# Dynamo: A Transparent Dynamic Optimization System

Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia
Hewlett-Packard Labs
Chadd Williams

SysChat

13 Feb 2002

# Outline

- Goals & Motivation
- Overview
- Startup Procedure
- Trace Information
- Fragment Information
- Signal Handling
- Performance Data
- Conclusions

# Goals

- Dynamo: a dynamic software optimization system
- Complement static compiler optimizations
- Operate on JIT output or statically compiled binary
- Focus: statically compiled binary

# Motivation

- Static compiler optimizations less effect because of new technology
  - Object Oriented Languages
  - Delayed binding (dll/so)
  - Dynamically generated code (JIT/dynamic binary translators)
- Dynamo operation is transparent
  - *Don’t require software vendors to change*
  - *Allow computer system vendors more control*

# Overview

- Dynamo *interprets* the application
- Watches for a "hot" trace
  - Consecutively executed instructions
  - Hopefully will be across static program boundaries
- Optimize the "hot" trace
- Produce an optimized code fragment
- Cache the fragment and jump to it during interpretation
- Watches overhead and bails when necessary

# Start up

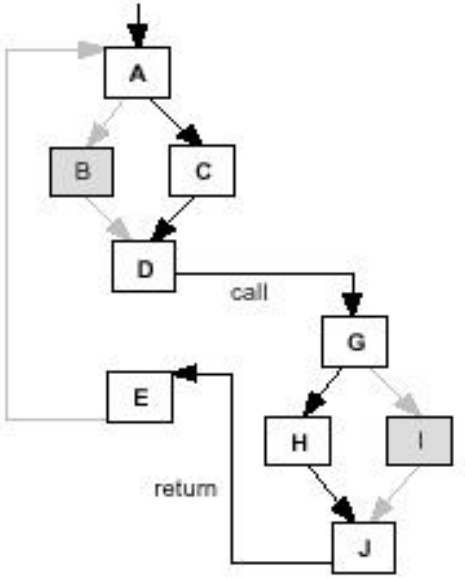
- User-mode shared library
- Application must invoke Dynamo
- Provides modified crt0.o to link with application
- Dynamo creates it own stack
  - Does not interfere with the program stack/context
- Dynamo mmaps memory for its own use
  - Fragment cache
  - counters

# Trace

- “A dynamic sequence of consecutively executed instructions”
- Start-of-trace
  - Target of backward-taken branches (loops)
  - Fragment cache exit branches
- End-of-trace
  - Backward taken branches
  - Taken branch to a fragment entry point

# Trace Selection

- Predictability! (not accuracy)
- Most Recently Executed Tail (MRET)
  - Renamed Next Executing Tail (NET)
  - Associated counter with start-of-trace
  - Trace is “hot” after a threshold number of executions
    - Begin recording instructions
    - Stop on end-of-trace
  - No profiling needed



# Trace Optimization

- Instructions converted to low-level intermediate representation
- Transform loops so the fall through is on trace
  - Loops are only allowed to target the start-of-trace
  - Otherwise a loop is a trace exit
- Conditional branches
  - Predicted target is on trace
  - Other target is in a Dynamo-maintained switch table
    - To other fragments
    - Exit to the interpreter

# Trace Optimization

- Unconditional branch at the end to return to interpreter
- Single-entry, multi-exit sequence of instructions
- Optimizer is non-iterative, two pass
- Optimizations
  - Redundant branch elimination
  - Redundant load removal
  - Redundant assignment elimination

# Trace Optimization

- Move all partially redundant instructions to off-trace compensation blocks
  - Only execute these blocks when necessary

