Dynamo: A Transparent Dynamic Optimization System

V. Bala, E. Duesterwald, and S. Banerjia, May 2000

Introduction: What is Dynamo?

- Run-time software optimizer
- Performs optimization on a *native* instruction stream
- Intruction stream can come from
  - a statically compiled native binary; or
  - a dynamically generated binary, e.g., by a JIT compiler
- Implemented entirely in software
- Provided as a user-mode DLL

Background: Why Dynamo?

- Greater degree of delayed binding due to OOP paradigms and modern software techniques
  - Functions and methods are looked up at run-time
  - Limits the size and scope available for static analysis by the compiler
- Modern software is shipped as collection DLLs (shared library)
  - Parts of DLL referenced at run-time
  - Static optimizations virtually impossible
- Generally, dynamic code generation environments make static optimization techniques impractical
- Reliance on independent software vendors to enable optimizations
  - System vendors not able to control the keys that unlock thei performance potential
- Current trend of offloading complexity from hardware to the compiler (CISC to RISC to VLIW progression)

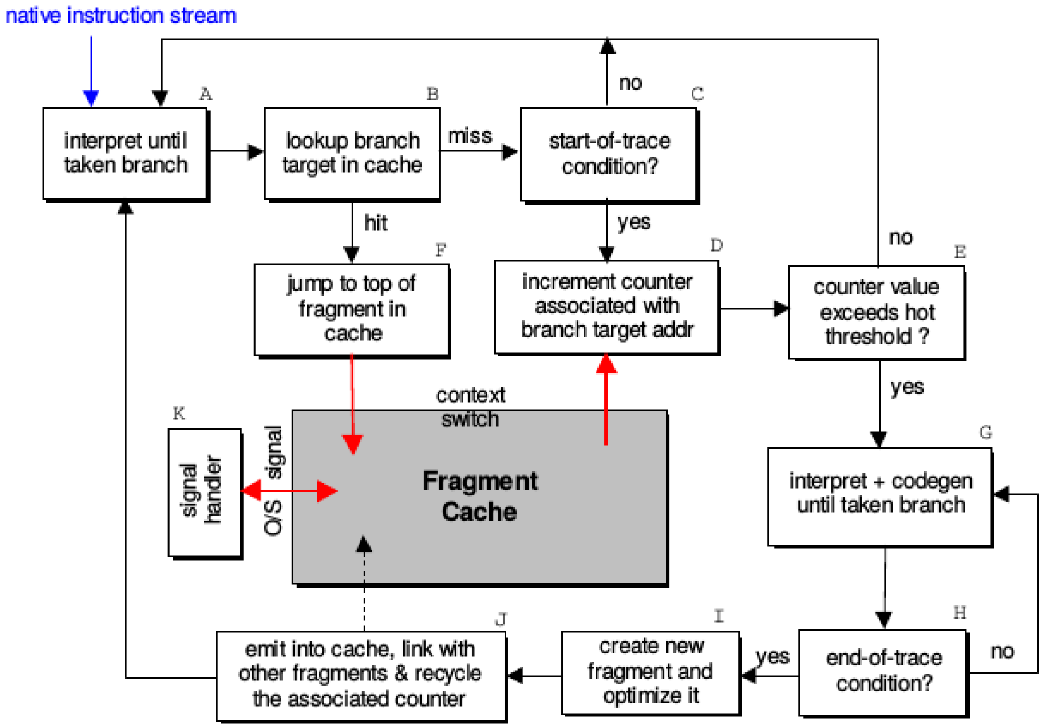


Figure 1. How Dynamo works

Overview

1. Startup and initialization
2. Fragment formation
3. Fragment linking
4. Fragment cache management
5. Signal handling

Startup and initializaion

- Dynamo is provieded as a user-mode DLL
- Entry point: `dynamo_exec`
- Saves a snapshot of the application context to an internal data structure
  - Application binary need not be perturbed in any way
- Swaps stack invironment
  - Application’s runtime stack is not interfered

app runs natively

app runs under Dynamo

<u>Application crt0 code</u>

```
...
...
push stack frame;
spill caller-save regs;
call dynamo_exec;
restore caller-save regs;
pop stack frame;
...
...
...
```

<u>Dynamo library code</u>

```
dynamo_exec:
  save callee-save regs to app-context;
  copy caller-save regs from stack frame
      to app-context;
  save stackptr to app-context;
  return-pc = value of link reg;
  swap Dynamo & application stack;
  // stackptr now points to Dynamo stack
  initialize internal data structures;
  call interpreter (return-pc, app-context);
  // control does not return here!
```

Overview

1. Startup and initialization
2. Fragment formation
3. Fragment linking
4. Fragment cache management
5. Signal handling

Fragment formation

- Definition: *Trace*
  - Sequence of consecutively executing instructions from the native instruction stream
- Definition: *Hot trace*
  - Sequence of instructions excuted many times, e.g. following the target of a backward branch (indicating a loop)
- Definition: *Fragment*
  - Optimized hot trace
- Definition: *Fragment cache*
  - The place where hot traces are linked together for reuse
- How are hot traces selected?

