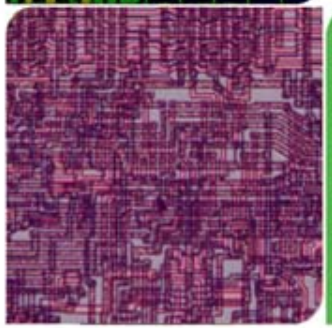
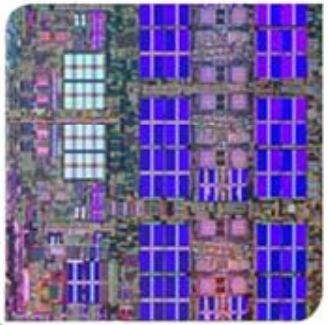



Performance Debugging Support for Many-Threaded Applications

Georgia Tech

comparch



Motivation

- Scalable many-core performance is hard
  - Parallel efficiency drops with more cores
    - Less performance per core
  - The drop in parallel efficiency ***grows*** with # of cores
    - Performance per core drops very quickly!
- For a given app, many potential reasons
  - Some problems are intrinsic to the application's code
    - Code wouldn't scale to N cores regardless of the hardware
  - Most have to do with the interplay between the two
    - This code can't scale to N cores with ***this*** hardware
- Very hard to figure out what to fix



Why is it so hard?

- Typical programmers lack the expertise
  - Need to understand intricate details of HW
    - And the myriad ways these interact with SW execution
  - Many-core everywhere – not just IQ>150 PhDs
    - A typical nuclear physicist => trained to learn and problem-solve
    - Performance problems easier than superstrings or quantum relativity
  - Typical progammers
    - Has very limited understanding of HW
    - Can't fix a problem caused by something they don't know exists
- Real reasons are often counter-intuitive
  - Real reason often considered and dismissed
    - "This code can't have load imbalance" (turns out that it does!)
    - "This code can't have excessive cache misses" (but it does!)



Solution: Better Tools

- Must report problems in an ***actionable*** way
  - "90% of the thread time spent waiting on a barrier"
    - This is what one gets from existing profiling tools
    - First reaction usually "No way! All threads run exactly the same code"
    - Spend 30-60 minutes checking if this is true
    - OK, it's true… look at the code, but it still seems impossible
  - 90% of the thread time spent waiting on the barrier, 60% of that due to different threads having different iteration counts of the for-loop at line 233, another 35% is due to different threads having different cache miss rates for the array access at line 700"
    - This is what we want
    - More believable, programmer can focus on how to avoid/fix it
    - Easier to estimate effort level and decide if it's worth it
      - E.g. 10 different causes, each with a 5% contribution will probably take much longer to fix than one cause with 50% contribution

