A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware

Kangkook Jee
Columbia University

Joint work with
Georgios Portokalidis[1] , Vasileios Kemerlis[1] ,
Soumyadeep Ghosh[2], David August[2], Angelos Keromytis[1]

[1]Columbia University, [2]Princeton University

Talk Outline

- Data Flow Tracking [VEE 12]
  - Popular Subject of Security Research
  - Tagging and tracking of Interesting data
- Taint Flow Algebra (TFA) [NDSS 12]
  - Intermediate Representation(IR) for DFT
  - Compiler optimization + DFT specific optimization
- ShadowReplica
  - Decoupling of execution and monitoring
  - Running each from different cores

Talk Outline

- Data Flow Tracking [VEE 12]
  - Popular Subject of Security Research
  - Tagging and tracking of Interesting data
- Taint Flow Algebra (TFA) [NDSS 12]
  - Intermediate Representation(IR) for DFT
  - Compiler optimization + DFT specific optimization
- ShadowReplica
  - Decoupling of execution and monitoring
  - Running each from different cores

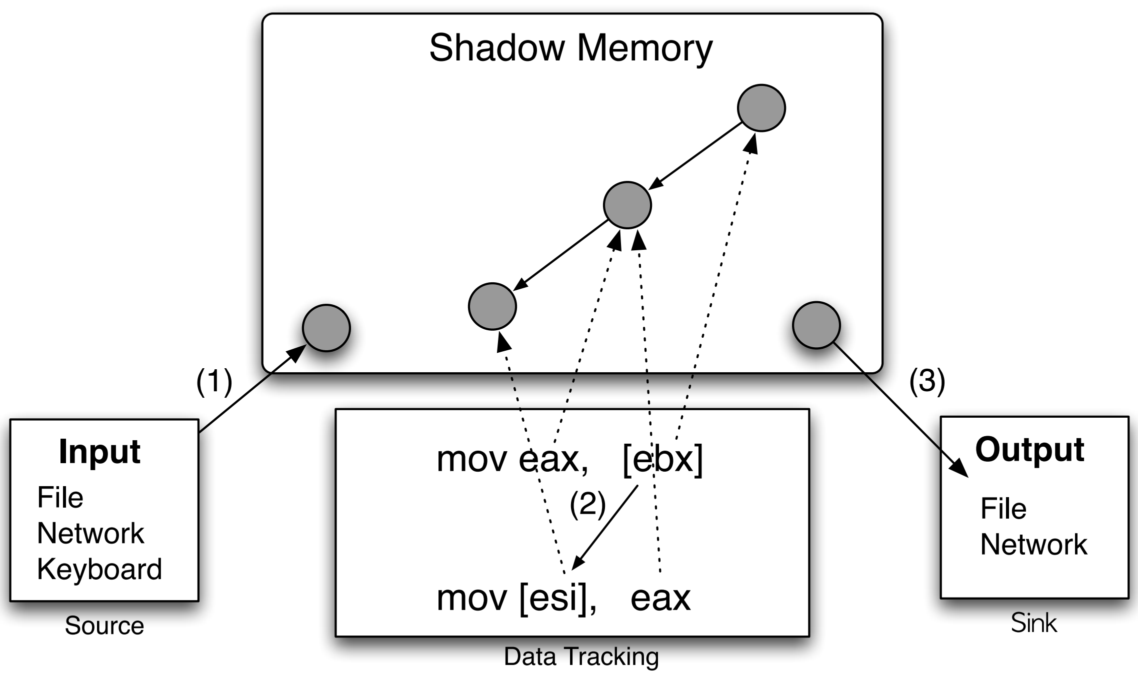
Data Flow Tracking (DFT)

- A great security tool with many applications
  - Tag input data and track them
  - Software exploits, Information misuse or leakage malware analysis …
- Implementation approaches
  - Hardware assisted: Raksha, RIFLE …
  - Source code based: GIFT …
  - Binary only: TaintCheck, Dytan, Minemu, Libdft …

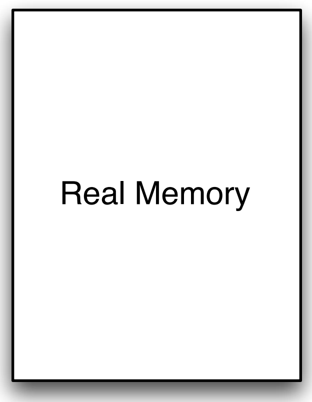
<u>Binary only DFT</u>: Most promising, but too slow!

DFT: Basic Aspects

- DFT is characterized by *three* aspects
  (1) **Data Sources**: program or memory locations where data of interest enter the system and is subsequently tagged
  (2) **Data tracking**: process of propagating data tags according to the program's semantics
  (3) **Data Sinks**: program or memory locations where checks for "tagged" data can be made

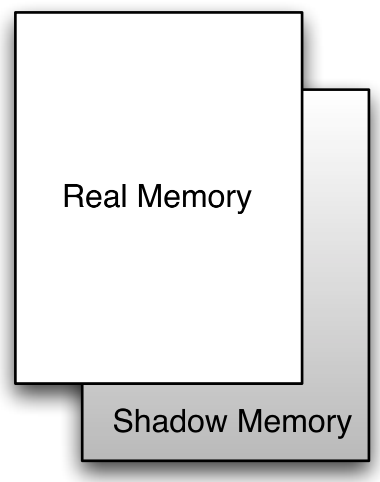


DFT Operation



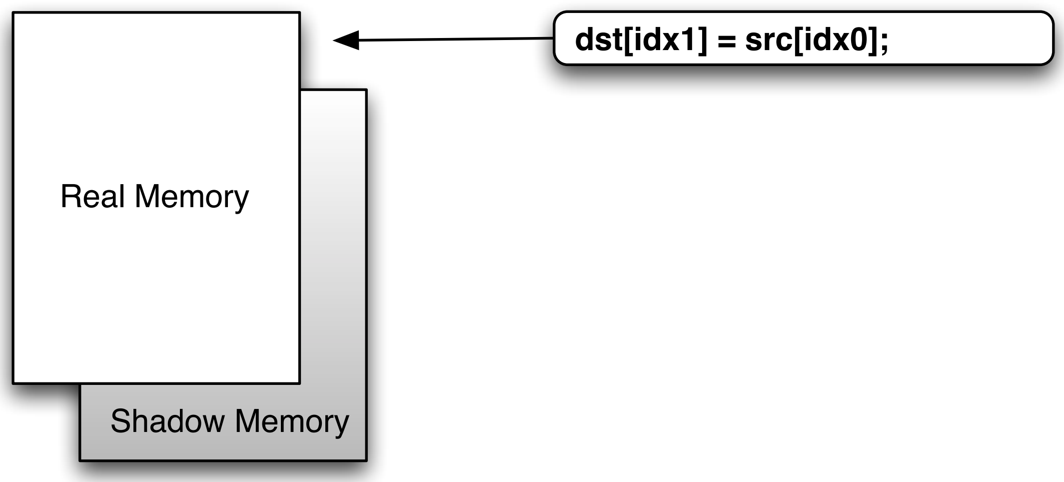
- Real Memory = Address space + register context

