A Case for Exploiting Memory-Access Persistence

Kim Hazelwood
Mark Toburen
Tom Conte

TINKER Research Group
Department of Electrical & Computer Engineering
North Carolina State University

Memory-Access Persistence: Motivation

- The memory gap is doubling every year
  - Processor speed growth per year: 60%
  - DRAM speed growth per year: 10%
- Larger caches, prefetching are not providing enough relief
  - Larger working sets
  - Access patterns that are difficult to predict
- Dynamic optimization provides resources for exploiting memory-access persistence

The Basic Idea:
Some Things Never Change

Common memory access patterns exist within and across program executions regardless of the input data

- Accessing the same addresses
- Accessing the same structures

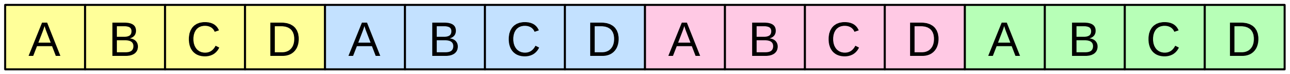
Dynamic optimization could easily track and eliminate the cost of these accesses if they exist to a significant degree!!

Presentation Outline

- Introduction to Persistence
- Insight: Does Memory-Access Persistence Exist?
- Exploiting Memory-Access Persistence
- Conclusions

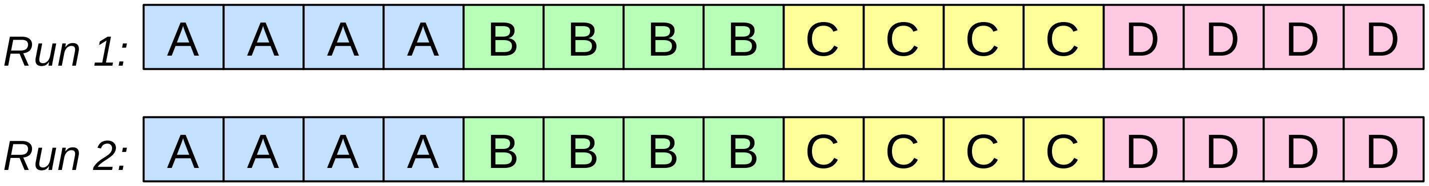
What is Persistence?

- The repetition of a common event or access of a common entity within and / or across runs of a program
- Two logical forms:
  - Intraprogram – persistence occurs within the current run of the program



A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D

  - Interprogram – persistence occurs across multiple runs of the program regardless of the input data set



Run 1:	A	A	A	A	B	B	B	B	C	C	C	C	D	D	D	D
Run 2:	A	A	A	A	B	B	B	B	C	C	C	C	D	D	D	D

Intraprogram Persistence

- Events or sequences of events that repeat throughout a single program run
- Examples:
    - Memory accesses
    - Branch directions
    - Instruction result values
- Exploiting intraprogram persistence is relatively easy using known techniques
    - Prefetching
    - Branch prediction
    - Value prediction

Interprogram Persistence

- Events or sequences of events that repeat across multiple runs of a program regardless of the input data set
- Interprogram memory-access persistence exists in two forms
  - Base – the same addresses are accessed across runs
  - Constant-offset – the same structures are accessed across runs but were allocated to different locations
- To what degree does it exist in either or both forms?
- How can we effectively exploit it?

Interprogram Persistence: Does It Exist?