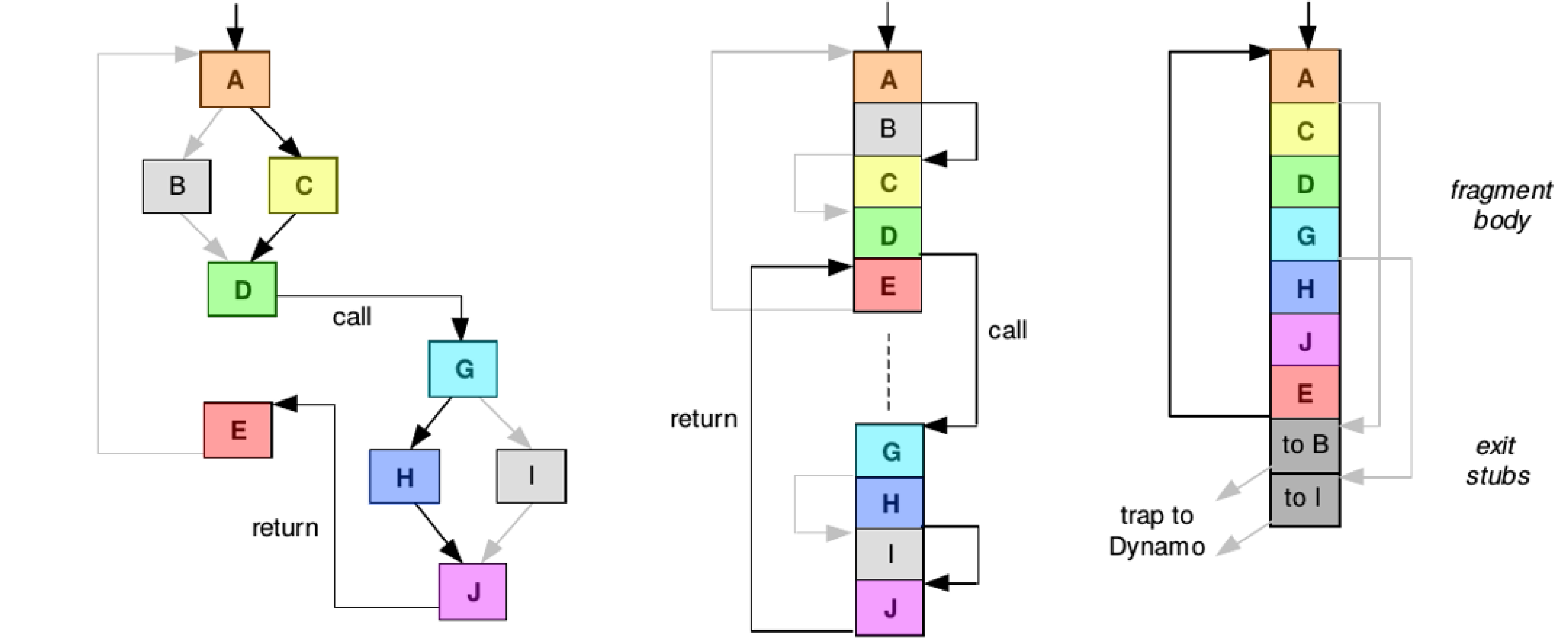
Fragment formation: Hot trace selection

- Dynamo uses a speculative scheme, *most recently executed tail (MRET)*, for hot-trace selection
  1. A counter is associated with certain *start-of-trace condition*, e.g. a backward branch
  2. The counter is incremented each time the associated start-of-trace condition occurs
  3. When counter exceeds som threshold value, switch to code generation mode and record hot trace until *end-of-trace* condition is reached
- Counters are only maintained for potentail loop headers (low memory footprint)

Fragment formation: Hot trace optimization

- Hot trace is converted into low-level IR
- Fall-through direction of indirect conditional branches remain on the trace
  - Transformed into direct conditional branch (less expensive)
- Direct unconditional branches are redundant and can be removed