DFT Operation



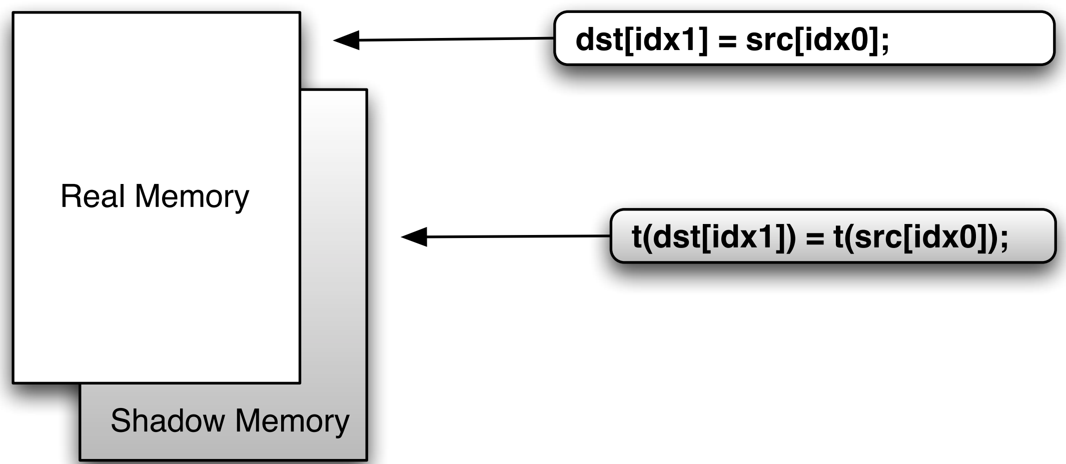
- Real Memory = Address space + register context
- Shadow memory to track metadata update

DFT Operation



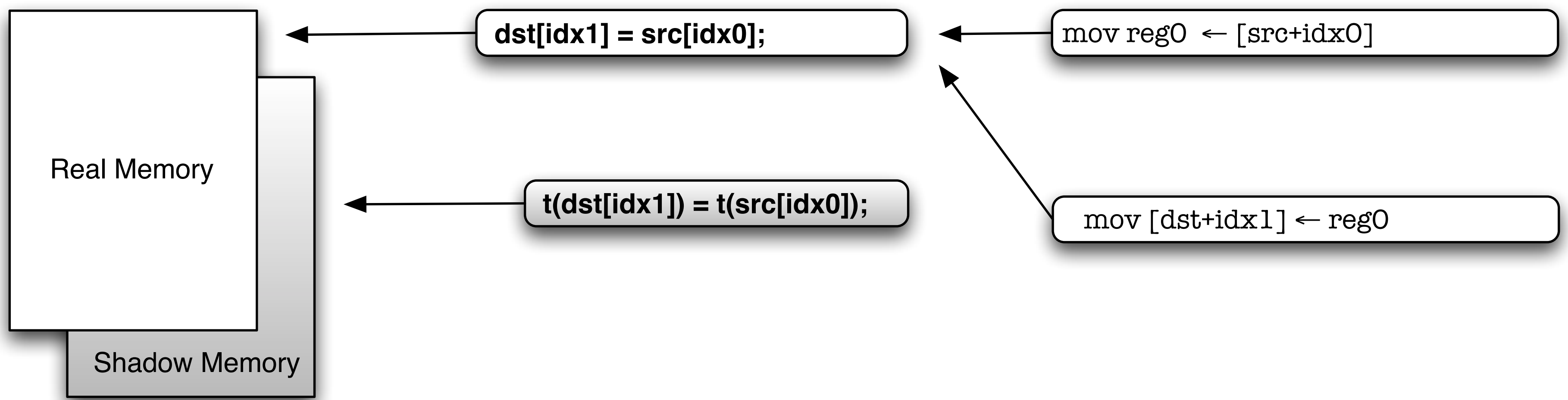
- Memory copy statement from the original execution

DFT Operation



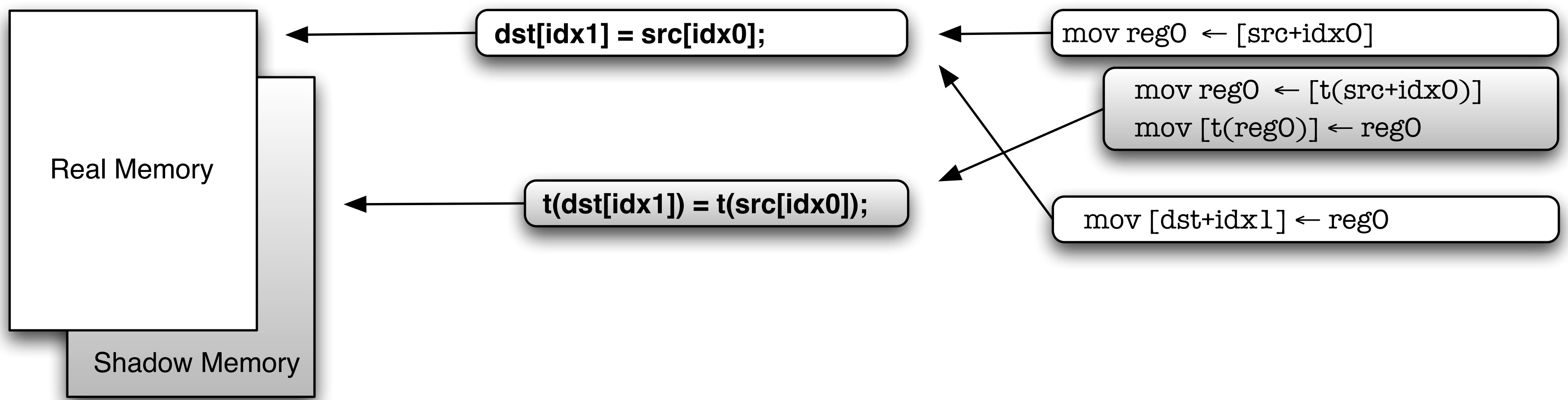
- Memory copy statement from the original execution
- Corresponding shadow memory update

DFT Operation



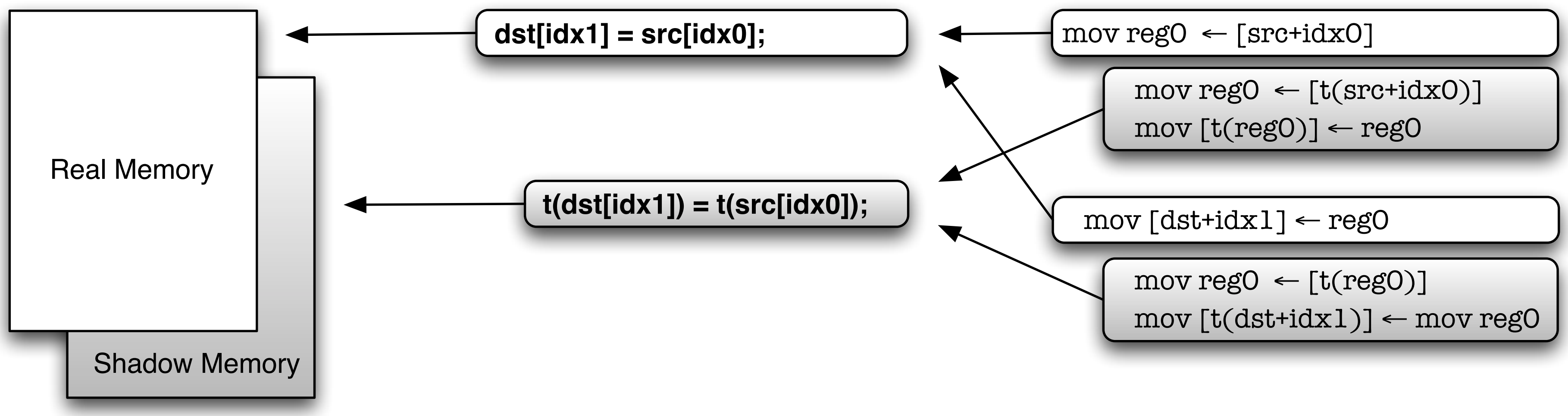
- Original operation translated into machine code
- It requires intermediate register repository (reg0)

DFT Operation



- Instruction level instrumentation to implement shadow update

DFT Operation



- 2 original instructions + 4 tracking instructions
- 2 instrumentation units

Why So Slow?

