Hardware Support For SDT:

The Use of Hardware Counters For Hot Spot Detection in Strata

Kevin Hirst, Kevin Scott, Kevin Skadron, Jack Davidson {krh2n, jks6b, skadron, jwd}@cs.virginia.edu

Abstract

Compilers have begun to push the limits of static program optimization and are showing diminishing returns in optimization of current techniques. While compilers are good at ordering data accesses and predicting behavior of processor structures such as cache and branch predictors, they cannot truly take advantage of run-time information. JIT compilers and profilers can take advantage of information collected during a program's execution, but cannot optimize programs for every execution. Strata is a dynamic optimization framework that has already been introduced by Kevin Scott. In the Strata architecture, the eventual goal is to implement speculative profiling of execution traces in frequently executed areas of running programs, or hotspots, and to use these profiles to optimize code during execution. By optimizing hotspots only, we maximize the chances that running a dynamic optimizer will be beneficial. In this paper, I will describe a low cost, hardware method of identifying hotspots and deciding where speculative profiling will take place.

1. Introduction

Modern compilers have developed many more sophisticated techniques in the past few years to improve program performance. Compilation techniques that reduce hazards associated with data access and branch resolution allow for better performance at run-time. Improvement of static compiler techniques, however, is showing diminishing returns in program performance and instruction throughput. Time can be better spent in analysis of these programs at runtime, and experiments should be run to determine how we should analyze programs at runtime.

Traditional factors that hinder program performance include cache misses, branch mispredictions, and structural hazards. Compilers have difficulty determining what the effect of these factors will be at runtime. While the compiler may arrange accesses to memory in a way that will favor cache hit ratios, the compiler does not know how well the cache will perform at runtime. The OS layer is another factor that the compiler cannot take into account. For example, process scheduling can cause certain data within a process to have a low hit rate due to cache evictions. A dynamic optimizer may be able to detect this behavior and reschedule instructions that can be executed during these cache misses.

Ideally, a dynamic optimizer would be able to monitor all of these factors and reorder or change instructions to best suite the execution environment of the processor and OS layer. Our experiments, however, are not just geared to running a dynamic optimizer on a given processor framework and analyzing the results. We propose that cooperation between the dynamic optimizer and the processor hardware can help us best exact optimization of the running program. More specifically, we wish to address the issue of *hot spot detection* in running programs. A *hot spot* is a frequently executed area of the program, usually contained in a main loop or function. We wish to identify these areas so we can determine where to best use our resources for optimization. In our addressing of hot spot detection, we wish to address the following questions.

- How do we best identify hot spots in running programs without unnecessary overhead?
- What method of hot spot detection will allow the programmer the most flexibility in choosing a method of optimization or in choosing a program area to focus optimization on?

The answers to these questions will provide processor manufacturers the information necessary to build processors that best support future needs for dynamic optimizers.

In general, our results showed that our methods of hot spot detection were low cost

compared to software methods, and detected program areas with a wide range of "hotness." Some areas were executed extremely frequently, and some were not executed as frequently, but most areas deemed hot by our detection method were executed with sufficient frequency to warrant optimization.

Our experiments are run using SimpleScalar, running a version of *Strata* that supports the PISA architecture. Section 2 of this paper will explain and justify our additions to the PISA architecture. Section 3 will detail previous work on dynamic optimizers with hardware support and hot path prediction, and Section 4 will give a breakdown of our benchmarks and our results. Section 5 will detail our conclusions and any further work needed on this research.

2 Methods

For this area of research, *Strata* was ported to the PISA architecture for simulation on the SimpleScalar system. For a detailed explanation of *Strata*'s architecture and the issues in porting this architecture to PISA, please refer to the appendix. Briefly, *Strata* is a system that builds code fragments from a running program's text segment into a fragment cache. Code from the running program is then executed from the fragment cache, and stats are taken from that execution, if need be. Optimization occurs on the individual fragments.

2.1 Hot Paths

The metric that *Strata*'s performance will be measured by is the simple equation $T_{new} + T_{opt} < T_{old}$. This simply means that in order for *Strata*'s overhead to be worthwhile, the new execution time of a program plus the time spent optimizing it must be less than the old execution time. Run-time optimization is a relatively expensive process. In addition to optimization costs, Strata incurs the cost of building fragments into the fragment cache. In order to circumvent these costs, optimizations enacted upon fragments must have a high gain. If a particular fragment is rarely executed, then optimization performed on that fragment is wasted due to the optimization costs. The simple equation $T_{gained} = (T_{old} - T_{new}) * x - T_{opt}$, where x is the number of times the fragment is executed, tells us the gain of optimizing a fragment. Hot paths are traces within the program, usually contained in loops, that have high x values. We wish to reserve our optimization for hot paths to maximize gain.

