On-line Profiling Techniques

Shivnath Babu
David Bloom
Rohit Gupta

Outline

- Introduction
  - The why, what, and how of profiling
- Papers
  - Relational Profiling from Wisconsin
  - TEST from Stanford
- Discussion

Introduction to Profiling

- Collect performance data of application(s)
  - Execution time (per procedure, per block)
  - #cache misses, memory access pattern
- Uses of profiles
  - Identify bottlenecks
  - Manual tuning (offline)
  - Automated optimizations
    - Offline with profile-driven compilers
    - Online with virtual machines

Breakdown of a Profiling Task

- ***Why:*** intended use of profiling
- ***What:*** identify data to be collected
- ***How:***
  - Data-collection infrastructure
  - Profile-usage infrastructure

Why and What of Profiling

- Edge profiling
  - Use: branch prediction, hyperblocks for high ILP
  - Data: branch taken/not, correlation across branches
- Thread-level speculation
  - Data: inter-thread dependencies
- Prefetching
  - Data: memory access patterns, cache miss addresses
- Low-overhead Garbage Collection
  - Data: Stores to reference variables

Why and What of Profiling (contd.)

- Instruction scheduling
  - Behavior in pipeline for sampled instructions
  - #retired instructions when instruction in flight
  - #wasted issue slots when instruction in flight
- OS policies
  - Superpage promotion to reduce TLB misses
  - Virtual-to-physical mapping to reduce conflict misses
  - Page migration/replication in NUMA
- Power management

How: Data Collection Infrastructure

- Event counters in Alpha, Pentium, MIPS
  - Counts events, delivers interrupt on counter overflow
  - Few: two counters in Pentium II
- Alpha’s ProfileMe
  - Monitor entire pipeline for sampled instructions
  - Interrupt overhead restricts sampling rate
- Research
  - Relational profiling: general-purpose infrastructure (next)
  - TEST: for one specific use (later in talk)
  - Programmable co-processor (Zilles/Sohi, discussion)

Relational Profiling

- Based on co-designed hardware/VM
- Service threads perform dynamic profile collection, dynamic recompilation in parallel with program execution