- Framework cost (virtualization cost)
  - DBI, Hypervisor instrumentation
- DFT cost
  - Accesses to shadow storage
- Naïve Implementation
  - No understanding of global context
  - No understanding of DFT semantics

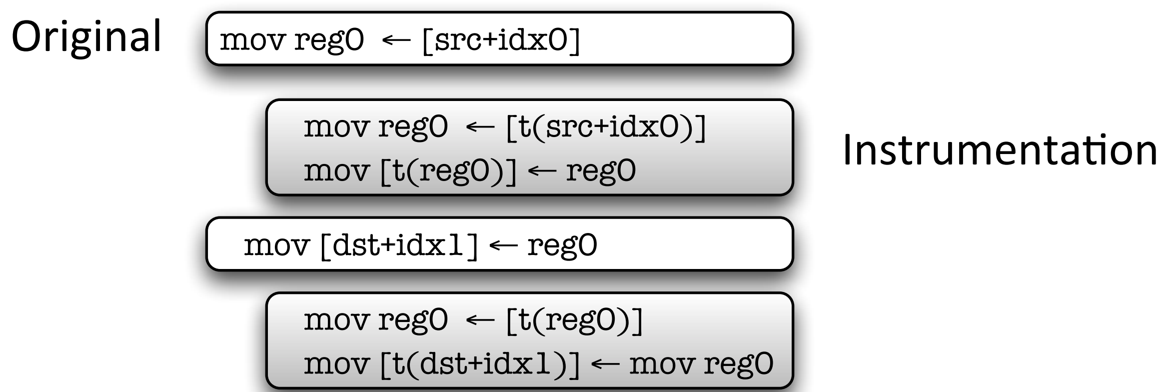
Talk Outline

- Data Flow Tracking
  - Popular Subject of Security Research
  - Tagging and tracking of Interesting data
- Taint Flow Algebra (TFA)
  - Intermediate Representation(IR) for DFT
  - Compiler optimization + DFT specific optimization
- ShadowReplica
  - Decoupling of execution and monitoring
  - Running each from different cores

Taint Flow Algebra

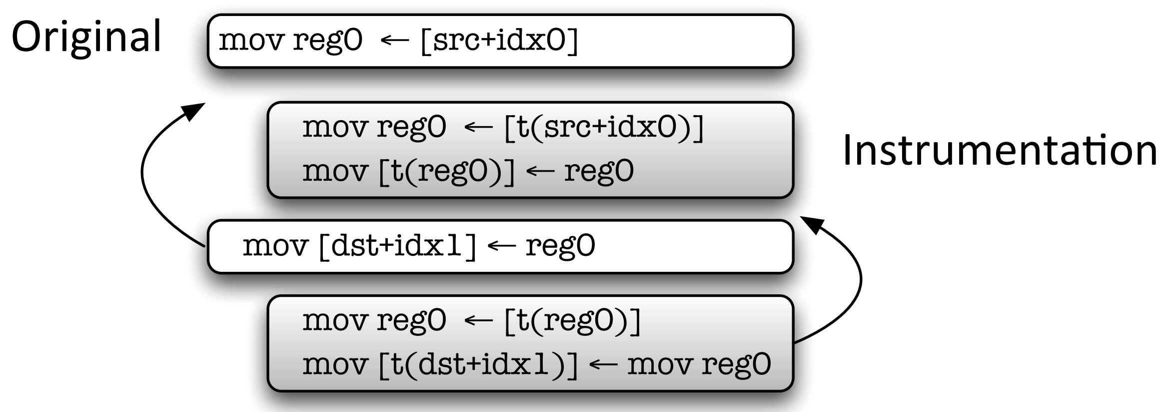
- Application specific analysis
- DFT specific analysis
- Integrated with *libdft*
  - High performance DFT tool [VEE 2012]
    - 1.46x ~ 8x slowdown (over native execution)
  - Designed for use with *Pin DBI framework*
  - Open source
    - http://www.cs.columbia.edu/~vpk/research/libdft

Optimizing DFT



- Each Instrumentation unit requires head/tail instructions
- *t( )* : shadow memory access cost

Optimizing DFT



- Re-locatable

Optimizing DFT

```
mov reg0  ← [src+idx0]
mov [dst+idx1] ← reg0
```

```
mov reg0  ← [t(src+idx0)]
mov [t(reg0)] ← reg0
mov reg0  ← [t(reg0)]
mov [t(dst+idx1)] ← mov reg0
```

- Less instrumentation units (2→1)

Optimizing DFT

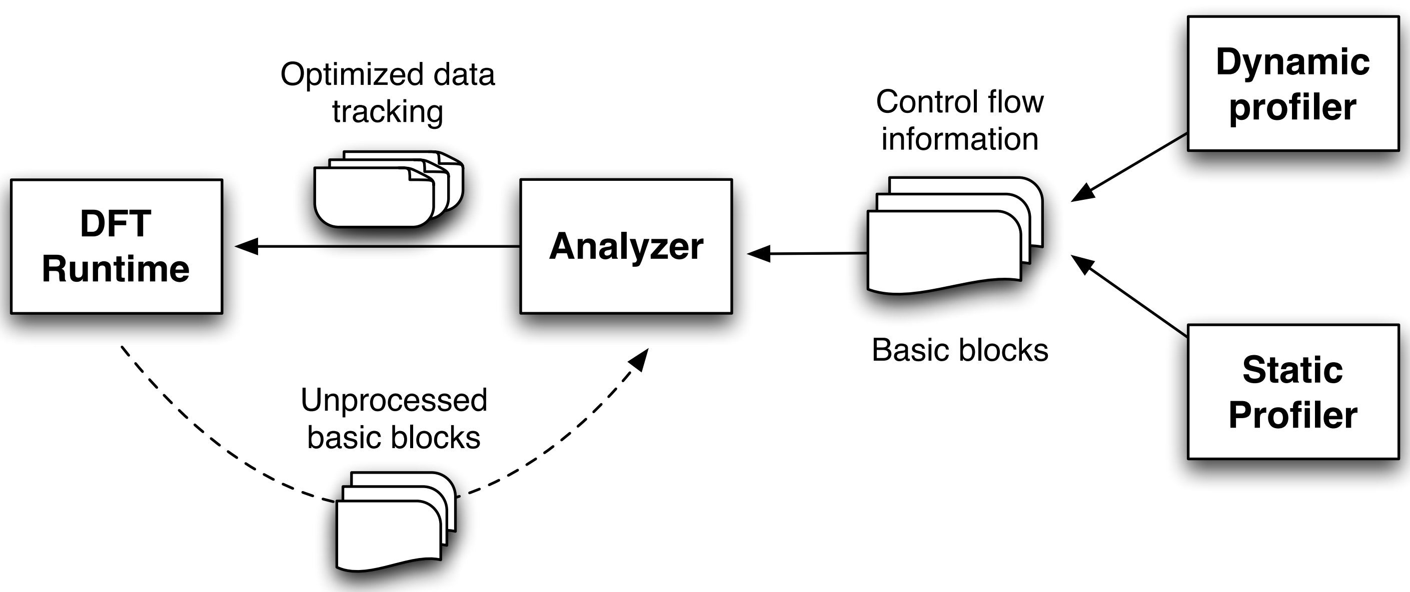
mov reg0 ← [src+idx0]
mov [dst+idx1] ← reg0

mov reg0 ← [t(src+idx0)]
~~mov [t(reg0)] ← reg0~~
~~mov reg0 ← [t(reg0)]~~
mov [t(dst+idx1)] ← mov reg0

- Less instrumentation units (2→1)
- Less tracking instructions (4→2)

Execution Model

- 3 Components
  - Profiler, Analyzer, DFT Runtime
- Static/offline analysis + Dynamic runtime
  - Feedback loop



Analyzer

- Taint Flow Algebra
  - Represent binary analysis result
  - IR tailored to capture DFT semantics
- Compiler optimization to TFA
  - Inner (intra) basic block:
    Dead code elimination, Algebraic simplification, …
  - Outer (inter) basic block:
    Data flow analysis
- DFT specific considerations
  - Valid location for each instrumentation unit
  - Number of instrumentation units

