Exploiting Multiple On-Chip Contexts in New Ways

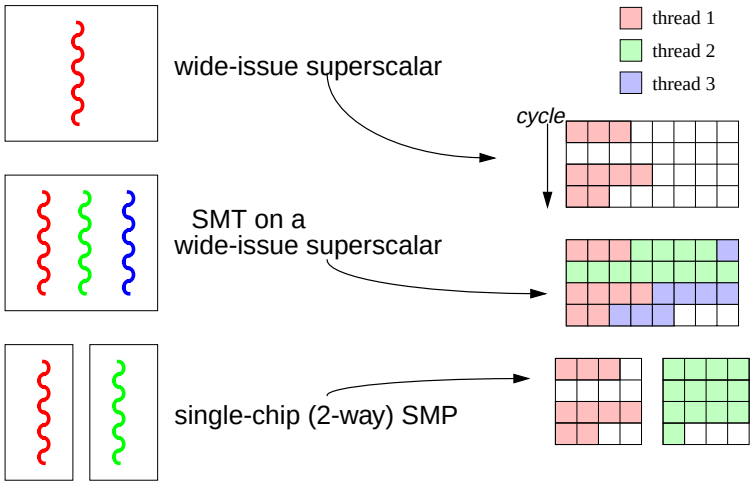
Eric Rotenberg

Dept. of Electrical and Computer Engineering
North Carolina State University
http://www.tinker.ncsu.edu/ericro
ericro@ece.ncsu.edu

High Perf. Microprocessor Trends

- Technology
  - Billions of transistors on a chip
  - Clock rate growth curve may not be dependable
- Implications to microarchitecture
  - Higher performance will rely increasingly on *parallelism*
    - Dictates multiple sources of parallelism
  - Billions of transistors helps, but how to use them?
    - Evolutionary: integrate more independent entities on a single chip
    - Single-chip Multiprocessors (SMP) and Simultaneous Multithreaded Processors (SMT)

SMP/SMT Processors



Exploiting SMP/SMT in New Ways

- SMP/SMT is happening
  - Compaq announced SMT on an 8-way superscalar
  - IBM announced 2 processor cores on a chip
- Opportunity
  - View SMP/SMT architecture as an enabling technology for new processing models
  - Look beyond improving *throughput*
    - Value-add other than performance (e.g. fault tolerance)
    - Speed up single programs in interesting ways

Talk Outline

✓Introduction: towards multiple on-chip contexts

Pursuing three ideas

- AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors
- Exploiting Large Ineffectual Instruction Sequences
- Exploiting Cross-Program Redundancy

• Summary (common underlying themes)

Technology and Fault Tolerance

• High clock rate, dense designs (GHz/billion transistors)

- low voltages for power management
- high-performance and “undisciplined” circuit techniques
- managing clock skew with GHz clocks
- pushing the technology envelope potentially reduces design tolerances in general

=> Entire chip prone to frequent, arbitrary transient faults

Microarchitecture and Fault Tolerance

• Conventional fault-tolerant techniques

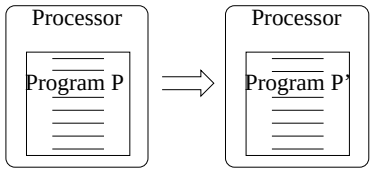
- Specialized techniques (e.g. ECC for memory, RESO for ALUs) do not cover arbitrary logic faults
- Pervasive self-checking logic is intrusive to design
- System-level fault tolerance (e.g. redundant boards/ computers) too costly for commodity computers

• A microarchitecture-based fault-tolerant approach

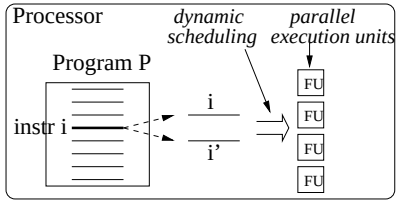
- Microarchitecture performance trends can be easily leveraged for fault-tolerance goals
- Broad coverage of transient faults
- Low overhead: performance, area, and design changes

