A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization

Matthew C. Merten

Andrew R. Trick   Christopher N. George
John C. Gyllenhaal   Wen-mei W. Hwu

IMPACT Research Group
University of Illinois at Urbana-Champaign

Objectives

- Design a hardware-based profiler with the following characteristics
  - No probe insertion allows for faithful preservation of execution behavior
  - Focus on code that is important without extensive software analysis
  - Swift and early determination of what code is important during profiling
  - No profiling overhead until important code is detected
  - Minimize memory and processor cycles required to collect profile of important code
- Such a system will allow for
  - Profiling on real inputs instead of sample inputs
  - Static program optimization of important code based on temporal relationships
  - Runtime optimization as focused and swift detection may be a necessary component of such a system

Outline

- Objectives
- Motivation and Hot Spot Definition
- Hot Spot Detection System
- Results
- Conclusion

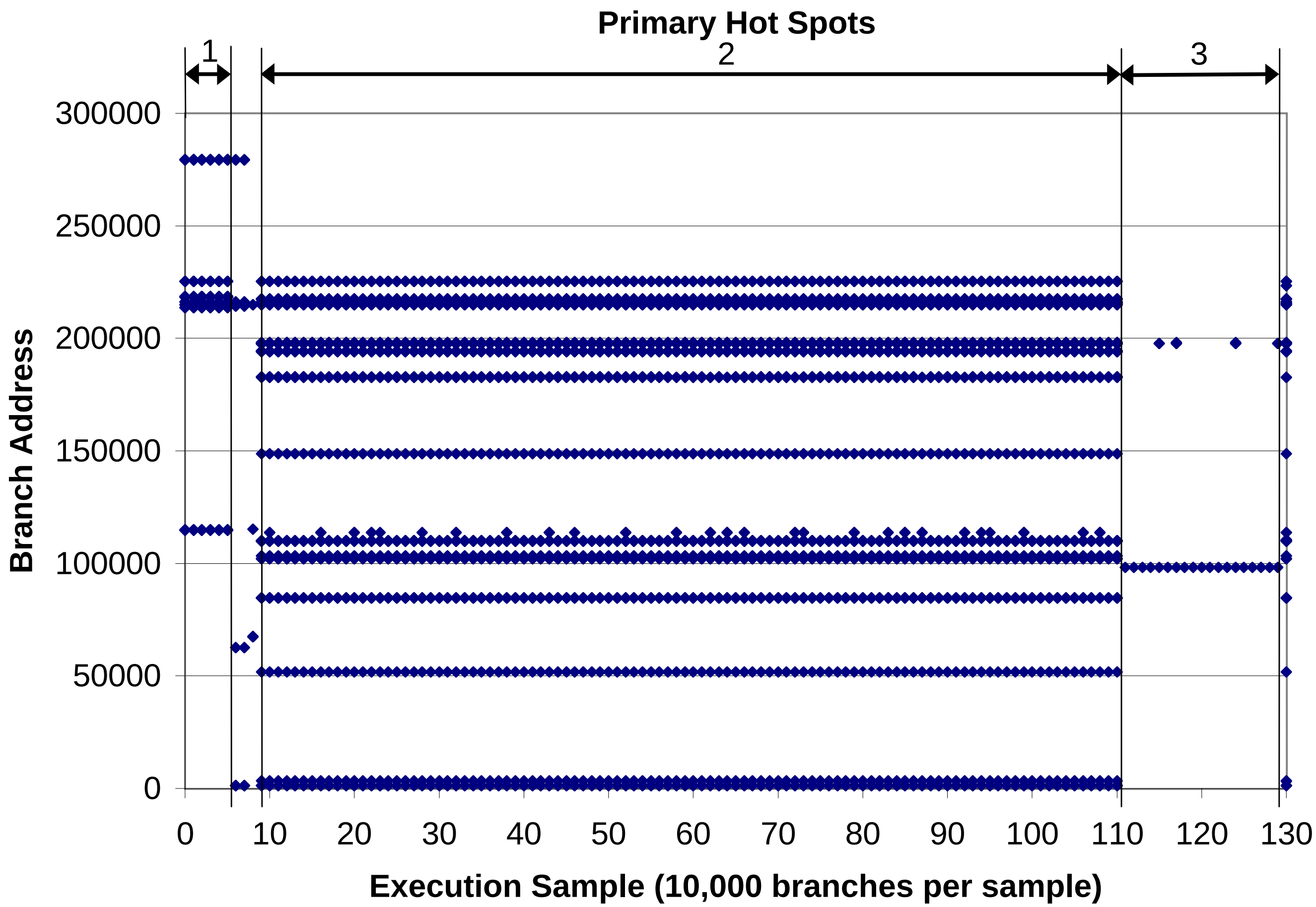
Supporting Runtime Optimization

- Current research challenges for runtime optimization
  - Reoptimize region of code that is currently important
  - Reoptimization may occur during execution or during idle time
  - Time and memory efficient
  - First step: timely detection of important blocks of code
- Existing continuous profiling mechanisms may not be ideal for runtime optimization:
  - Statistical sampling[Dean97], and the Profile Buffer[Conte96]