TFA Optimization

```
1: mov ecx, esi
2: movzxb eax, al
3: shl ecx, 0x5
4: add edx,0x1
5: lea esi, ptr [ecx+esi]
6: lea esi, ptr [eax+esi]
7: movzxb eax, ptr [edx+esi]
8: testb al, al
9: jnzb 0xb7890200
```

(a) x86 instruction

- Per basic block analysis
- Gray instructions: non-tracking instructions

TFA Optimization

1: mov ecx, esi	1: ecx1 := esi0
2: movzxb eax, al	2: eax1 := 0x1 & eax0
3: shl ecx, 0x5	3:
4: add edx,0x1	4:
5: lea esi, ptr [ecx+esi]	5: esi1 := ecx1 | esi0
6: lea esi, ptr [eax+esi]	6: esi2 := eax1 | esi1
7: movzxb eax, ptr [edx+esi]	7: eax2 := 0x1 & [edx0+esi2]
8: testb al, al	8:
9: jnzb 0xb7890200	9:
(a) x86 instruction	(b) TFA transformation

- Translated into TFA
- Input operands, output operands

TFA Optimization

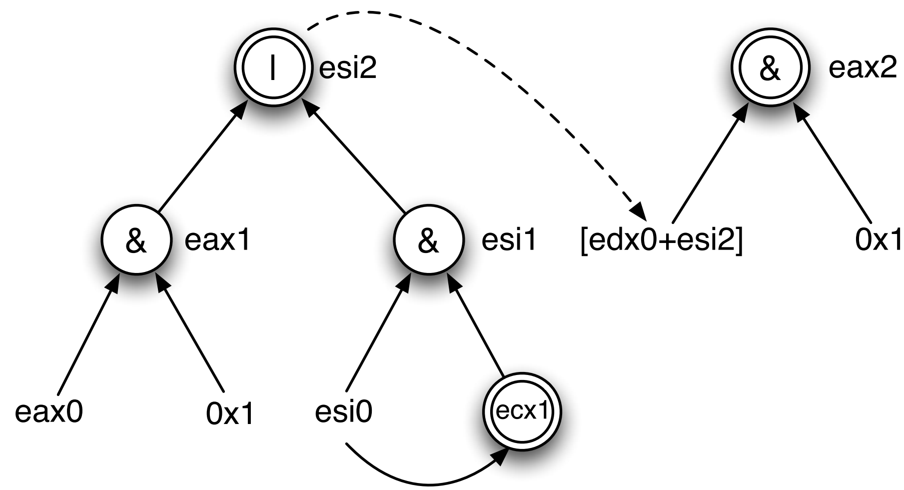
(a) x86 instruction	(b) TFA transformation	(c) TFA optimization
1: mov ecx, esi	1: ecx1 := esi0	1: ecx1 := esi0
2: movzxb eax, al	2: eax1 := 0x1 & eax0	2:
3: shl ecx, 0x5	3:	3:
4: add edx,0x1	4:	4:
5: lea esi, ptr [ecx+esi]	5: esi1 := ecx1 | esi0	5:
6: lea esi, ptr [eax+esi]	6: esi2 := eax1 | esi1	6: esi2 := 0x1 & eax0 |esi0
7: movzxb eax, ptr [edx+esi]	7: eax2 := 0x1 & [edx0+esi2]	7: eax2 := 0x1 & [edx0+esi2]
8: testb al, al	8:	8:
9: jnzb 0xb7890200	9:	9:

- Output operands are expressed in terms of input operands
- Data flow analysis to remove irrelevant outputs

TFA Optimization

1: mov ecx, esi
2: movzxb eax, al
3: shl ecx, 0x5
4: add edx,0x1
5: lea esi, ptr [ecx+esi]
6: lea esi, ptr [eax+esi]
7: movzxb eax, ptr [edx+esi]
8: testb al, al
9: jnzb 0xb7890200

(a) x86 instruction



DAG Representation

- DAG Representation
- Express root nodes in terms of leaf nodes

DFT Runtime

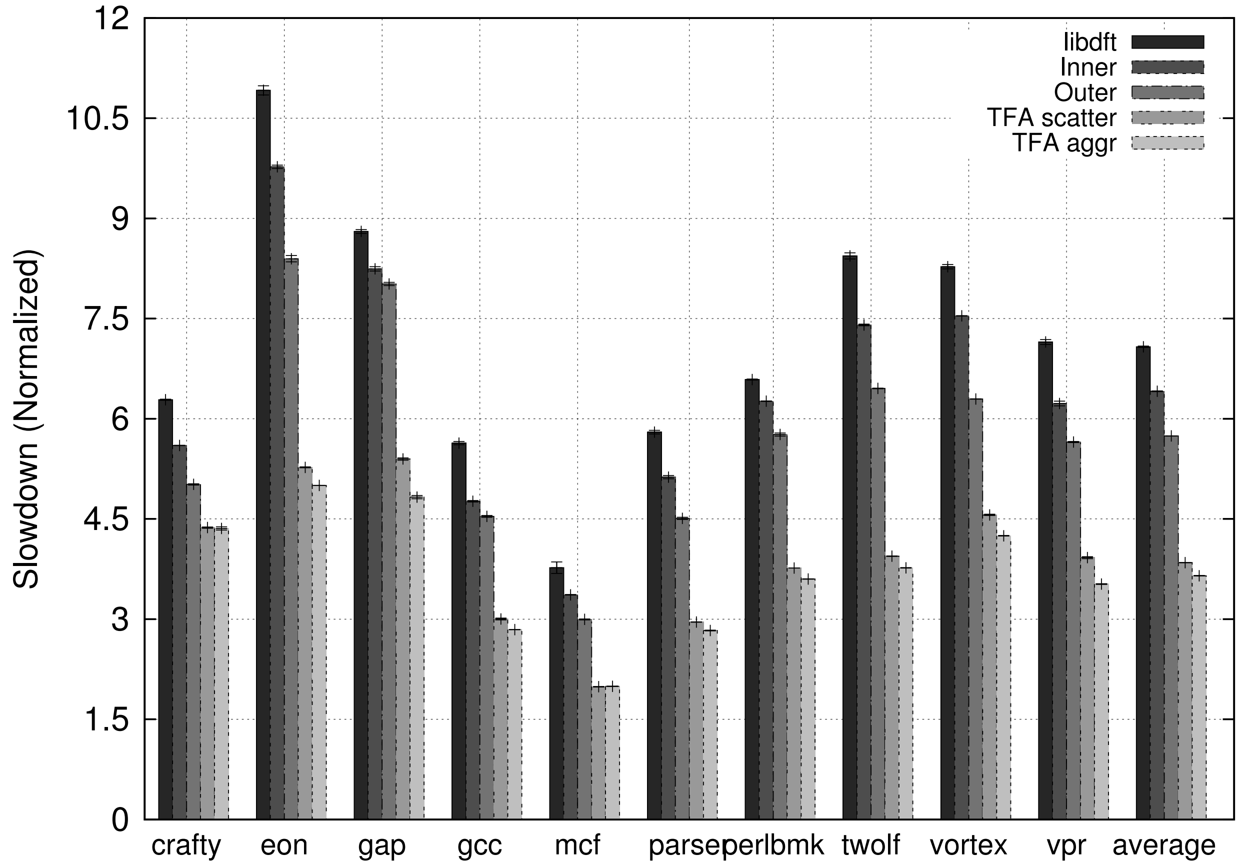
- Generate/Inject optimized tracking code to the baseline DFT platform
  - Translate optimized TFA
- Our prototype extends libdft
- Code generation of libdft/PIN-aware C code
  - A function per each instrumentation unit
  - e.g., Firefox: 50K customized functions

Evaluation

- Optimization schemes
  - Code reduction: Simple dead code eliminations
    - Inner, Outer
  - Code generation: Optimized tracking codes
  - TFA Scatter, TFA Aggregation

