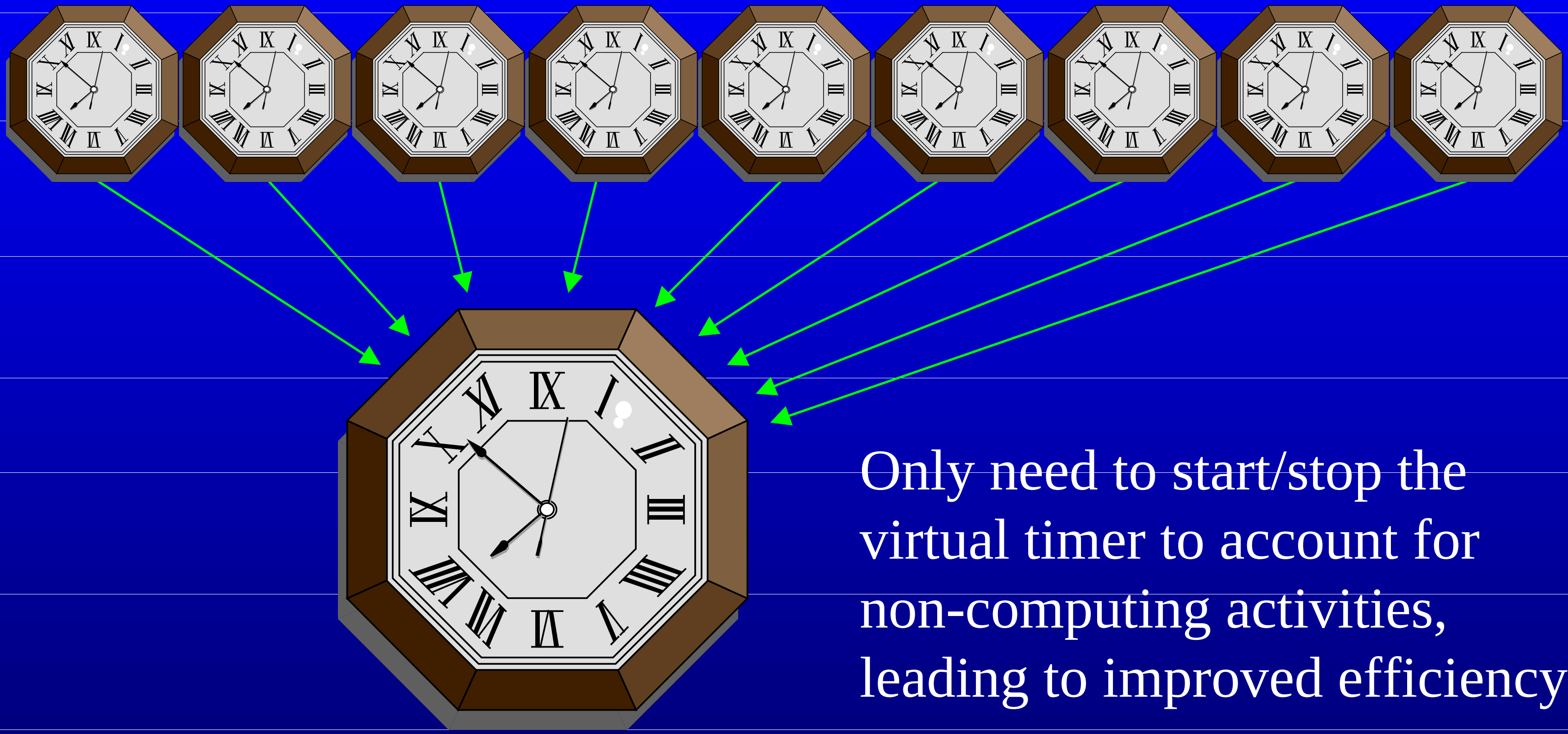
Trampoline guard benefits

- Trampoline guards prevent mini-tramps from being executed when the function/base-tramp was reached (via snippet code in a mini-tramp) from inside a base-tramp
- Avoids instrumentation recursively calling itself or any other instrumentation
- Provides additional safety and flexibility with dynamic instrumentation
- Guards can be disabled/removed for extra speed

Virtual timers

- Want to exclude all time spent in unproductive non-computing activities:
  - Synchronizations that use busy waiting (e.g., MPI send, MPI receive, Spinlock)
  - Performance tool activities (e.g., sampling, flushing)
  - Thread queuing
- Build each metric instance timer on top of (per-thread) virtual timers
  - Logically simpler and cleaner implementation
  - More efficient since only need to start/stop virtual timers instead of lists of individual metric timers

Virtual timer replaces many actual timers



Only need to start/stop the virtual timer to account for non-computing activities, leading to improved efficiency

Dyninst revision

Dynamically instrument

- ***anything*** that can be called/coded
- ***anywhere*** in process address space
- ***anytime*** ~~it’s not active, or blocked in syscall~~
- ***efficiently*** (further improvements in progress)
- ***safely*** (remove guards at own risk)


Now with
Ketchup!
Paradyn™
dyninst
inside