- Goal 1:
  - Determine the invariance in data cache miss addresses and use as an indicator of base persistence
- Experiment 1:
  - Measure base persistence across program runs by monitoring distinct misses (DM)

$$\text{Base Persistence} = (DM_{in2} - (DM_{total} - DM_{in1})) \; / \; DM_{total}$$

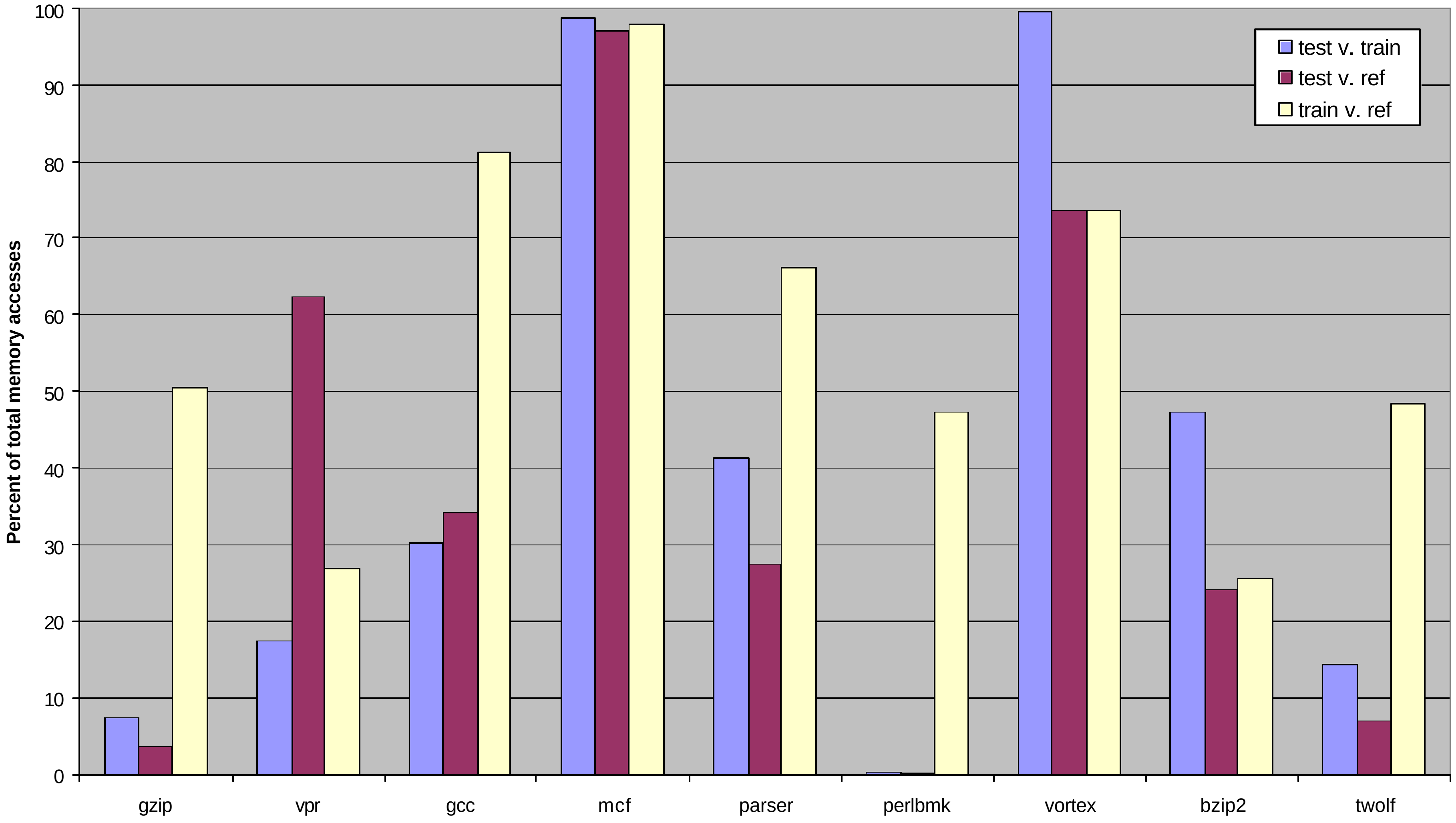
  - L2 D-cache configuration: 256 KB, 4-way set associative
  - Benchmarks: SPECint2k
  - Input sets: SPECint2k test, reference, and training inputs
- Limitations:
  - Does not reflect dynamic frequency of matching addresses
  - Does not account for constant-offset persistence