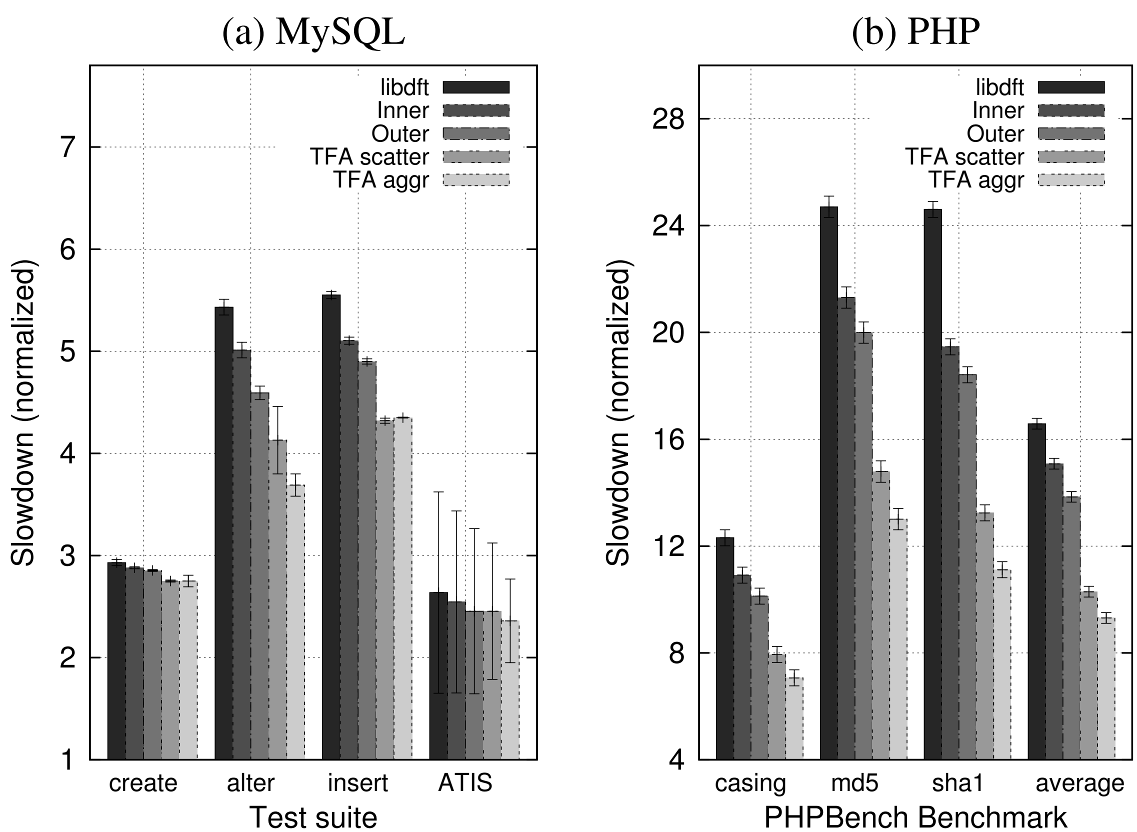
Category	Optimization schemes	CFG Consideration	TFA Optimization	Aggregation
Code reduction	Inner	No	No	No
	Outer	Yes	No	No
Code generation	Scatter	Yes	Yes	No
	Aggregation	Yes	Yes	Yes

Evaluation: SPEC CPU2000



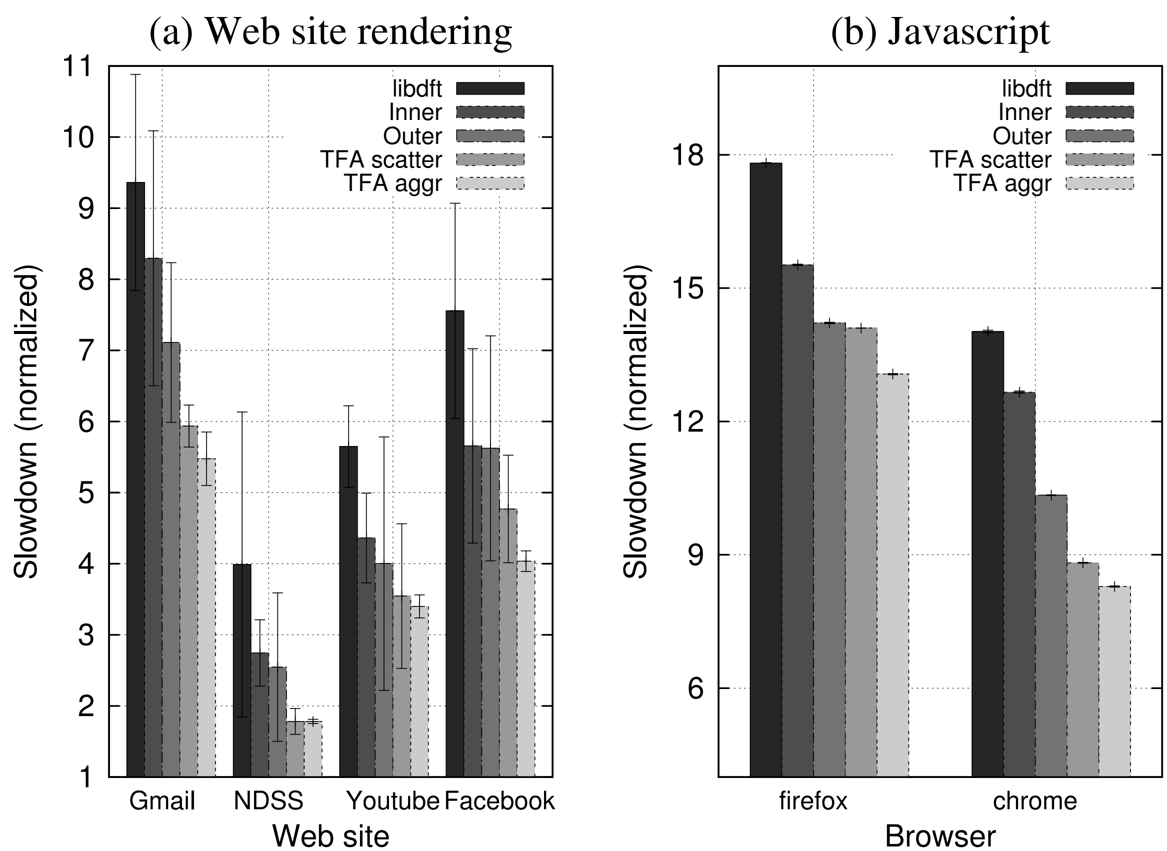
- CPU intensive workloads
- TFA's speedup over libdft: on average 1.90x (the largest 2.23x)
- ~3x slowdown over the native execution

Evaluation: Server applications



- Mysql's own benchmark suite (sql-bench) and PHP micro benchmark suite (PHPBench)
  - Plotted representative subsets

Evaluation: Client Applications



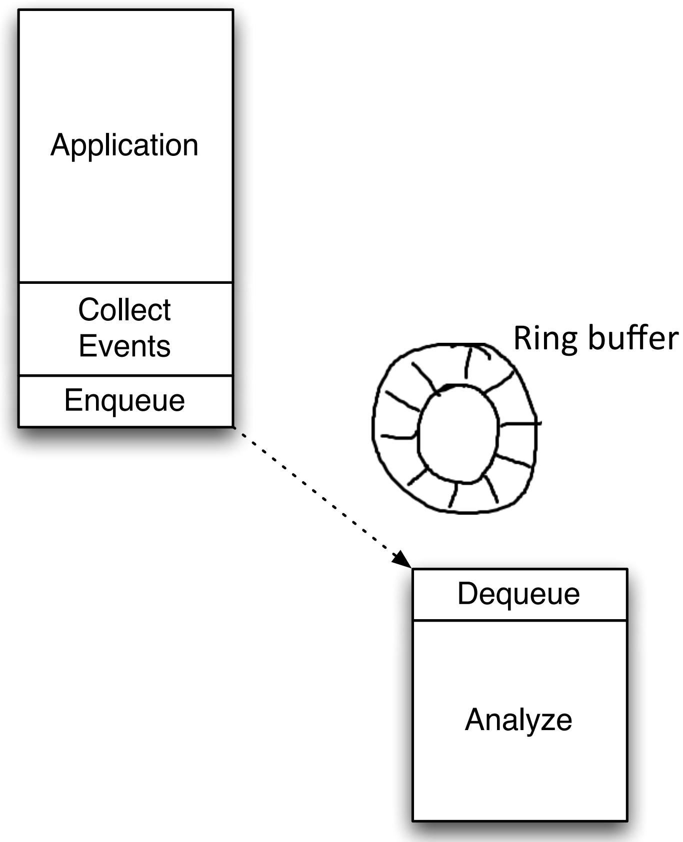
- Rendering measurement for Alexa's Top 500 sites and NDSS 2012 site
  - For Firefox web-browser
- Dromaeo (http://www.dromaeo.com) Javascript benchmark suite
  - For Firefox and Google Chrome web-browser

