Dynamic Binary Optimization

A comprehensive comparison between Dynamo, DynamoRIO, ADORE/Itanium, and ADORE/Sparc

Chih-Yung Liang & Shih-Kai Lin

June 8, 2017

Department of Computer Science and Information Engineering,
National Taiwan University

Outline

Dynamic Binary Optimization

Optimization Mechanisms

Comparison

Conclusion

Dynamic Binary Optimization

Introduction

Dynamic Binary Optimization (DBO) aims to improve the performance of an executing native program.

Why Static Optimization is Insufficient?

Analysis Scope

Cross-module optimization is complex and even unavailable, especially for dynamic linked/shared libraries.

Machine Dependent Optimization

Code optimized statically are not optimal for every micro-architecture.

Behavior Information

Behavior of some program vary greatly for different inputs. However, static optimizers can only rely on past profile data, which is not as accurate as those retrieved in runtime.

Aggressiveness

Aggressive optimizations applied speculatively by static optimizers are not revertible even they are eventually not giving good effect on the performance or causing incorrectness.

Challenges

- The **overhead** of performing optimizations are parts of execution time.
- The dynamic optimizer has to realize the **hotspot** of the program.
- The optimizer has to be **transparent** to the target program.

Timeline

PLDI'00	Dynamo (PA-8000)	*Dynamo: A Transparent Dynamic Optimization System*
CGO'03	DynamoRIO (IA-32)	*An Infrastructure for Adaptive Dynamic Optimization*
JILP'04	ADORE (Itanium 2)	*Design and Implementation of a Lightweight Dynamic Optimization System*
MICRO'05	ADORE (UltraSPARC)	*Dynamic Helper Threaded Prefetching on the Sun UltraSPARC® CMP Processor*

Optimization Mechanisms

Mechanisms

Startup

In convenience to modify the target program, it is better to have the dynamic optimizer share the **same address space** with the target program. As a result, optimizers are often built as shared libraries and loaded into the target process using following methods:

- The target application **explicitly call** a routine starting the optimizer.
  - Dynamo
- Use a modified C runtime (`libc_start_main`) to start the optimizer.
  - Dynamo
  - ADORE
- Have the program loader to preload it by setting the environment variable `LD_PRELOAD`.
  - ADORE

Interpretation

The optimizer has to realize the hotspot of the executing program, which is usually viewed as code that are **executed many times**.

Dynamo starts to run a program using a lightweight software interpreter to count execution frequencies of code sequences.

Frequently executed code sequences are selected as traces to optimize. Generated native versions of traces (called fragment) are stored in the code cache. After that, future executions of these traces use the native code in the code cache instead of interpreting.

Trace

Dynamo defines *start-of-trace* as **targets of backward-taken branches**, which are likely loop headers, and fragment cache exit branches.
End-of-trace are **backward-taken branches** or taken branches whose targets correspond to the entry point.

A fragment is built as an optimized **single-entry, multi-exit** contiguous sequence of instructions.

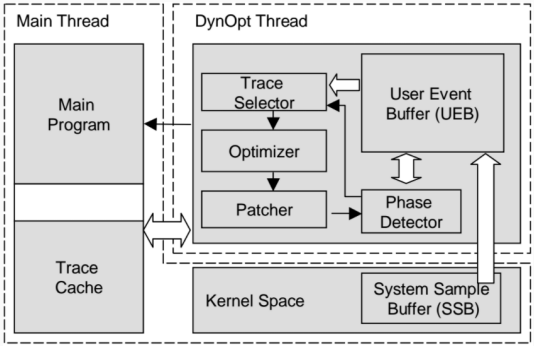
Cached Interpretation

To overcome the high overhead caused by the interpretation, DynamoRIO **caches** translations of frequently executed code. It copies **basic blocks** into code cache and executes them natively. At the end of a block, the machine state is saved and control is returned to DynamoRIO. The interpretation overhead is furthermore decreased with following additional optimizations:

System Type	Normalized Execution Time	
	crafty	vpr
Emulation	~ 300.0	~ 300.0
+ Basic block cache	26.1	26.0
+ Link direct branches	5.1	3.0
+ Link indirect branches	2.0	1.2
+ Traces	1.7	1.1

Sampling-based Profiling

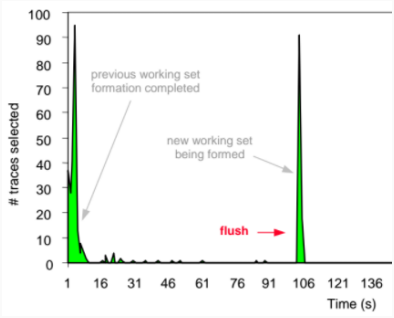
ADORE utilizes *Itanium 2*'s **Performance Monitoring Unit** (PMU) and some system services to figure out the hotspot of an executing program, so the target program is **run natively**. The trace building and optimizing are done in **the other thread**, which is run on the other physical core.



