Binary Code Patching: An Ancient Art Refined for the 21st Century

Barton P. Miller

bart@cs.wisc.edu
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

Triangle Computer Science Lecture
UNC/NCSU/Duke

Duke UNIVERSITY

Paradyn

Overview

Code patching is an ancient (or at least, old) art.

The last 25 years have produced some significant advances, bringing code patching from an mystical art to an everyday tool.

I'll talk about:

- Some old history (or nostalgia).
- Binary Rewriting
- Dynamic Instrumentation: application and kernel
- Control-flow instrumentation

1961: The DEC DDT Debugger

Allowed code patching of the image in memory.

Used on the DEC PDP-1 mini-computer.

DDT = "DEC Debugging Tape".

Descendent of the FLIT debugger on the 1957 TX-0 computer from MIT Lincoln Labs.

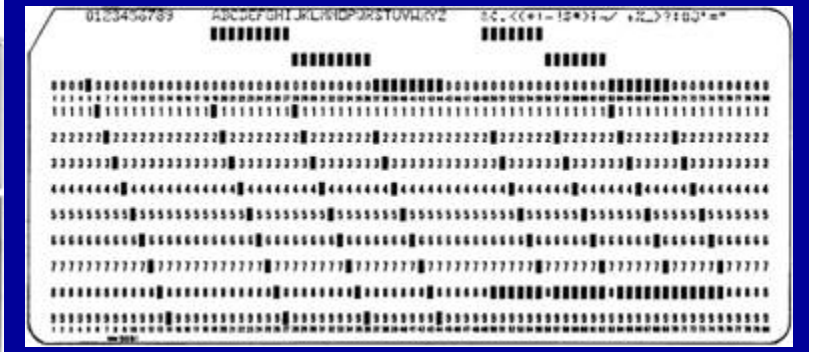
The "Tape"

1972: Hand-Patching a Binary

The time to read a card deck, re-assemble and re-link could be lengthy:

E.g., a Basic compiler on the IBM 1130

- 400 card/minute x 5000 cards = 12 minutes
- At least 10 minutes to assemble
- Similar amount of time to link.



1972: Hand-Patching a Binary

Leave patch areas (BSS's) scattered throughout the code (e.g., between functions).

When the program was in memory (core!), hand assemble code fixes in the patch area.

Overwrite an instruction in the original code to jump to the patch.

Often need to include the overwritten instruction in the patch area.

1989: Pixie

In 1989, DEC (Earl Killian) introduced Pixie, a tool for inserting simple instrumentation code into a binary.

- Simple profiling: basic counts.
- Expensive instrumentation:
  - Run-time processing of indirect jumps and calls, extensive register saves.
- Performance data used to feedback to optimizing compiler.
- Notable as the earliest binary instrumenter.
- Operated on the MIPS/Ultrix.

1992: A Burst of New Activity

In 1992, code patching became a general purpose technique, making the transition from hack to technique:

- Binary rewriting: take an object file as input and produce a new, modified object file as output. Operates before execution.
  Pioneer system: OM from DEC (Srivastava and Wall)

- Dynamic instrumentation: modifying a program while it is executing.
  Pioneer system: Paradyn/Dyninst from Wisconsin (Miller and Hollingsworth)

Binary Rewriting

Benefits:

- Do not need the source code.
- Can do fine-grained instrumentation.
- Sometimes do not need symbol tables.

Drawbacks:

- Static: if you want to change the instrumentation, you have to re-process the binary.
- Only one or a few platforms supported.
- Often requires "managed" code.
- Scarcity of tools.

Examples of use:

- Performance profiling.
- Tracing: addresses, basic blocks, functions . . .
- Sandboxing: modifying a binary to restrict its behavior (IDS).
- Simulation: intercepting operations to simulate different ones.
- Memory debugging: checking dynamic allocate/free and memory references for correct operation.
- Code optimization: optimizing based on the binary code.

Dynamic Instrumentation

Benefits:

- Do not need the source code.
- Can change the code as the program executes.
- Can instrument running programs (such as servers).
- Sometimes do not need symbol tables.

Drawbacks:

- More complex than static rewriting: execution state of the program restricts the kind of changes you can make.
- Few tools.

Examples of use:

- Performance profiling.
- Tracing: similar to rewriting, though more expensive per op.
- Cyber forensics: controlling suspicious code.
- Memory debugging: checking dynamic allocate/free and memory references for correct operation.
- Dynamic code optimization.

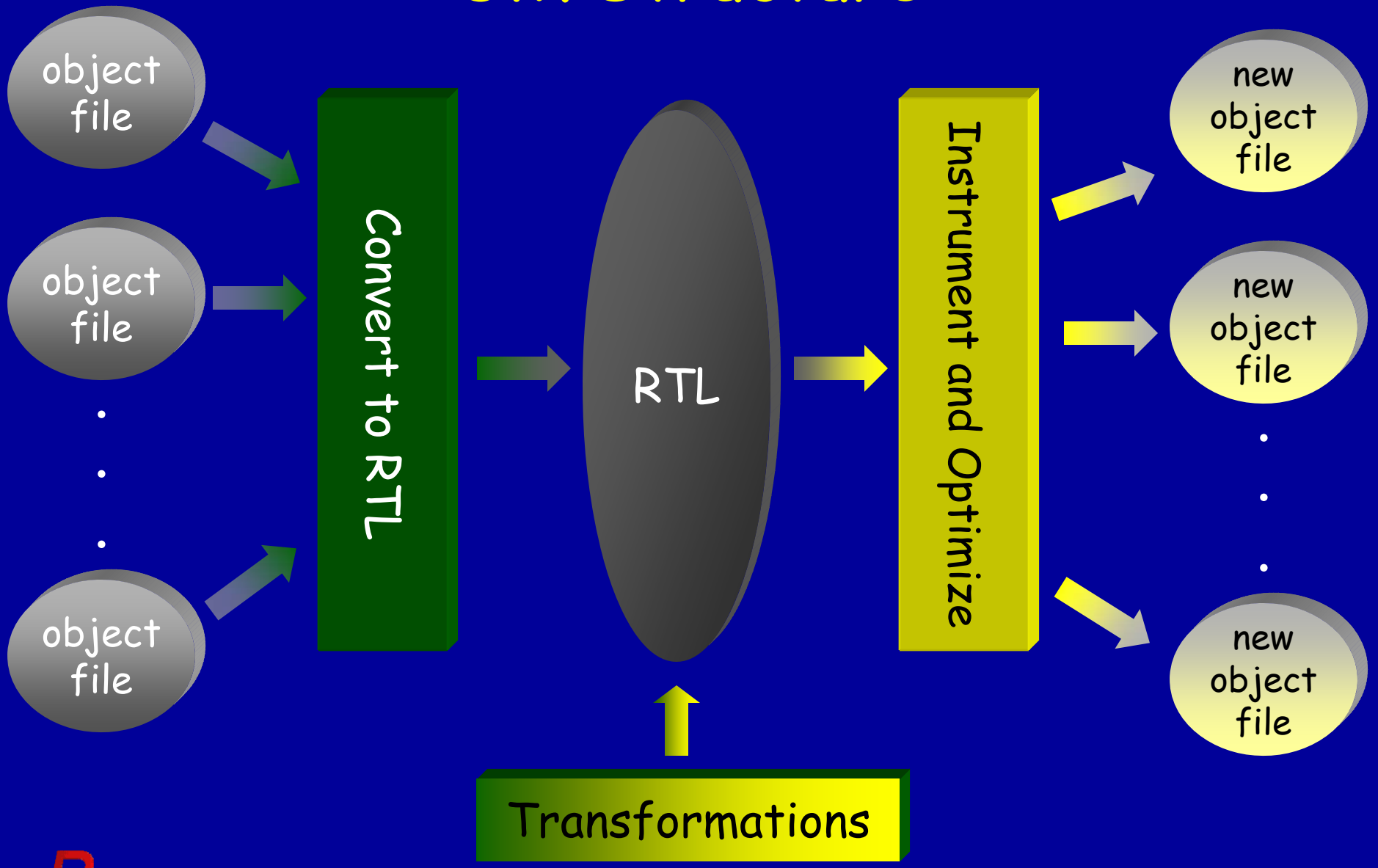
The OM Binary Rewriter

- The first general purpose rewriter.
- Requires symbol tables.
- Supports some interesting analyses and optimizations, e.g.:
  - Live variable analysis.
  - Dead code elimination.
  - Hoisting code out of loops.
- DEC Alpha processor.
- Basis for the ATOM tracing tool
  - Allows inserting calls almost anywhere in the program.
- Works on object (.o) files, not executables (a.out/.exe).

Side Note: Object Files vs. Executables

- Object files contain relocation information:
  - + All external address references (code and data) are labeled.
  - + Labeling simplifies one of the hardest parts of binary analysis: identifying and understand embedded addresses.
  - + With this address reference information, moving code around is much simpler.
  - - Requires you to have the object files.
- Executable files do not contain this information.
  - - Must work harder to operate on executables because less (possibly no) information is available in symbol tables.
  - - Restricts the techniques you can use for inserting or modifying code: efficiency, not correctness issue.
  - + Can operate on a much larger variety of programs.

OM Structure



Some Other Binary Rewriters

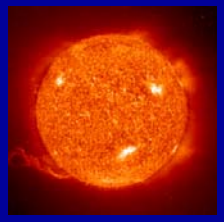
- EEL, 1995 (University of Wisconsin). SPARC/Solaris.
  - Operates on executables and needs symbol table.
- Etch, 1997 (University of Washington). IA32/Windows.
  - Operates on object files.
- LEEL, 1999 (University of Singapore). IA32/Linux.
  - EEL interface on IA32.
  - Operates on executables; doesn't need symbols.
  - Limited implementation.

Some Other Binary Rewriters (con'd)

- Vulcan, 2001 (Microsoft Research). {IA32,IA64,MSIL}/Windows.
  - Requires managed (Visual Studio) object files. Includes some dynamic features.
- BIT, 2003 (University of Arizona). IA32/Windows.
  - Based on the PLTO ('01) link-time optimizer.
- FIT, 2004 (University of Ghent). Alpha, ARM, IA32, IA64, MIPS.
  - Requires special compiler to produce annotated object files. Based on the DIABLO link-time optimization engine.