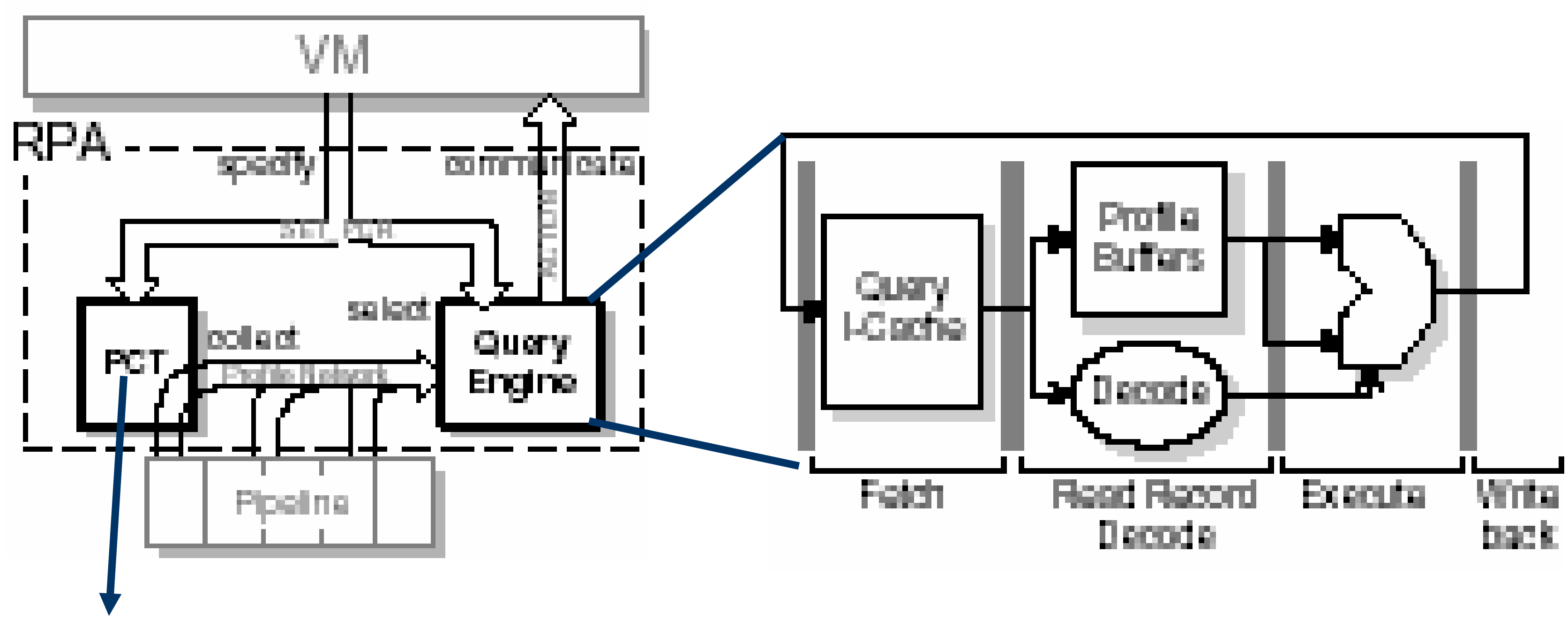
Relational Profiling Model

- 2 basic query formats
  - Instruction-based queries : “For certain instructions, what events occurred?”
  - Event-based queries: “For some events, which instructions were involved?”

Relational Profiling Architecture

- Assembly Language: specify instruction which includes sampling rates, selection criteria, and action
- 64-bit instruction contains 2 comparisons, a branch and an action

Query Engine



Profile Control Table: stores query info (rates, information to be collected) and query PC

Query Engine: 4 stage pipeline to execute queries and communicate with VM. Profile Network and buffer limits number of queries executed in parallel

Results

- Ran simulations to find optimum buffer size and number of profile networks – Garbage collection and Edge Profiling applications
- 8 Profile Buffers and 4 profile networks results in <1.7% slowdown for all cases
- Suggested improvement is to only store information at instruction retirement, not at all stages of pipeline

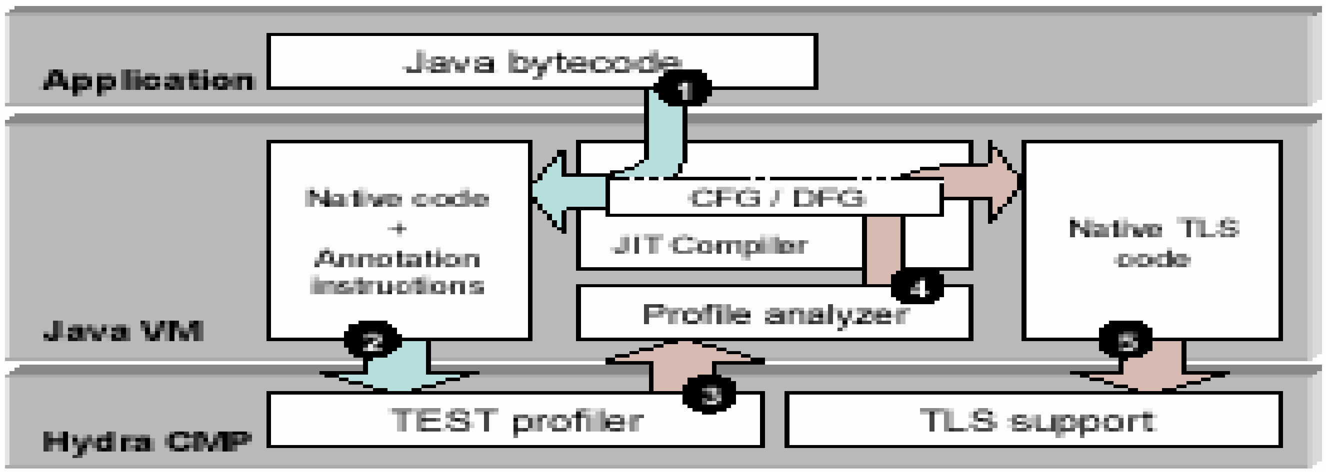
Critique

- Applications studied are fairly simple
- No discussion of area overhead of profiling hardware
- L2 cache conflicts between service and main processor
- Extensions: Dynamic control of sampling rate to decrease profiling stalls

TEST

- Hardware profiler for analyzing candidate threads for speculation
- Currently does loop-level analysis
- Used in conjunction with Hydra, a Java VM, and JIT to dynamically extract TLS
  - Create the Java Runtime Parallelizing Machine (Jrpm)