2.2 Hot Path Detection

The detection of hot paths within the running program is our central issue. Our philosophy was to provide a simple, flexible, solution, and let the individual programmer decide how to use our solution. The method of hot path detection in *Strata* is based upon a structure of hardware counters, which keeps track of how many times certain individual fragments are executed. The counter structure is an associative cache, meaning that its identifying tag is simply the program of the beginning instruction of the fragment. Each entry, or counter, also contains the number of times that the instruction at that PC has been executed. Each cycle, the processor sends the current PC to the counter structure, and the counters update themselves, using their own comparators and adders. If the PC matches a particular counter, that counter simply adds 1 to its current total. The entry contains a *callback* address, to be explained briefly. The counters are created using an instruction we added to the PISA architecture.

The counter initialization instruction has the format described in this table.

Counter Opcode	**Reg1**	**Reg2**	**Unused**
Bits 63..32	Bits 31..24	Bits 23..16	Bits 15..0

The counter instructions themselves use indirect addressing. This means that the address to monitor is the value contained in the register pointed to by reg1, and the callback address is the value in the register Reg2. These counter instructions are placed at the taken trampoline for branches with a backward taken target, where the backward taken target has already been built into the fragment cache. If the backward taken target has not already been built into the fragment cache, then we do not know which address to monitor, and cannot issue a counter instruction. Once the counter instruction has been issued into the taken trampoline and subsequently executed, we update the taken trampoline such that that counter instruction will not be executed again, and therefore we occur no additional penalty. Please see Appendix A for more on fragment layout in Strata. The layout of the trampoline is detailed in Figure 1.

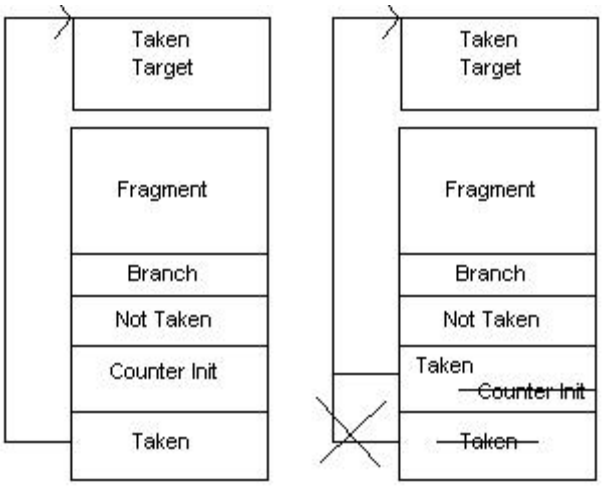


Table 1 – Counter Instructions

```
-- branches forward across jump if branch is not taken
BNE       R12, 0, 8  -- 2 insts ahead
Jump      Not_Taken_Target
Add       SP, SP,  -8
Store     T2, SP, 0
Store     T3, SP, 4
LoadImm   T2, Taken_Target
LoadImm   T3, Profiler_Address
Counter   T2, T3
Load      T2, SP, 0
Load      T3, SP, 4
Add       SP, SP,  8
Jump      Taken_Target
```

Figure 1 The counter initialization instruction(s) are later replaced with the actual jump to the taken target, since a second counter initialization is redundant.

The code is explained in Table 1– The stack is decremented by 8 to save the temporary registers, while the callback and monitoring address is sent to those registers, and the counter instruction follows, along with a restore of the temporary registers.

Strata maintains a list of addresses needing to be fixed in order to replace not-taken and taken trampolines which activate the fragment builder the first time a branch is executed. These trampolines call the builder with the target address of the branch, which is determined conditionally at execution time. When a branch's target fragment is built, the branch in the fragment which jumps to that target is "fixed" such that it no longer jumps to the Strata builder, but the fragment that contains the target. In the same way, the counter is "fixed" so that it no longer executes every time a taken branch is executed. Table 1 contains a code example.

There are several advantages to this approach. This architecture is flexible because it allows the user to start gathering information about the execution of certain fragments with a penalty of only a few instructions per counter. It allows the programmer to use the information gathered by the counters in a method of his or her choosing. Another advantage our paper addresses is the actual savings, measured in processor cycles, of using our hardware counters versus normal software methods in hot path detection. This is a significant part of the rationale for implementing such primitives in hardware.