Specialize Rewriter Tools

- QPT, 1992 (Univ. of Wisconsin)
  - Original rewriting engine for EEL. Operates on executables and need symbols.
  - Counts execution frequency of basic blocks and control flow edges.
- ALTO/PLTO/ILTO, 1991, 2001, 2002 (Univ. of Arizona)
  - Link-time code optimizers for Alpha, IA32 and IA64: dead code elimination, constant propagation, function in-lining, and load/store forwarding.
- Squeeze, 2002 (Univ. of Arizona and Ghent)
  - Link-time code compression, with execution-time code expansion.
- DIABLO, 2004 (Univ. of Ghent).
  - Link-time optimizer, also used for code compression.



Squeeze

DIABLO
Diablo is A Better Link-time Optimizer

Dynamic Instrumentation: Do It On-the-Fly

Running programs are objects to be easily manipulated. Kinds of manipulations might include:

- Instrumentation
- Analysis
- Control

The Vehicle:The Dyninst API

A machine-independent library for machine level code patching.

- Eases the task of building new tools.
- Provides the basic abstractions to patch code on-the-fly

Truth in advertising: the idea of a clean, portable interface was a post-implementation realization.

Dynamic Instrumentation

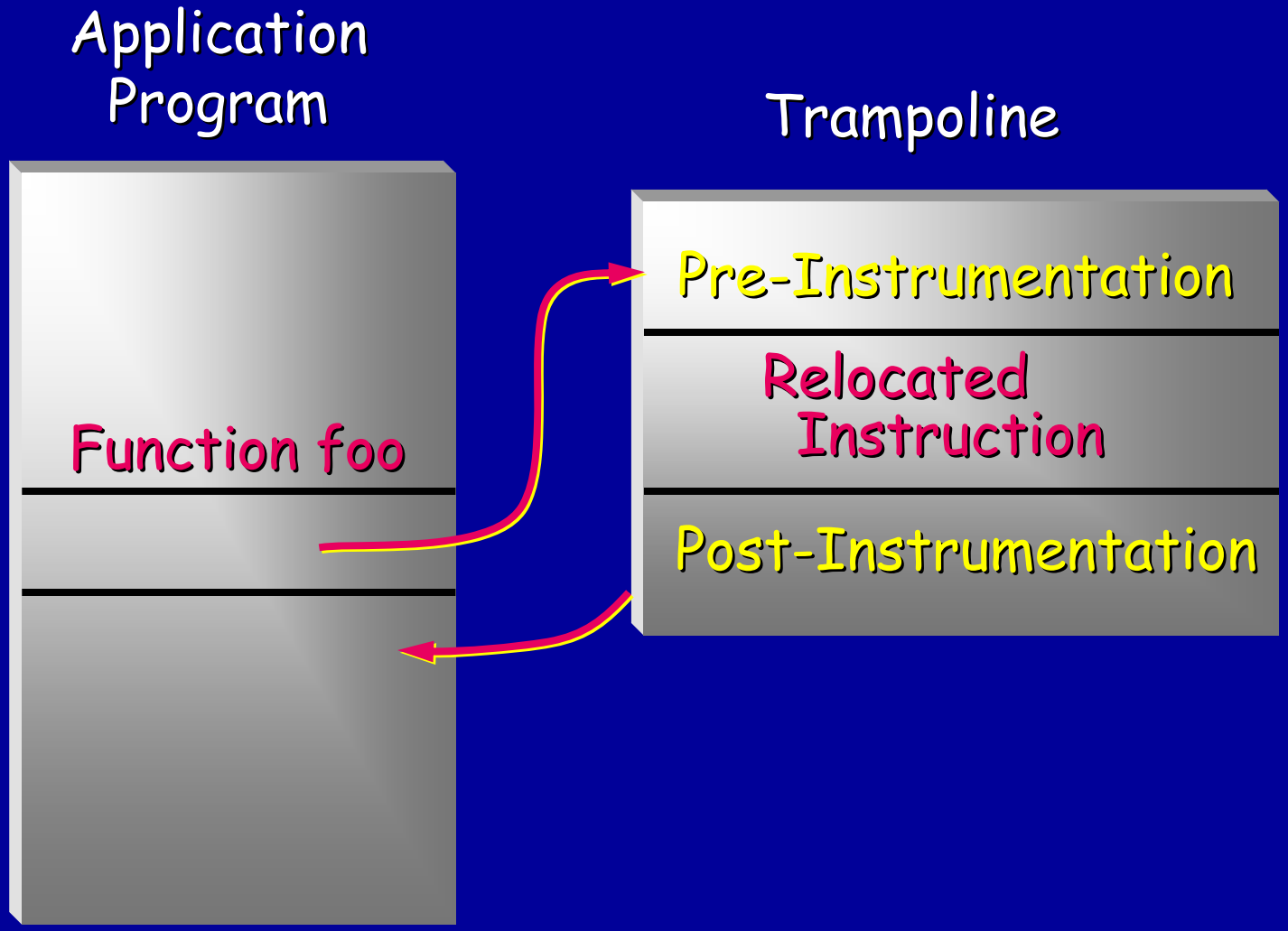
A familiar list:

- Does not require recompiling or relinking
  - Can instrument without the source code (e.g., proprietary libraries).
  - Can instrument without linking (relinking is not always possible.
  - Saves time: compile and link times can be significant in real systems.
- Instrument optimized code.
- Instrument stripped code.
  - Linux utilities.
  - Cyber forensics

Dynamic Instrumentation (con'd)

- Only instrument what you need, when you need
  - No hidden cost of latent instrumentation.
  - Enables "one pass" tools.
- Can instrument running programs (such as Web or database servers)
  - Production systems.
  - Embedded systems.
  - Systems with complex start-up procedures.

The Basic Mechanism



The Dyninst Interface

- Machine independent representation
- Object-based interface to build Abstract Syntax Trees (AST's)
- Write-once, instrument-many (portable)
- Hides most of the complexity in the API

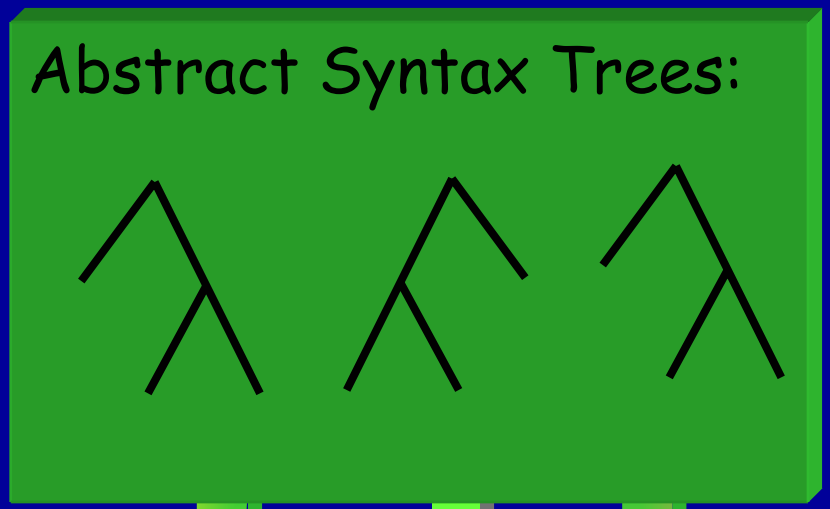
  Easy to build tools: e.g.:

  - A function entry/exit tracer: 75 lines of C++.
  - An MPI tracer: 250 lines.
  - Process hijacker: 700 lines.

Basic Dyninst Operations

- Code Analysis:
  - Basic blocks, functions, modules, variables
  - Call graph (inter-procedural CFG)
  - Intra-procedural control-flow graph (including natural loop detection)
  - Operate on stripped binary code, but opportunistically use symbol information.
- Program instrumentation
- Program modification
  - Both of these use run-time code generation
  - Based on machine independent code interfaces
- Process control:
  - Attach/create process
  - Monitor process status changes
  - Callbacks for fork/exec/exit
  - Inferior operations: malloc, load module, inferior RPC

Machine Independent Code



SPARC Code

```
sethi %hi(ctr)
ld [. . .],%o1
add %o1,%o1,1
st %o1,[. . .]
```

```
incl ctr
```