Time Redundancy Spectrum

Program-level time redundancy

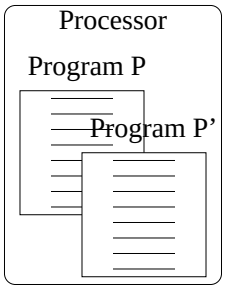


Instruction re-execution

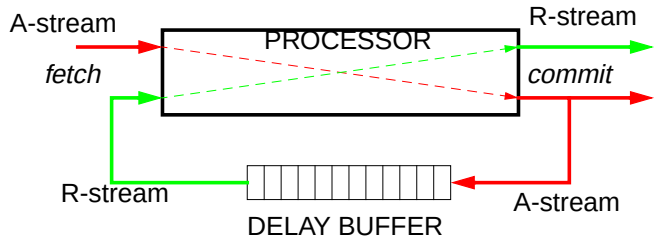


Time Redundancy Spectrum

AR-SMT
time redundancy



AR-SMT High Level



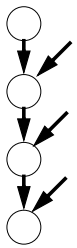
- "A" => "**A**ctive stream"
- "R" => "**R**edundant stream"
- "SMT" => "**S**imultaneous **M**ulti**T**hreading"

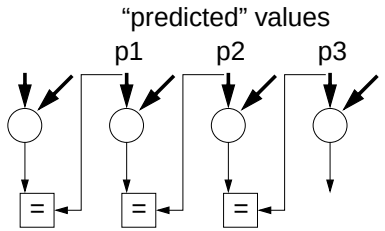
AR-SMT: Fault Tolerance

- Delay Buffer
  - Simple, fast, hardware-only state passing for comparing thread state
  - Ensures time redundancy: the A- and R-stream copies of an instruction execute at different times
  - Buffer length adjusted to cover transient fault *lifetimes*
- Transient fault detection and recovery
  - Fault detected when thread state does not match
  - Error latency related to length of Delay Buffer
  - Committed R-stream state is checkpoint for recovery

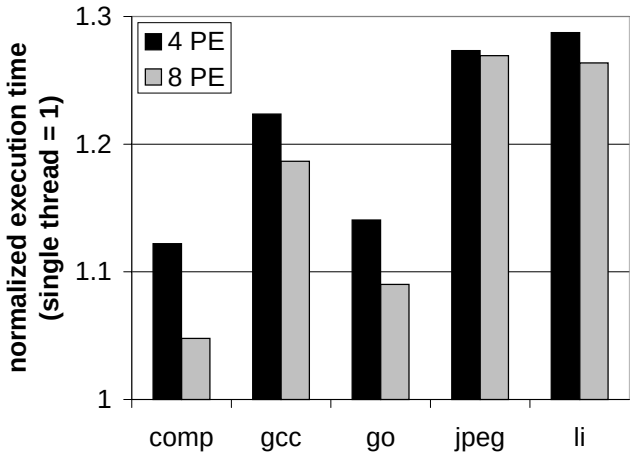
AR-SMT: Low Overhead

- Low hardware and design overhead
  - Leverages underlying microarchitecture (SMT)
- Low performance overhead
  1. SMT-ness: all the same benefits of general SMT (utilization)
  2. R-stream has perfect control and data "predictions" from the A-stream (via delay buffer)!

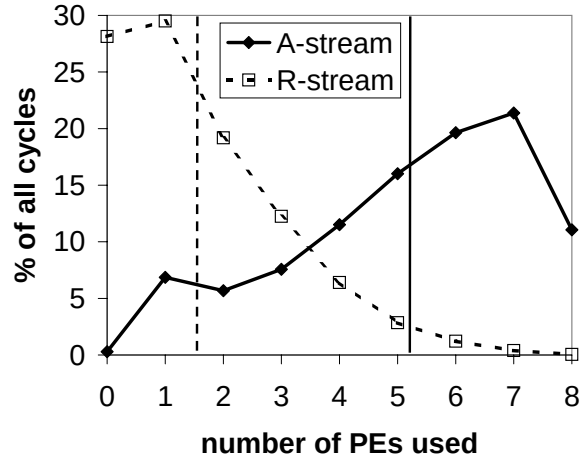




AR-SMT: Performance Results



AR-SMT: Performance Results



AR-SMT Summary

- Technology-driven performance improvements
  - New fault environment: frequent, arbitrary transient faults
- Leverage microarchitecture performance trends for broad-coverage, low-overhead fault tolerance
  - SMT-based time redundancy
  - Control and data “prediction”
- Introducing a second, redundant thread increases execution time by only 10% to 30%