Experiment 1: Observations

- Based on cross-program DM measurements, benchmarks fall into two categories
  1. $DM_{total} \sim= Max(DM_{in1}, DM_{in2})$
  2. $DM_{total} > Max(DM_{in1}, DM_{in2})$
- All but the Test v. Train case for bzip2 fall into category 1
- So base persistence is determined by the difference in DMs between input sets as well as the total number of DMs across input sets
  - Input sets varying significantly in size will tend to demonstrate less base persistence

NC STATE UNIVERSITY

Experiment 1:
Results



Experiment 1: Observations (cont.)

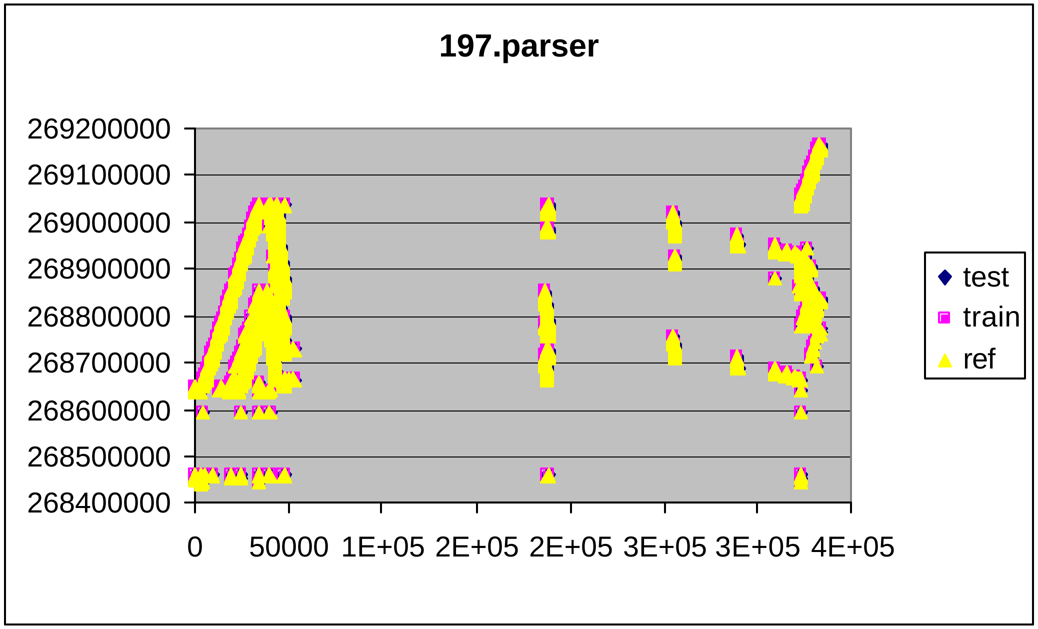
- Benchmarks fall into four categories based on DM differences between input sets
  1. $DM_{test} \sim= DM_{train} \sim= DM_{ref}$: mcf
  2. $DM_{test} << DM_{train} < DM_{ref}$: gzip, gcc, parser, twolf, perlbmk
  3. $DM_{test} < DM_{ref} << DM_{train}$: vpr
  4. $DM_{test} \sim= DM_{train} << DM_{ref}$: bzip2, vortex
- In general benchmarks show low to moderate levels of base persistence among all input combinations. *Why*?
  - Large variations in input set size
    - More persistence between input sets of similar size
  - constant-offset persistence is not accounted for so differences in DMs may not reflect true persistence levels

Things aren't always what they seem!!

Interprogram Persistence: Does It Exist?