Fragment formation: Hot trace optimization



Fragment formation: Hot trace optimization

- Most optimizations involve redundancy removal
  - Remove or convert branches
  - Remove conditional load operations
  - Remove dead code
- Conventional optimizastions
  - Copy propagation
  - Constant propagation
  - Strength reduction
  - Loop invariant code motion
  - Loop unrolling
- On-trace redundancies placed in off-trace *compensation blocks* at bottom of trace (more on this later)

Fragment formation: Emit fragment

- Emits optimized hot trace into fragment cache
- Two steps:
  1. Emit generated code from fragment body IR
  2. Emit unique fragment exit stubs for every exit and loop-back branch in trace
- Exit stubs transfer control from the Dynamo fragment cache to the Dynamo interpreter

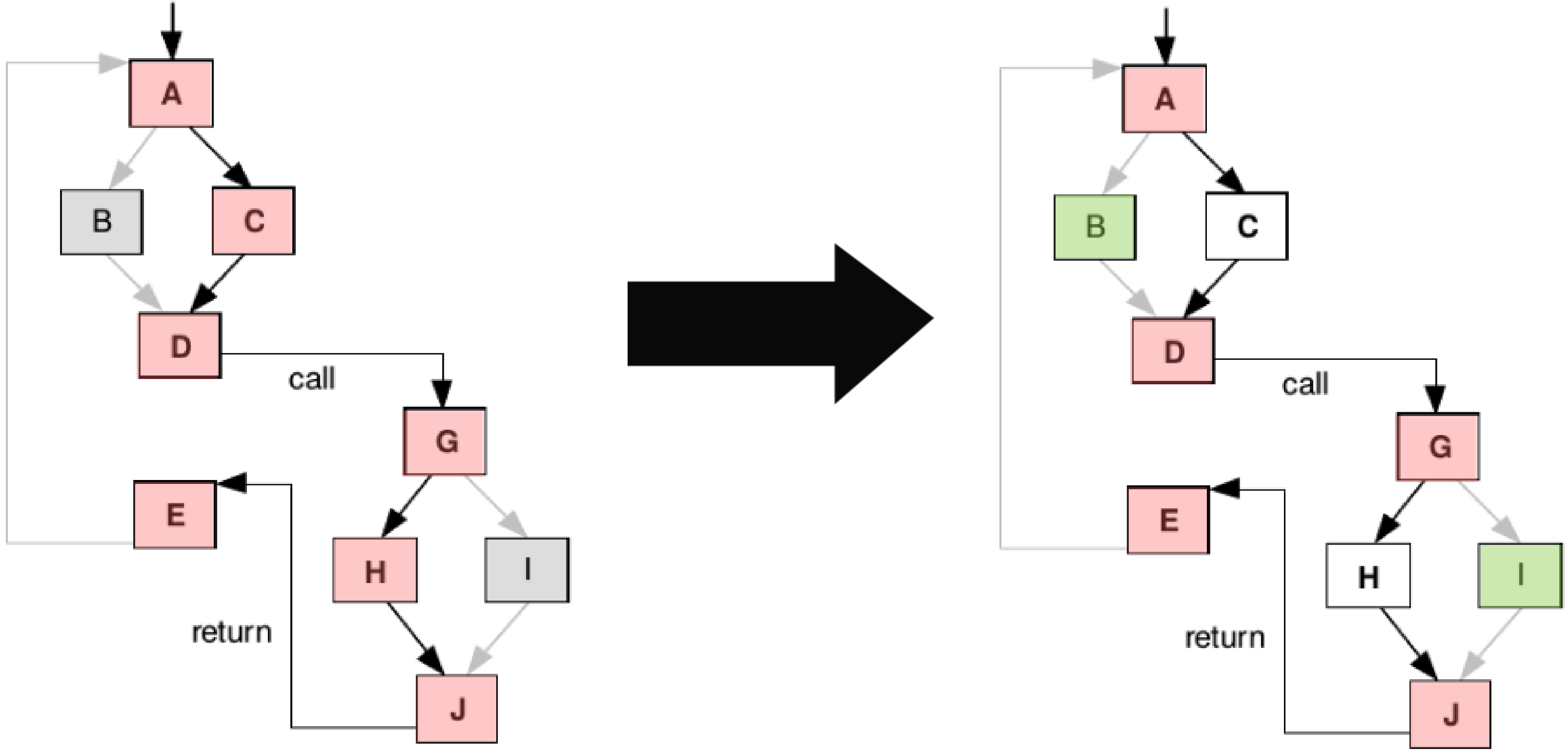
Overview

1. Startup and initialization
2. Fragment formation
3. Fragment linking
4. Fragment cache management
5. Signal handling

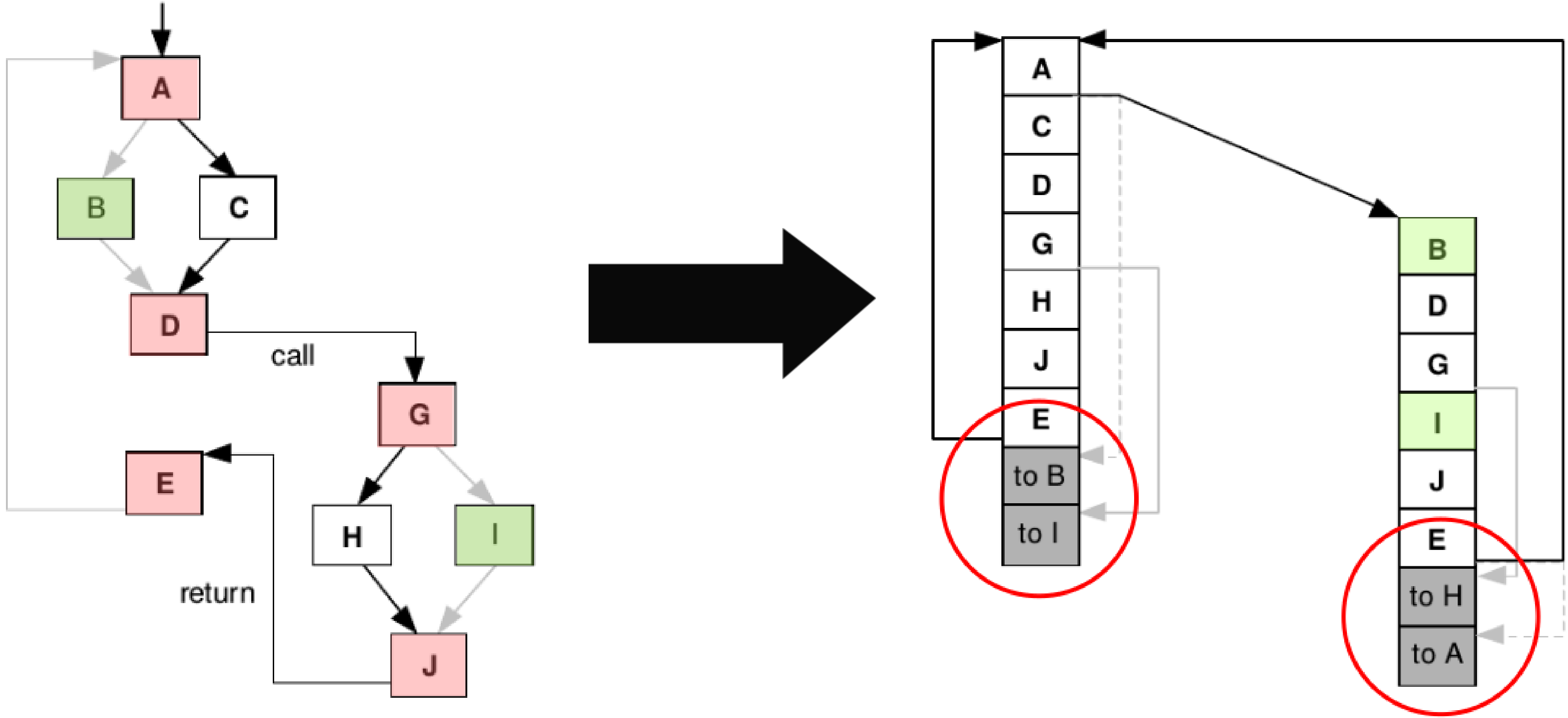
Fragment linking

- Linking involves patching block exit branches so that the target address becomes the entry of another fragment