Other Interesting Perspectives

- D. Siewiorek. “Niche Successes to Ubiquitous Invisibility: Fault-Tolerant Computing Past, Present, and Future”, *FTCS-25*
  - (Quote) *Fault-tolerant architectures have not kept pace with the rate of change in commercial systems.*
  - Fault tolerance must make unconventional in-roads into commodity processors: leverage the commodity microarchitecture.
- P. Rubinfeld. “Managing Problems at High Speeds”, Virtual Roundtable on the Challenges and Trends in Processor Design, *Computer*, Jan. 1998.
  - Implications of very high clock rate, dense designs

Talk Outline

✓Introduction: towards multiple on-chip contexts

✓Pursuing three ideas

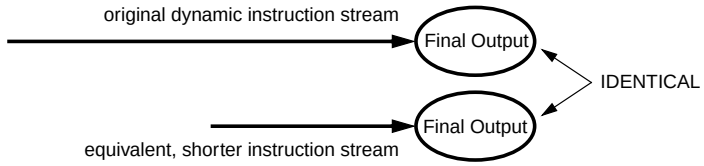
- ✓AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors
- Exploiting Large Ineffectual Instruction Sequences
- Exploiting Cross-Program Redundancy

• Summary (common underlying themes)

Equivalent Dynamic Instr. Streams

• The program is merely a specification
- Only requirement is to obtain correct, final output
- The processor executes the full dynamic instruction stream to meet this requirement

• Hypothetical question
- Are there shorter, equivalent instruction streams that produce the same exact output?



Equivalent Dynamic Instr. Streams

• Thought experiment
- Run the full program and lay out the dynamic instruction stream
- By trial and error, remove dynamic instructions and discard them if the final program output does not change

• Result: as little as 20% of the original dynamic program can produce the same result

Ineffectual Instructions

• Criteria for identifying ineffectual instructions

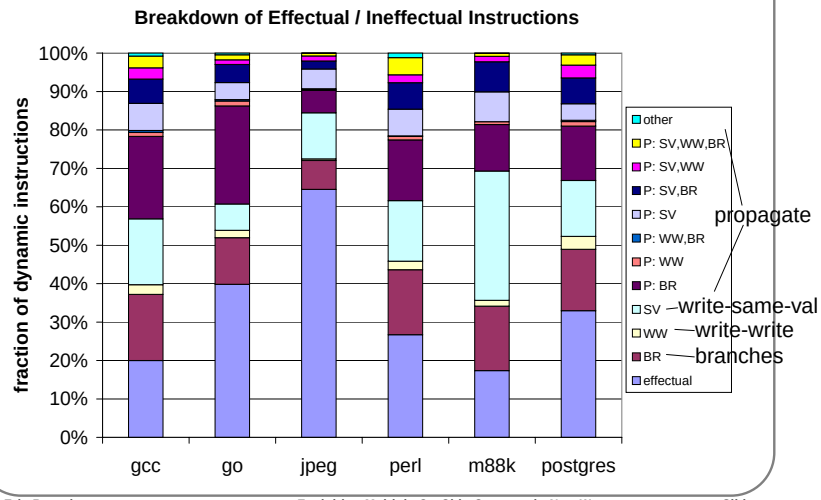
1. Write to a location followed by another write, and no intervening read; *first* write is ineffectual
2. Write the same value to a location as already exists at that location; *this* write is ineffectual
3. Any *correctly predicted* branch is considered ineffectual because the effect is predicted in advance

• Propagate ineffectual status backwards
- Computation leading solely to ineffectual instructions can be marked as ineffectual as well

Experiment

- Identify ineffectual instructions *ideally* but *not optimally*
  - Based on our three criteria and back-propagation
  - Requires maintaining a dataflow graph and register/ memory tracking state
  - Dataflow graph limited to 64K nodes
  - A 2^{16}-entry *gshare* branch predictor is used to identify correctly predicted branches (3rd criterion)