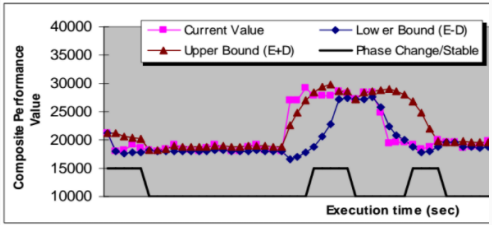
Phase Detection

Programs have phases. It is more meaningful to optimize code in **stable phase**.

Dynamo **flushes** its code cache when the fragment creation rate has a sharp increase.



ADORE monitors **PC** and **CPI** for phase stability and performs optimizations only when stable.



Mechanisms

Optimizations in Dynamo

Conservative Optimizations

- Redundant branch elimination
- Redundant assignment elimination
- Copy/constant propagation
- Strength reduction
- Loop unrolling

Aggressive Optimizations

- Dead code removal
- Code sinking
- Loop invariant code motion
- Redundant load removal

Since Dynamo is interpretation-based and use little hardware support, it is hard to realize and optimize various types of performance behavior, such as **cache** and **branch prediction**.

Optimization Examples in DynamoRIO

DynamoRIO is a **framework** for implementing dynamic analysis and optimizations. The paper describes several example DynamoRIO clients.

- Redundant load removal
- Strength reduction
- Indirect branch dispatch
- Custom traces

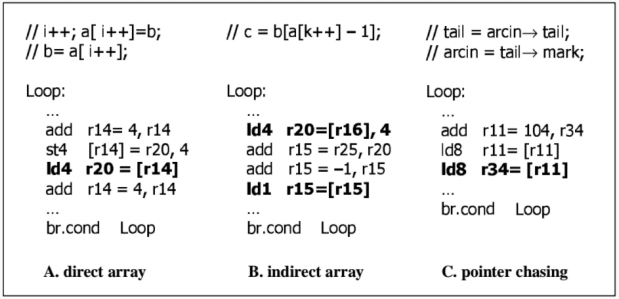
Optimization in ADORE/Itanium (Prediction-based Prefetch)

ADORE focuses on inserting **cache prefetches** into fragments to decrease the amount of cache misses.

Steps

1. Tracking delinquent loads
2. Data reference pattern detection
3. Prefetch generation
4. Prefetch code optimization and scheduling

Data Reference Pattern



Optimization in ADORE/SPARC (Execution-based Prefetch)

Execution-based profiling can provide better prefetching *coverage* and *accuracy*. It is especially suitable for the dual-core UltraSPARC, which has **private L1 caches** for each core and a **shared L2 cache**.

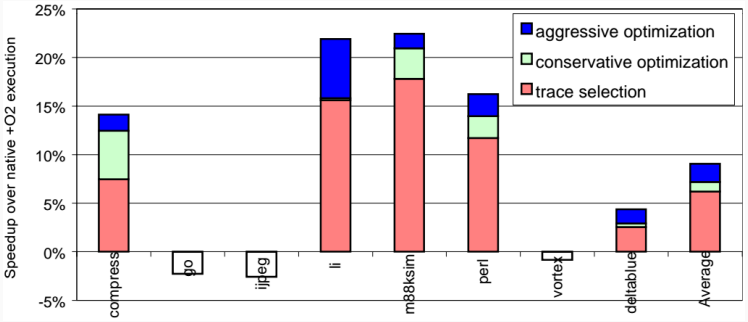
Extended from ADORE/Itanium, it makes the optimization thread also a helper thread, which executes a **distilled version** (called scouting code) of the forward slice starting from the missing load. The distillation includes:

- Memory stores and unrelated instructions are ignored, leaving only delinquent loads and instructions changing the control flow.
- Delinquent loads are replaced by **strong prefetches** unless their results are used by subsequent computations.
- Otherwise, delinquent loads are converted into **non-faulting loads**.