Fragment linking



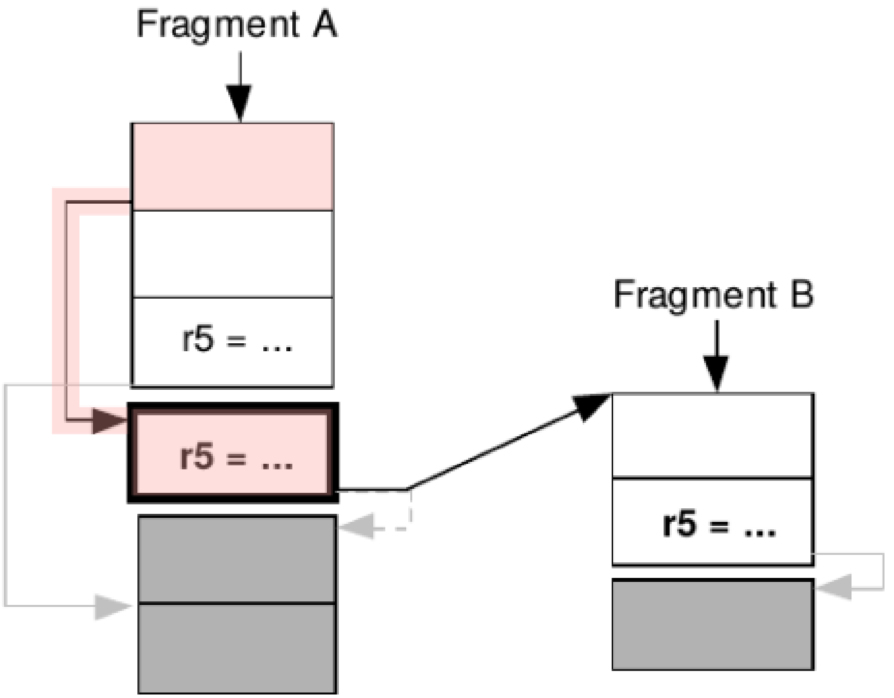
Fragment linking



Fragment linking

- Fragment linking is crusial for performance
- Prevents expensive exits from the fragment cache back to the intepreter
- Provides an opportunity to remove compensation blocks
  - On-trace redundancies are sunk into off-trace compensation blocks