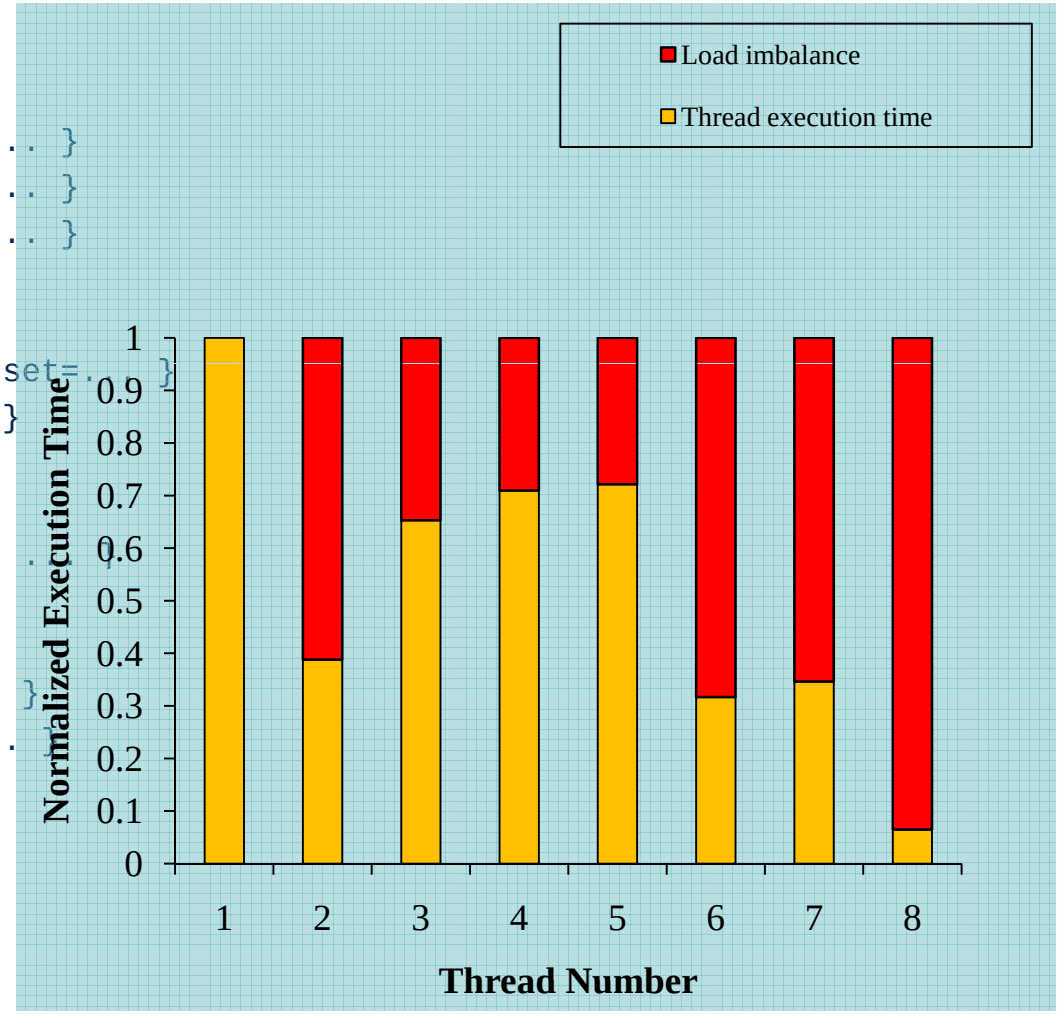


Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset=... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ... }
598 BARRIER(...);
```

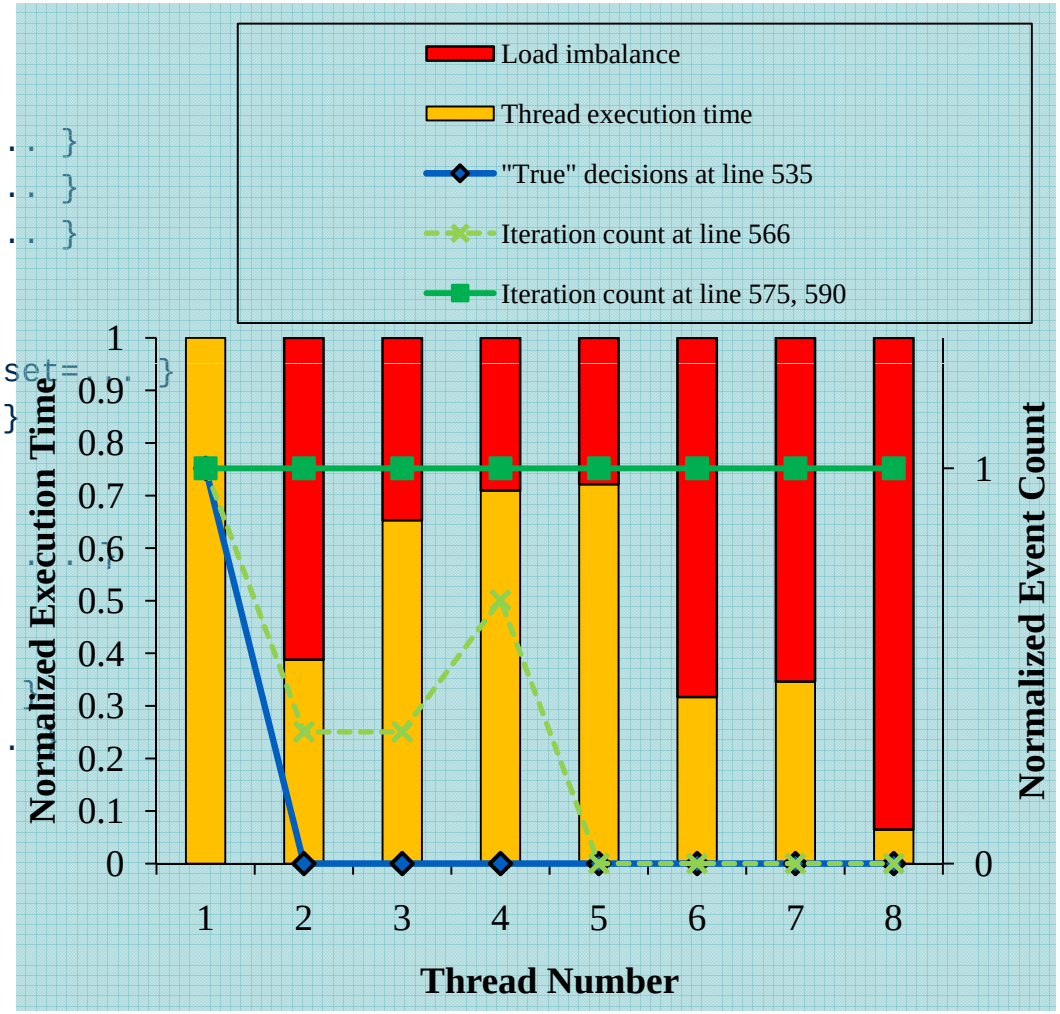
Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset=... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ...
598 BARRIER(...);
```