Results: Ineffectual Instructions



Ineffectual Regions (IRs)

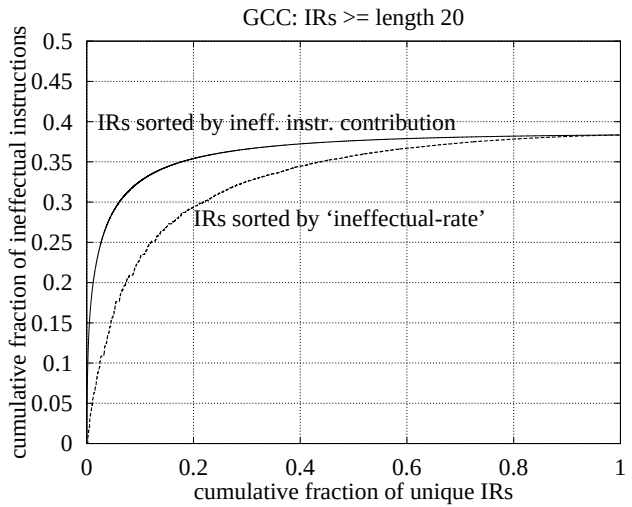
- Ineffectual regions
  - Isolated ineffectual instructions are uninteresting
  - Identify long runs of ineffectual instructions
- Real example from benchmark *go*

```
found = false;
ptr = head;
while (ptr) {
    if (ptr->key == key) {
        ...never entered
    }
    else
      ptr = ptr->next;
}
```

```
s = connect_bamboo(...);
if (s != NO_SQUARE) {
  ... never entered
}
```

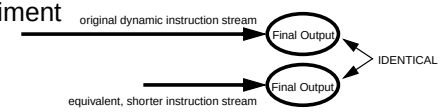
- Sources of ineffectual behavior are not just the compiler, ISA, and ABI => the *algorithm* as well
  - Understanding sources is future work

Results: Ineffectual Regions



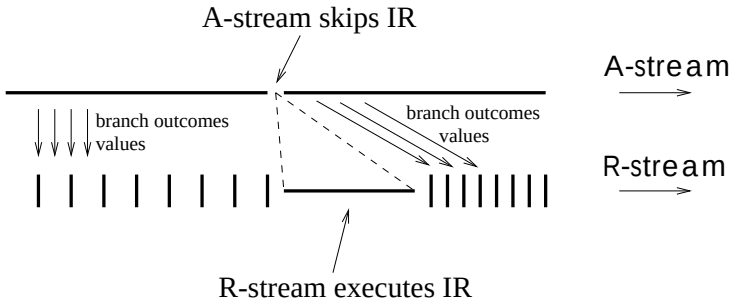
Architectural Implications

- Back to thought experiment



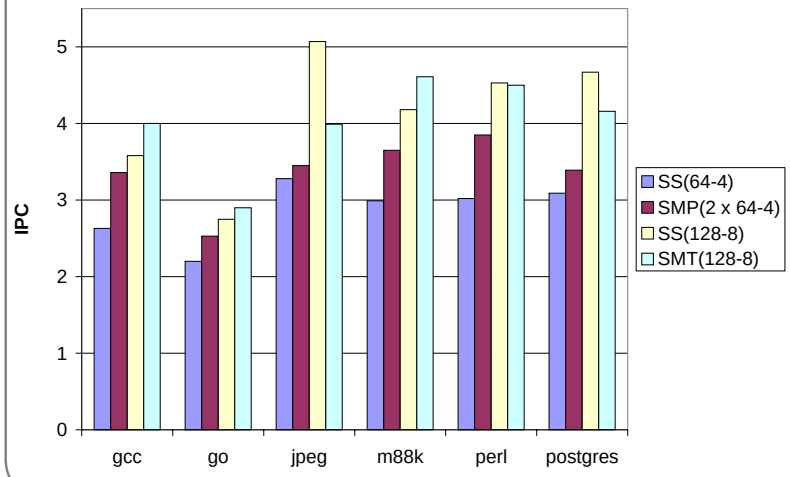
- *IRs provide means to create another version of program*!
  - Detect and learn about IRs via an IR-predictor
  - Dynamically create a short version of the program using IR-predictor
    - Fetching is redirected to point just after a predicted IR
    - If correct, IR can be skipped without corrupting state
  - Run original version of program in a separate context
    - Validate the shorter version, similar to AR-SMT!
    - Verify that 1) implied branch predictions in IR are correct and 2) the region is truly ineffectual