IA32 Code

MIPS Code

```
la   $t0,ctr
lw   $t1,($t0)
addi $t1,$t1,1
sw   $t1,($t0)
```

Dynamic Instrumentation Challenges

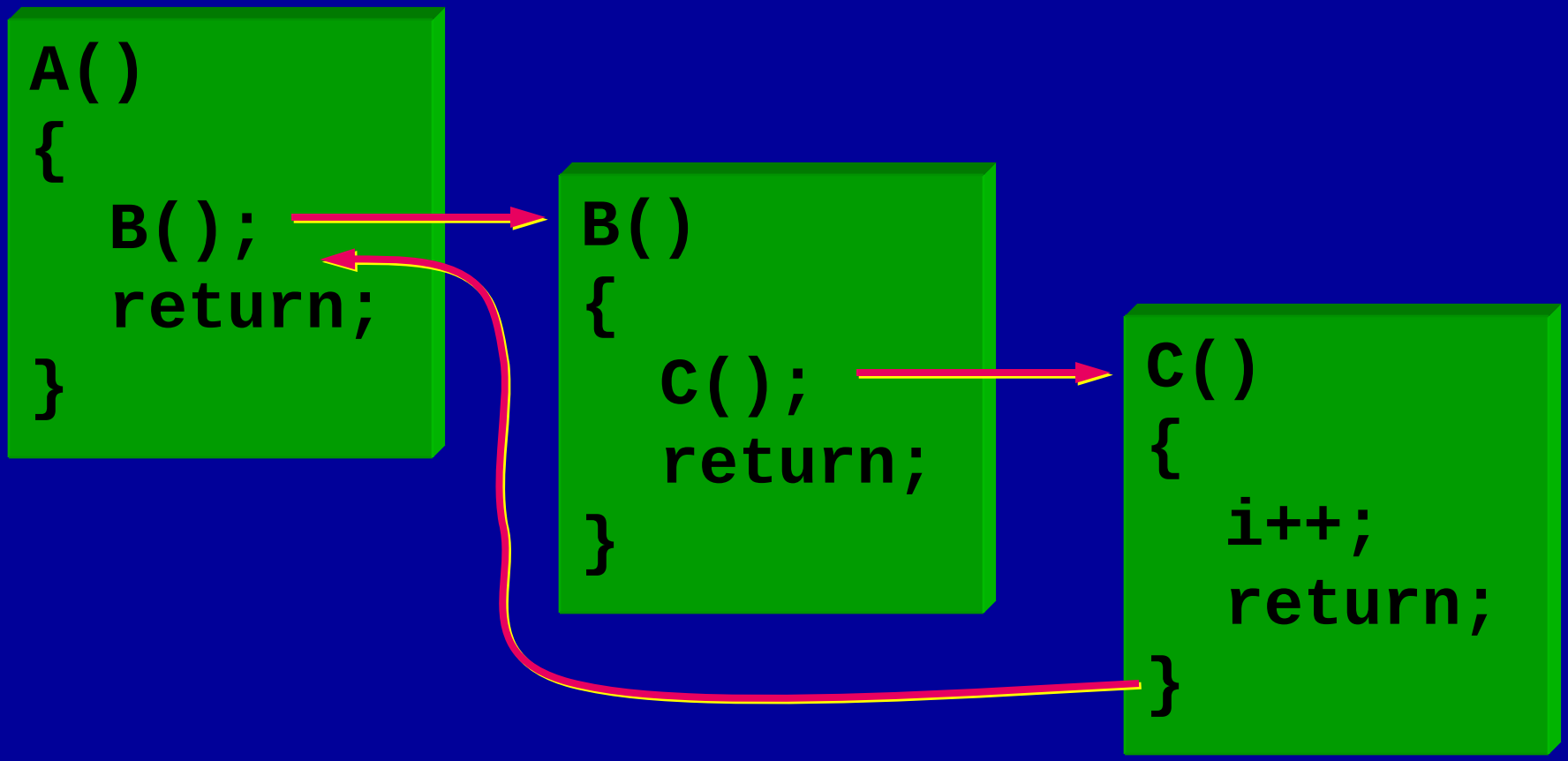
Parsing binary code a bit of a black art. You can get the 80% of the functionality in the first 20% of the time. Getting that last 20% will take you years . . .

- Finding key code idioms (function entry, exit, call site).
  - Function exit is often the toughest in code with symbols.
  - Function entry can be hardest in stripped code.
- Finding space for jump to trampoline
  - Long jumps (2-5 words or 5 bytes)
  - Short code sequences
  - Small functions
  - One (or few) byte instructions.
  - Code relocation is the key technique (huge topic).

Dynamic Instrumentation Challenges

- Fighting compiler optimizations (we instrument optimized code)
  - No explicit stack frame (leaf and intermediate functions)
    - Identifying parameters and local variables.
    - Walking the stack for safety: note that stack walking is an important general tool.
  - Data in code space (e.g., jump tables)
  - De-optimize code on the fly (e.g., tail calls)

Example: Tail-Call Optimization



There is no return point in B to instrument, so create one.

Example: Tail-Call Optimization

```
B: save
   . . .
   call C
   restore
```

```
C()
{
   i++;
   return;
}
```

Example: Tail-Call De-Optimization

```
B: save
   . . .
   mov %i0,%o0
   mov %i1,%o1
   mov %i2,%o2
   call C
   nop
   mov %o0,%i0
   mov %o1,%i1
   mov %o2,%i2
   ret
   restore
```

```
C()
{
  i++;
  return;
}
```

1999: Kerninst, Dynamic Instrumentation in the Kernel

Like Dyninst, but for kernels . . .

- The Kerninst API is (almost) identical to the Dyninst API
- Inserts runtime-generated code into kernel
- Dynamic: everything at runtime
  - No recompile, reboot, or even pause
- Runs on unmodified commodity kernels
  - Linux/IA32, Linux/Power, Solaris/SPARC.

Kerninst

- kerninstd: kernel instrumentation server
  - Basic instrumentation primitive: *splicing*
    - Put instrumentation code in a *trampoline.*
    - Patch a jump at the instrumentation site.
- kperfmon: performance measurement tool
  - Generates measurement instrumentation code
  - Communicates with kerninstd to insert that code
- Optimization framework

Vision: Autonomous, Evolving Kernels

- Measure, design improved code, install it
  - All at run-time
  - All on a commodity OS kernel
- Envision full suite of dynamic optimizations
  - Specialization
  - Constant propagation
  - Inlining
- Also do other development steps at run-time
  - Debugging, installation of patches

Run-time I-Cache Optimization

- Code positioning [Pettis & Hansen 90]
  - An example of an evolving kernel algorithm
  - Reorder basic blocks to improve I-cache performance
    - What's new: at run-time *and* on a kernel
  - Procedure splitting
    - Segregate hot vs. cold basic blocks
  - Basic block positioning
    - Reorder blocks to facilitate straight-lined execution
    - Try to make hottest branches untaken
    - Requires edge counts

Code Positioning Steps

Measure root function

- Is there poor I-cache performance?

Measure block counts

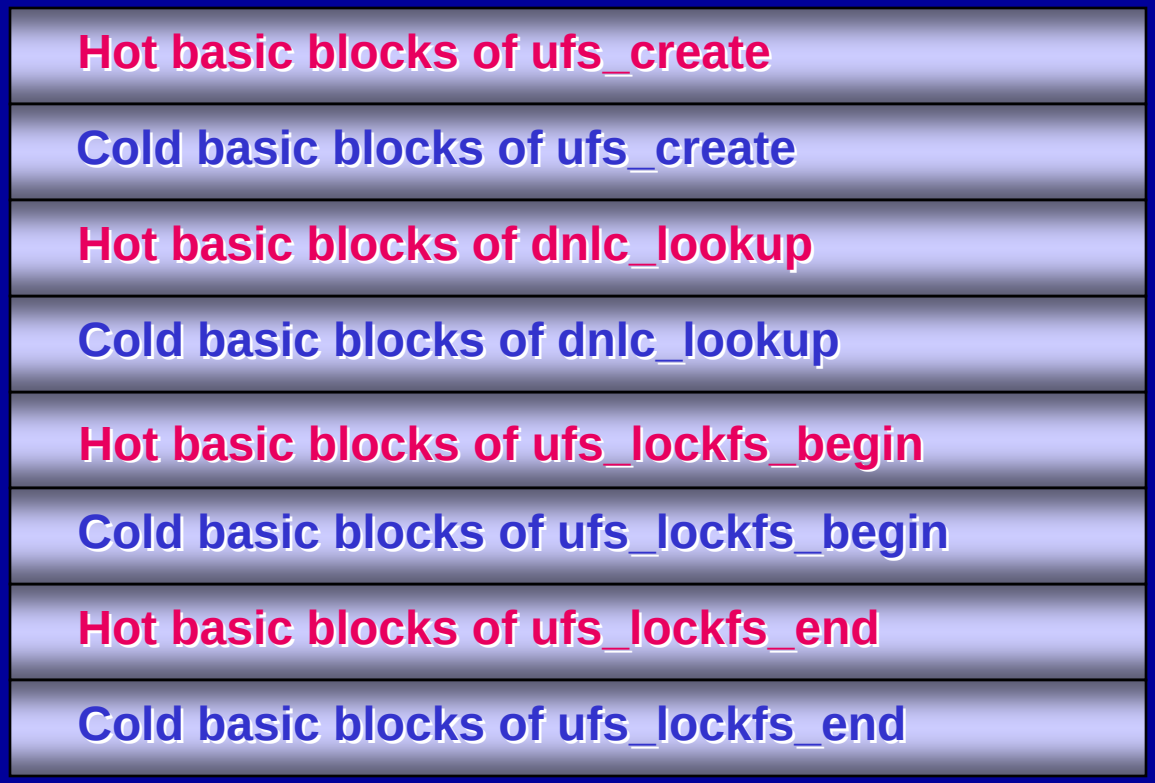
- Of the root function and its descendants
  - A traversal of the call graph
- Weed out descendants with no "hot" basic blocks
  - Hot: executed > 5% as often as root function is invoked