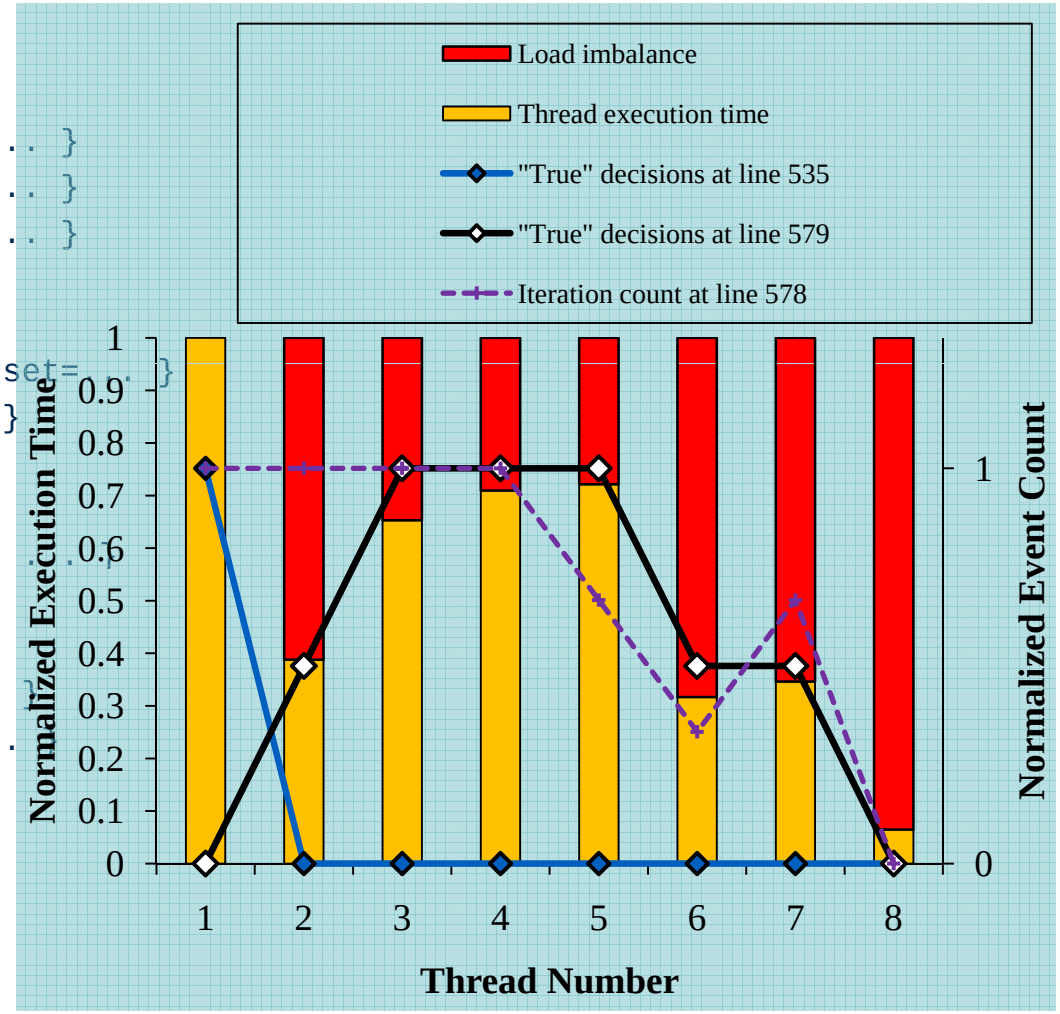
Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset= ... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ...
598 BARRIER(...);
```



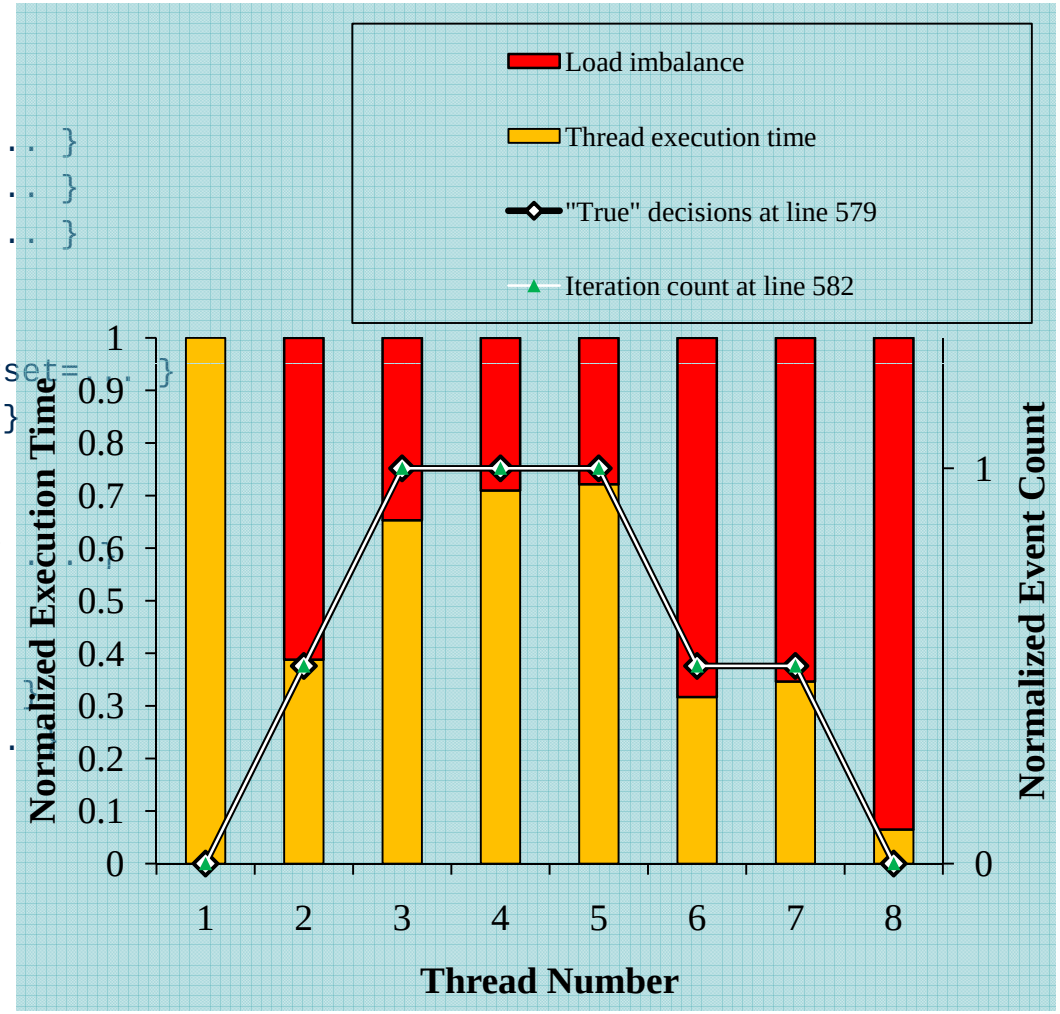
Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset= ... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ... }
598 BARRIER(...);
```