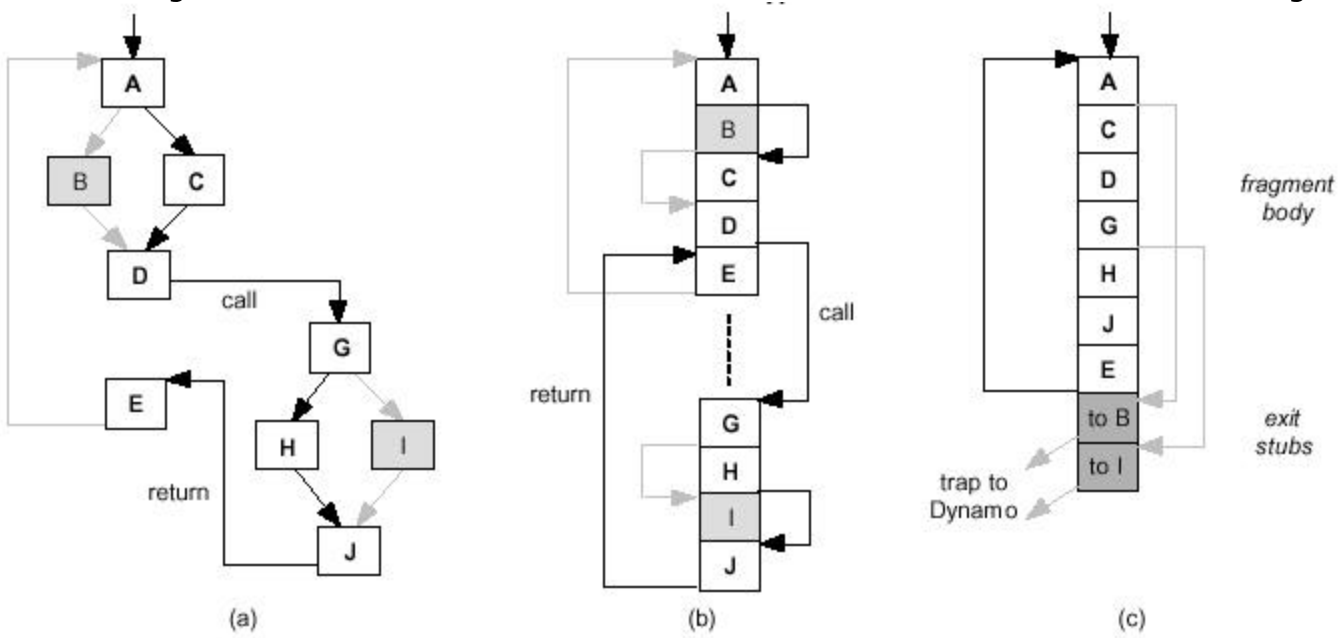



**Figure 3. Control flow snippet in the application binary, (b) Layout of this snippet in the application program's memory, and (c) Layout of a trace through this snippet in Dynamo's fragment cache.**

# Fragment Generation

- Fragment: optimized version of the trace
- Stored in a fragment cache
- Trace may be split into two fragments
- Emit fragment code
- Emit fragment exit stubs
  - Transfer control to Dynamo interpreter
  - Each stub is entered by only one exit branch

# Fragment Linking

- Patch fragment exit block to jump to another fragment entry
- Essential for performance
  - Avoid jumping back to the interpreter
  - *Disabling linking produces a order of magnitude slowdown*
- Allows for the removal of redundant code in the compensation blocks
- Disadvantages:
  - Removal of individual fragments expensive
  - Relocation of fragments is difficult (?)

# Fragment Linking

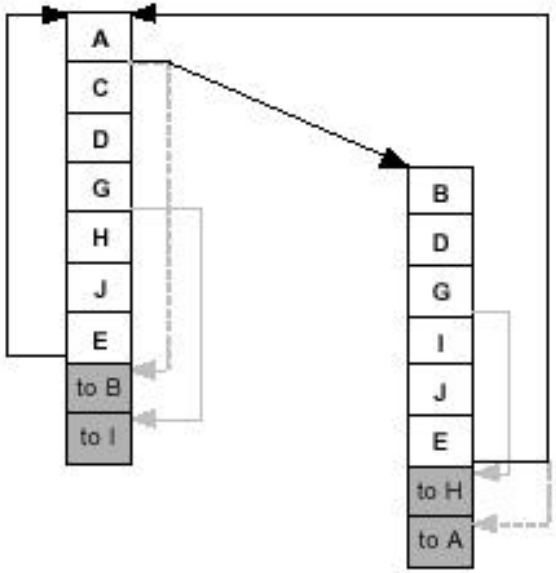



**Figure 4. Example of fragment linking**

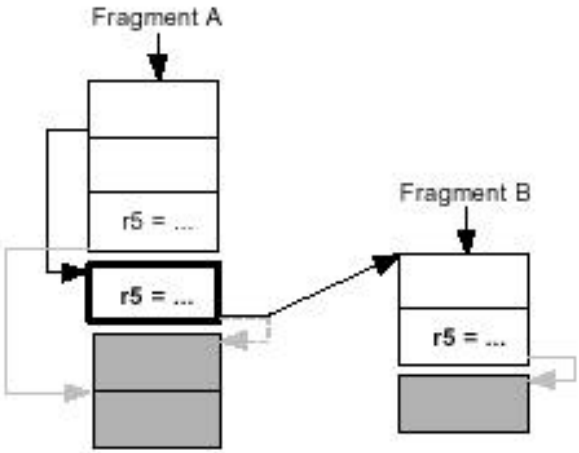



**Figure 5. Example of link-time optimization**

# Fragment Cache

- No complicated management scheme
- Keep the fragments close together -- Locality!
- Need to flush fragments as current working set changes
- Hard to remove one fragment
- “Novel pre-emptive flushing”
  - Flush the entire cache at once
  - Triggered by high fragment creation rate