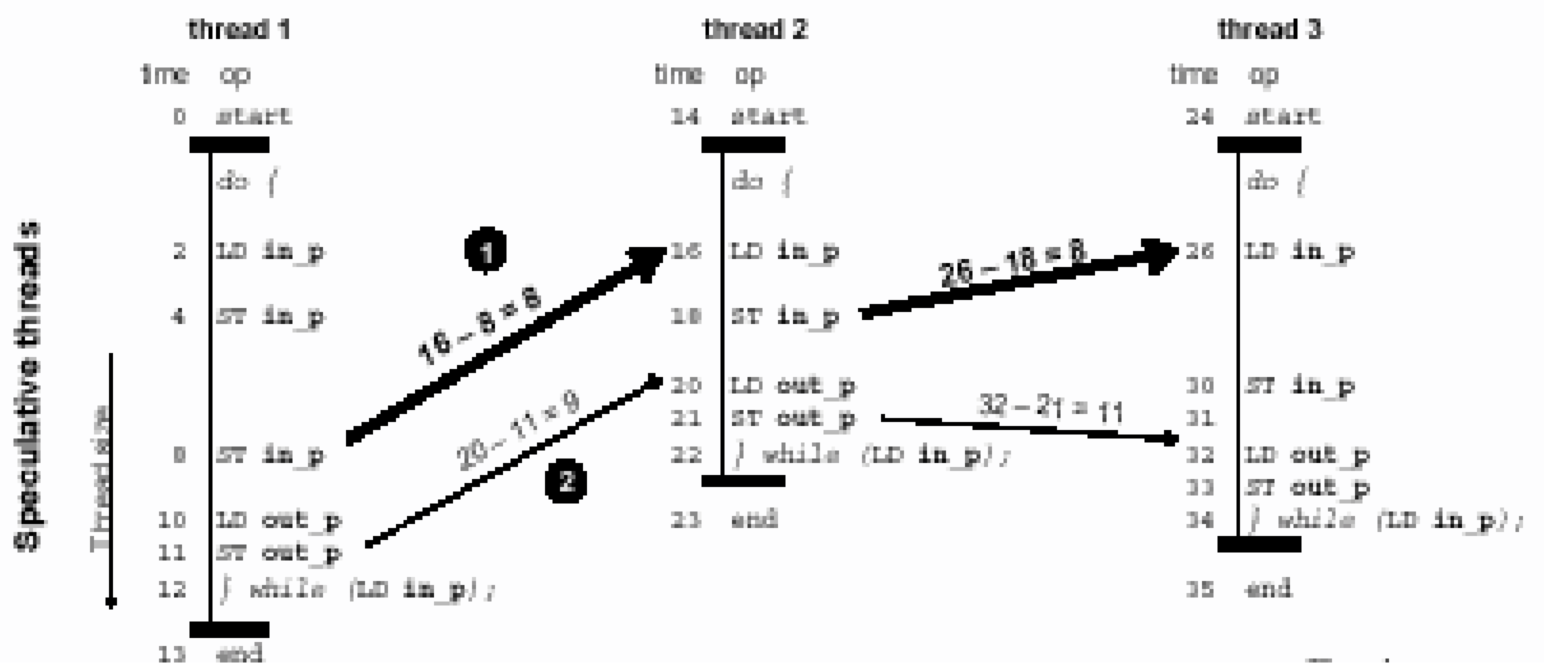
Jrpm



1. Identify possible STLs by analyzing bytecodes and compile natively with annotation instructions.
2. Run annotated program sequentially, collecting TEST profile statistics on potential STLs.
3. Post-process profile statististics and choose STLs that provide the best speedups.
4. Recompile code with TLS instructions for selected STLs.
5. Run native TLS code.

Load Dependency Analysis

– Determine critical arc lengths for various STLs by



Speculative State Overflow Analysis

- Check that speculative state for a given STL can fit in the L1 caches and store buffers
  - Also requires timestamp comparisons and counters

Hardware Implementation

- 3 main components
  - Dynamic compiler must insert annotation instructions into the code
  - Hardware comparator banks to perform the timestamp comparisons to calculate the critical arcs and the state overflow analyses and store the results into counters
  - The store buffers that are used to hold writes during the speculative execution are used during the profiling to hold the timestamp values needed for analysis