Fragment linking



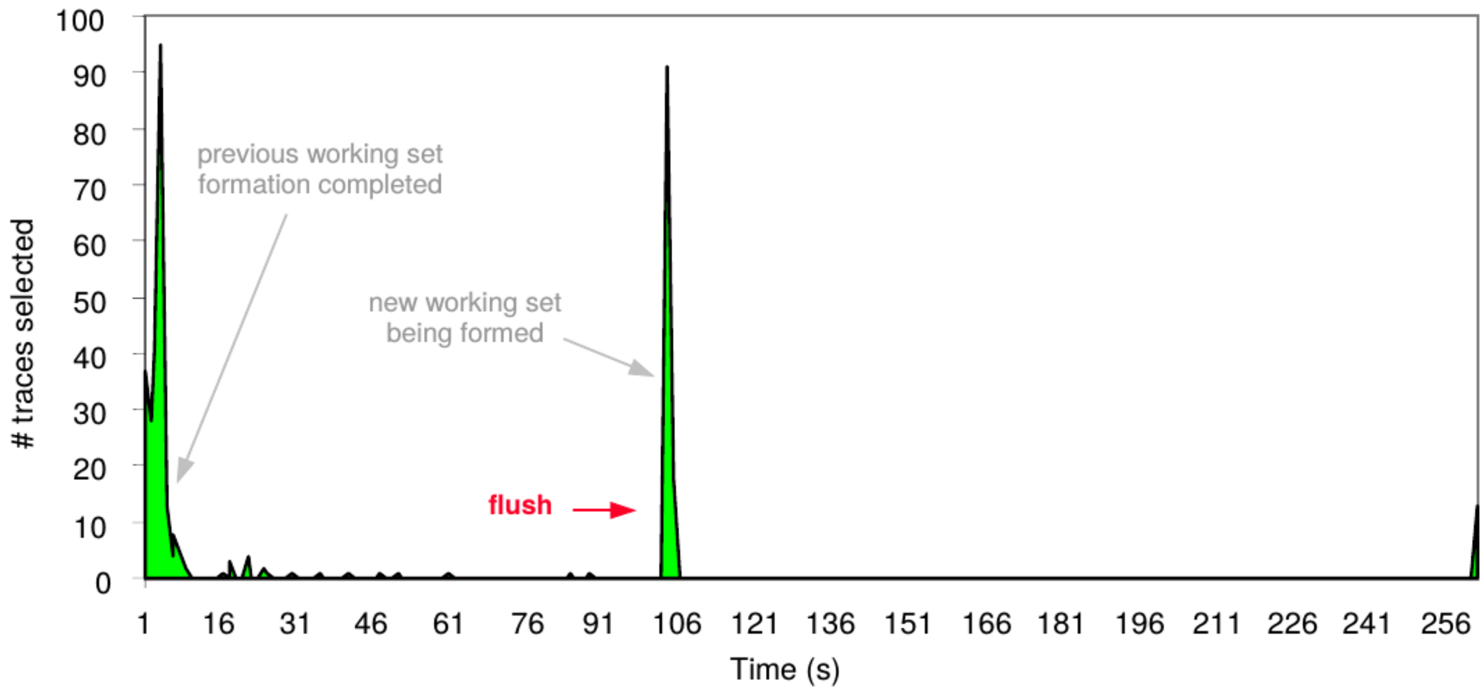
Overview

1. Startup and initialization
2. Fragment formation
3. Fragment linking
4. Fragment cache management
5. Signal handling

Fragment cache management

- Employs a pre-emptive flushing heuristic to periodically remove cold traces from the cache
- Essentially “free” since control is predominantly being spent in Dynamo anyway

