Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots

Kenneth C. Barr

Thesis Defense
August 25, 2006
My thesis, a bird’s eye view

Computer architects rely heavily on software simulators to evaluate, refine, and validate new designs.

Simulators are too slow!
My thesis, a bird’s eye view

Computer architects rely on heavily on software simulators to evaluate, refine, and validate new designs.

My thesis research provides…

- Software structures and algorithms to speed up performance simulation
- Approach
  - Amortize time-consuming process of warming detailed models in a multiprocessor simulator
  - Cache coherent memory system: store one set of data to reconstruct many target possibilities
  - Branch predictors: lossless, highly compressed traces

© 2006, Kenneth C. Barr
Detailed performance simulation

Baseline Configuration

Configuration_1

Configuration_N

Target Configuration
(//cache size, pipeline stages, number of cores, etc./)

Benchmark Program

Detailed Simulator

Baseline Results

Results_1

Results_N

Host computer

Performance Results
(cycles-per-inst, cache miss rate, power, etc.)
Why is detailed software simulation slow?

How slow?
- 5.9 trillion instructions in SPECINT 2000
- Actual 3.06 GHz Pentium 4
  \( \approx 31 \) minutes
- “Fast,” uniprocessor, user code only, detailed simulator
  \( \approx 1 \) Minsts/sec:
  \( \approx 68 \) days
- Our 4-CPU simulation with OS and memory system
  \( \approx 280 \) Kinsts/sec:
  \( \approx 244 \) days

- Out-of-order, superscalar pipeline
- Cache coherent memory system
- Resource contention (buses, ports)
- Statistics gathering, power modeling
- Multiple runs to observe variation
Intelligent sampling gives best speed-accuracy tradeoff for uniprocessors (Yi, HPCA `05)

- Single sample:
  - Detailed: ignored

- Fast-forward + single sample:
  - ISA only: detailed, ignored

- Fast-forward + Warm-up + sample:
  - ISA only: detailed, ignored

- Selective Sampling (SimPoint)

- Statistical Sampling

- Statistical sampling w/ Fast Functional Warming (SMARTS, FFW)
  - ISA+µarch

Online sampling:
- too much time required for fast-forwarding and warming
Snapshots amortize fast-forwarding, but require slow warming or bind to a particular $\mu$arch

- ISA snapshots (registers & memory)
  - Slow due to warm-up, but allows any $\mu$arch

- ISA+$\mu$arch → “concrete” snapshots
  - Fast (less warm-up), but tied to $\mu$arch
  - …or huge

- $\mu$arch-independent snapshots (MINSnaps)
  - Fast, NOT tied to $\mu$arch
Agenda

Introduction and Background

Memory Timestamp Record (MTR)
- Multiprocessor cache/directory MINSnap
- Evaluation: versatility, size, speed

Branch Predictor-based Compression (BPC)
- Lossless, specialized branch trace compression as MINSnap
- Evaluation: versatility, size, speed

Conclusion
The MTR initializes coherent caches and directory

Modern memory system
  – Multi-megabyte caches
  – Cache coherence

Warming with trace is prohibitive
  – Lots of storage
  – More time: must simulate each memory access

MTR reconstructs state of many targets from concise summary of trace
Memory Timestamp Record: related work

Single-pass cache simulators
- Stack based algorithms: [Mattson et al. 1970]
- SMP extensions: [Thompson 1987]
- Arbitrary set mappings, all-associativity: [Hill and Smith 1989]
- Faster algorithms, OPT, direct-mapped with varying line sizes [Sugumar and Abraham 1993]

MTR improvements
- Like Thompson, supports SMP, but we add support for directory and silent drops.
- Smaller size
- No upper bounds
- Parallelizable
- Separates snapshot generation from reconstruction
What is the Memory Timestamp Record (MTR)?

MTR is abstract picture of an multiprocessor’s coherence state

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
What is the Memory Timestamp Record (MTR)?

MTR is abstract picture of an multiprocessor’s coherence state
- Fast snapshot generation
- Concrete caches and directory filled in prior to sampling

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MTR example: generation

MTR contains one entry per memory block; locality keeps it sparse.