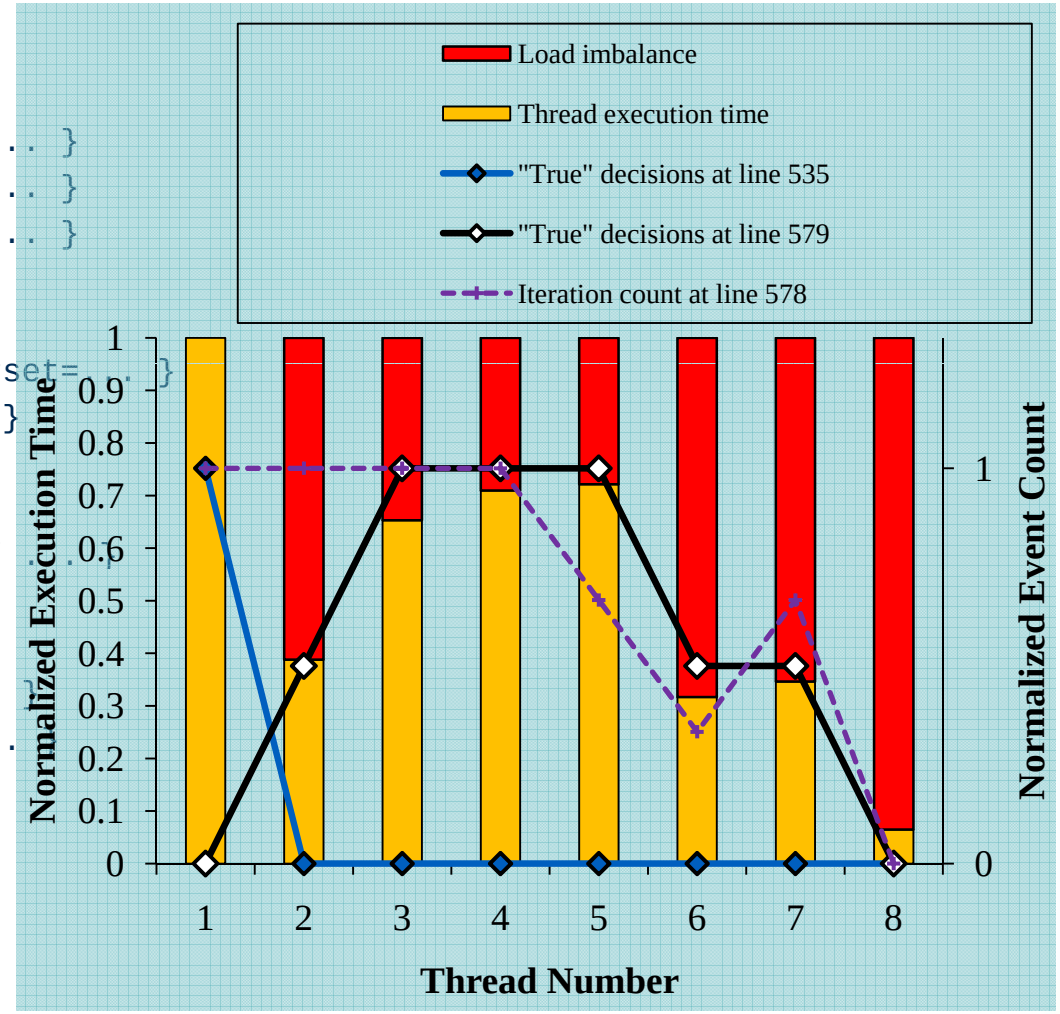
Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset= ... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ...
598 BARRIER(...);
```



Imbalance Example (Radix)

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset= ... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ...
598 BARRIER(...);
```





Assigning Blame

- Exculpate the clearly innocent
  - Event counts that are the same in all threads
  - That event can't be causing imbalance

  } Trivial
- Group the potential suspects
  - Event counts that go together (strongly correlated to each other)