  - Designed to collect average behavior: more temporal context desired
  - Entire profile maintained in memory: reduce to important code
  - No analysis of behavior performed: need to recognize behavior change
  - Possible to configure them to aggressively profile time slices
    - Costly software comparison of slices
    - Responsiveness to changes in program behavior is still unclear
- We've designed a hardware profiling system that addresses these issues

Program Behavior Observations

- Execution often occurs in distinct phases
- Often contain intensely executed tasks that are good candidates for runtime optimization
  - We denote them as *hot spots*
  - Small static code size: important for a reasonable optimization cost
  - Long running period: large, positive effect on performance
  - Large dynamic execution percentage within the period: potential for significant performance gain due to optimized code
- Not always fully optimized because of wide range of program functionality and code-size issues
- Some optimizations may benefit from more specific runtime information
  - An example will be illustrated in the next few slides

Runtime Optimization: Motivational Example

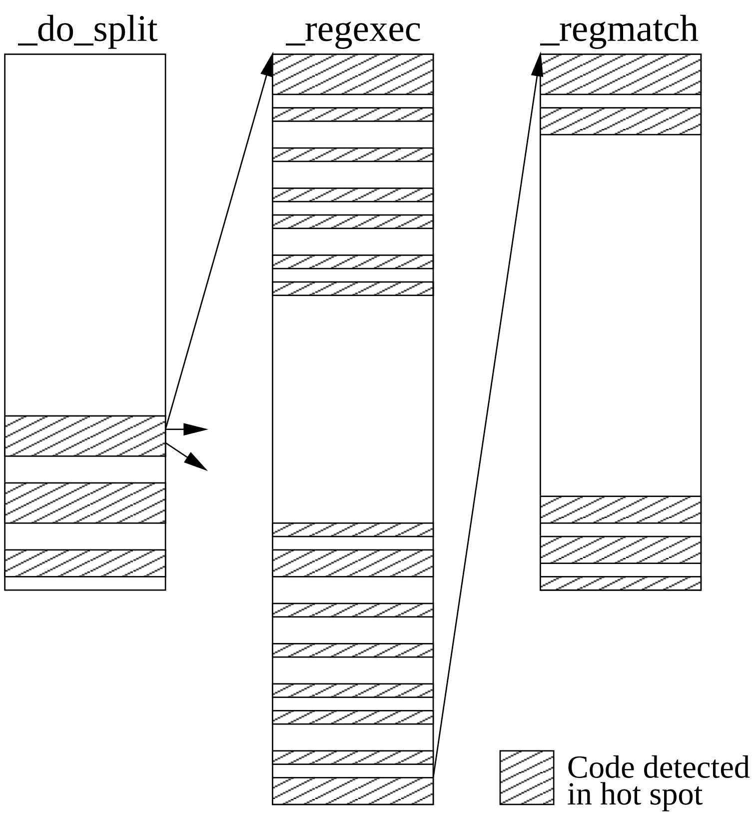


- Lots of calling
- Possible cache conflicts

- 134.perl from SPEC95 running training input jumble.pl contains three intensely executed regions of code (hot spots) that are fairly unique to the particular input.

Runtime Optimization: Motivational Example Cont.