MTR:

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
<td>CPU1</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Trace:

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Read e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Read b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Read c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write b</td>
</tr>
</tbody>
</table>
MTR example: generation

New access times overwrite old (self-compressing)

Memory Trace:

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Read e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Read b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Read c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write b</td>
</tr>
<tr>
<td>5</td>
<td>Read c</td>
<td></td>
</tr>
</tbody>
</table>

MTR:

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
<td>CPU1</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2006, Kenneth C. Barr
1. Choose target
2. Coalesce
   (determine contents)
3. Fixup
   (determine state)
MTR example: reconstruction

Choose target
– Two sets, two ways

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2006, Kenneth C. Barr
### MTR example: reconstruction

**Coalesce**
- What are the contents of CPU’s cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>…</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

#### CPU0’s cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MTR example: reconstruction

Coalesce
- What are the contents of CPU’s cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

CPU0’s cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MTR example: reconstruction

Coalesce
- What are the contents of CPU’s cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>... 4</td>
<td>CPU1</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

CPU0’s cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
MTR example: reconstruction

Coalesce
- What are the contents of CPU’s cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>CPU0</th>
<th>...</th>
<th>CPUUn-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Last Readtime</td>
<td>Last Writetime</td>
<td>Last Writer</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

CPU0’s cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
MTR example: reconstruction

Coalesce
– What are the contents of CPU’s cache?
– Determine which blocks map to same set
– Only ways most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

CPU0’s cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>Set 1</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

© 2006, Kenneth C. Barr
Coalesce
- What are the contents of CPU’s cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

CPU0’s cache

<table>
<thead>
<tr>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>b</td>
</tr>
<tr>
<td>Set 1</td>
<td>e</td>
</tr>
</tbody>
</table>

CPU1?

<table>
<thead>
<tr>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>b_{write}</td>
</tr>
</tbody>
</table>
Fixup: determine correct status bits
MTR example: fixup

Reads prior to a write are invalid, valid writes are dirty, etc…

<table>
<thead>
<tr>
<th>Block Address</th>
<th>CPU0</th>
<th>…</th>
<th>CPUn-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>…</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td>CPU1</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Which cache has the most recent copy of ‘b?’

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>b_{writ}</td>
<td>4</td>
</tr>
</tbody>
</table>
## MTR example: directory reconstruction

### MTR:

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Last Readtime</th>
<th>Last Writetime</th>
<th>Last Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Directory:

<table>
<thead>
<tr>
<th>Block Address</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S</td>
<td>CPU0</td>
</tr>
<tr>
<td>b</td>
<td>M</td>
<td>CPU1</td>
</tr>
<tr>
<td>c</td>
<td>S</td>
<td>CPU0</td>
</tr>
<tr>
<td>d</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>S</td>
<td>CPU0</td>
</tr>
</tbody>
</table>

(Silent drop)
Evicts cannot be recorded in the MTR, but many can be inferred: isEvictedBetween()

**MTR:**

<table>
<thead>
<tr>
<th>address</th>
<th>CPU0</th>
<th>CPU1</th>
<th>Writetime</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>n+k</td>
<td>n</td>
<td>CPU0</td>
<td></td>
</tr>
</tbody>
</table>

**CASE A:**

- CPU0 writes b
- CPU0 reads b
- b = dirty

**CASE B:**

- CPU0 writes b
- CPU0 writes b'
- CPU0 evicting b
- CPU0 reads b
- b = clean

Time:
- n
- n+k
The MTR supports many popular organizations and protocols

Snoopy or directory-based
Multilevel caches
  – Inclusive
  – Exclusive
Time-based replacement policy
  – Strict LRU
  – Cache decay
Invalidate, Update
MSI, MESI, MOESI
Evaluation / Results: Detailed, full-system, execution-driven, x86, SMP simulation

Bochs
Multiprocessor, full-system, x86 emulator (4-way Linux 2.4.24)

Memory Timestamp Record

cclite
Detailed memory system
Parallel Benchmarks

NASA Advanced Supercomputing Parallel Benchmarks:
- FFT, sort, diff. eqns., matrix manipulation
- OpenMP (loop iterations in parallel)
- Fortran

2 OS benchmarks
- dbench: (Samba) several clients making file-centric system calls
- Apache: several clients hammer web server (via loopback interface)