This approach to hot path detection also has a low hardware cost. An individual counter has a comparator, an adder, a tag, an unsigned integer, and a 32-bit callback address. This structure is detailed in figure 3. This may seem fairly large, but the size of the structure will depend on the number of counters that are actually needed. This can vary highly from application to application.

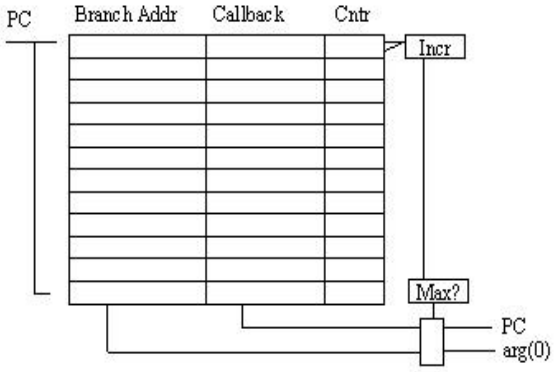


Figure 2 - the counter structure. In this figure, the top line is a match, and the counter is incremented and checked versus the threshold. If it matches, the PC is latched out from the callback address and fed the branch address as an argument.

Finally, the counter implementation would not be on the critical path of the processor. As shown in figure 3, the PC would simply be broadcast to the counter structure and checked versus all of the existing tags. The

counter is incremented if necessary. Therefore, there is no hardware slowdown of other instructions in normal program execution.

We limited our hot spot detection to loop constructs within original program code. Figure 1 shows the placement of the counter instructions, and the method by which *Strata* avoids executing the counter instructions more than once. Because we limited our focus to program loops, counters are only created at backward taken branch targets, since backward branches represent these loops. The disadvantage of this approach is that functions called many times outside of loops will not be fully optimized. However, we will still experience worthwhile gain from these functions, because the hot paths within these functions will be executed many more times than the function itself, and those hot paths will be well optimized. However, our counters are flexible enough for any programmer to use a different method, if he or she chooses.

When a counter associated with a loop head reaches a particular threshold, we consider that area of the program to be hot, and at that point we would begin speculative profiling of that loop. At this point, however, we do not turn off the hardware counter monitoring this loop. Because we want a fair assessment of how worthy these loops are of optimization, the counter continues to keep track of the number of times that loop is executed, despite reaching the threshold. Keeping the counter active helps us determine what percentage of program execution time a loop is occupying. Since the hardware monitoring is a transparent function (i.e. there are no instructions required to continue the monitoring) the executing program will incur no instruction penalty as a result of the continued monitoring, except where a profiler might decide to read a particular counter. The amount of time we should spend optimizing a loop is probably correlated to its percentage of its execution time by a hyperbolic function. That is, we should spend more and more time optimizing loops with large percentages of execution, but keep in mind that any optimizer will produce diminishing returns after a certain number of times. This topic will be covered as we expand the capabilities of *Strata*.

The *callback* address associated with the counter is the address of our speculative profiler, and the processor calls that function using the associated PC as a parameter when the counter reaches our threshold. The method and implementation of speculative profiling will be left to a future paper, since we are only interested in hot spot detection here. Speculative profiling keeps track of the most commonly executed path within the loop or hot spot, and puts a greater priority on optimization of that path. We considered a reasonable threshold to be four executions. Using this metric, we assume that optimization of a loop would equal the loop in number of instructions executed. Therefore, if we achieved a 25% faster solution by optimization, which is a reasonable assumption, then we have broken even.

In order to check our results versus a similar software method, we implemented an identical structure to our hardware counters in software as well. We keep the same data as our hardware counters (a callback address and a counter), except we implement them as a standard array in main memory. Since main memory is inexpensive, we keep a large array of these structures and use a hash function to map the branch PC to certain slots. The hash function is only a cost of a few instructions, but there are several loads which drive the cost of initializing and/or incrementing a counter to 25-30 instructions, and introduce the possibilities of data cache misses. Table 2 explains the software method and its drawbacks.

Table 2 – Software Counters

This is the equivalent software implementation – the hash function must be called and a match determined on the corresponding counter in memory. If a match is not found in the hashed slot, the counter in that slot is initialized to zero, and the callback and branch addresses are written. If a match is found, the counter is incremented.