Mechanisms

Performance - Dynamo

- Average speedup: 9%
  - Mainly from trace selection
- Long execution time
  - ijpeg performs badly
- Stable working set
  - li, m99skim, perl, and compress
  - go and vortex on the contrary

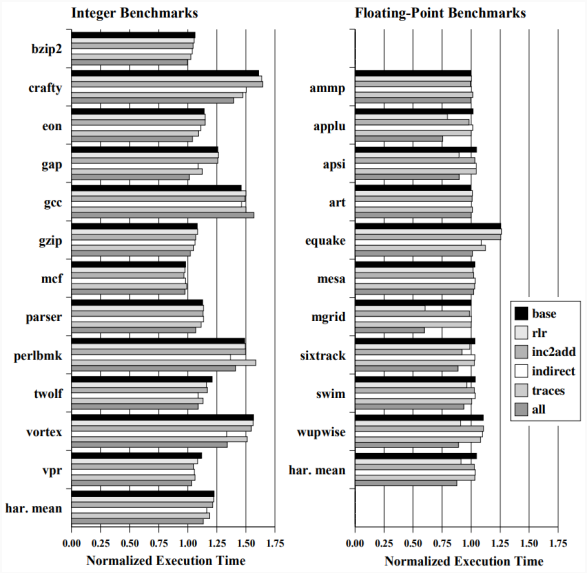


Performance - Dynamo

- Specific to the architecture (code generated by the compiler, implementation)
  - Branch misprediction penalty is 5 cycles.
  - Indirect Branch are always mispredicted.
- Large instruction cache
  - Gain from I-cache locality improvement is not significant.
- Unified instruction and data TLB with only 96 entries
  - Better locality of the working set in the fragment cache results in better performance.

Performance - DynamoRIO

- FP benchmark: 12% speedup on average
  - Redundant load removal (`rlr`)
    - Static compilation is limited by basic block.
  - Strength reduction
    - Optimized for specific architecture
    - `inc` is slower than `add 1` on *Pentium 4* but faster on *Pentium 3*
  - Indirect branch dispatch
    - Inline indirect branch targets when building the trace



Performance - DynamoRIO

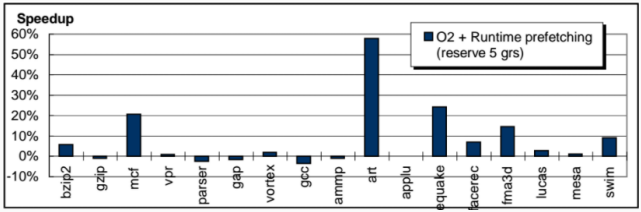
The overhead comes from:

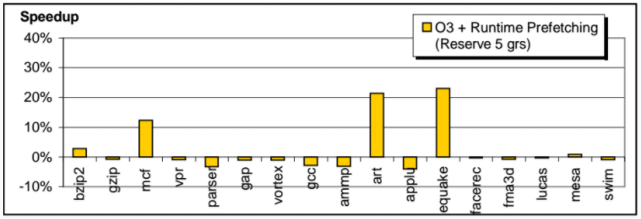
- Indirect branches handling
  - Indirect branches (including **returns** and **indirect calls**) are translated into indirect jumps.
  - The hardware has return address predictors but no indirect jump predictors.
  - Misprediction cost is higher than native.
- Dealing with `eflags` changes caused by introduced code.

Performance - ADORE

Cache misses on Itanium machines have significant impact on performance.

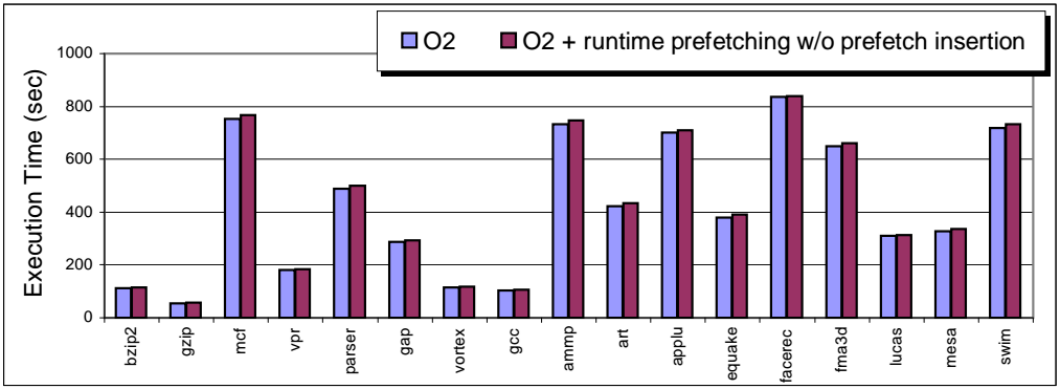
- O2 + reserve registers
  - Without static prefetch
  - Speedup: 3%–57%
  - No effect: -2%–1%
- O3 + reserve registers
  - With **static prefetch**
  - Speedup: 11%–22%
  - No effect: -3%–2%