Fragment cache management

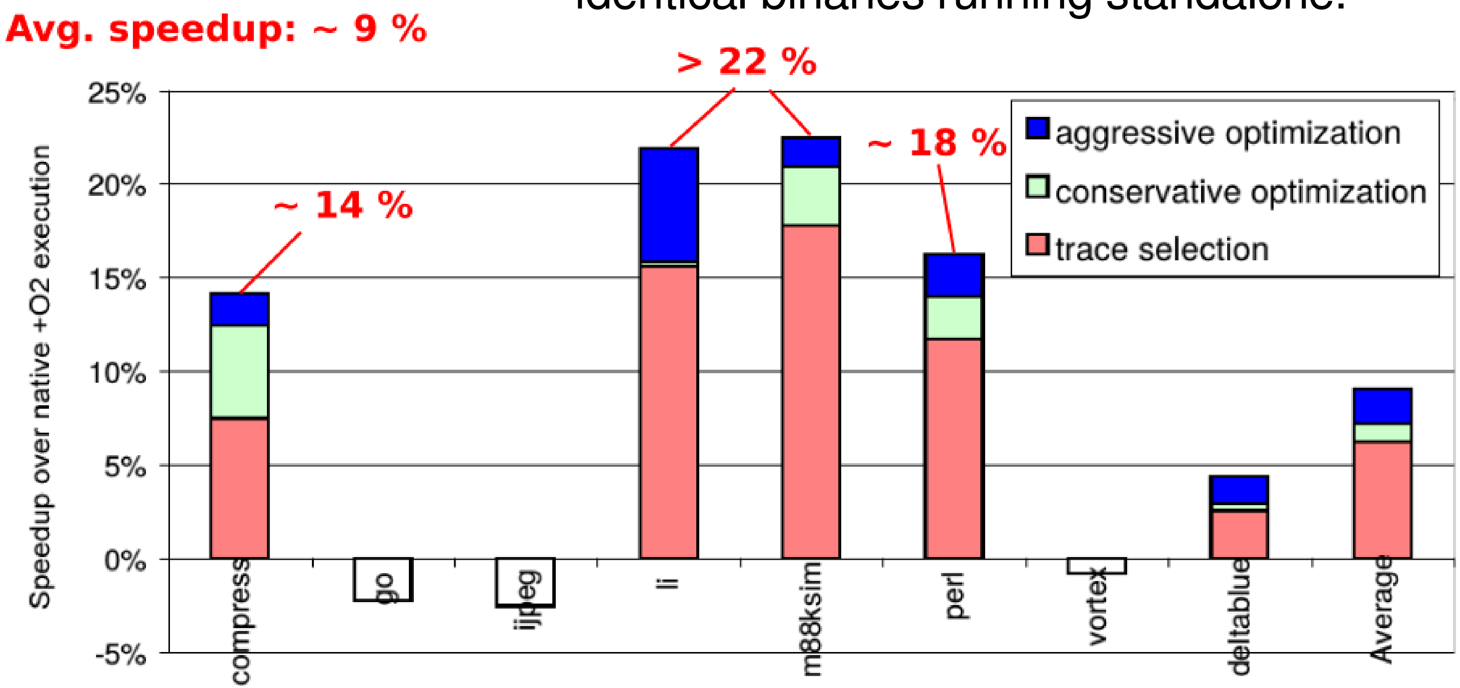


Performance data

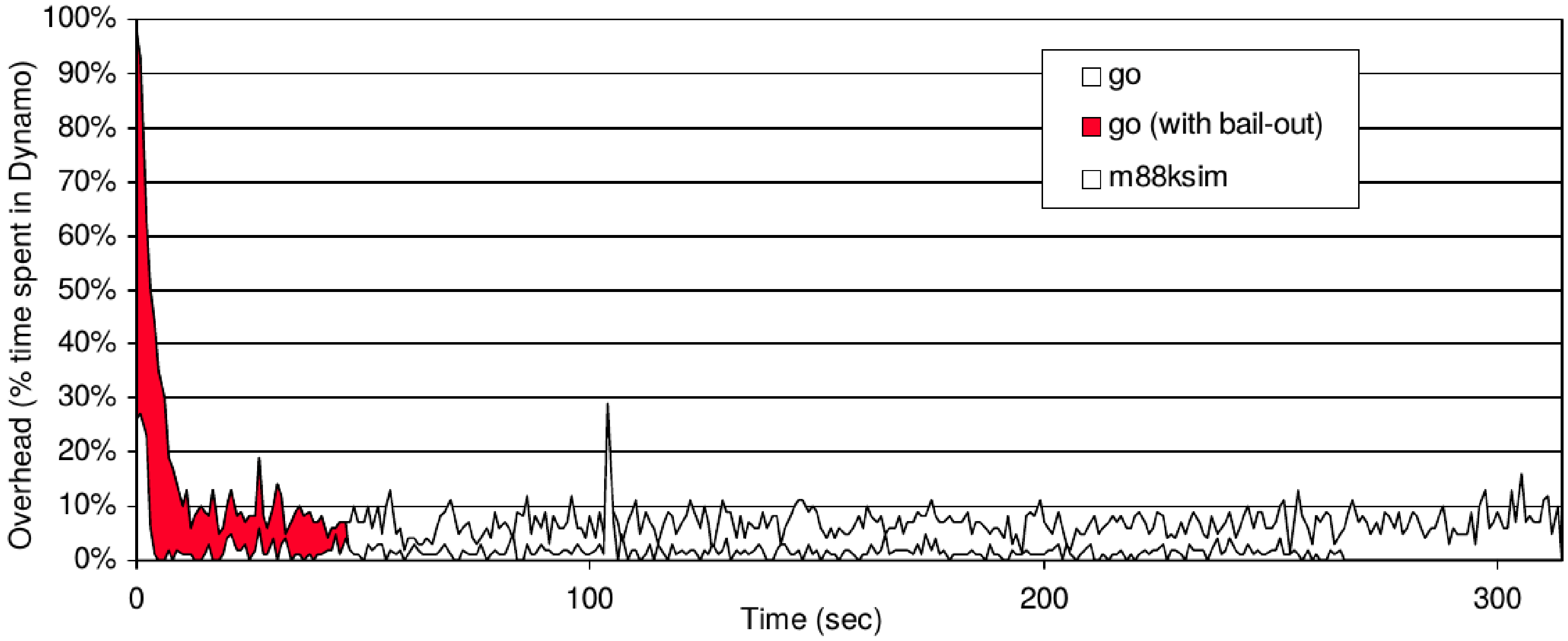
- SpecInt95 integer benchmarks and a commersial C++ code, *deltablue*
- HP C/C++ compiler with +O2 optimization level
- Single processor HP PA-8000 workstation
- 150 K fragment cache