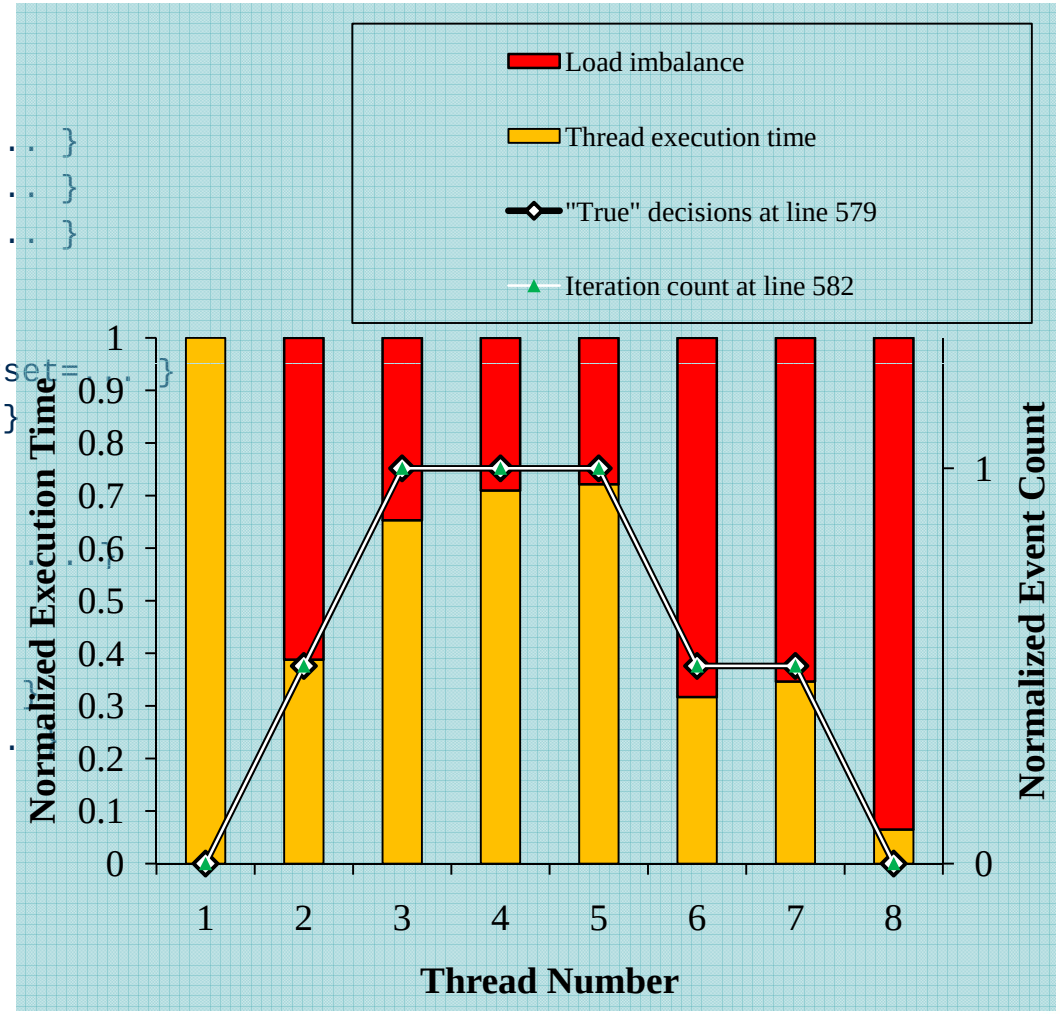
  } Statistical clustering
- Identify group leaders
  - Events that lead to other events in the group

  } Next few slides
- Find the leaders to blame
  - Which leader gets which share of the blame

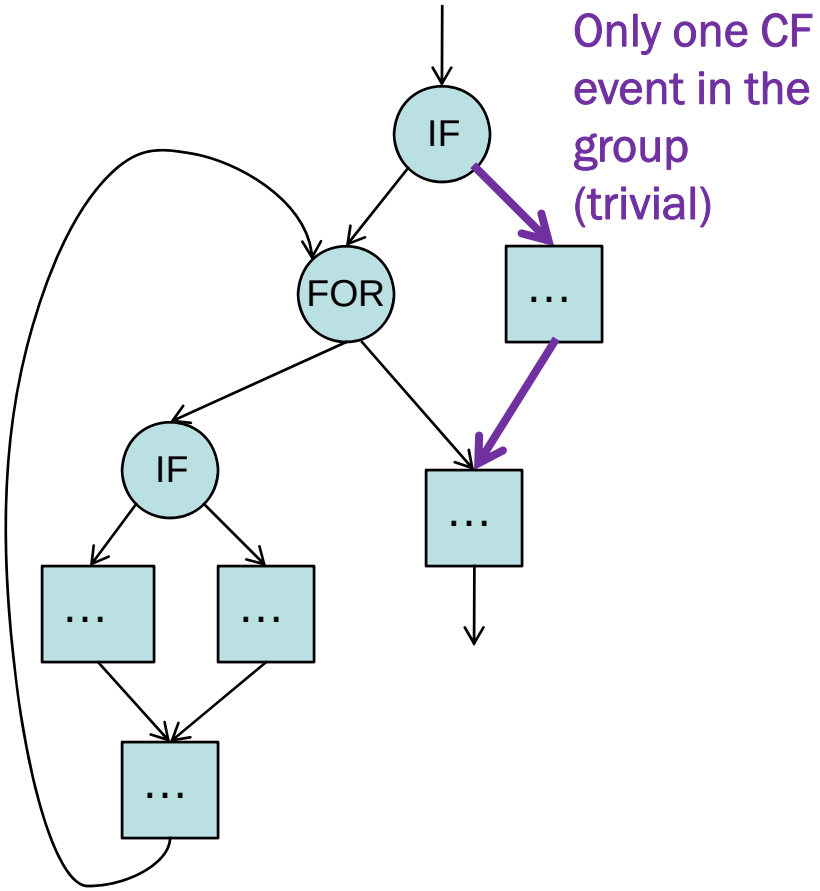
  } Statistical regression
- Report
  - Group leader events
  - Their share of the blame
  - Typical solutions for that type of event