Talk Outline

- Data Flow Tracking [VEE 12]
  - Popular Subject of Security Research
  - Tagging and tracking of Interesting data
- Taint Flow Algebra (TFA) [NDSS 12]
  - Intermediate Representation(IR) for DFT
  - Compiler optimization + DFT specific optimization
- ShadowReplica
  - Decoupling of execution and monitoring
  - Running each from different cores

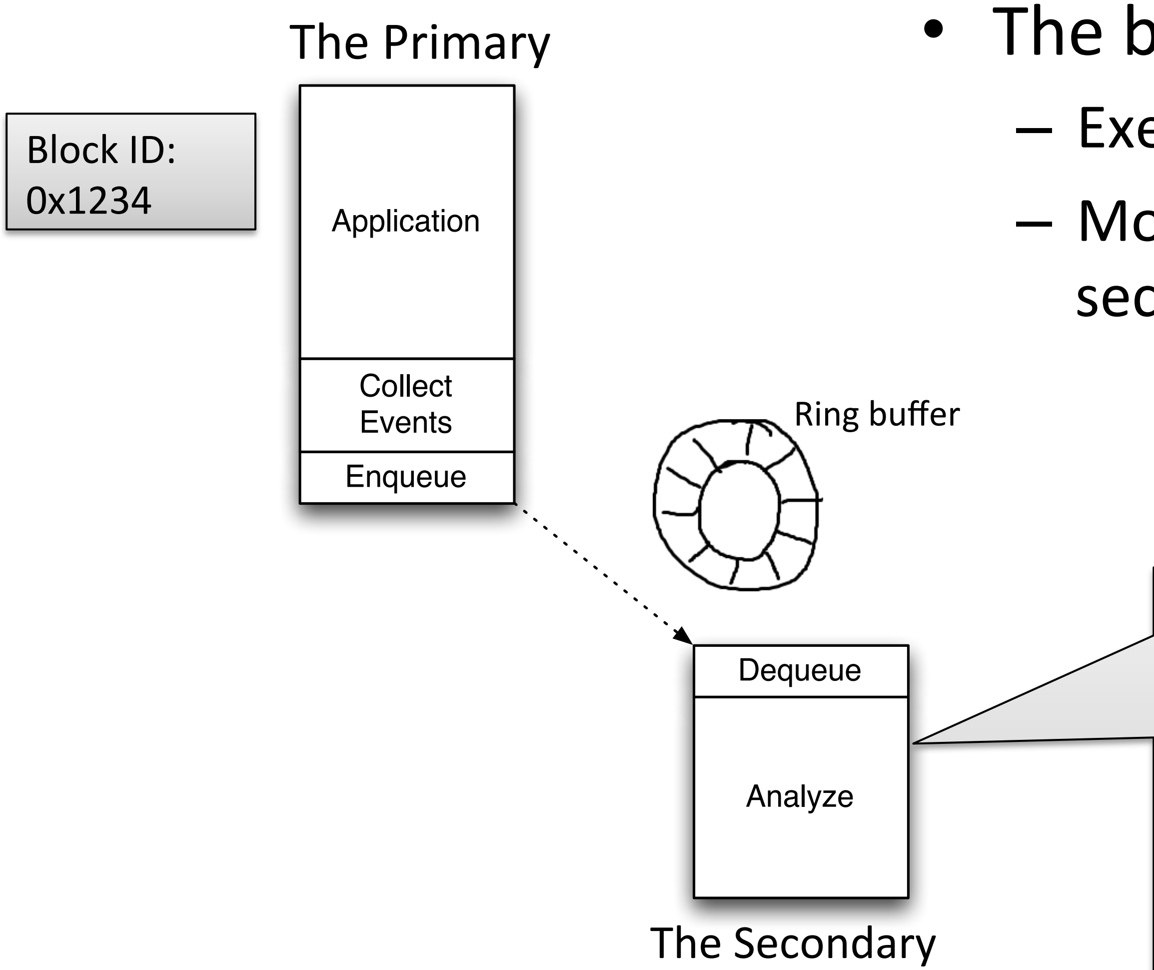
ShadowReplica: Parallelized DFT

- Basic idea
  - Offload tracking overhead to the separate core
- Challenge
  - Communication cost is too high
  - Synchronization
- Approach
  - Minimize the overhead to the primary
  - Maximize the utilization of the secondary



ShadowReplica: Execution Model

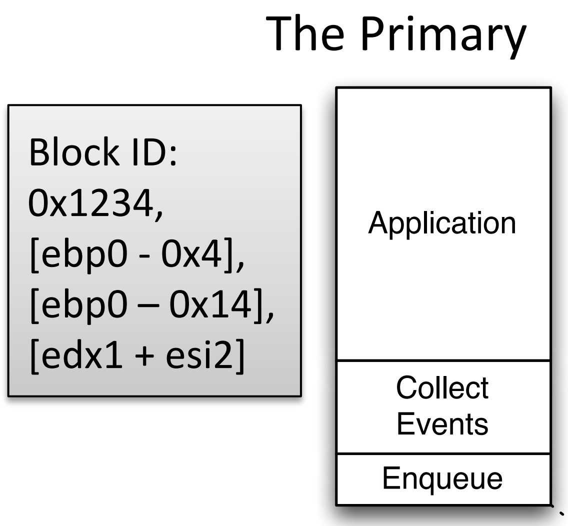
- The block 0x1234
  - Executed from the primary
  - Monitored from the secondary

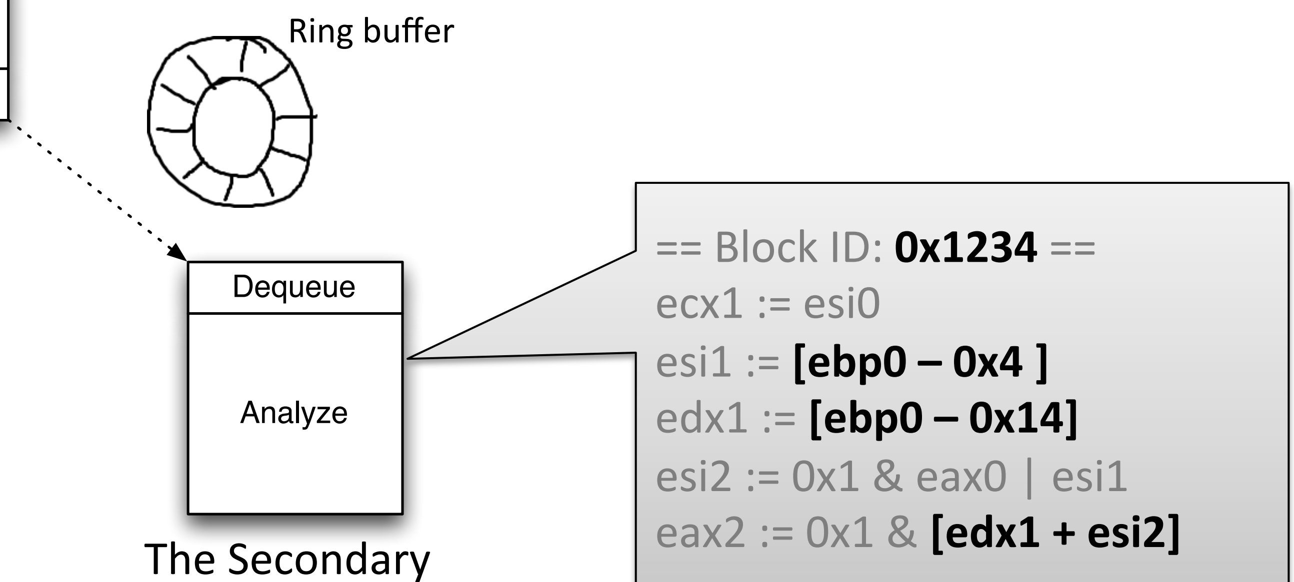



```
== Block ID: 0x1234==
ecx1 := esi0
esi1 := [ebp0 – 0x4 ]
edx1 := [ebp0 – 0x14]
esi2 := 0x1 & eax0 | esi1
eax2 := 0x1 & [edx1 + esi2]
```