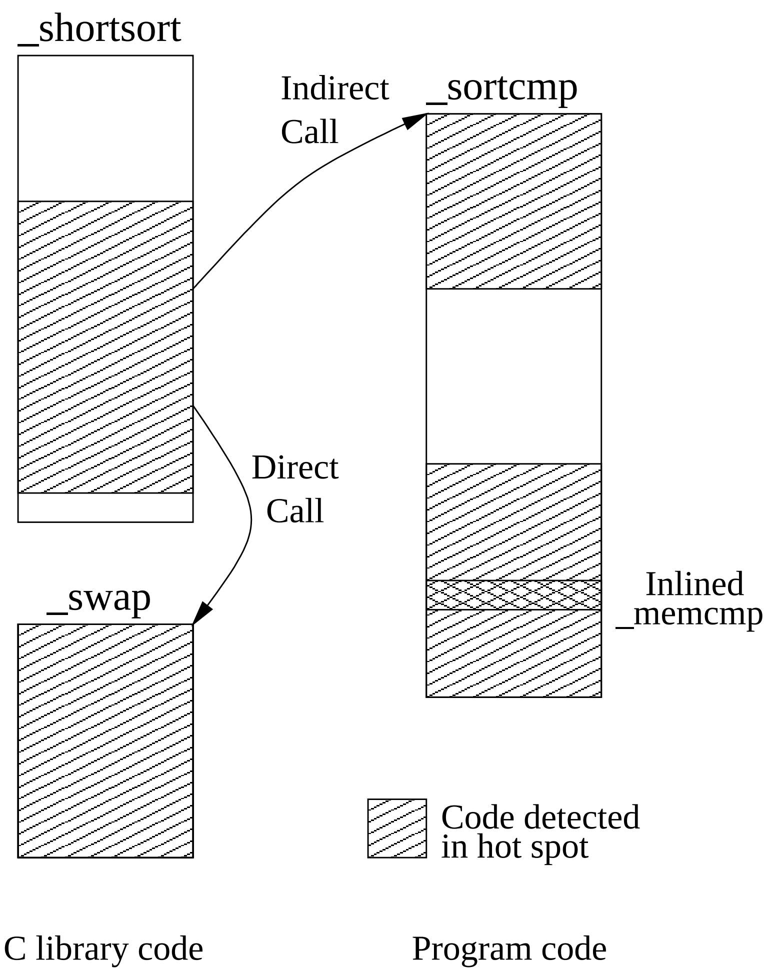
- Input script calls _*do_split* to break up an input word into individual letters; _regexec and _regmatch called with empty regular expression



  - Location: _do_split and _regexec separated by 64KB
  - Other perl library functions called from _do_split
  - Total hot spot code: 216 instructions
  - Excellent candidate for intelligent block placement and for partial inlining
  - Calling _regexec with the empty regular expression is not typical

Runtime Optimization: Motivational Example Cont.

- Second, *sort* is called to alphabetically sort the letters



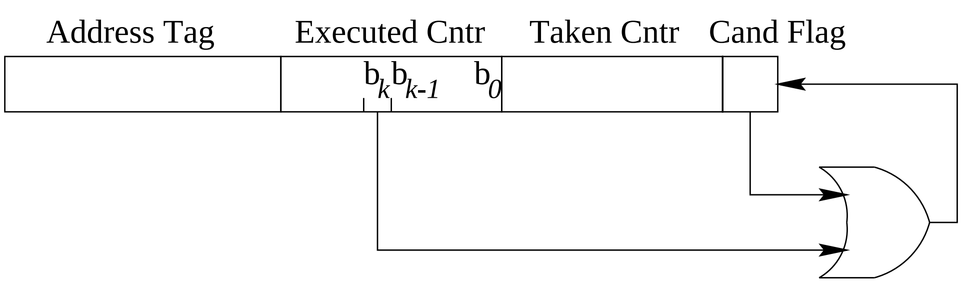
  - Location: potential cache problems
  - Indirect call: pointer analysis reveals several possible targets
  - _memcmp in _sortcmp: x86 string compare used, optimized for long strings
  - Total hot spot code: 74 instructions
  - Excellent candidate for partial inlining and optimization for single characters