Emit optimized group of functions

The Optimized Function Group

Root function is ufs_create

Other functions are the hot subset of ufs_create's call graph descendants



Use code replacement on root function to install

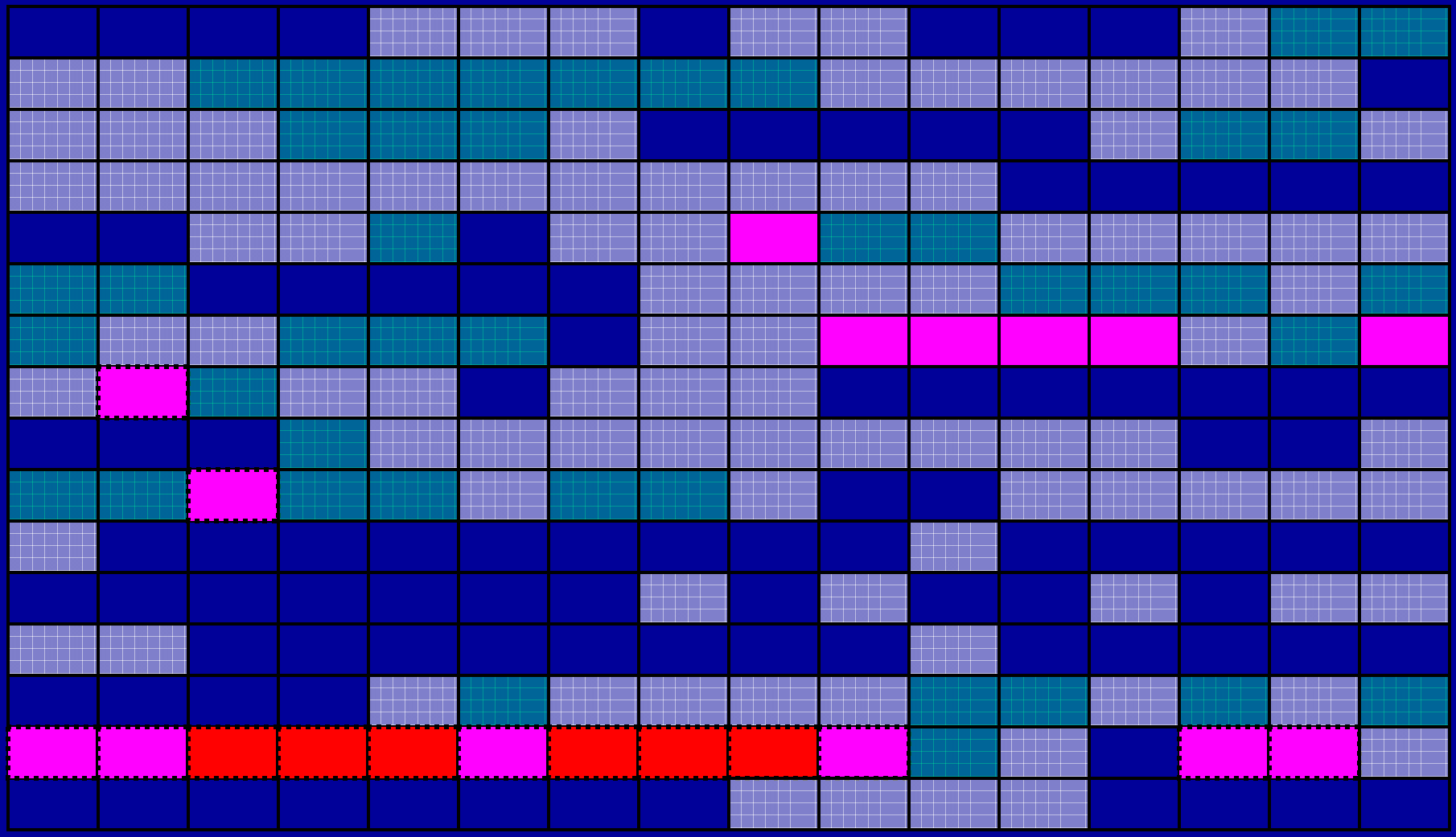
Case Study of Code Positioning

- Benchmark: mirror Paradyn papers Web page
  - 10 simultaneous connections
  - Perform code positioning on tcp_rput_data
    - Forwards data from IP to socket (on the hot path)
    - A large function (12K+ excluding callees)
      - So a good candidate for code positioning
- Macro results
  - About 7% end-to-end speedup in completing benchmark.

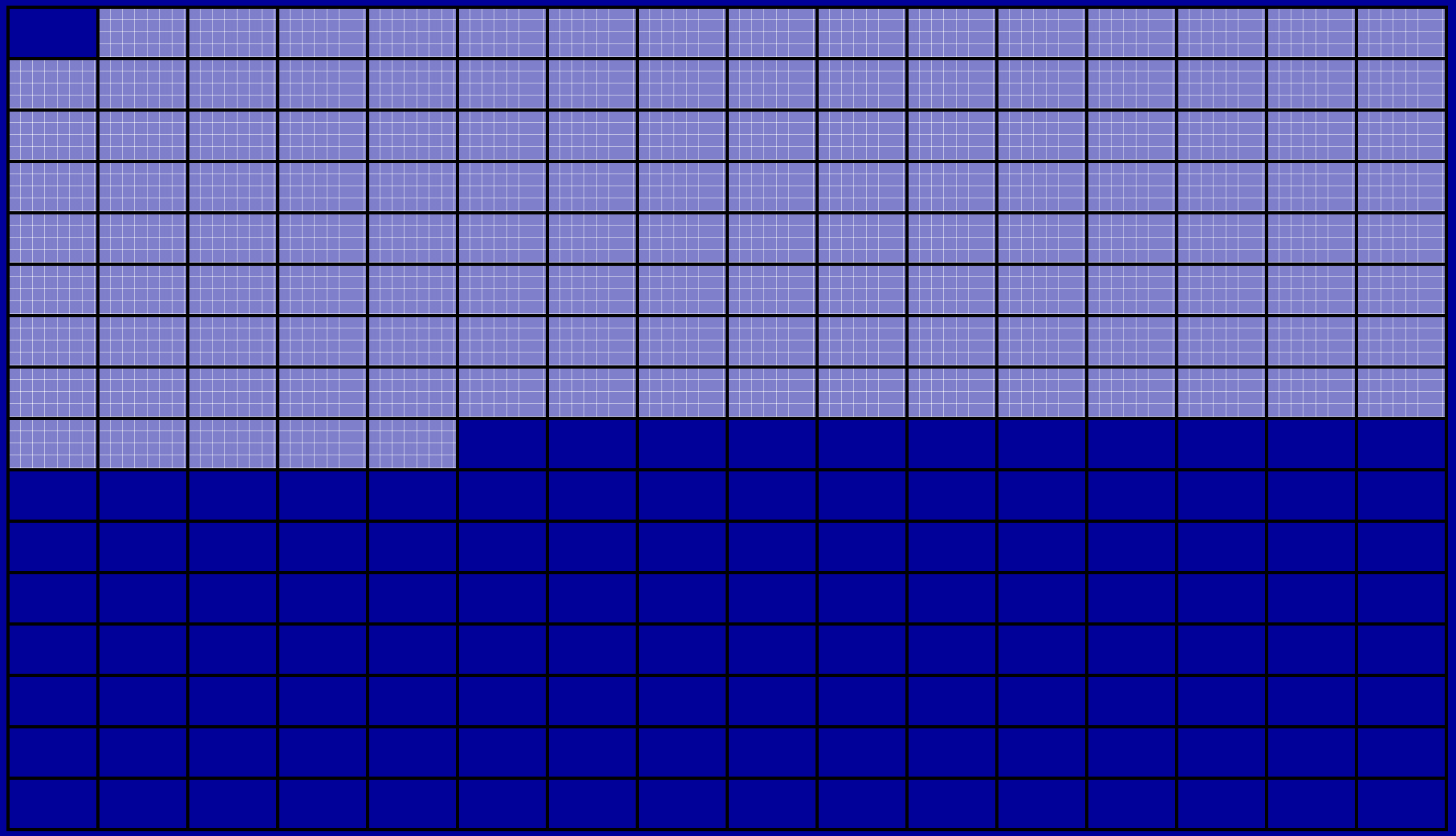
Results (cont'd)

- 22 functions in the group
  - 4260 bytes of hot code, 14624 bytes of cold code
  - Most functions had all hot blocks covered in 1 path
- Micro-results (per invocation of tcp_rput_data)
  - Virtual time per invocation
    - From 6.6 μs to 5.4 μs (reduction of 17.6%)
  - I-cache stall time per invocation
    - From 2.4 μs to 1.55 μs (reduction of 35%)
  - Branch mispredict stall time per invocation
    - From 0.38 μs to 0.20 μs (reduction of 47%)
  - IPC: from 0.28 to 0.38