Cooperating Threads Model



- New (non-intrusive) hardware components
  - IR-detection hardware and IR-predictor
  - Queue to communicate outcomes from A-stream to R-stream
  - Write checkpoint buffer to repair A-stream if corrupted

Results (*perfect IR-predictor*)



Summary

- Hypothesized and identified smaller equivalent program
  - Equivalent programs comprising 20% of original
  - Due to unreferenced writes, non-modifying writes, and correctly-predicted branches (and computation chains)
- More useful to identify large ineffectual regions (IRs)
  - IRs provide means to speculatively create second, shorter version of program
  - IRs of length 20 or greater contribute 15-40% of all ineffectual instructions
  - Only 5-10% of all unique IRs account for most of these
- Cooperating threads architecture: potential for 30% speedup leveraging SMP/SMT cores

Talk Outline

✓Introduction: towards multiple on-chip contexts

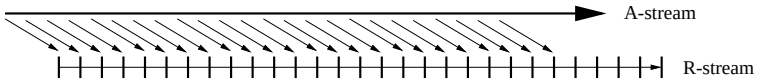
✓Pursuing three ideas

- ✓AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors
- ✓Exploiting Large Ineffectual Instruction Sequences
- Exploiting Cross-Program Redundancy

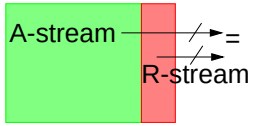
• Summary (common underlying themes)

AR-SMT revisited

• Where AR-SMT performance is coming from

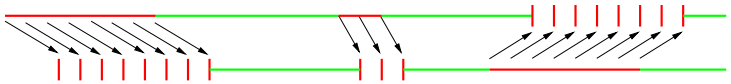


- R-stream ILP is increased because A's past is R's future
- R-stream execution time is not improved and A/R-stream have same average IPC
- R-stream requires less window resources => A-stream is degraded less



Cross-Program Redundancy (XPR)

• Generalize this to *independent* but *partially redundant* programs!



Potential XPR Applications

• Applications
- XPR is likely to be special-purpose
  - Close similarity across different runs of a program
  - Heavy batch environment
  - Ability for O/S to predict XPR and co-schedule
- Examples
  - Databases (Decision Support, Web Engines): multiple similar/related queries
  - Simulations: design-space searches

Research Aspects

1. Ensuring redundant regions are close in time
   - O/S co-schedules to start things off
     - Use high-level program information to predict XPR
   - But need the hardware to continuously fine-tune
     - Detect regions of XPR
     - Delay/advance threads to line up XPR regions

note: schedule other SMT threads in delayed regions...

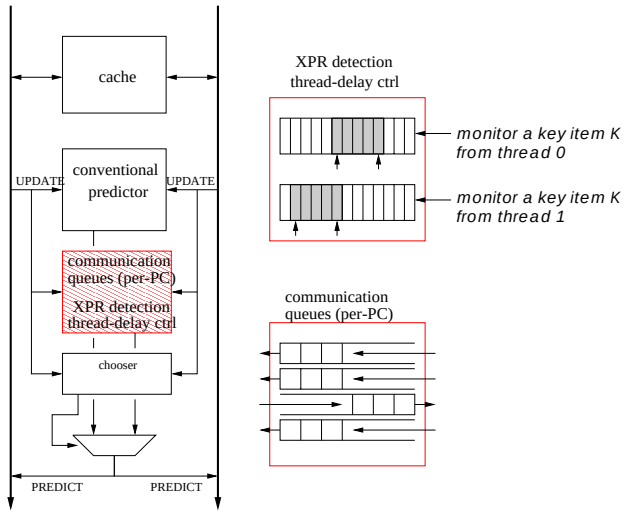
Research Aspects

2. Types of information to communicate
   - Implicit: improve cache hits
     - Beneficial for shared data with poor reuse
   - Explicit: control and data predictions
     - Beneficial for branches/data that are unpredictable using self-program information