The value of the address kept in the hashed slot is returned in register T0, the address of the slot is in T1

```
BNE       R12, 0, 8  -- 2 insts ahead
Jump      Not_Taken_Target  - not taken trampoline
AddImm    SP, SP , -16
Store     T0, SP, 0
Store     T1, SP, 4
Store     T2, SP, 8
Store     T3, SP, 12
LoadImm   T1, Slot_Addr – determined by strata builder
LoadImm   T2, Taken_Target – determined by strata builder
JAL       Address_Fetch  - fetches address into T0
BEQ       T2, T0, 32  -if the addresses match skip 4 insts
LoadImm   T3, Profiler_Address – determined by builder
Store     T2, T1, 0 – store new address of counter
Store     Z0, T1, 4  - set new counter to 0
Store     T3, T1, 8 – store profiler address
Load      T3, T1, 4
AddImm    T3, T3, 1 – increment counter
Store     T3, T1, 4   - store it back
LoadImm   T0, T0, 4
BNE       T0, T3, 16 – if counter <> 4 skip two insts
AddImm    T1, T2, 0 – put address in parameter register
JAL       Profiler_Address
Load      T0, SP, 0
Load      T1, SP, 4
Load      T2, SP, 8
Load      T3, SP, 12
AddImm    SP, SP , 16
Jump      TakenTarget
```

In addition to the instructions that are in Table 2, the software method incurs a penalty for a *jump-and-link* to fetch the address in the proper slot for comparison with its taken address, and it incurs a delay for two branch determinations, as there is no branch delay slot in the PISA architecture. Another drawback of the software method of profiling is that without adding a high additional cost to Strata for keeping track of which branches have reached their execution threshold, we cannot turn off these instructions or rewrite the branch in the same method as we do for hardware counters. Therefore, the software method will not discontinue profiling of a branch when the threshold of four executions is reached, and will continue to incur execution penalties. When a speculative profiler has been implemented, we will be able to remove software instrumentation, but at the cost of keeping lists of those locations and information on how to remove them. Our hardware method of profiling will continue to increment a counter after it reaches the threshold, but it will not incur instruction penalties since this occurs as part of the processor's natural functions.

3 Related Work

Many dynamic optimization systems have been written in the last few years. As with other areas of research, many of these systems incorporate similar techniques and methods. We will review some of the more recent methods of hot path profiling.

Hot path profiling has become a popular topic within the realm of dynamic optimization. SDT papers in general recognize the value of reserving optimization for frequently executed program blocks. We look at a few methods that are software based only. Kistler and Franz [3] incorporate *trace scheduling* into their path profiling code. Their algorithm starts at the beginning of a procedure and collects the hot trace by selecting the most frequently taken fork until the end of the procedure. By selecting traces until they have full coverage of the basic blocks, they claim to have divided the entire hot path of the program into corresponding traces. By profiling the entire program unselectively, they have introduced large overhead into their system, and their results show that the hot path profiling hurts execution time. Patel and Lumetta[4] also construct traces based on selection of highly biased branches. Their algorithm removes branches and replaces them with assertion instructions after they show highly biased behavior. Assertions are fired if the branches do not follow the predicted path, and the trace is eliminated. Areas with highly biased behavior are constructed into traces and executed as such, whereas areas without such behavior are executed normally. The optimizer then focuses on the constructed traces. This demonstrates an ability to focus on highly executed areas, but profiling must be occurring at all times so the optimizer can focus on wanted areas. These solutions could clearly be implemented using counter instructions, which would only execute a speculative profiler after a hot spot is detected, rather than depending on a profiler running during a program's entire execution.

Finally, Hwu et. al [2] propose a hardware architecture for dynamic optimization that focuses primarily on hot spot detection and determination heuristics. They propose an architecture that sits at the retirement stage of the processor and collects information on branches. A *branch behavior buffer* collects information about the paths of particular branches. Frequently executed branches are collected into a list of *candidate* branches, and infrequently executed branches are *non-candidate* branches. As the program executes, the *hot spot detector* determines the ratio of candidates to non-candidates, and trace collection starts when the ratio exceeds a certain amount. This architecture would seem to limit hot spot detection to short, tight loops, considering that the hot spot detection operates off of a local ratio. Since branches are removed from candidacy as a function of their execution percentage to all other recent branches, a loop with a large number of branches inside would be invalidated simply as a matter of the total number of branches. Trace collection would not occur no matter how many times the loop was executed, and optimization would therefore not occur. The primary advantage of the simple hardware counters over Hwu's hot spot detector is that we may use software to implement any of a number of hot spot detection or trace collection algorithms – including long loops that are branch heavy.