Our Profiling and Hot Spot Detection System

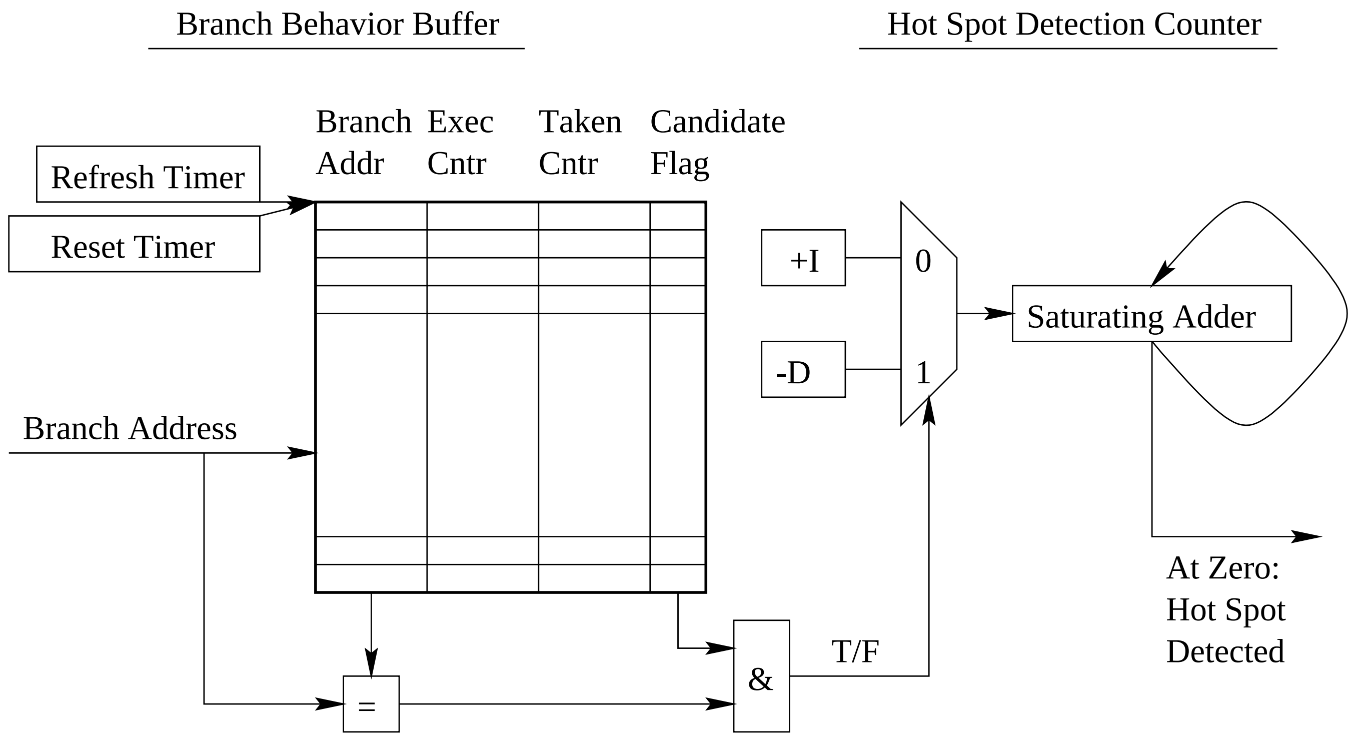
- Three system components
  - Hot Spot Detector
    - Branch Behavior Buffer (BBB): collects profiles of MFU branches
    - Hot Spot Detection Counter (HDC): monitors dynamic execution percentage accounted for by the MFU branches
  - Monitor Table
    - Maintains a collection of previously discovered hot spots
    - Monitors current execution enabling Hot Spot Detector when execution strays from known hot spots
    - Prevents redetection
  - Operating System Support
    - Software that reads the BBB entries when a hot spot is detected
    - Adds detected hot spots and optimized code to the Monitor Table
    - Calls the runtime optimizer
- All hardware located in retirement stage, off of the critical path

Hot Spot Detector

- Branch Behavior Buffer
  - Indexed on branch address (blocks derived from branches)
  - Collects executed and taken direction profiles
  - Maintains approximately a Most-Frequently-Used (MFU) replacement policy
  - Branches with executed counter above threshold are locked into table by setting the candidate bit
    - Called *candidate branches*
  - A refresh timer triggers periodically flushing non-candidate branches