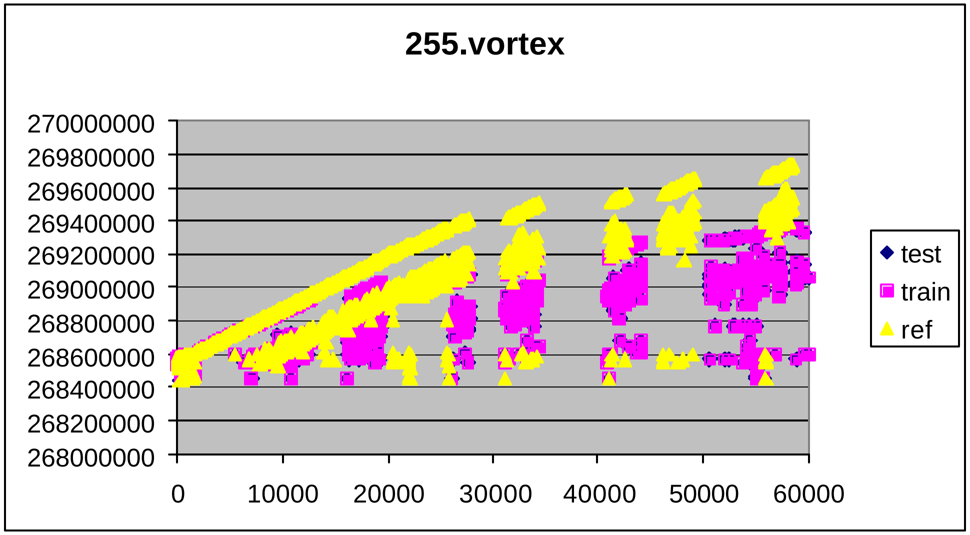
- Goal 2:
  - Observe phases of distinct memory access behavior
  - Establish existence of constant-offset persistence
    - Dynamically-allocated data structures will not always get allocated to the same physical location each time the program is run
- Experiment 2:
  - Plot memory access patterns over time for program runs using varying input sets
  - Examine snapshots of the execution for:
    - Addresses that repeat temporally between input sets
    - Address shifts that occur temporally between input sets

Interprogram Persistence: Experiment 2



Overlap indicates base persistence between all three runs

Interprogram Persistence: Experiment 2



Address shift between test / train and ref illustrates

constant-offset persistence

Experiment 2: Observations

- Significant amounts of base persistence as per the results in Experiment 1
- Clear examples of constant-offset persistence in gzip and vortex which indicates that there is potentially much more persistence than Experiment 1 indicates