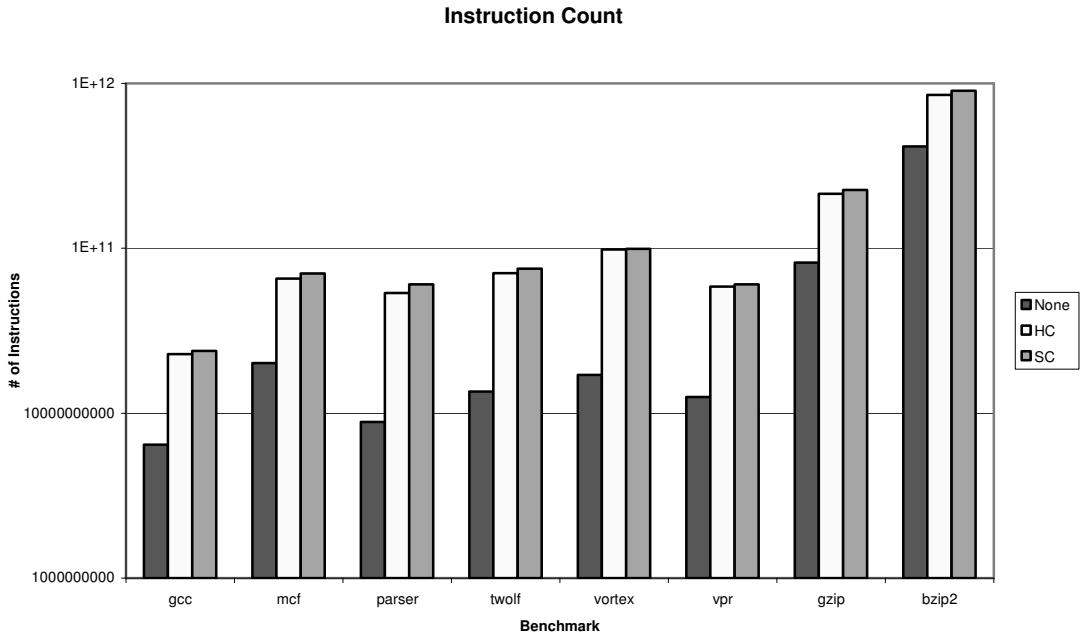


Figure 3 - Instructions executed per benchmark

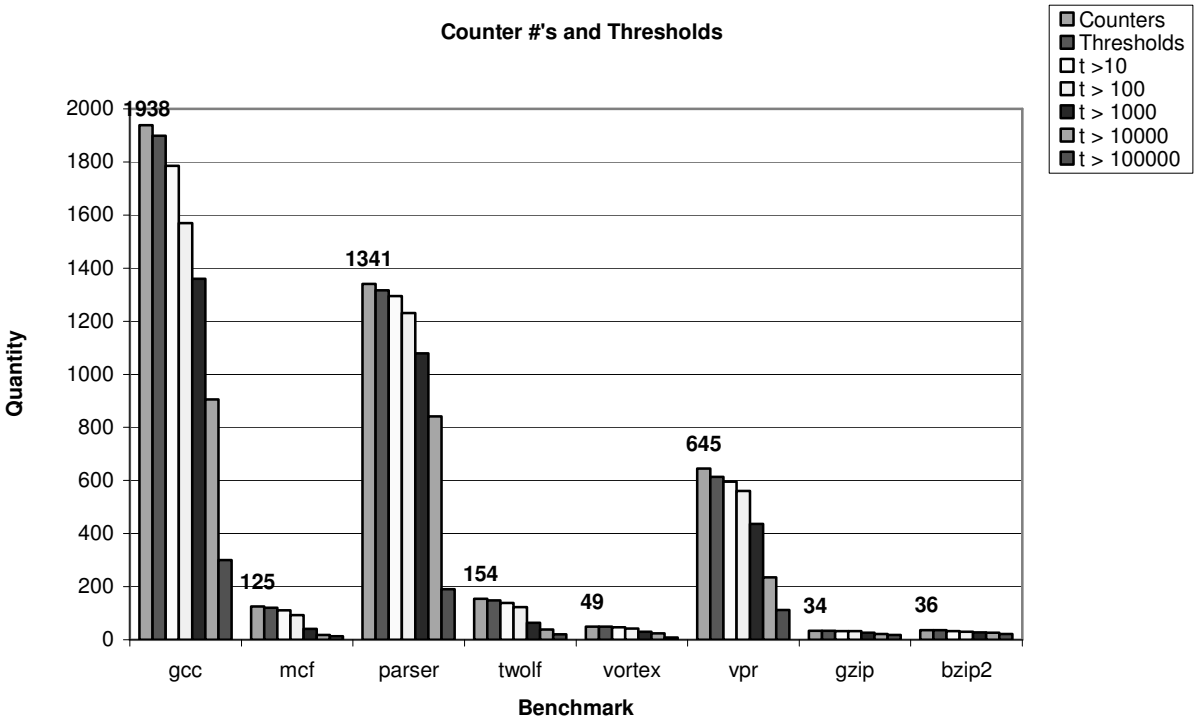


Figure 4 - Number of Counters Used Per benchmark, and execution thresholds reached per threshold level