I-Cache Footprint: Before

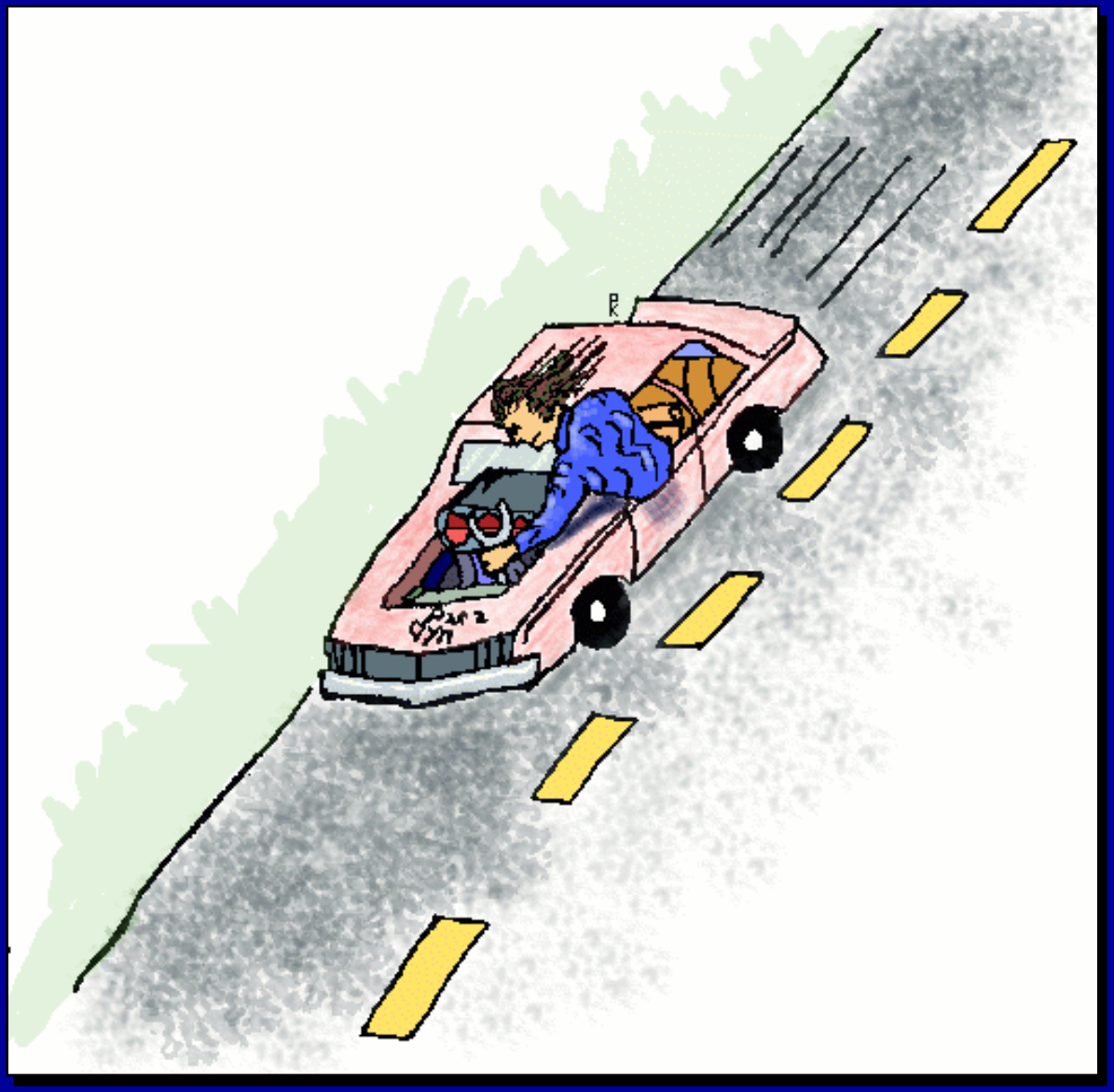


I-Cache Footprint: After



0 1 2 3 4

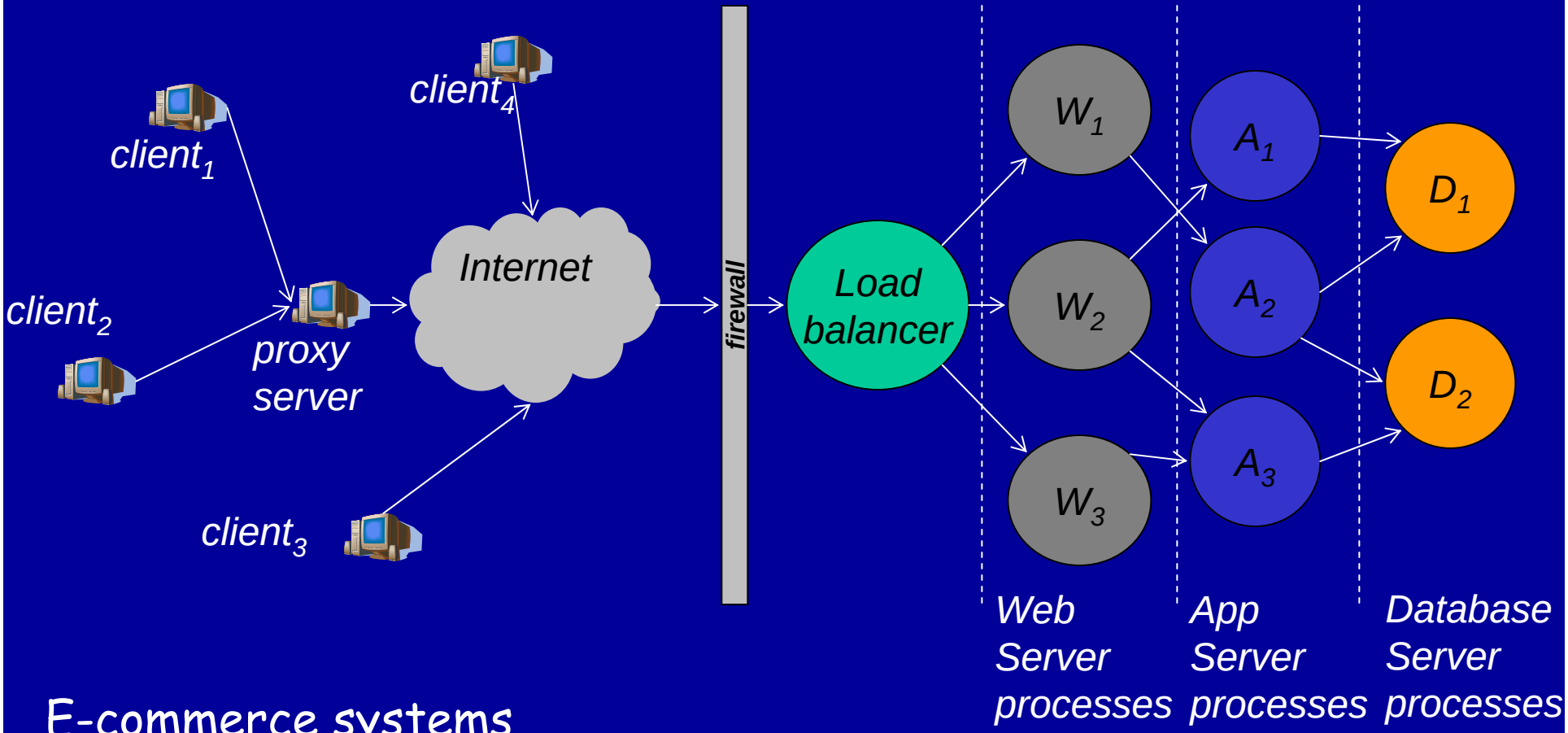
Evolving Kernels: The Big Picture



2004: Instrumentation that Follows Control Flow

- Goal: Finding the causes of bugs and performance problems in complex, multi-tier systems.
  - E-commerce systems
  - Grid computing environments
- Systems are complex and non-transparent
  - Multiple components, different vendors
- Anomalies are common in production software
  - Intermittent problems
  - Environment-specific ("It works for me" syndrome)
- Traditional profilers are not sufficient

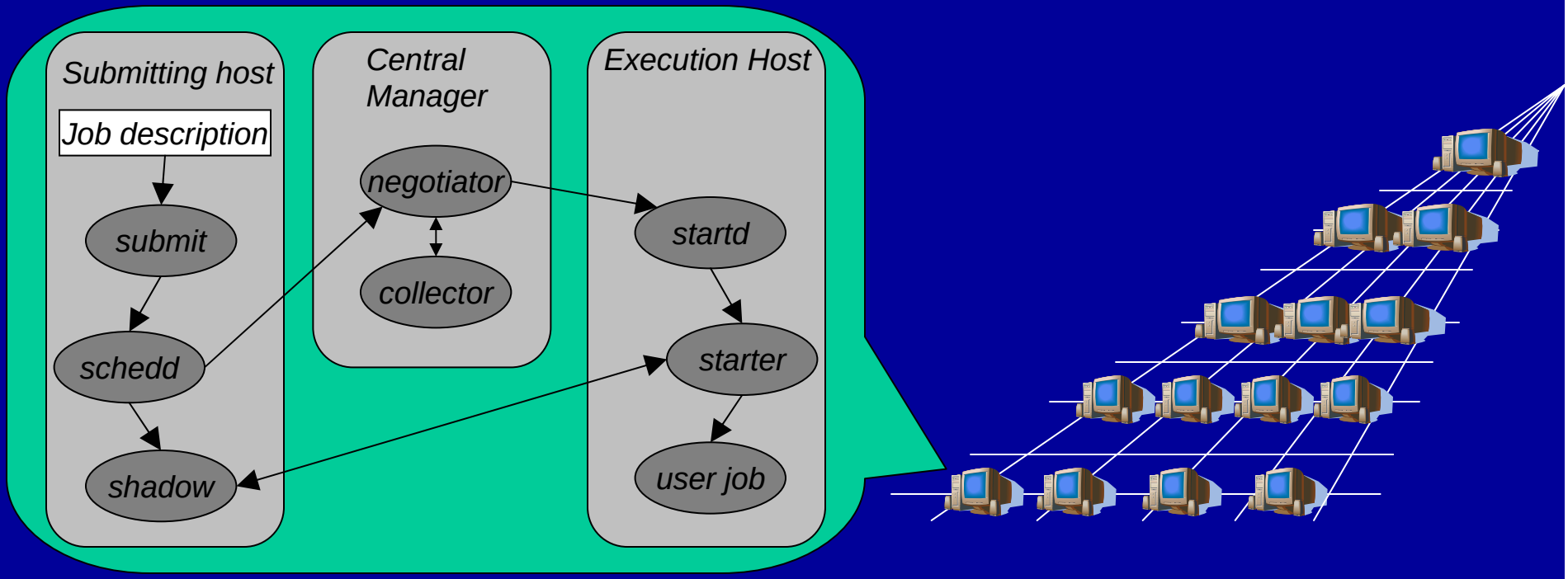
Common Environments



E-commerce systems

- Multi-tier: Clients, Web, DB servers, Business Logic
- Hard to debug: vendors have SWAT teams to fix bugs
  - Some companies get paid $1000/hour

Common Environments

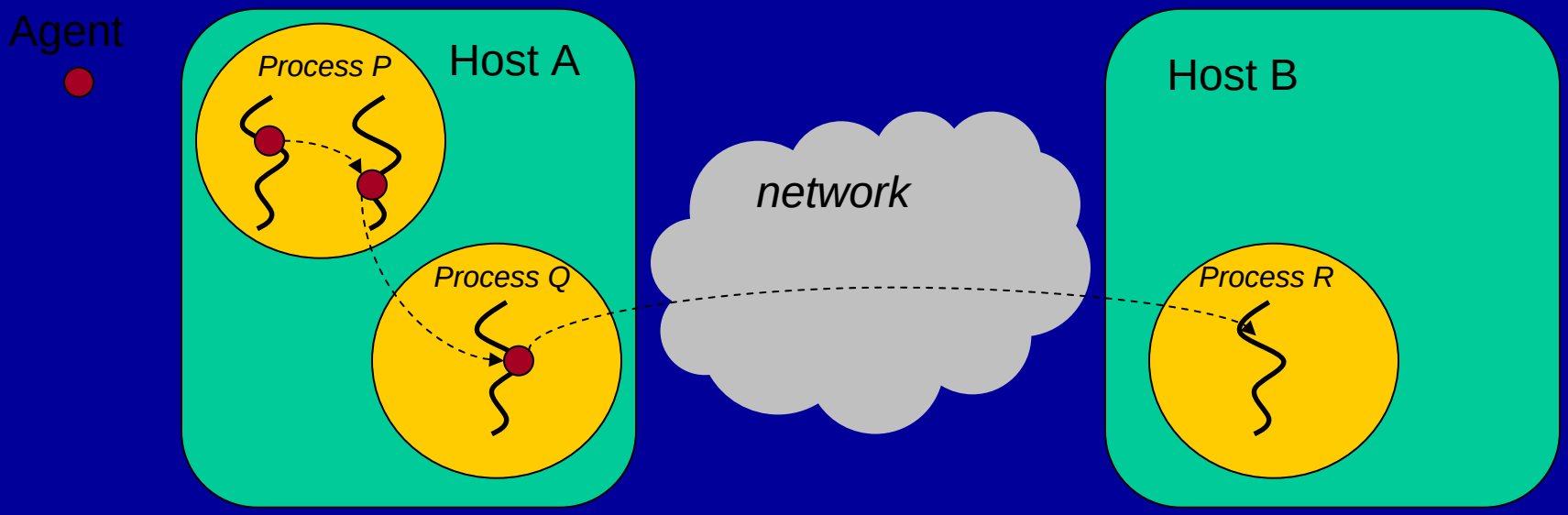


- Clusters and HPC systems
  - Large-scale: failures happen often (MTTF: 30 – 150 hours)
  - Complex: processing a Condor job involves 10+ processes
- The Grid: Beyond a single supercomputer
  - Decentralized
  - Heterogeneous: different schedulers, architectures
- Hard to detect failures, let alone debug them

Data Collection is Key

- Some basic goals:
  - We needs the details.
    - Do not discard important facts.
    - Collect system-wide data.
  - Autonomous
    - Rely minimally on human help.
  - Low perturbation
    - Try to have small affect the execution behavior.