# Signal Handling

- Dynamo intercepts all signals
- Runs the applications signal handler in the interpreter
- Asynchronous signals (keyboard) are queued until the fragment cache is exited
  - To ensure the fragment cache is exited all fragments are unlinked upon receiving a signal
  - Gradually relinked as executed

# Signal Handling

- Synchronous signal handling
  - Cannot be postponed
  - Expects the process context to be in a certain state
    - This may not be the case with code optimization
  - Dynamo keeps an optimization log to recreate ‘correct’ process context
- Backs off some optimizations if “suspicious” instructions are encountered
  - Dead code removal
  - Code sinking
  - Also flush fragment cache

# Performance Data

- Integer benchmarks
  - SpecInt95
  - deltablue (commercial C++ code)
    - Incremental constraint solver
- HP C/C++ compiler
- HP PA-8000
- HP-UX 10.20
- Fixed size 150Kbyte fragment cache

# Performance Data

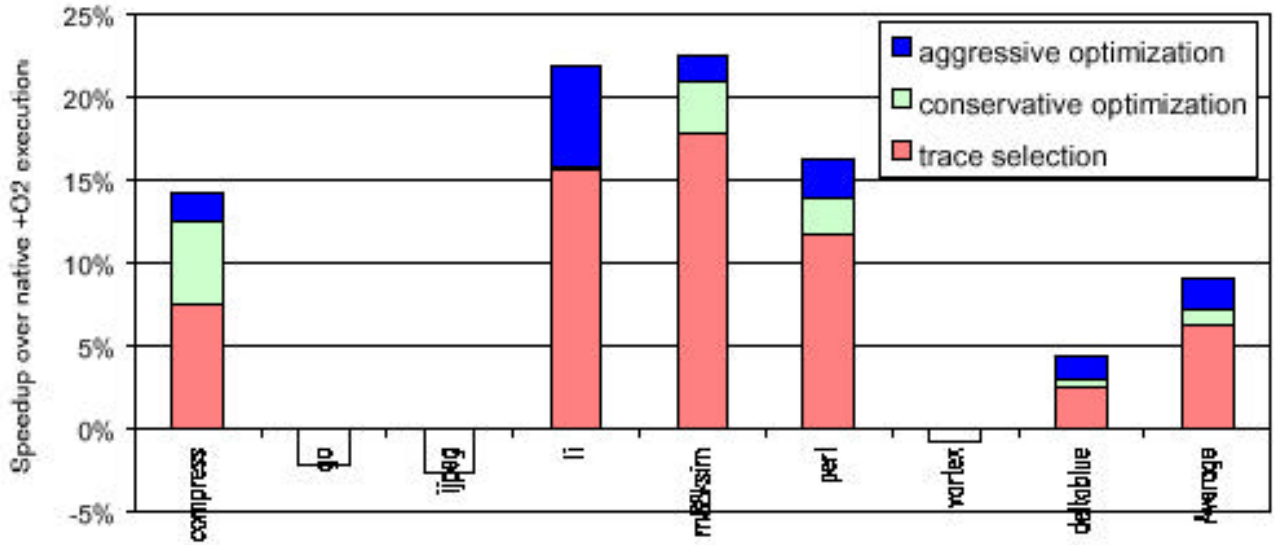



**Figure 7. Speedup of +O2 optimized PA-8000 binaries running on Dynamo, relative to the identical binaries running standalone. The contributions from dynamic inlining due to trace selection, conservative trace optimization and aggressive trace optimization are shown. Dynamo bails out to direct native execution on *go* and *vortex*.**

# Performance Data

- Compiled with +O2
- Stable working set – good results
- Unstable working set/short runtime -- bail
- Forming a fragment improves speed
  - Inlining code!
- Fragment optimization
  - **3% of total gains** – (why bother?)
  - 1/3 of this due to *conservative* optimizations