Hot Spot Collection Hardware

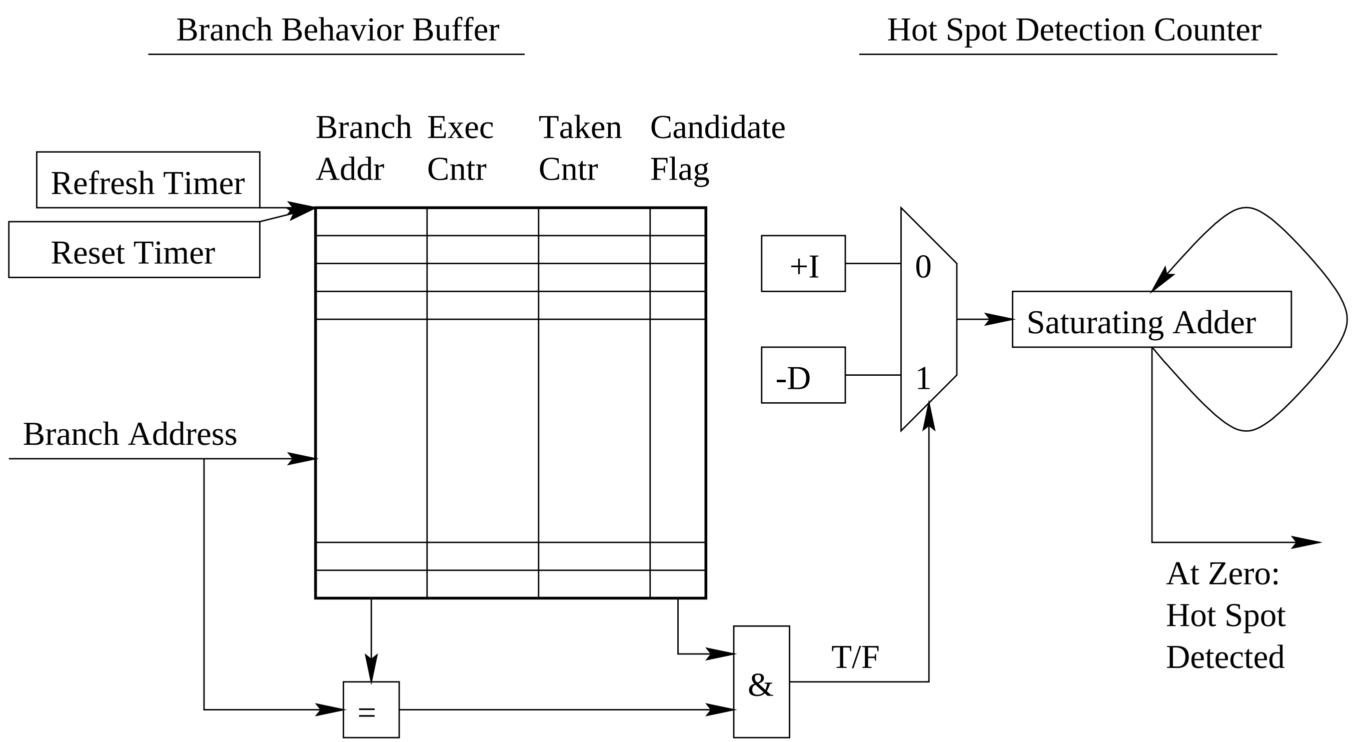


- Example BBB Configuration:
  - Branches entries: 2048
  - 2-way set assoc. indexing
  - Profile weight counters: 9 bits
  - Branches of $> .4\%$ dyn. exec.
    - Refresh timer: 4096 branches
    - Candidate branch thresh: 16
  - Reset timer: 65535 branches

Hot Spot Collection Hardware

- Hot Spot Detection Counter
  - Candidate flag indicates large dynamic execution percentage and presence in the potential hot spot
  - Threshold execution percentage is the minimum percentage of execution required to be of the candidate branches for them to be called a hot spot
  - Saturating counter:
    - decremented by D on execution of a candidate branch
    - incremented by I on execution of a non-candidate branch or table miss
    - initialized to max value, detection when reaches zero
  - I, D, and the size of the counter are determined by the threshold execution percentage
  - Actual execution percentage:
    - Slightly above minimum, slow movement of counter toward zero
    - Well above minimum, faster movement toward zero
  - Much less frequently, a reset timer triggers clearing of entire table