Cilk checkers: AI search plies in parallel
- uses spawn/sync primitives (dynamic thread creation/scheduling)
We compare three simulation methods

Full detailed simulation

Functional fast forwarding (FFW)

Memory Timestamp Record (MTR) with online sampling

Hypothesis

- Both FFW and MTR should be accurate and fast
- MTR should be faster than FFW
- To be useful, FFW and MTR must answer questions in the same way as a detailed model, but faster
MTR results: difficult to quantify accuracy

Methodology
– Eight runs per benchmark
– Vary CPU timing to induce different thread interleavings

Bar shows the median of eight runs, with ticks for min and max. Each run is a valid result!
Open problem: can’t have true confidence intervals without independent random samples of entire population of possible interleavings
Replicating “detailed”-mode stats less crucial than accurate answers to design questions

Change from MSI to MESI

– Blocks are loaded “Exclusive” if no other sharers
– Less traffic for read-modify-write
Replicating “detailed”-mode stats less crucial than accurate answers to design questions

With respect to reply message types, the MSI vs. MESI change is dramatic.

- All fast-fwd bars move with the detailed bar.
- Movement beyond range of detailed runs

Discover evicts (isEvictedBetween()) to more closely match detailed run

- Less drastic timing variations helps, too
Size of MTR: 2-8 times smaller than compressed memory trace

bzip2 compression – 128 Kinsts/sample

(Note: plot shows reduction. Higher is better.)

© 2006, Kenneth C. Barr
Online sampling: MTR faster than FFW

- MTR spends less time in fast-forward (up to 1.45x faster)
- Less work in common case
- Result can be used to initialize multiple targets
Snapshot-driven simulation: Reconstruction speed scales with touched lines

Reconstruction speed:
- MTR has costlier transition than FFW, but
- Reconstruction scales with *touched lines*, not total accesses

© 2006, Kenneth C. Barr
Agenda

Introduction and Background
Memory Timestamp Record (MTR)
  – Multiprocessor cache/directory MINSnap
  – Evaluation: versatility, size, speed

Branch Predictor-based Compression (BPC)
  – Lossless, specialized branch trace compression as MINSnap
  – Evaluation: versatility, size, speed

Conclusion
Why can’t we create \( \mu \)-arch-independent snapshot of a branch predictor?

In cache, an address maps to a particular cache set.

In branch predictor, an address maps to many locations. We combine address with history to reduce aliasing and capture context.

- Same branch address……………..
- In a different context……………..

In a cache, we can throw away LRU accesses

In a branch predictor, who knows if ancient branch affects future predictions?!
If a $\mu$arch independent snapshot is tricky, can we store several branch predictor tables?

Suggested by
- TurboSMARTS / Livepoints
  SIGMETRICS ’05 / ISPASS ’06
- SimPoint Group: HiPEAC ‘05

Not always an option
- If you generate snapshots via hardware dumps, you can’t explore other microarchitectures

Requires predicting the future
- If it takes two weeks to run a non-detailed simulation of a real workload you don’t want to guess wrong
If a µarch independent snapshot is tricky, can we store several branch predictor tables?

Suggested by
- TurboSMARTS / Livepoints
  SIGMETRICS ’05 / ISPASS ’06
- SimPoint Group: HiPEAC ‘05

Not always an option
- If you generate snapshots via hardware dumps, you can’t explore other microarchitectures

Requires predicting the future
- If it takes two weeks to run a non-detailed simulation of a real workload you don’t want to guess wrong

“Several branch predictor tables” aren’t as small as you think! They multiply like rabbits...
One predictor is small, but we need many. Example: 8KB quickly becomes 1000’s of MB.

\[8 \times 1000 = 8 \text{ MBytes}\]
\[78 \approx 10 \times 78 \text{ MBytes}\]
\[3.7 \approx 48 \times 3.7 \text{ GBytes}\]
\[59 \approx 16 \times 59 \text{ GBytes}\]

\[\begin{align*}
P &: \text{gshare with 15 bits of global history} \\
\text{n:} &: 1 \text{ Billion instructions in trace sampled every million insts} \\
\text{requires 1000 samples} &\times 1000 = 8 \text{ MBytes} \\
\text{m:} &: 10 \text{ other tiny branch predictors} \\
48 \text{ benchmarks in SPEC2000} &\times 48 \approx 3.7 \text{ GBytes} \\
16 \text{ cores in design?} &\times 16 \approx 59 \text{ GBytes} \\
\text{Now, add BTB/indirect predictor, loop predictor…} \\
\text{Scale up for industry: 100 benchmarks, 10s of cores} \\
\end{align*}\]
Don’t store collection of concrete snapshots! Store entire branch trace… with BPC