ShadowReplica: Runtime Information

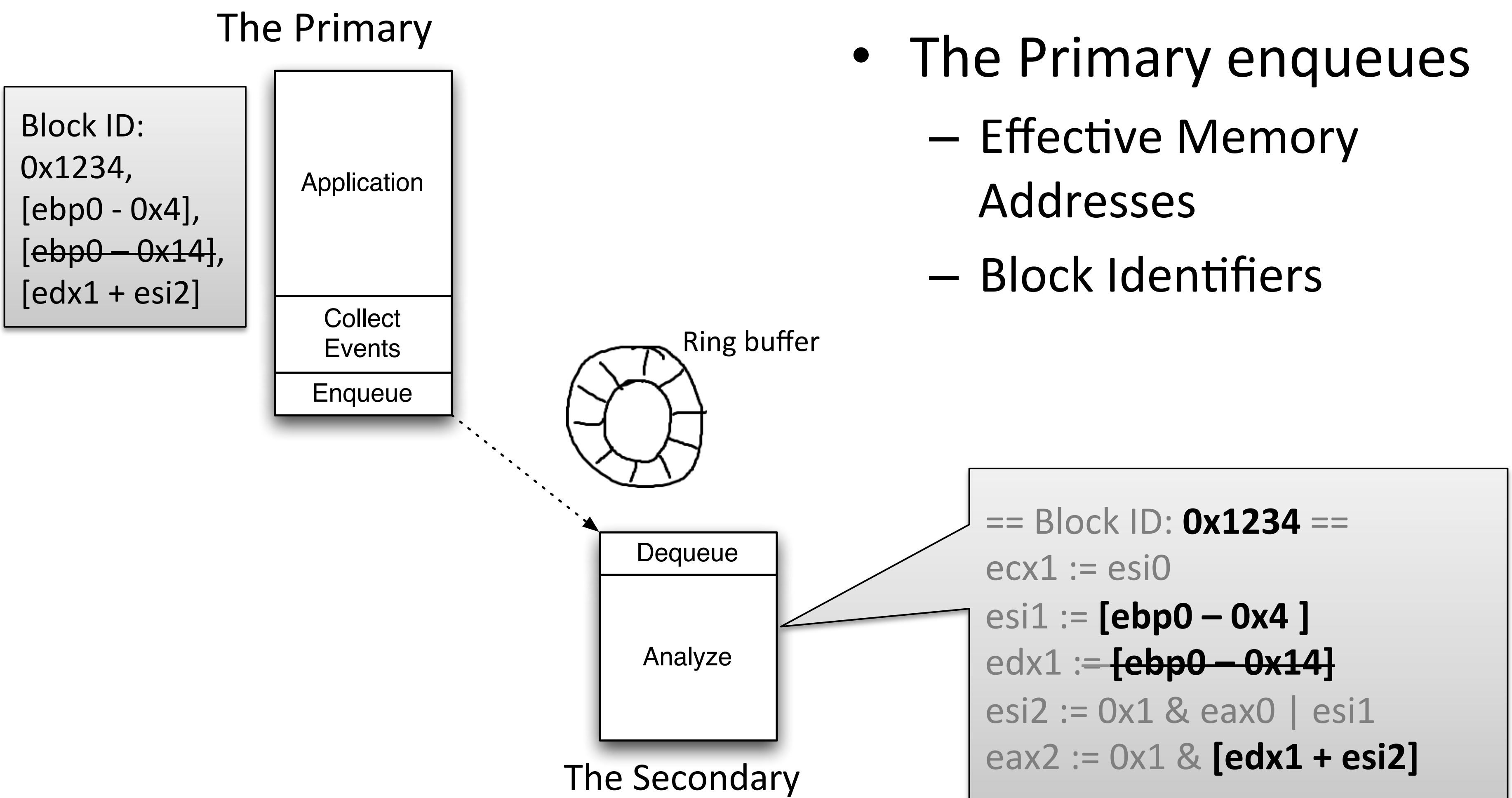
- The Primary enqueues
  - Effective Memory Addresses
  - Block Identifiers





ShadowReplica: Runtime Information

- The Primary enqueues
  - Effective Memory Addresses
  - Block Identifiers



Optimizing Effective Addresses

- Intra Block Optimization
  - Linear Lower Bound
    - 3 Register Variables: eax0, eax1, ebx0
    - 4 Memory Operands: One redundancy

Block ID: 0x1234

[eax0 + ebx0]
[eax1 + 2 × ebx0]
[ebx0 + 4 × eax1]
[eax0 + 2 × ebx0]

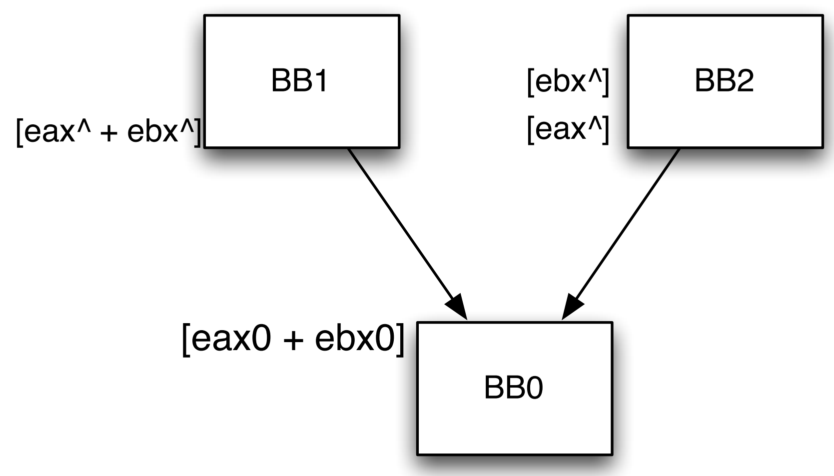
Optimizing the Primary: Effective Addresses

- Intra Block Optimization
  - Linear Lower Bound
    - 3 Register Variables: eax0, eax1, ebx0
    - 4 Memory Operands: One redundancy