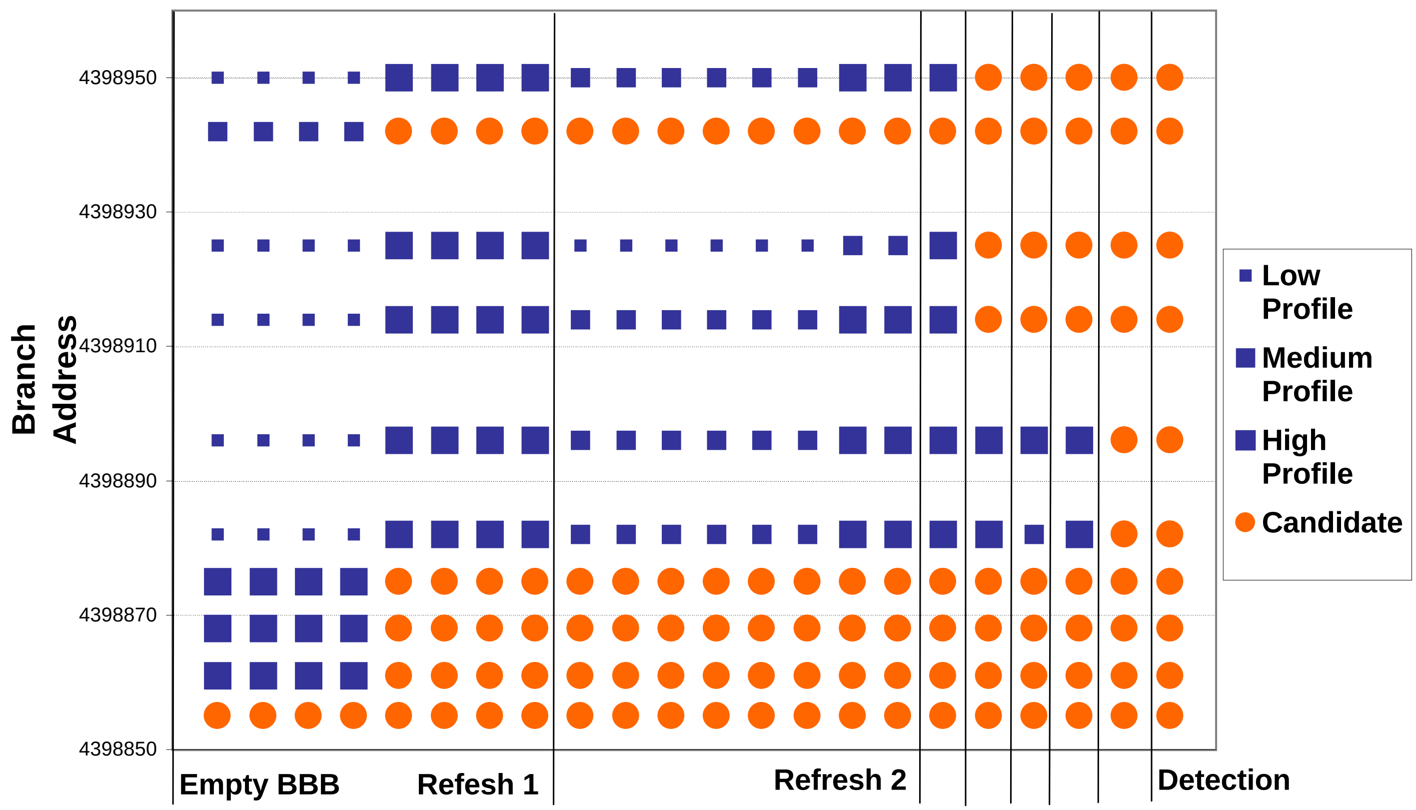
Hot Spot Collection Hardware

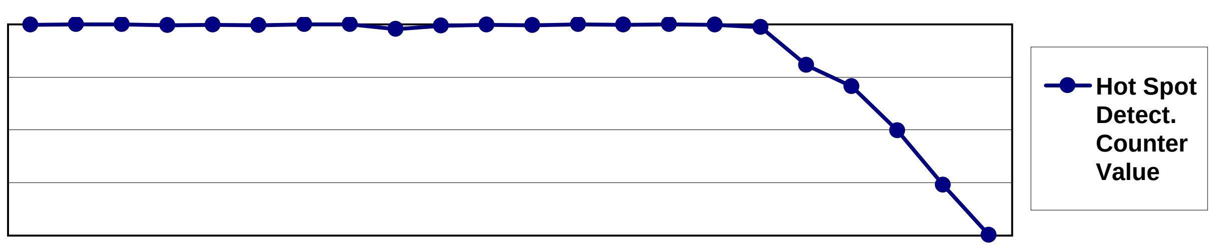


- Example HDC Configuration:
  - Counter size: 13 bits
  - Candidate branch decrement: 1
  - Non-candidate branch increment: 2