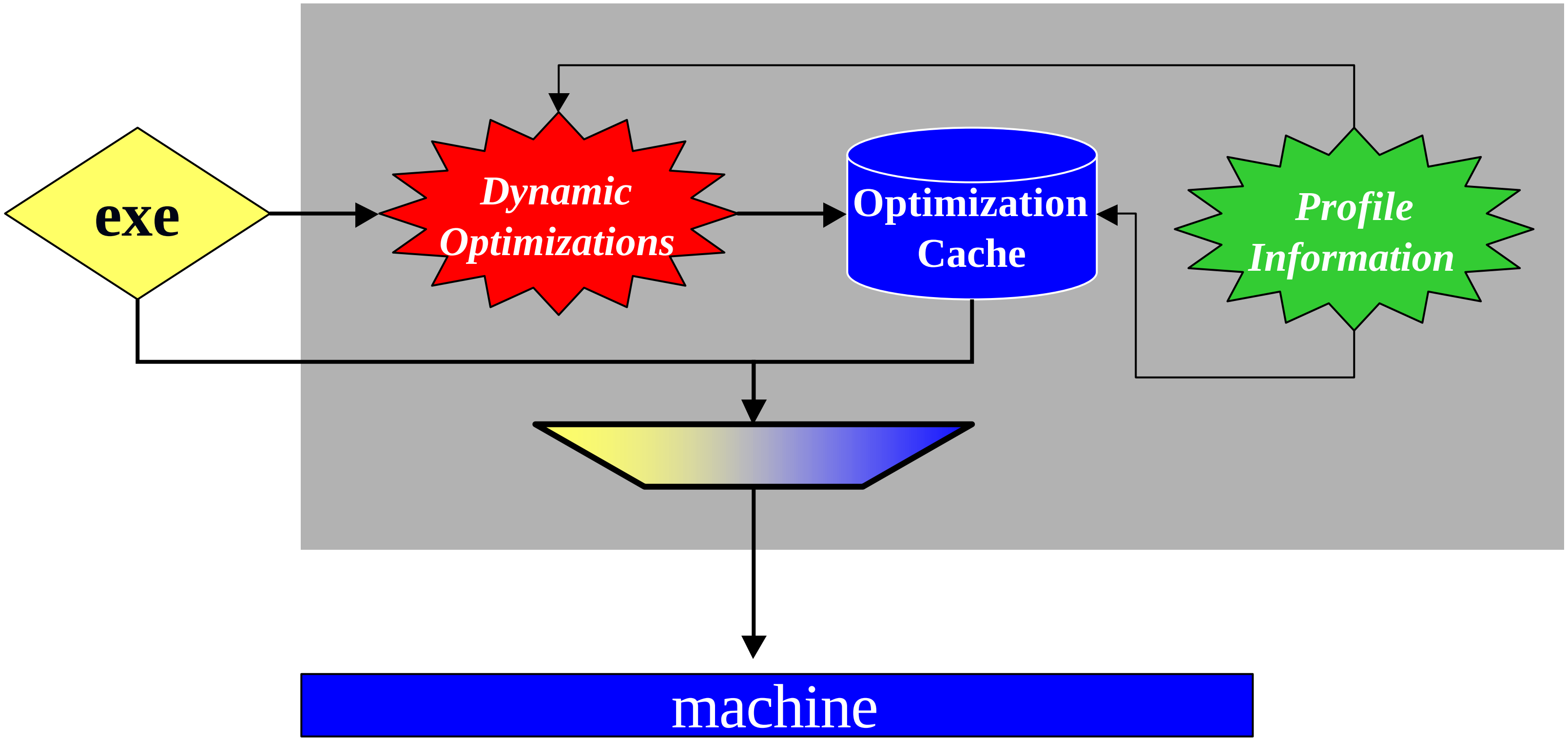
Interprogram Persistence: Further Analysis

- Examine base persistence in a dynamic context to get a clearer picture of its extent
- Determine the contribution of constant-offset persistence through correlation of misses to specific instruction info
- Study the frequency of specific memory-access clusters
- Look at how persistence varies over a wider variety of input sets
- Study how persistence varies with cache size and organization

Exploiting Persistence

- Exploiting memory-access persistence and persistence in general requires two primary capabilities:
  - A mechanism for constantly collecting non-statistical profile information
  - A mechanism for altering the current program in order to take advantage of persistence later on in its execution
- Dynamic optimization systems provide both!!

High-Level View of a Dynamic Optimizer



Dynamic Optimization for the Memory Wall

Application

Dynamic Optimizer

Hardware

Current dynamic optimizers are:

- Transparent to the application and user
- Able to intercept profile information regarding the executing application
- Able to store information within and between program executions

Each of these features make dynamic optimization a great candidate for exploiting memory-access persistence

Exploiting Persistence: Open Issues

- Need real mechanisms for collecting memory-access profiles
  - General-purpose programmable hardware profiling
- Need dynamic-optimization algorithms for analyzing and optimizing programs to exploit memory-access persistence

Conclusions

- Current latency tolerance mechanisms for data cache misses are not providing enough relief for the memory wall
- Memory-access persistence occurs in varying forms and to varying degrees within and across program runs regardless of the input data set
- Dynamic optimizers provide the type of framework necessary to exploit this persistence
- Need further research in the areas of detecting memory-access persistence, algorithms for effectively leveraging this persistence, and how to find and exploit other forms of persistence

Contact Information

Kim Hazelwood	kim_hazelwood@ncsu.edu
Mark Toburen	mark_toburen@ncsu.edu
Tom Conte	conte@ncsu.edu

TINKER Research Group
North Carolina State University
www.tinker.ncsu.edu