Group Leaders – Control Flow

```
534 BARRIER(...);
535 if (MyNum != (...)) {
540   while ((offset & 0x1) != 0) { ... }
549   while ((offset & 0x1) != 0) { ... }
557   for (i = 0; i < radix; i++) { ... }
560 } else {
562 }
566 while ((offset & 0x1) != 0) { offset= ... }
575 for(i = 0; i < radix; i++) { ... }
578 while (offset != 0) {
579   if ((offset & 0x1) != 0) {
582     for (i = 0; i < radix; i++) { ... }
585   }
589 }
590 for (i = 1; i < radix; i++) { ... }
594 if ((MyNum == 0) || (stats)) { ... }
598 BARRIER(...);
```

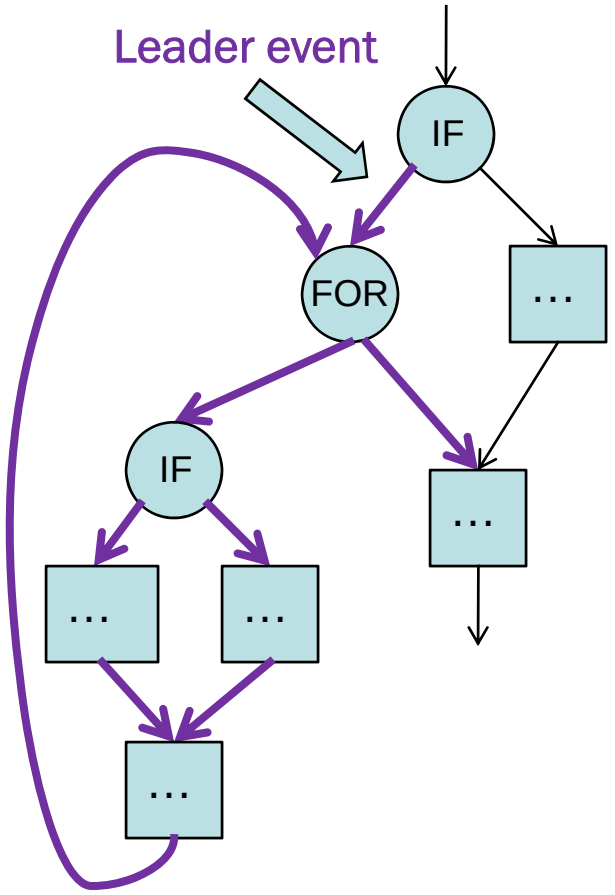


Group Leaders – Control Flow Events

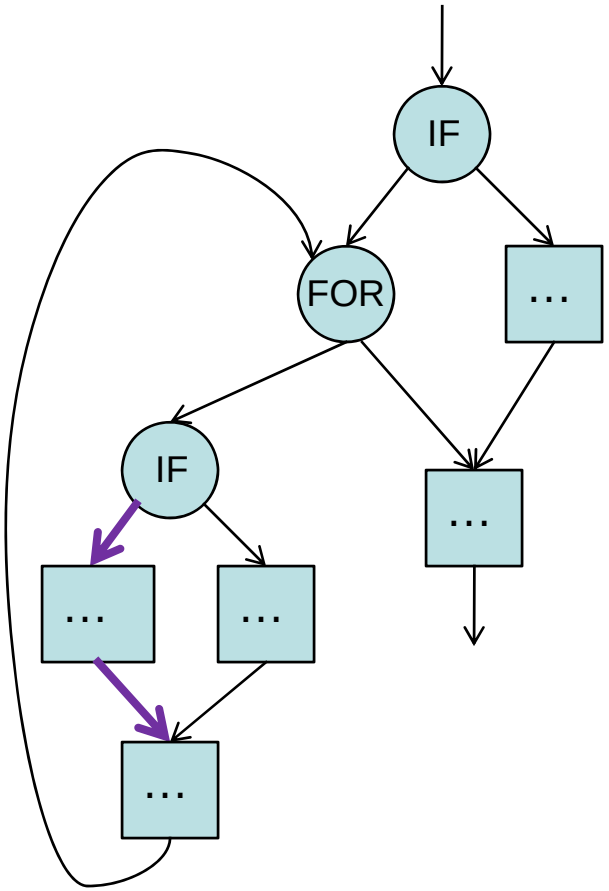


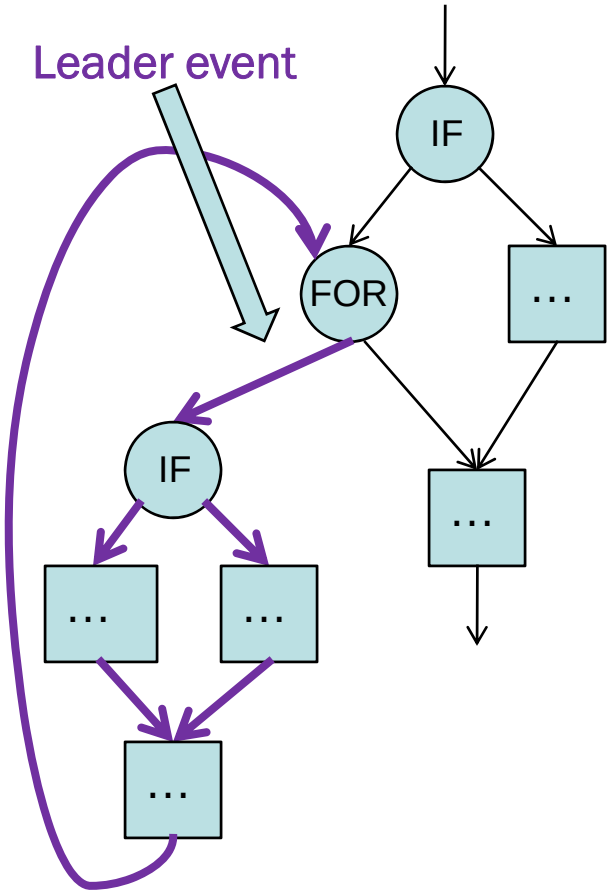
Only one CF event in the group (trivial)

Leader event



Group Leaders – Control Flow Events





Control Flow Leader Events

- Group leader is the decision that causes control flow to enter the group
  - This decision creates the difference among threads
    - Thread 1 tends to have more iterations of loop X than thread 0
    - Thread 1 tends to take “true” path of an “if” more often than thread 0
- Other decision’s execution counts simply follow
  - No *additional* differences among threads created
    - Thread 1 has a larger total # of iterations of loop Y than thread 0, but that’s because loop Y is nested within loop X
    - Thread 1 has a larger total # of iterations of loop Y than thread 0, but that’s because loop Y is on the “true” path of the “if”