> 66% of executed branches must be candidates to be detected as a hot spot

Contents of the Branch Behavior Buffer



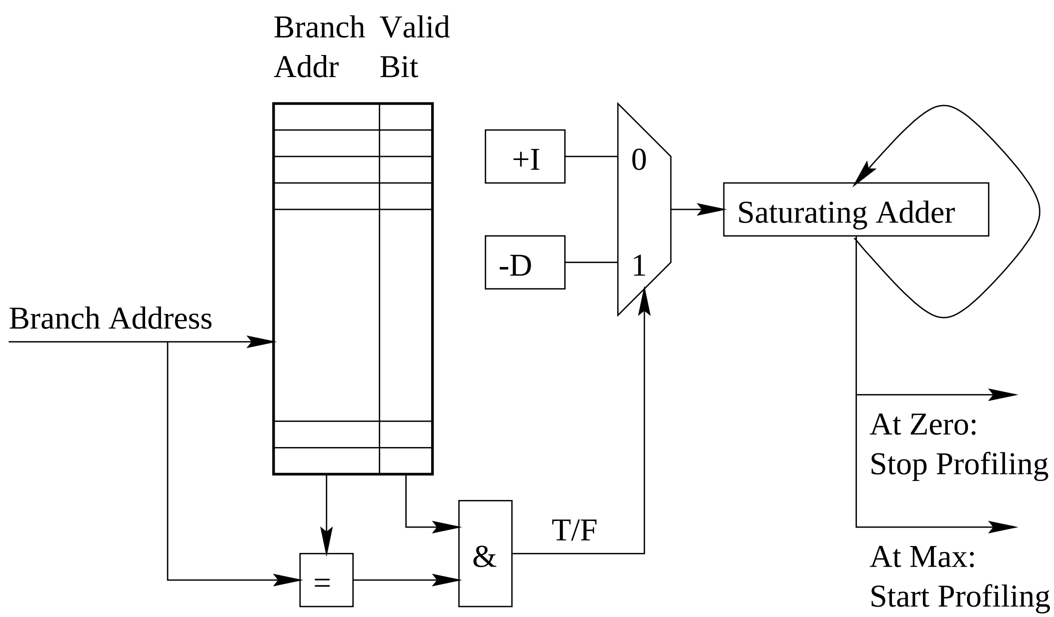


Matthew C. Merten
IMPACT Research Group

Monitor Table Hardware

- Monitor Table
  - Purpose: enable the search for a hot spot when executing in code not previously detected as being in a hot spot
  - Contains Entries for branches in all hot spots (this requirement will be relaxed later)
  - Threshold percentage is the minimum percentage of execution required to be in previously detected hot spots
  - Saturating counter, similar to HDC
    - Decremented by D on execution of a hot spot branch
    - Incremented by I on execution of non hot spot branch
    - Initialized to max value
    - Disable hot spot detector when reaches zero, enable detector when reaches max value
  - I, D, and the size of the counter are determined by the threshold percentage

Monitor Table Hardware



- Example HDC Configuration:
  - Monitor Counter size: 12 bits
  - Hot spot branch decrement: 1
  - Non hot spot branch increment: 1

$>$ 50% of executed branches must be in hot spots, otherwise the detection hardware is enabled

Operating System

- OS notified only upon detection of a hot spot (infrequently)
- BBB branch data copied into OS memory
- OS installs hot spot in Monitor Table
- OS determines when to call optimizer
- Deploys optimized code and installs it in the Monitor Table
- May have control over increment/decrement values of saturating counters
  - Adaptive control