Performance - ADORE

- Overhead is trivial (average: 2%).
  - Contiguous sampling, phase detection, trace optimization

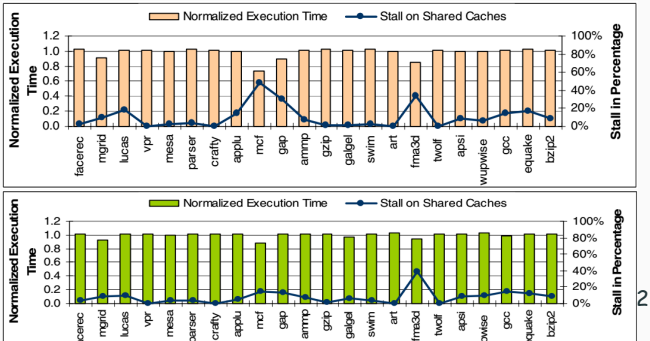


Performance - Helper-threaded prefetch

- Max speedup: 35% (mcf)
  - Memory Level Parallelism (MLP) helps hide latency by overlapping cache misses.
- Overhead: 1%-2%

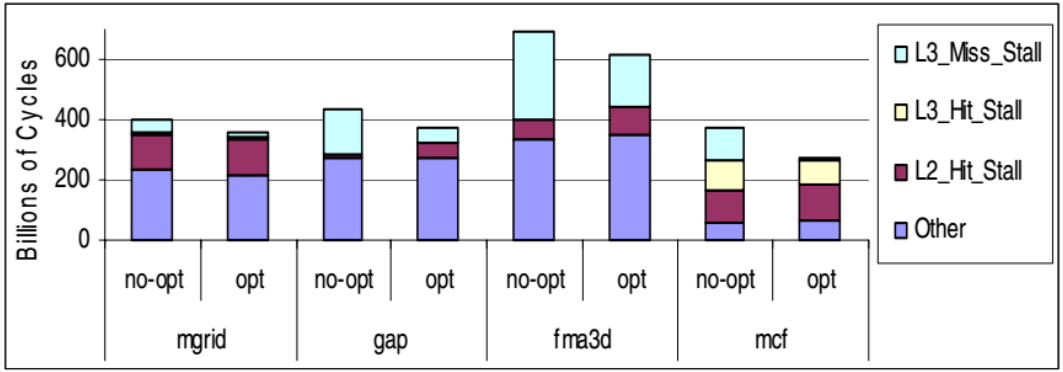
Fluent

SPEC2000



Performance - Helper-threaded prefetch

- Particularly effective on hiding L3 cache miss latency.
- Does not decrease L1 data cache miss penalty (`L2_Hit_Stall`).



Comparison

Comparison

	Dynamo	DynamoRIO	ADORE/Itanium	ADORE/SPARC
Profile (Overhead)	Interpretation (High)		Sampling-based profiling (Low)	
Target architecture	HP PA-RISC	IA-32	Itanium 2	UltraSPARC (Panther)
Optimization	Redundant load/branch/assignment elimination, copy/constant propagation, strength reduction, LICM, and unrolling		Prediction-based prefetch	Execution-based profiling
Speedup	Avg: 9% Max: 22%	Floating avg: 12% Max: 40%	3–57%	Max: 35%
Mainly gain from	Code layout and indirect branch		Decreased cache miss penalty	
Performance impact when no gain	High (Bailout)	High (Mean 12%)	Low (Avg: 2%)	Low (1–2%)

Conclusion

Conclusion

- Dynamic optimization can deal with some problems that cannot be done statically.
  - Cache misses & branch misprediction patterns are difficult to optimize at static time.
  - Find the hot code and optimize.
- Sampling-based profiling has negligible overhead.
  - But need more time to collect enough data.
- Static + Dynamic integration?
  - Annotation on may-be-hot code.
- Multi-threaded programs are increasing.
  - ADORE/COBRA can optimize for multi-cores.

Q & A

Current DynamoRIO

DynamoRIO announced 7.0 release candidate (4 Feb, 2017) which contains support for AArch64. Until now, DynamoRIO supports:

Architectures:

1. IA-32
2. AMD64
3. ARM
4. AArch64

Operating Systems:

1. Windows
2. Linux
3. Android

It is now a open source project with binary packages available:
https://github.com/DynamoRIO/dynamorio

There will be a tutorial on DynamoRIO held at **CGO 2017**.