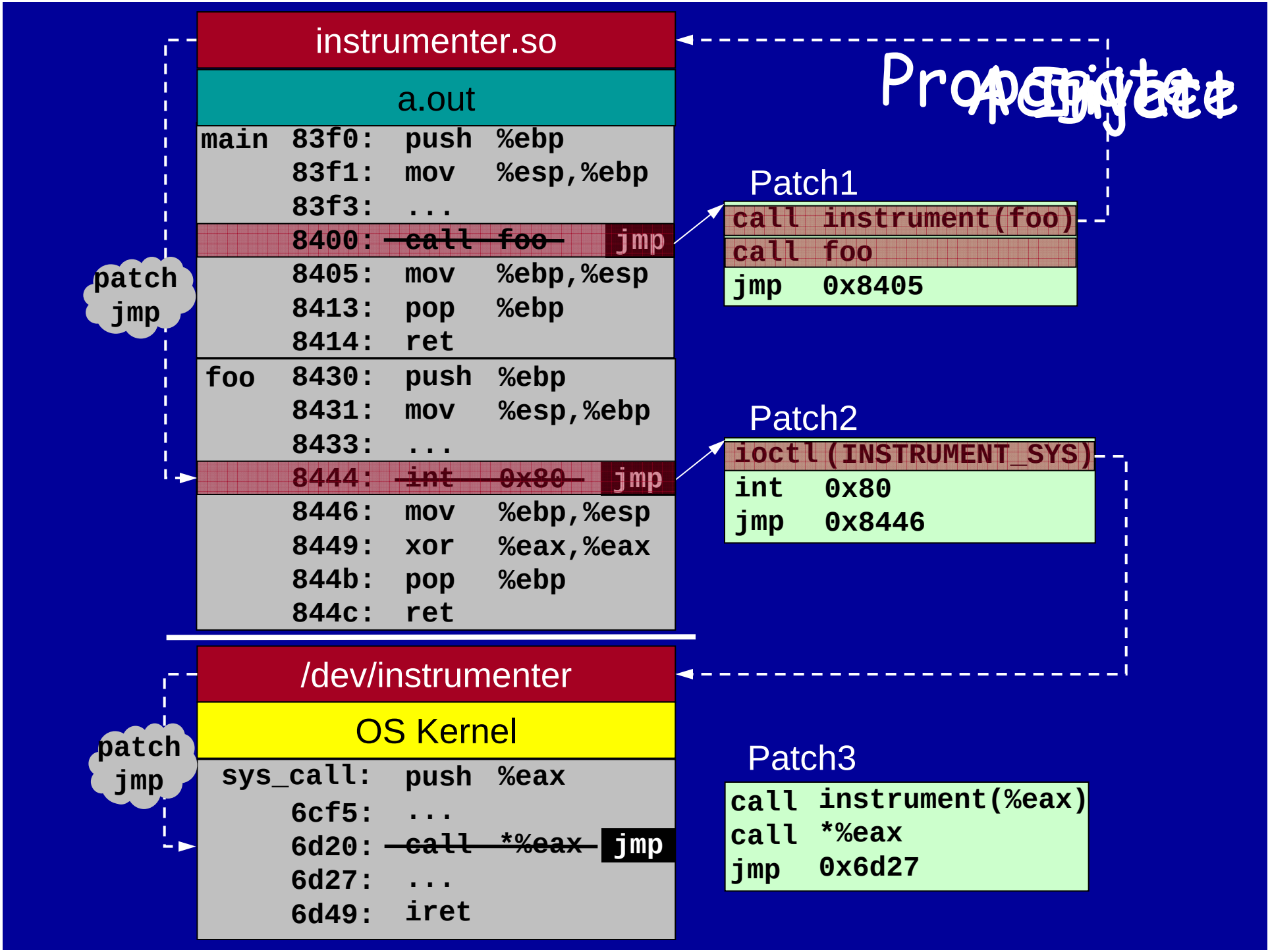
Vision: Self-Propelled Instrumentation



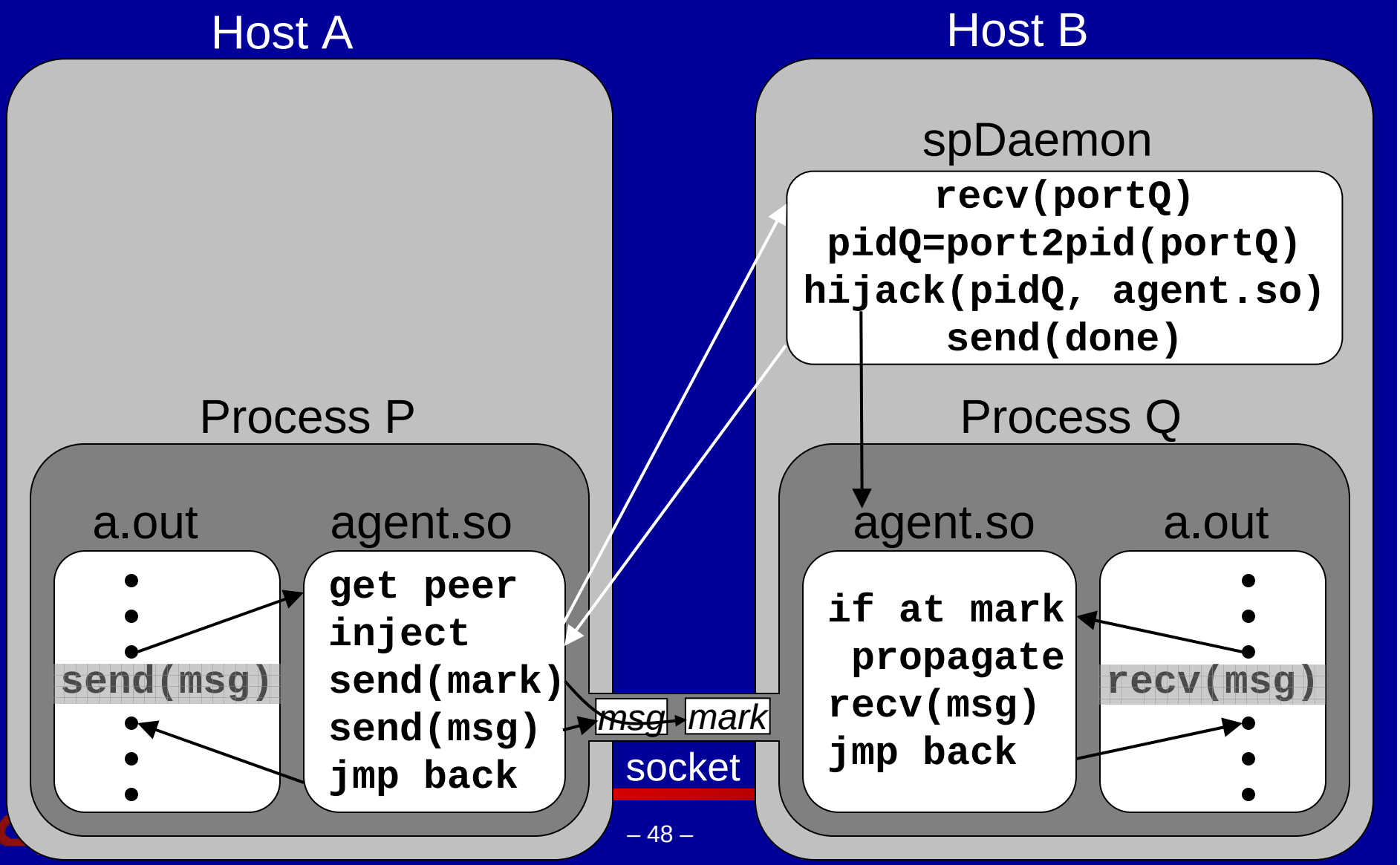
- An autonomous agent for system exploration
  - Follows execution of components (threads)
  - Propagates across component boundaries
  - Collects run-time data
  - Looks for anomalies in the data