BPC = Branch Predictor-based Compression
Entire branch trace
  – inherently microarchitecture-independent

Traces!? 
  – Fewer branches than memory operations 
  – Easier to predict branches than memory accesses
  • Easy to compress well (< 0.5 bits/branch)
  • Fast to decompress (simple algorithm)
BPC compresses branch traces well and quickly warms up any concrete predictor.

Simulator decodes branches
BPC Compresses trace
  – Chaining if necessary
General-purpose compressor shrinks output further
  – PPMd
Reverse process to fill concrete predictors, one branch at a time
BPC uses branch predictors to model a branch trace. Emits only unpredictable branches.

Contains the branch predictors from your wildest dreams! Hurrah for software!

- Large global/local tournament predictor
  - 1.44Mbit
  - Alpha 21264 style
- 512-deep RAS
- Large hash tables for static info
  - Three 256K-entry
- Cascaded indirect predictor
  - 32KB leaky filter
  - path-based (4 targets)
  - PAg structure
BPC Compression

Input: branch trace from functional simulator

0x00: bne 0x20 (NT)
0x04: j 0x1c (T)
0x1c: ret (T to 0xc4)

Output:

- If BPC says “I could have told you that!”
  (Common case): no output

- If BPC says “I didn’t expect that branch record!”
  < skip N, branch record >

Update internal predictors with every branch.
BPC Decompression

Input: list of pairs < skip N, branch record >

< 0, 0x00: bne 0x20 (NT) >
< 0, 0x04: j 0x1c (T) >
< 13, 0x3c: call 0x74 >

Output:

if (skip==0)
   emit branch record
   // update predictors

while(skip > 0)
   BPC says “let me guess!”
   emit prediction – guaranteed correct
   // update predictors
   // decrement skip
We produce long chains of good predictions represented by single <skip, branch record>.
With BPC, choice of predictor is implicitly provided, not included in output stream.

Value Predictor-based Compression
(Burtscher et al., 2003-2005)
Championship Branch Prediction
(Stark et al. w/ Micro, 2005)

BPC:
Results: Size. BPC-compressed traces are smaller than a concrete snapshot in all cases.

BPC smaller than other compression techniques in almost all cases.
Results: Scaling. BPC-compressed traces grow slower than concrete snapshots

Growth
- BPC has shallow slope, adapts to phase changes
- concrete scales with \( mnP \)
- Concrete = one Pentium 4 style predictor
  - BPC is 2.7x smaller (avg)
  - But if \( m=10 \) predictors → BPC is 27x smaller!

Both grow with number of benchmarks and cores
Results: Speed. BPC compresses well and decompresses fast

Best region: upper left fast and small
BPC is faster than other decompressors
…and sim-bpred
BPC+PPMd faster than PPMd alone
Conclusion

**Goal:** fast, accurate simulation for multiprocessors

**Approach:** Summarizing Multiprocessor Program Execution with Versatile, \( \mu \)arch-Independent Snapshots

**Thesis Contributions**

- **Memory Timestamp Record (MTR):**
  - Versatile: a microarchitecture-independent representation of coherent caches and directory
  - Fast: easy to create, \( O(touched \ lines) \) reconstruction
  - Small: self-compressing, sparse

- **Branch Predictor-based Compression (BPC):**
  - Versatile: compressed trace, lossless
  - Fast: decompression faster than general purpose algorithms and functional simulation
  - Small: compressed branch traces are smaller than concrete branch predictor snapshots

© 2006, Kenneth C. Barr
Acknowledgements

Krste Asanović
  – Guidance, contributions, perspective, opportunity

Michael Zhang: Bochs/cclite infrastructure
Heidi Pan: Corner cases
Joel Emer: Internship opportunity, BPC idea