Performance data

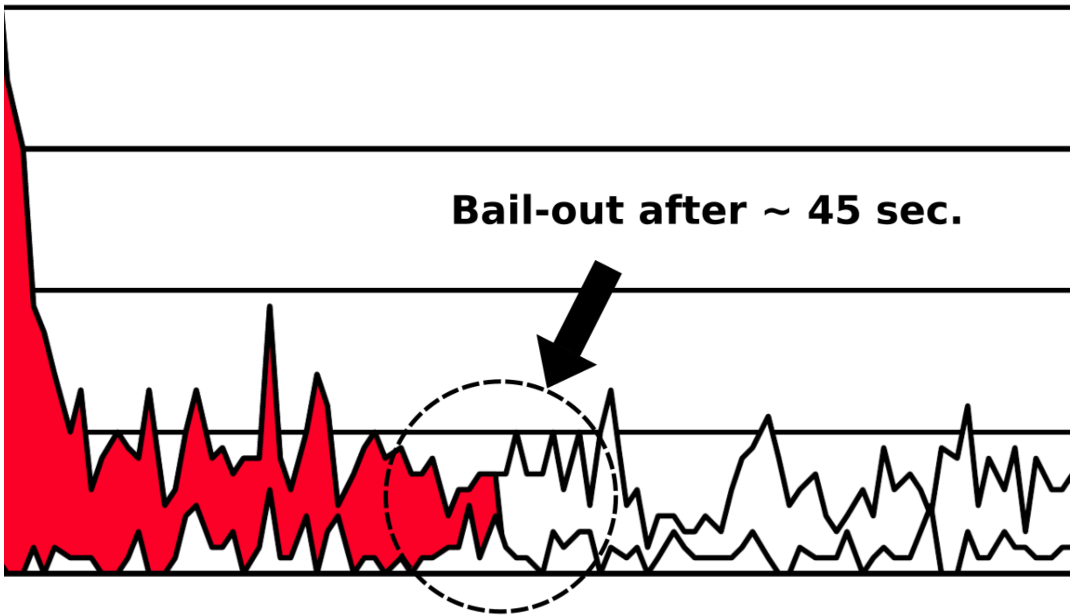
Speedup of +O2 optimized PA-8000 binaries running on Dynamo, relative identical binaries running standalone.



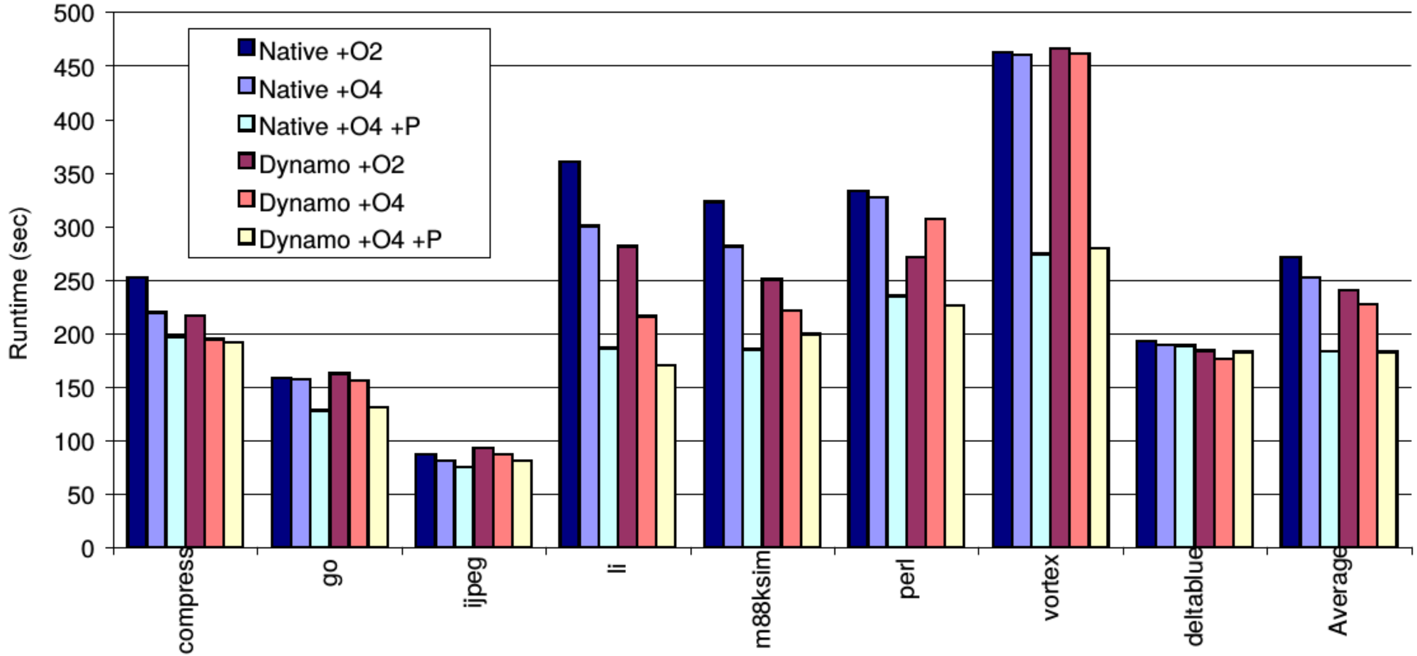
Performance data



Performance data



Performance data



Conclusion

- Complements the static compiler
- Focuses on “run-time only” opportunities (that the compiler might miss)
- No user intervention
- Client-side performance delivery mechanism
- Provides significant benefits even on highly optimized binaries