    - Thread 1 takes the “false” path of an “if” more times than thread 0, but that’s because the whole “if-then-else” statement is nested inside loop X (or another “if” statement)



Other Causes of Imbalance

- Imbalance can also be caused by
  - Unequal cache miss rates in private (e.g. L1) caches
  - Unequal cache miss rates in shared (e.g. L3) caches
  - Unequal waiting when grabbing the CPU-MEM bus
  - Unequal waiting when obtaining a lock
  - Any other unequal behavior on delay-causing events
- Same “leader” problem for these events
- Example: Th0 has more L3 misses than Th1
  - If same # of L2 misses, L3 is he real reason
  - If # of L3 mises in each thread is proportional to its L2 miss count, L3 is not the real reason (is it L2?)

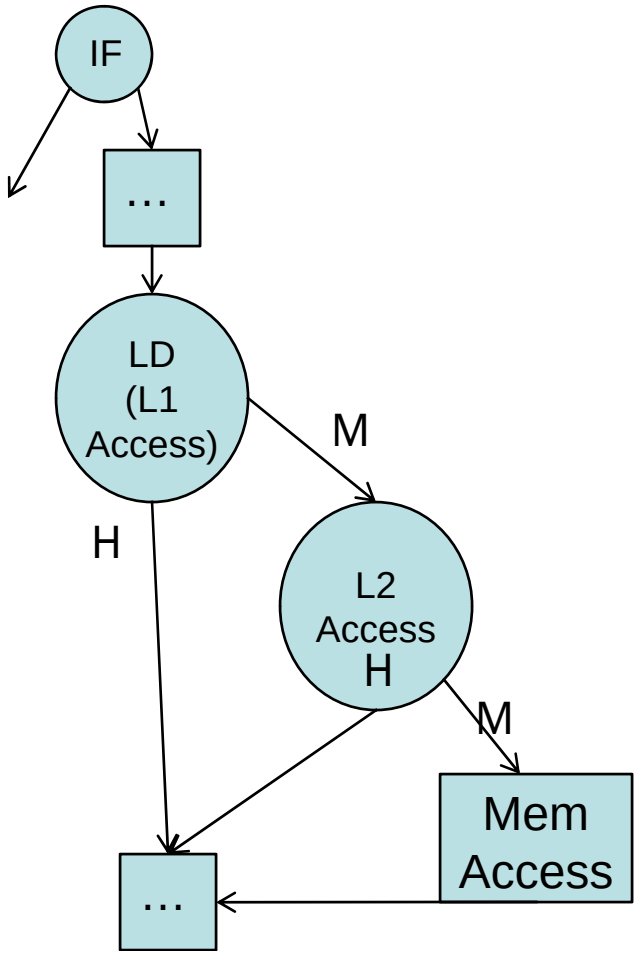


Example

- Example: Th0 has more L3 misses than Th1
  - The real reason can be a control-flow decision
    - Execute a load more times in Th0 than Th1
    - Same miss rate in both threads, but Th0 has more misses than Th1
    - In this case, the control-flow decision, the L1 miss count, the L2 miss count, and the L3 miss count are in the same group
  - The real reason can be different L1 behavior
    - Similar number of executions for the instruction
    - Number of misses in L1 differs, correlated to imbalance
    - Number of L2 and L3 misses proportional to # of L1 misses
    - Now L1, L2, L3 misses on that instruction are in the same group
  - The real reason can be different L2 behavior
  - Or different L3 behavior