Research Aspects

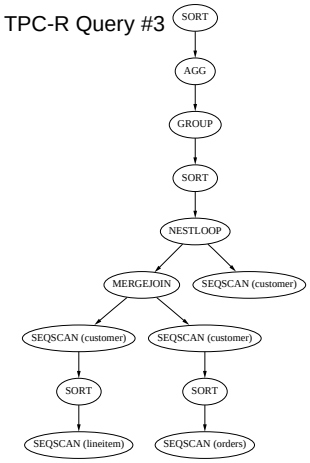
3. Passing information
   - Caches: no explicit communication, just line programs up to ensure cache hits
   - Predictors: two options
     - Handle like caches (rely on positive interference of existing, shared predictor)
     - Pass predictions explicitly via *queues*

High-Level View

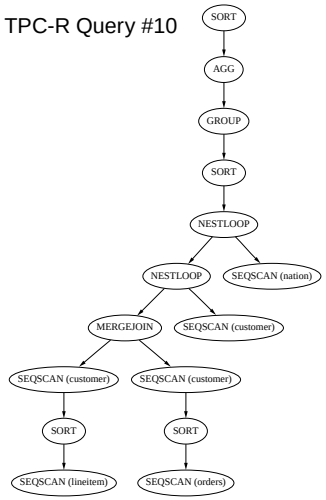


Postgres/TPC-R Example

TPC-R Query #3

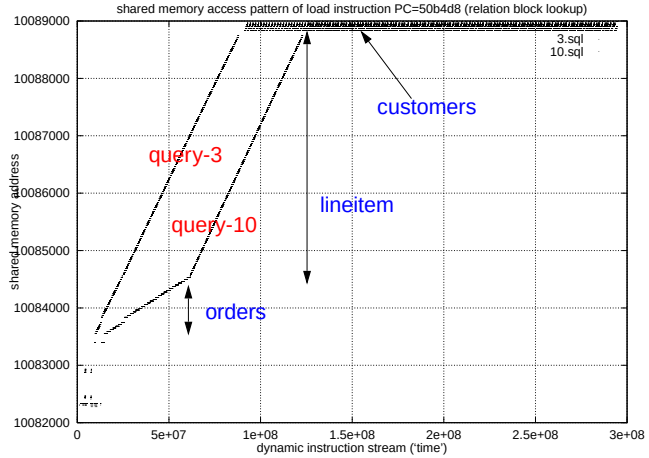


TPC-R Query #10



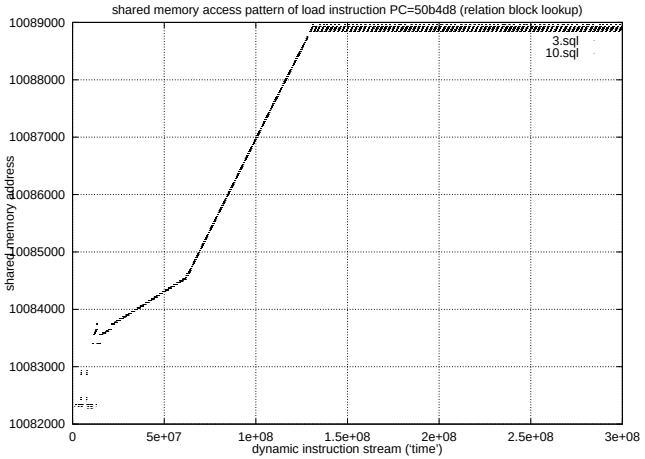
XPR Detection / Thread-Delay Control

- Note the similar access pattern of a *particular* load: the key



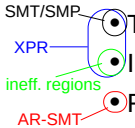
XPR Detection / Thread-Delay Control

- Detect XPR and align the queries by monitoring the key



Summary

- Vision
  - *Flexibility and functionality*: dynamically choose among diverse goals with same underlying architecture



  - Throughput
  - Improve performance of a single program
  - Fault tolerance
  - *Common theme*:
    - Explicit cooperation between redundant or partially-redundant threads