Comparator Bank

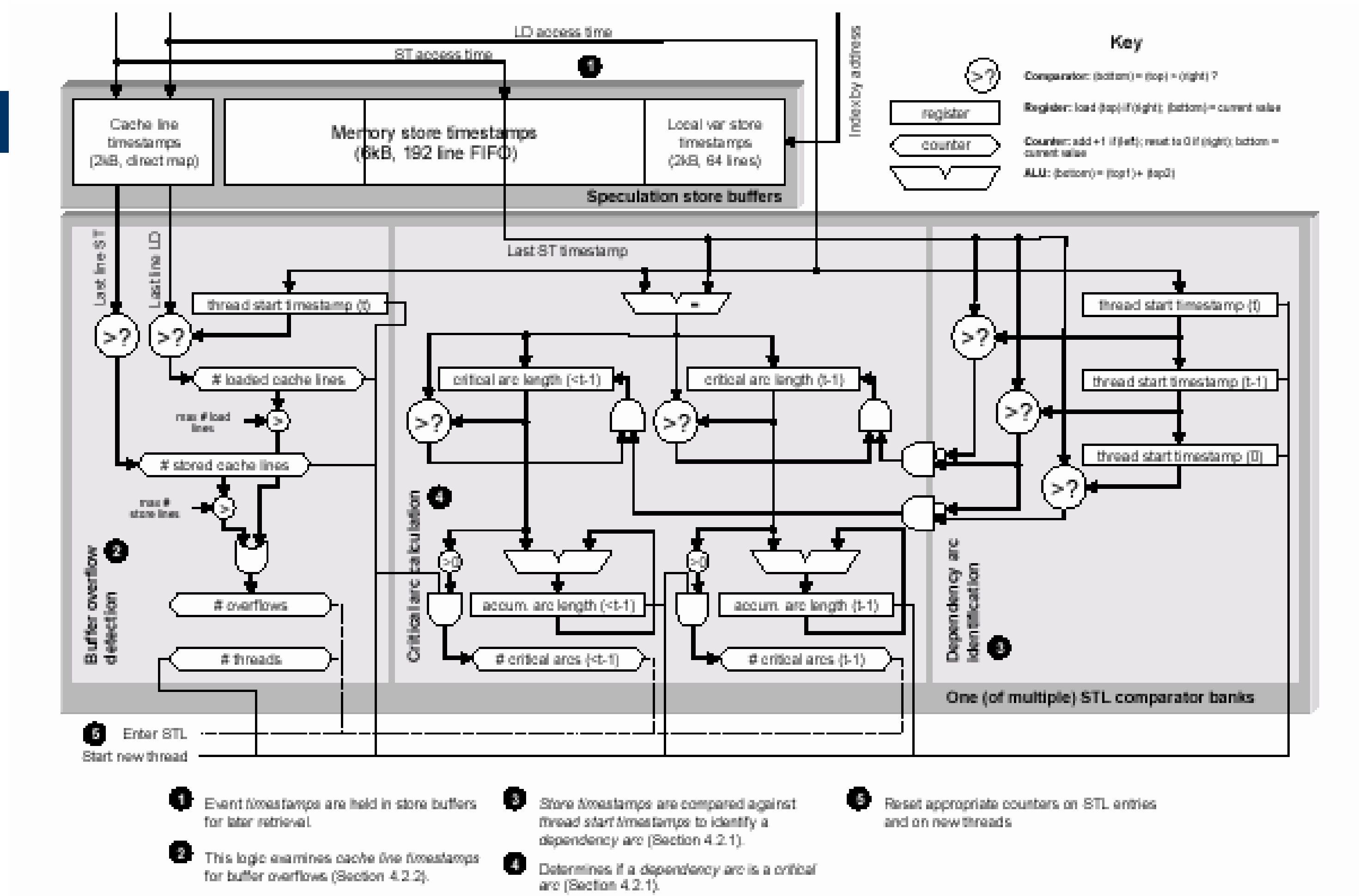


Figure 7 – Block diagram of one comparator bank.

Results

- Relative predicted vs. actual speedup quite good
- Speedup achieved with speculative execution of loop threads is promising
- Other benefits of dynamic parallelization (besides simplifying the task) is the ability to make STL selections that are input data dependent

Critique

- Currently only loop-level speculation is performed
  - Do not give reasons why other decompositions won’t work well…they just say they won’t
- Does not specify a mechanism for re-analyzing the code, in case behavior has drastically changed

Discussion

- Hardware/software support for profiling (30 minutes)
- Profiling infrastructure for CMPs (20 minutes)
- Accuracy of profiles (10 minutes)

Hardware / Software support for Profiling

- What profiling primitives do you need?
  - E.g., opcode filtering, instruction sampling, …..
  - Is the Relational Profiling Architecture sufficient?
- How much hardware support is needed?
  - TEST? Counters? Query engine? Programmable co-processor?
- What are the deciding factors – cost, performance, extensibility?

Profiling Infrastructure for CMPs

- ***Why*** do you want to profile?
  - Identify parallelism / efficient speculation?
  - (Re)configure the CMP?
- ***What*** do you want to profile?
  - Inter-thread dependencies?
  - Memory-access patterns?
- ***How?***
  - Would the Relation Profiling Architecture work?
  - Case for heterogeneity in CMP?

Accuracy of Profiles

- Tradeoff between profile accuracy and optimization capability
  - Accurate profiles ➔ higher collection overhead!
  - Compress / summarize profiled data?
- When is profile “accurate”?
  - What are the convergence metrics?
- How soon should profile reach service thread?
- How to detect changes in profiles?