# Performance Data

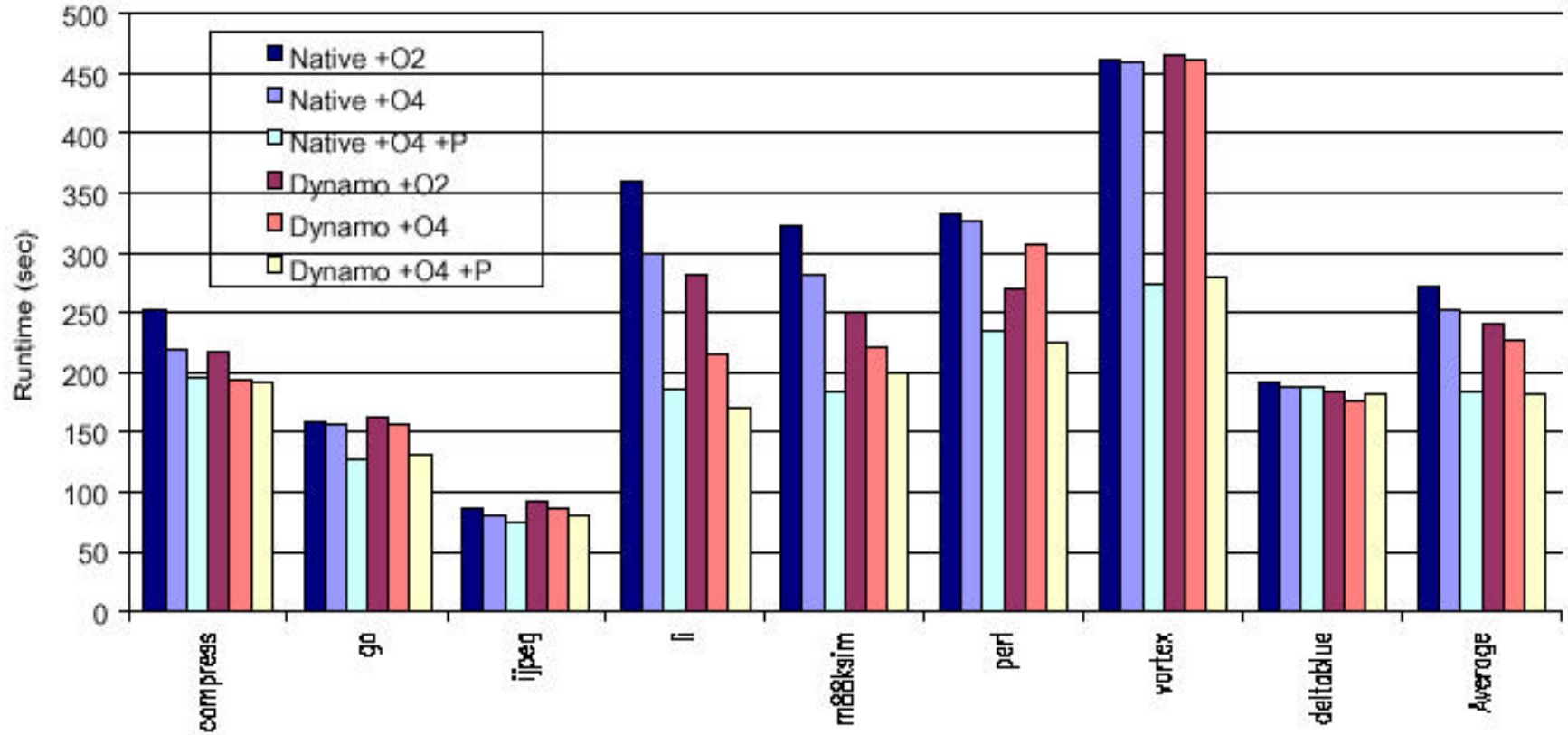



**Figure 8. Dynamo performance on native binaries compiled at higher optimization levels (the first 3 bars for each program correspond to the native runs without Dynamo, and the next 3 bars correspond to the runs on Dynamo)**

# Performance Data

- Compiled at higher optimization levels
- Average performance of +O2 binaries improved to +O4 levels
- Improves performance of +O4 binaries
- Dynamo does not improve much on profiled code

# Conclusions

- Data meant to establish worst case behavior
  - Binaries meant to be tough to improve
    - No late binding
    - Highly optimized
  - Still works well
- Dynamo gets a *huge* performance boost for merely inlining the code
- Will this work for large programs?
- Why not instrument code to find hot start-of-trace?