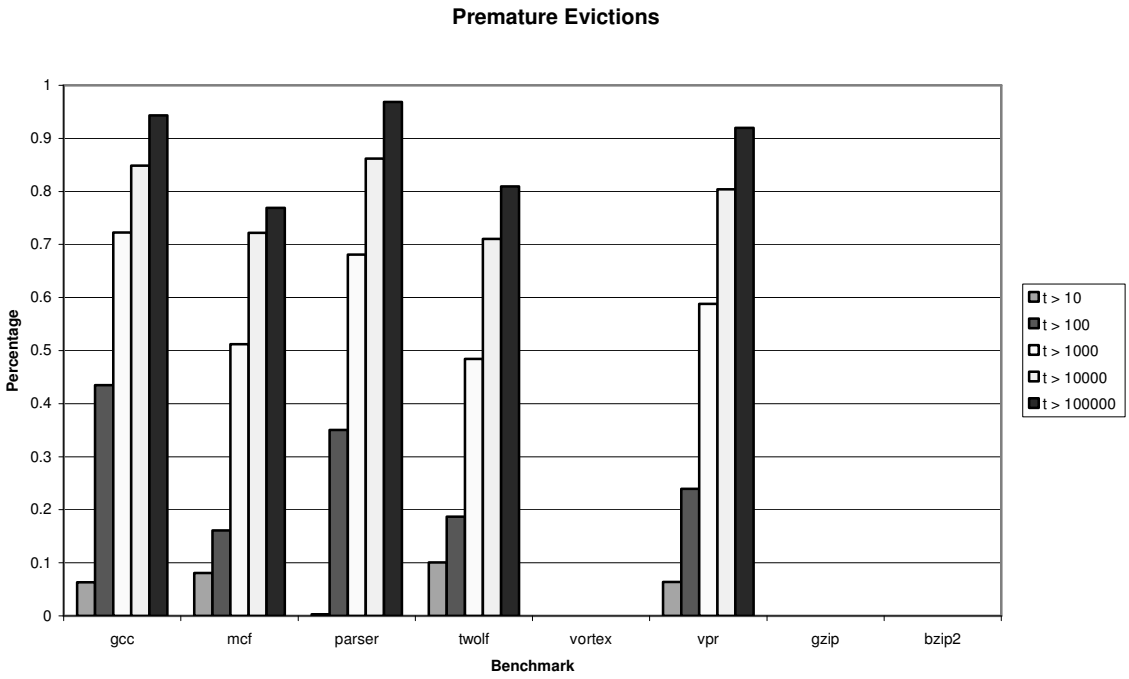


Figure 5 - Premature Evictions Percentage per Threshold Level

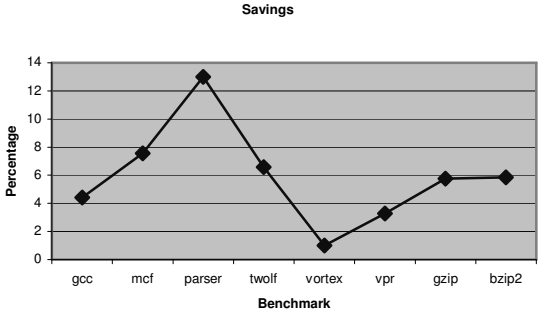


Figure 6- Instruction Savings per benchmark

4 Results

We took statistics on eight spec2000 benchmarks, running each benchmark from start to completion. Our focus is primarily on the savings that our hardware counters offer over similar software profiling. In addition, we collected the number of thresholds reached by individual counters, and the total executions of instructions at each monitored address. A monitored address is an address with an associated hardware counter. Figure 3 shows the number of instructions executed per benchmark, with the hardware counters, and with efficient software counters, and Figure 6 shows the savings of hardware counters as a percentage of overall instruction count.