Block ID: 0x1234

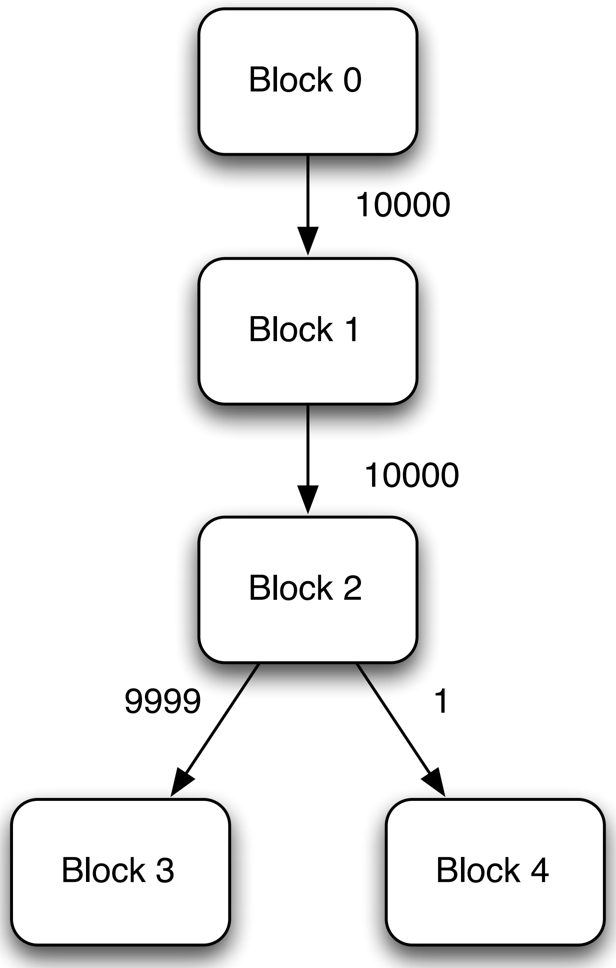
[eax0 + ebx0]
[eax1 + 2 × ebx0]
[ebx0 + 4 × eax1]
[eax0 + 2 × ebx0]

- Inter Block Optimization
  - Data Flow Analysis
  - Use-Def equation based on *Inferablity*



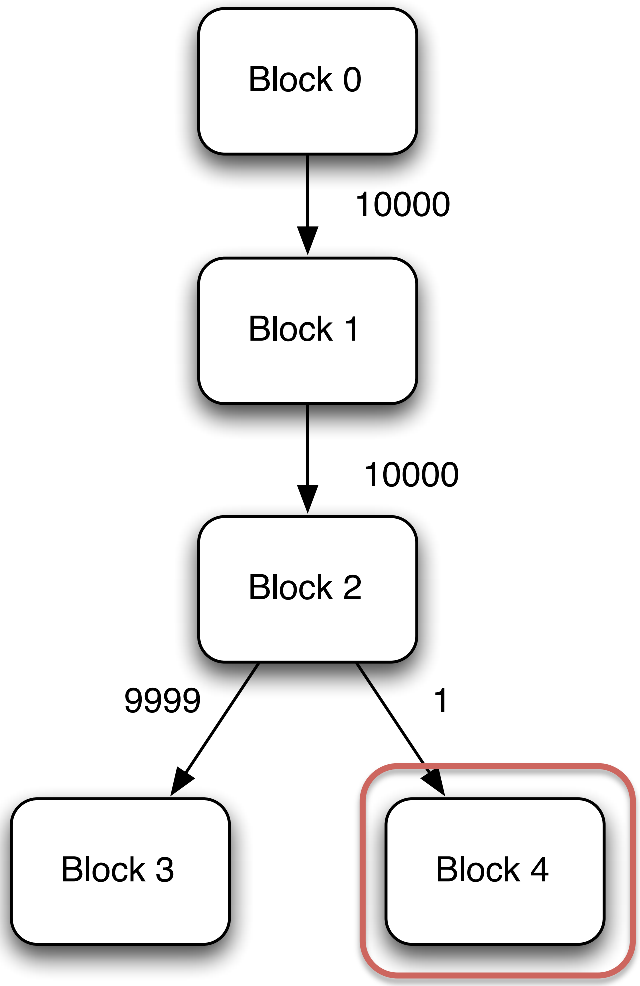
* ‘^’ indicates top-most version for the register variable in the block

Optimizing the Primary : Block Identifiers



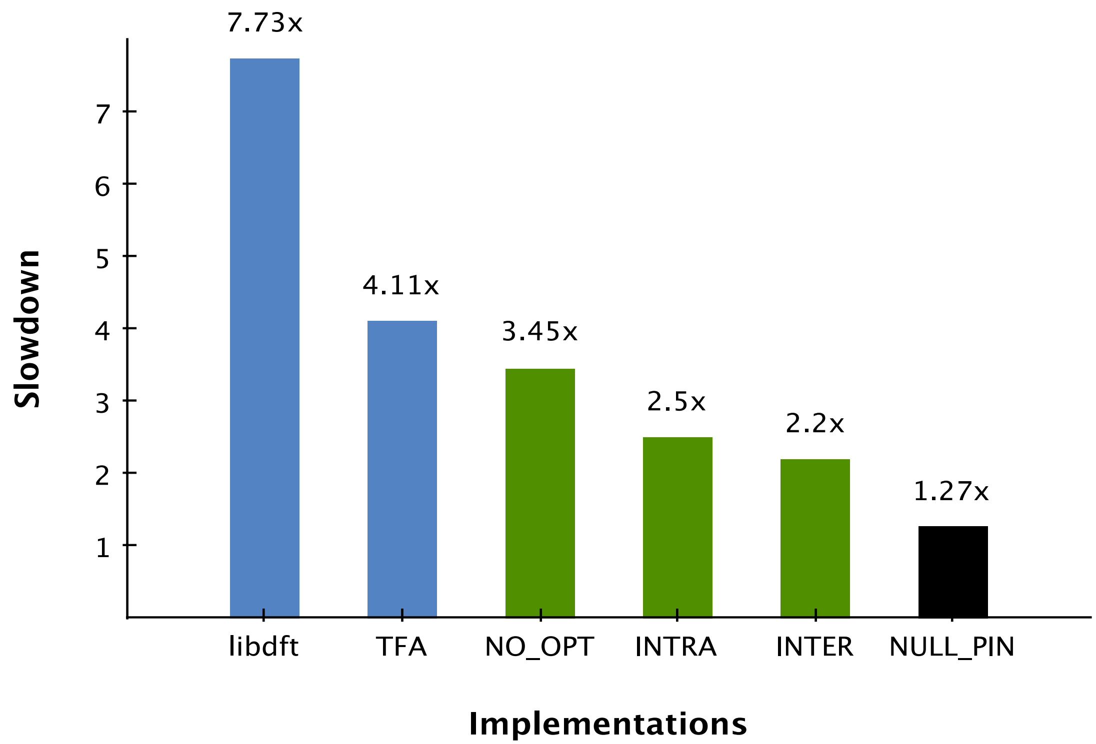
- Minimize Block ID trace leveraging
  - Control Flow Graph (CFG)
  - Execution count

Optimizing Block Identifiers



- Minimize Block ID trace leveraging
  - Control Flow Graph (CFG)
  - Execution count
- Only Block 4 require instrumentation
  - 1 instrumentation & 1 execution
  - Rest can be restored from the secondary

Preliminary Evaluation:
Bzip2 Compression



- The primary performs Bzip2 compression against a Linux kernel
- The cost of dumping EAes and Block trace from the primary
- Instrumentation for the primary is implemented using PIN DBI

- NO_OPT: All EAes and BB traces
- INTRA: Intra Block optimization
- INTER: Intra + Inter Block optimization

Candidate Analyses for ShadowReplica

Analyses	Shadow Memory	Data Dependency	Checking
Data Flow Tracking	✓	✓	✓
MemCheck	✓	✓	✓
LockCheck	✓	✗	✓
Method Counting	✗	✗	✗
Call Graph Profiling	✗	✗	✗
Path Profiling	✗	✗	✗
Cache Simulation	✓	✗	✗

- Each analysis have different performance implications
  - Instrumentation frequencies, size of analysis routines
- Data dependency across multiple updates inhibits the parallelization of analysis
- Checking operations requires synchronization between an application and an analysis

Conclusion

- Current binary-only DFT implementations are sub optimal
  - No consideration for DFT semantics
  - No consideration for global context
- Proposed a novel approaches scheme that
  - Combines static and dynamic analysis
  - Segregates execution and tracking logic
- Huge Speedups for real-world applications
  - TFA: ~2x
  - Shadow Replica: Constant upper bound to the overhead