Handling on non-CF events

- Hierarchy of events
  - Each such event treated as a “decision”
  - Then apply the same algorithm as before





Implementation

- Lots of details and improvements
  - Need good “distance” metric for clustering
  - Handling of weakly correlated decisions
    - E.g. if-then-else introduces some imbalance,
      a nested loop adds more on top of that,
      a loop within that adds even more, etc.
  - Scoring of reported results
    - Score ~ urgency with which to address this cause of imbalance
    - Related to % of imbalance caused and how much imbalance exists



Results

- Applied this to Splash-2 and PARSEC benchmarks
  - Already highly optimized (no low-hanging fruit)



Results

- How many “causes” end up being reported
  - 0-3 control flow decision points
  - 0-9 load/store instructions
- What are the typical scores
  - On a scale from 0 to 1, from 0.07 to 1.00
- How do we know the right things are reported
  - Used a simulator to “erase” reported misses, then verified that imbalance reduction is as expected
    - Reported instructions are responsible for <10% of all cache misses
    - Imbalance reduction >90% when these misses are eliminated
  - Examined reported control-flow causes of imbalance

Verifying the report for LU

- Highest imbalance of all the apps we used
  - 90% of execution time when using 64 cores
- Three lines of code reported (all control-flow)

Rank	Address	Score	Code point (func.)
1	0x4018b4	0.9455	lu.C:668 (lu)
2	0x4014dc	0.0086	lu.C:595 (lu)
3	0x40187c	0.0033	lu.C:660 (lu)

```
668 if (BlockOwner(I, J) == MyNum) { /* parcel out blocks */
669   B = a[K+J*nblocks];
670   C = a[I+J*nblocks];
671   bmod(A, B, C, strI, strJ, strK, strI, strK, strI);
672 }
…
556 long BlockOwner(long I, long J)
557 {
558   return (J%Ncols)+(I%Nrows)Ncols;
559 }
```

Removed 61% of the imbalance

Verifying the report for volrend

- Second-highest imbalance
  - 46.9% when using 64 cores
- Two lines of code reported

Rank	Address	Score	Code point (func.)
1	0x4068ec	0.9991	render.C:38 (Render)
2	0x447884	0.0002	pthread_mutex_unlock.c:52

```
31 Render(int my_node) /* assumes direction is +Z */
32 {
33   if (my_node == ROOT) {
34     Observer_Transform_Light_Vector();
35     Compute_Observer_Transformed_Highlight_Vector();
36   }
37   Ray_Trace(my_node);
38 }
…
298 Render(my_node);
300 if (my_node == ROOT) {
307   WriteGrayscaleTIFF(outfile, image_len[X], ... );
310   WriteGrayscaleTIFF(filename, image_len[X], ... );
312 }
```

This is reported because of inlining and compiler's instruction scheduling (-O3)

Line 300 reported when using -O0



What's next?

- Release a Pin-based tool
  - Can identify control-flow causes of imbalance
  - Other events rely on HW simulation
    (need event counts attributed to specific code points)
- Improve accuracy of HW-event reporting
  - Different miss rates in different threads
  - We want to report the mechanism behind it
    - Some threads have a larger working set?
    - Different data layout in different threads?
- Apply a similar approach to other perf. problems
  - Lock overhead, contention, and convoying
  - Destructive sharing and other resource-sharing



Acknowledgments

- Student: Jungju Oh
  - Second year student, will look for internships ☺
- Collaborators
  - Chris Hughes, Intel
  - Guru Venkataramani, GWU (former student)
- Support
  - NSF (1/2 student/year) and SRC (1/2 student/year)
  - More support would be welcome ☺
    - Progress would be a lot faster with 2-3 students



Other work

- Support for tainting at intranet level
  - Automated, uncircumventable “tags” go with data
  - Provenance tracking, disclosure prevention, etc.
  - NSF-funded, joint work with Alex Orso, Nick Feamster
- Support for efficient multi-grain checkpointing
  - Checkpoint often for quick rollback, but also allow rollback to long-ago state
    - Can be used for recovery when error detection latencies vary
    - Can be used for reverse-execution debugging
  - Partly NSF-funded, the student is Ioannis Doudalis
- Electric and electromagnetic side channel
  - Signals on CPU pins carry more info than specs say
  - Mobo wires == transmission antennas for some of this
  - Very little understanding of underlying mechanisms
    - How does internal signal X end up phase-modulating external signal Y?
  - Seed funding from NSF (for 1 year)
    - Joint work with Alenka Zajic, our resident signal processing and electromagnetics expert
    - Two first-year students (one CS, one ECE) and one 20GHz oscilloscope ☺