Steps in Tracing Flows

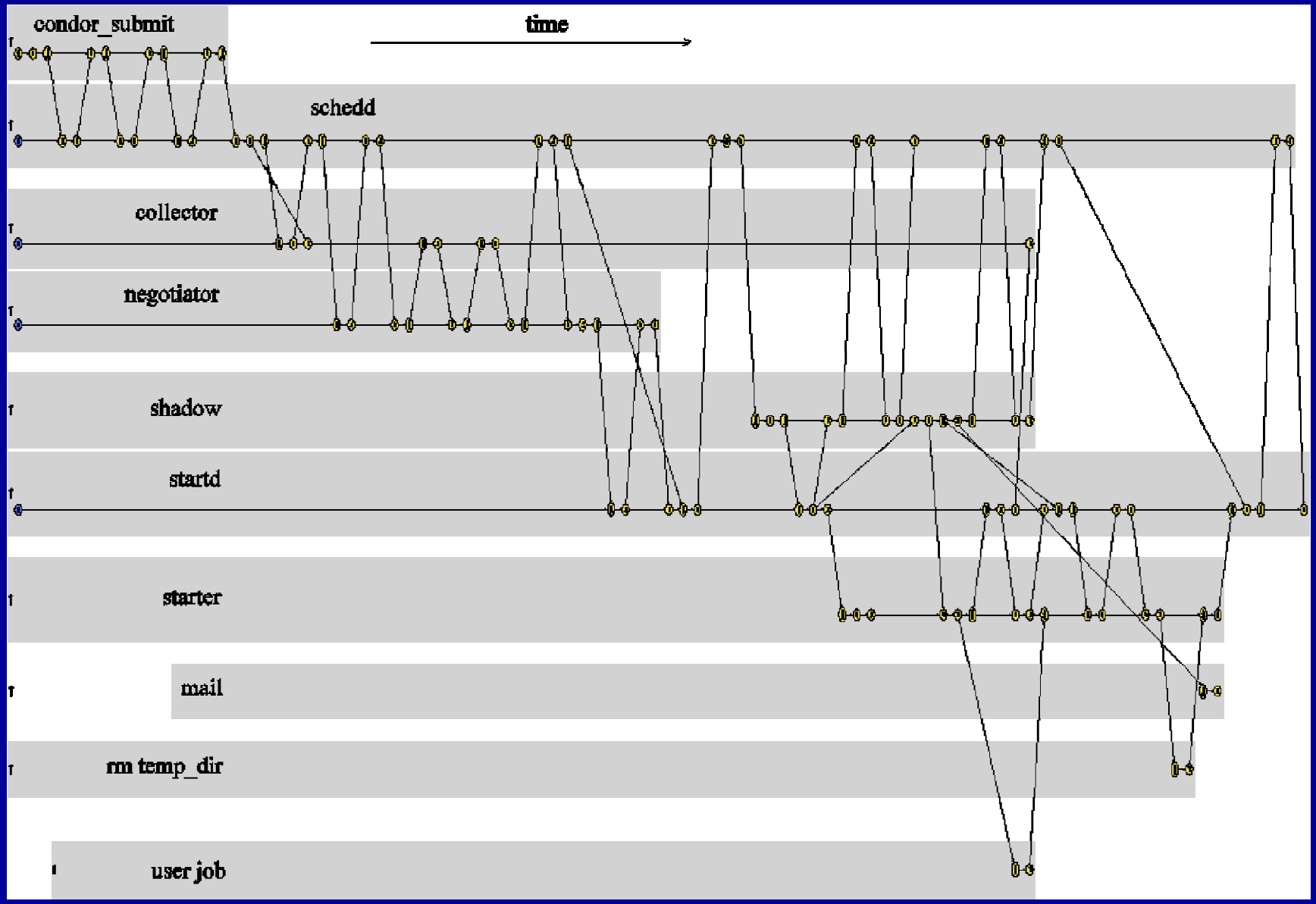
- Self-propelled instrumentation
  - System-wide control-flow tracing
- Separating concurrent flows
  - Interleaved executions within a process make analysis and diagnosis difficult.
- Automated diagnosis
  - Finds nodes or flows that appear anomalous
    - We use unsupervised and 1-class learning
  - Identify functions that are causing the anomaly.


instrumenter.so
a.out
main 83f0: push %ebp
83f1: mov %esp,%ebp
83f3: ...
8400: call foo
jmp
8405: mov %ebp,%esp
8413: pop %ebp
8414: ret
foo 8430: push %ebp
8431: mov %esp,%ebp
8433: ...
8444: int 0x80
jmp
8446: mov %ebp,%esp
8449: xor %eax,%eax
844b: pop %ebp
844c: ret
patch
jmp
Patch1
call instrument(foo)
call foo
jmp 0x8405
Patch2
ioctl(INSTRUMENT_SYS)
int 0x80
jmp 0x8446
/dev/instrumenter
OS Kernel
sys_call: push %eax
6cf5: ...
6d20: call *%eax
jmp
6d27: ...
6d49: iret
patch
jmp
Patch3
call instrument(%eax)
call *%eax
jmp 0x6d27