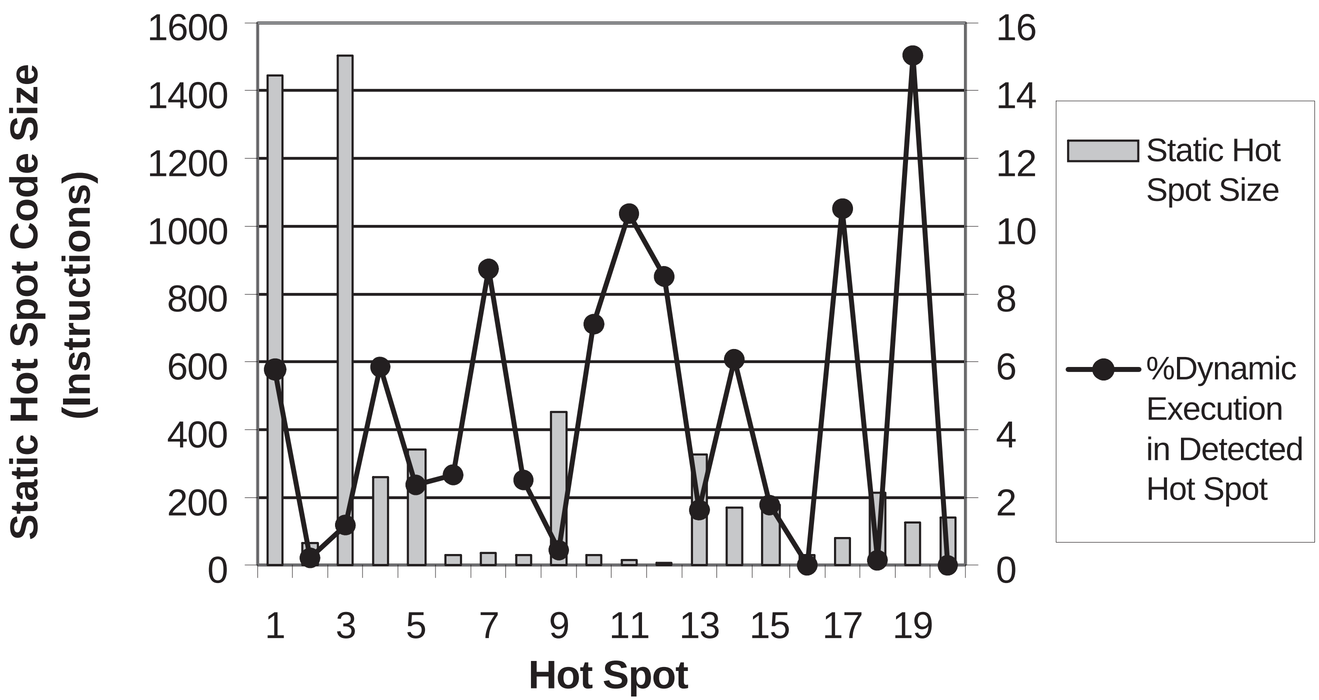
Further Refinements to Basic Design

- Multiprocess Support
  - Low utilization of BBB for most processes
  - Always utilizes Monitor Table
  - Prohibitively costly to context switch tables
  - Solution: global monitor table that enables and disables BBB for a particular process at a particular time
- Better BBB
  - Reduce size by profiling only conditional and indirect branches
  - Index on branch address and branch target (or direction) which may provide better behavior monitoring (eliminates taken counter)
- Smaller Global Monitor Table
  - 1st-level coarse-grained (range) table: best for deployed optimized code
  - 2nd-level additional per branch table (like the current table)

Simulation Methodology

- Collected complete instruction address traces executing on WindowsNT via SpeedTracer hardware from AMD
  - Traces all instructions including NT kernel, drivers, other processes, and DLLs
  - Filtered traces down to specific process and DLLs via code segment
- Benchmarks:
  - Spec95Int benchmarks VC++ 6.0 compiled *optimized for speed* and *inline where suitable*
  - General distribution versions of Word, Excel, Adobe PhotoDeluxe, and Ghostview
- Hot spot detector hardware parameters as used in the previous examples

Hot Spot Collection Results: PhotoDeluxe



PhotoDeluxe: manipulated 1.3 MB TIFF and exported to .pd image.

- 390M dynamic insts.
- 5485 static insts. in hot spots
- 1.68% static insts. executed in hot spots
- 94.3% dyn. insts. in hot spots
- 91.0% dyn. insts. in detected hot spots

Hot Spot Collection Results

Benchmark	# hot spots	# static insts. in hot spots	% static executed insts. in hot spots	% total exec. in hot spots	% total exec. in detected hot spots	Dyn. insts. in hot spots after detection
099.go	6	2398	3.46	37.84	35.39	31.7M
124.m88ksim	4	1576	2.78	93.03	92.30	110M
126.gcc	47	17665	8.90	58.42	52.12	617M
129.compress	7	918	2.12	99.93	99.81	2.87B
130.li	8	1447	3.00	91.28	90.88	137M
132.ijpeg	8	2556	3.48	91.07	91.00	1.42B
134.perl	5	1738	2.13	88.43	85.99	2.01B
147.vortex	5	2161	1.76	72.30	71.93	1.58B
MSWord(A)	5	3151	1.17	91.36	91.08	296M
MSWord(B)	21	12541	2.40	69.13	62.04	566M
MSExcel	25	18936	2.94	60.01	54.85	88.2M
PhotoD.(A)	20	5485	1.68	94.31	90.97	354M
PhotoD.(B)	14	4192	1.78	94.24	90.81	98.5M
Ghostview	33	8938	2.82	73.39	72.55	2.30B

Summary of the hot spots found in the benchmarks.

Future Work and Conclusion

- Future work
  - Further analysis of hot spots
  - Application to static compilers and runtime optimizers
    * Apply temporal behavior information in a static compiler
    * Runtime optimization system with Monitors, Optimizations, and Deployment Mechanisms
    * Region-based inter-function optimization [Hank95]
  - Advanced design for indirect branches and smaller hardware sizes
  - New types of counters for better runtime optimization support
- Developed practical hardware-based profiling method to detect hot spots
  - Consists of a few tables and counters located off the critical path
  - Minimal operating system support and overhead
  - Swift and early hot spot detection
  - On average, finds 2.9% of executed static code in hot spots which represents 79.6% of program's execution only missing 2.4% during detection.