Table 3 – Average number of executions at addresses with associated counters

Benchmark	Execution Quantity
vortex	244078
gcc	56049
twolf	124987
mcf	314984
vpr	98473
gzip	73666494
bzip	123110447
parser	95768

Table 3 is introduced as the average number of executions per monitored address(address with an associated counter). The instruction savings results in Figure 6 show an increase in instruction count of 1.5% to 7.5%. This is a wide range of values, but readily explainable because software methods of hot path detection cannot update their collected data without additional instruction overhead. There are instruction penalties for each counter initialization and increment, as well as address

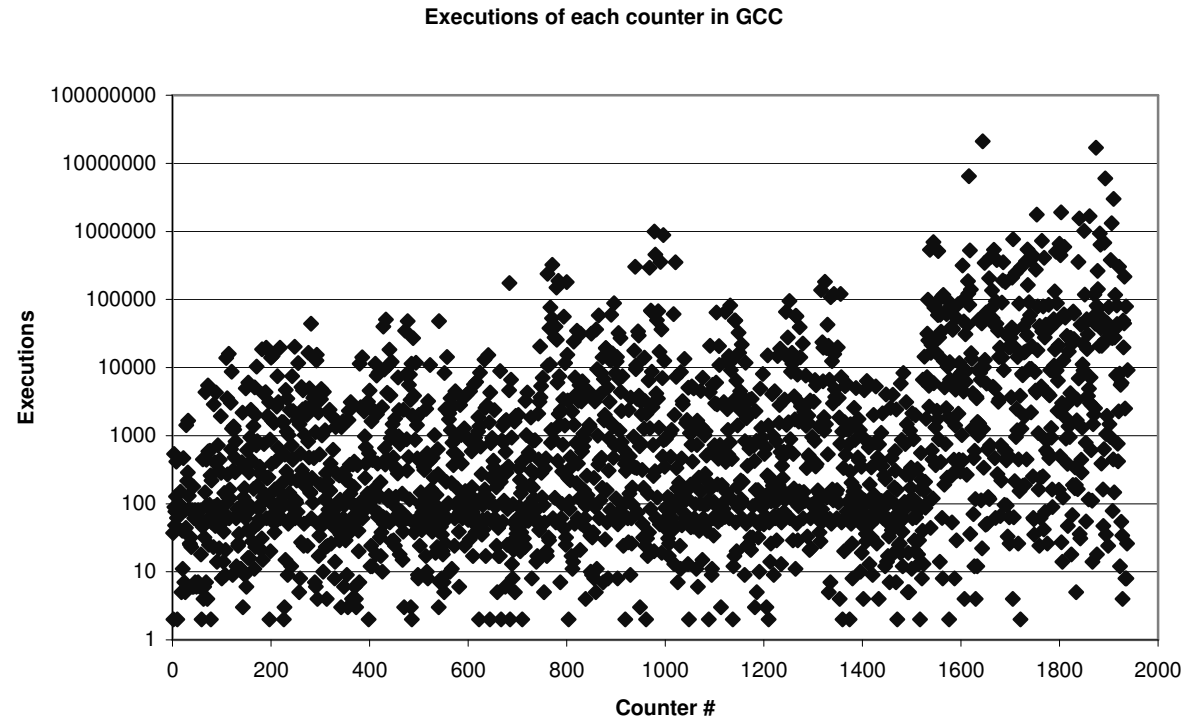
lookup penalties. The parser, twolf, and mcf benchmarks sustained the highest penalties for maintaining software counters. Parser had a high number of loops for its instruction count, and mcf and twolf incurred large penalties because their loops executed many more times than other benchmarks, meaning that they incurred the penalty for each execution. Parser spent more time executing instructions within its loops than other benchmarks because the coverage of loops in the parser program was higher, probably in string comparison algorithms. *Gcc* averaged half an order of magnitude lower in terms of execution of instructions at loop addresses. Since the number of instructions executed in *gcc* was not an order of magnitude lower, its overhead was much lower than *parser*. The benchmarks *twolf* and *mcf* each instantiated a fair number of counters, and averaged a higher number of executions per monitored address as *parser*, but since there were fewer monitored addresses, the software overhead was not as high. The *bzip, gzip,*and *vortex* had few monitored addresses, but *bzip* and *gzip* executed instructions at their monitored addresses extremely frequently, therefore encountering the software penalty a high number of times, and saving around five percent.

In Figure 3, the instruction count of benchmarks that were not instrumented with the Strata architecture were much lower than those instrumented with Strata. This is because of the lack of an indirect branch cache. Thus, every time an indirect branch was encountered, the fragment builder in Strata had to determine where the location of the next fragment was. In a benchmark such as gcc, the overhead was particularly great due to the number of procedure calls, while bzip had very few, and therefore incurred less instrumentation penalty. For a more detailed look at the Strata architecture look at Appendix A.