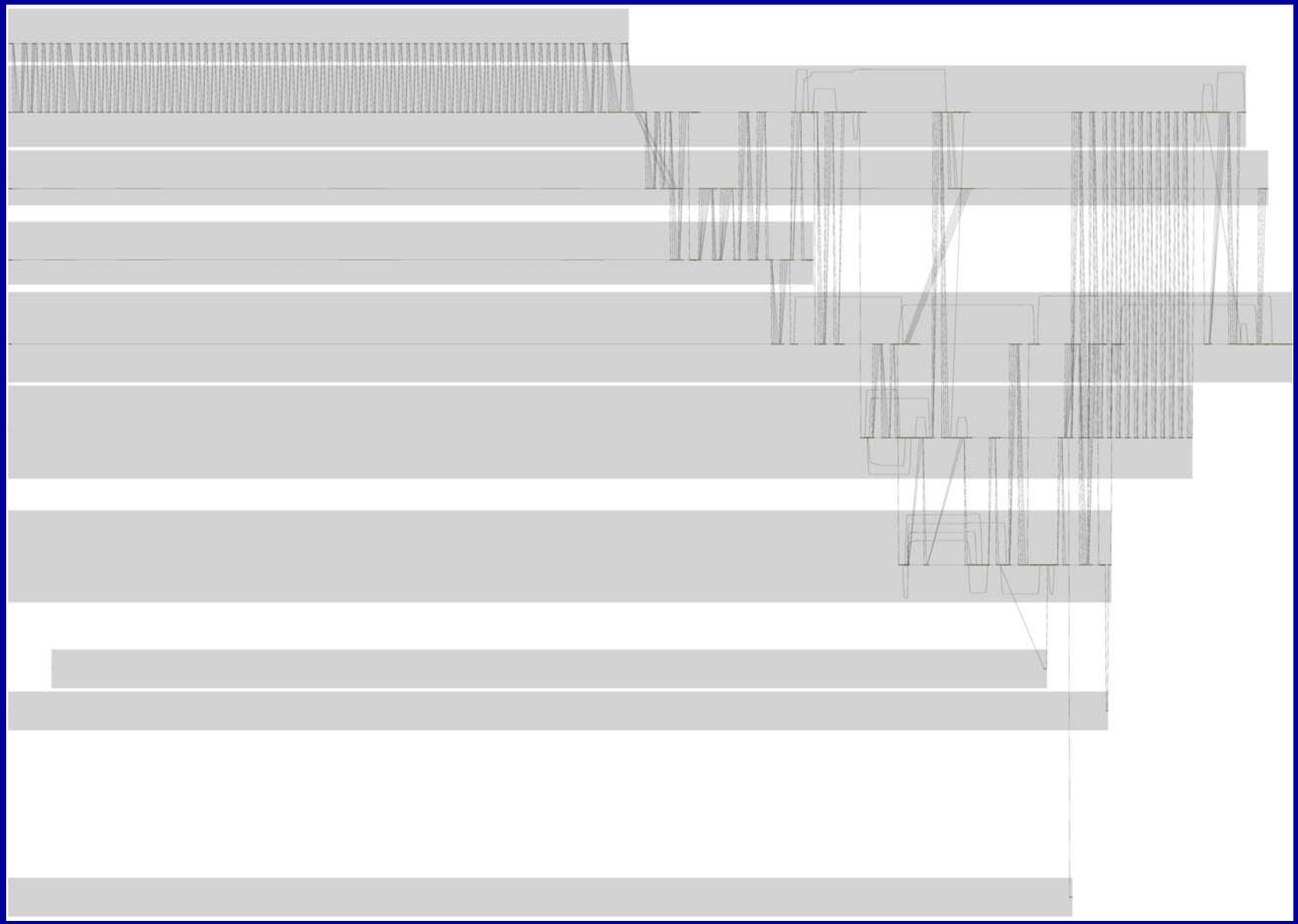
Cross-process Propagation



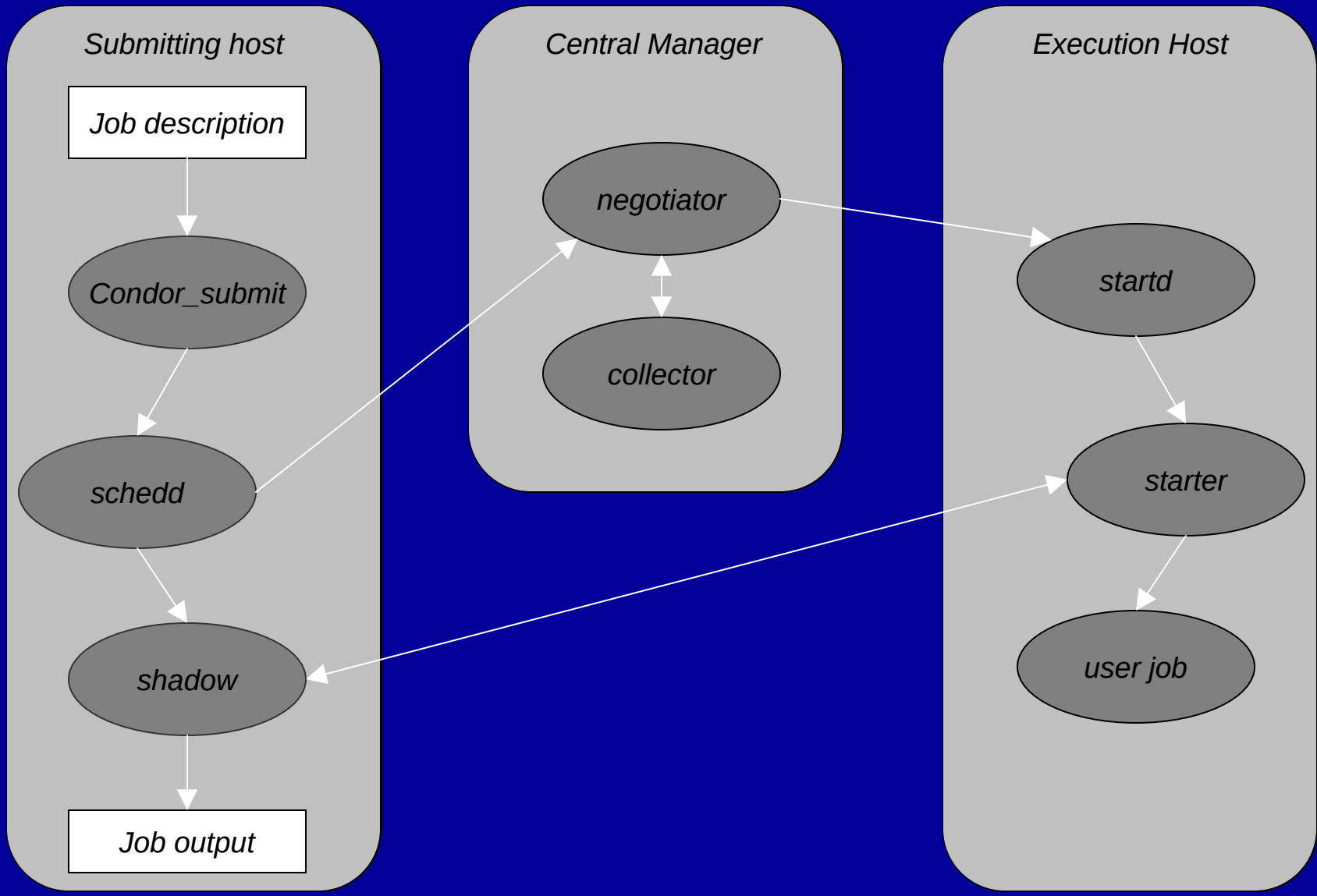
PDG for One Condor Job



PDG for One Condor Job



Experimental Study: Condor



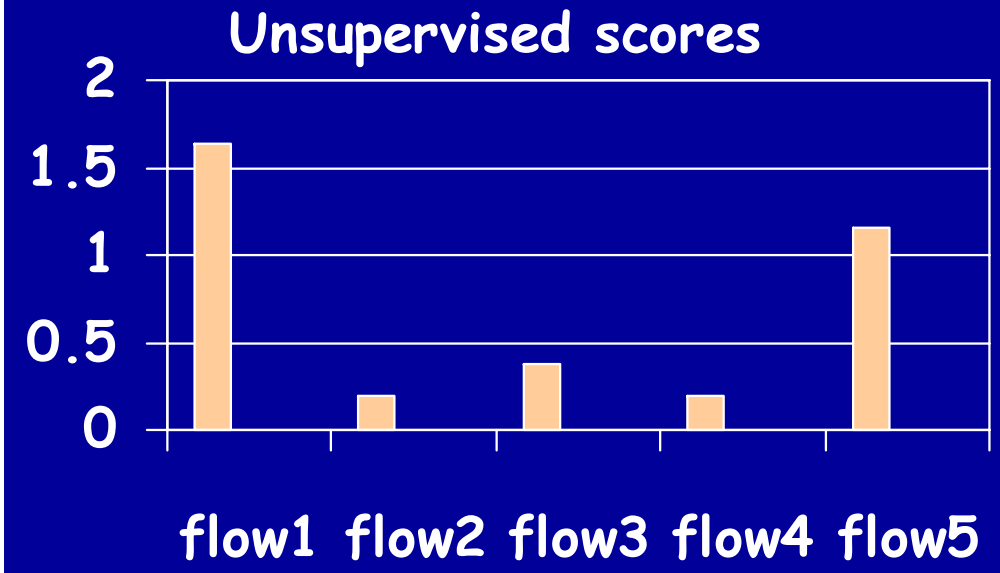
Job-run-twice Problem

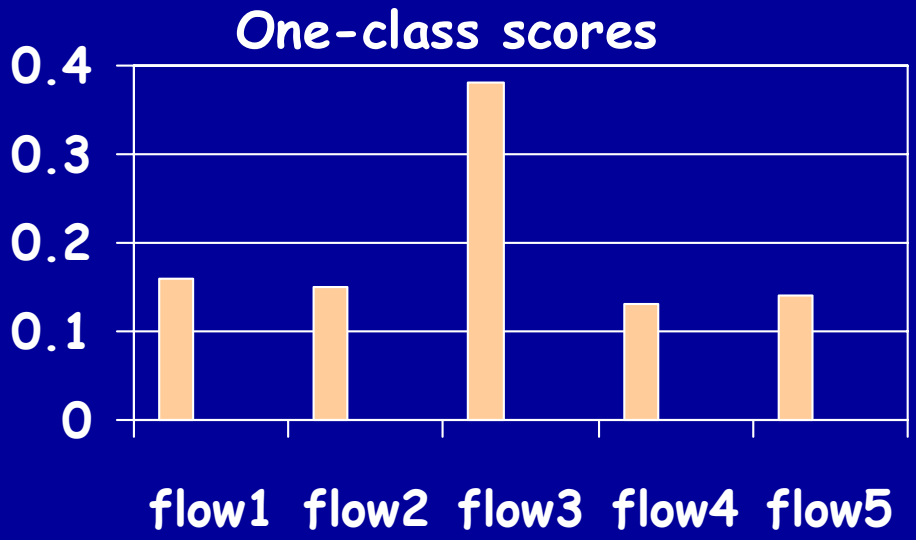
- Fault handling in Condor
  - Any component can fail
  - Detect the failure
  - Restart the component
- Bug in the shadow daemon
  - Symptoms: user job ran twice
  - Cause: intermittent crash after shadow reported successful job completion
  - Observed at a customer cite (Fermilab)

Debugging Approach

- Submit a cluster of several jobs
  - Start tracing condor_submit
  - Propagate into schedd, shadow, collector, negotiator, startd, starter, mail, the user job
- Separate the trace into flows
  - Processing each job is a separate flow
- Identify anomalous flow
  - Use unsupervised and one-class algorithms
- Find the cause of the anomaly

Finding Anomalous Flow



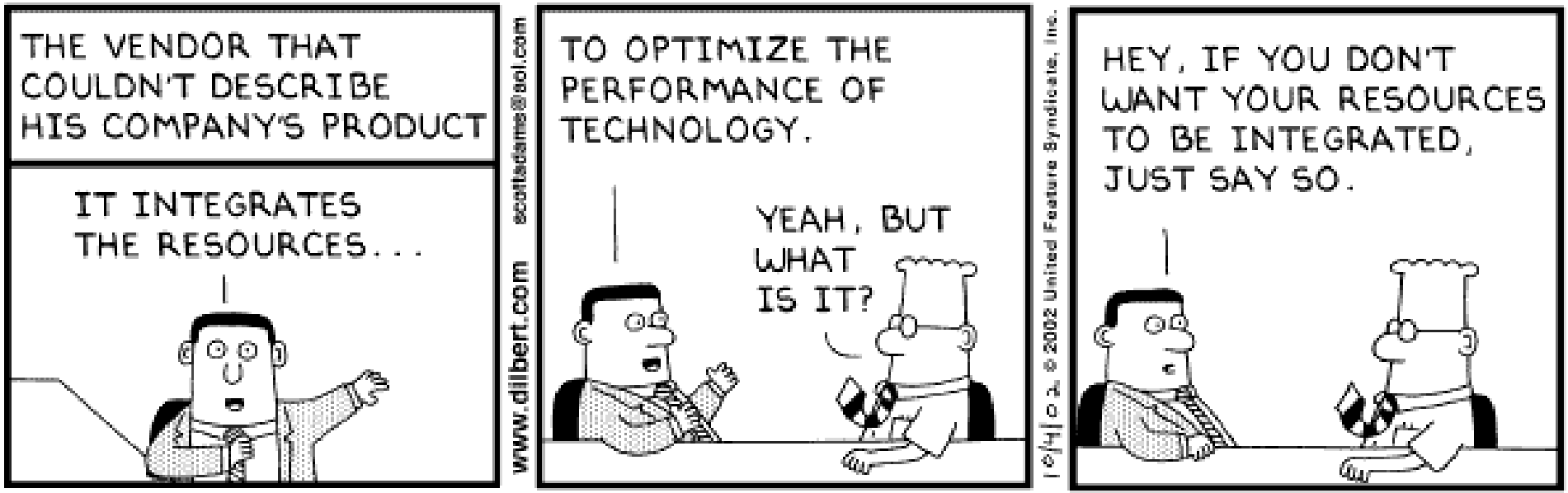


- Suspect scores for composite profiles
- Without prior knowledge, Flows 1 and 5 are unusual
  - Infrequent but normal activities
  - Use prior known-normal traces to filter them out
- Flow 3 is a true anomaly

Finding the Cause

- Computed coverage difference
  - 900+ call paths in anomalous run, not in good run
- Eliminated effects of earlier difference
  - 37 call paths left
- Ranked the paths
  - 14th path by time / 1st by length as called by schedd:
    main
    → DaemonCore::Driver
    → DaemonCore::HandleDC_SERVICEWAITPIDS
    → DaemonCore::HandleProcessExit
    → Scheduler::child_exit
    → DaemonCore::GetExceptionString
  - Called when shadow terminates with a signal
- Last function called by shadow = failure location

Vision: Self-Propelled Instrumentation



Some Closing Thoughts

"The most successful technologies have low usage complexity in spite of substantial internal complexity"
Alfred Spector, VP IBM, October 2003 (SOSP)

- The phone system
- The Web
- Automobiles
- Dyninst API ☺

We can (must!) leverage the great advances of the last 30 years in compiler technology and binary analysis.

Dynamically adaptive tools are needed to deal with dynamically adaptive systems.

Increasing complexity and scale of systems will continue to push the limits of our tools.

Need to resolve the conflict between security concerns and binary patching.

How to Get Copies of Dyninst and Kerninst:

Free for research use.

Dyninst release 5.0.1

- Supports Solaris (SPARC), Windows (IA32), *AIX (Power)*, Linux (IA32), Linux (IA64), Linux (AMD64/EM64T).

Kerninst release 2.1

- Supports Linux (IA32), Linux (Power), Solaris (SPARC).

http://www.paradyn.org

http://www.dyninst.org

paradyn@cs.wisc.edu