Figure 4 displays the number of counters instantiated in each benchmark, and the number of counters that reached the four-execution threshold, thus instantiating a would-be speculative profiler. In addition, Figure 4 displays the number of counters that reached the thresholds of 10-100000 executions in increasing orders of magnitude. Similarly to cache structures, the counters had an eviction policy. The counters were implemented as a circular buffer. That is, the counters were kept in a 64-element array with an index pointer. When a counter is instantiated, it replaces the current counter at the structure pointed to by the index pointer. The index pointer is then incremented to point to the next structure, and if the index pointer is incremented past the number of elements, it is modified to point to the beginning of the structure. So each counter is subject to possible eviction from the counter structure. The numbers in Figure 4 show the number of executions at monitored addresses *without* evictions. Therefore, figure 5 shows numbers that are representative of an infinitely large collection of counters. Figure 5 displays the overall number of counters that would reach the corresponding threshold (t) of executions, but would be evicted before they could reach that threshold, as a result of the counter structure implementation.

Overall, these figures show that a large percentage of counters would reach at least a 100-execution threshold without being evicted from a 64-entry counter structure. In the three benchmarks with less than 64 counters, every counter would of course never be evicted. In benchmarks with greater numbers of instantiated counters, however, there was a sharp curve in the percentage of counters that would be evicted as the corresponding threshold increased. In gcc and parser, around seventy percent of speculative profiling candidate loops would be evicted prematurely if the execution threshold were set at 1000 executions, simply due to the sheer number of counters that the program required. By the time counters reached 100,000 executions, around 95% or more were often evicted.

These results show that optimal detection of “hotness” of loop heads would probably be obtained by setting thresholds at around 100 executions. This way, most counters would reach the threshold without being evicted.



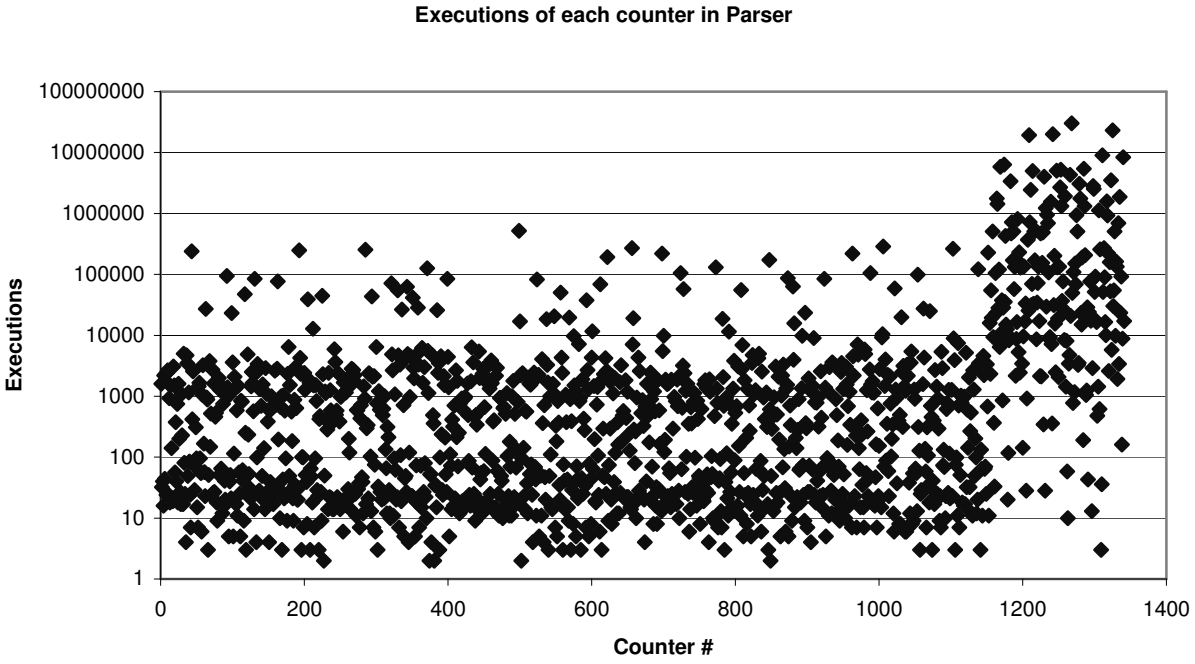


Figure 7 - Scatter Plots for Counter Benchmarks PARSER and GCC

Finally, figure 7 shows a detailed scatter plot of the number of executions for each counter instantiated in the benchmark *gcc*. There are obvious areas of concentration for our data points in gcc. Around 100 – 1000 executions, there is a clear concentration, and, even across the whole graph, an obvious majority of points rest between 100 and 10000 executions. This supports our conclusions regarding the eviction data in Figure 5. From this data, we can see that the area of concentration revolves around the optimal area of counter thresholds reached versus premature evictions. In other words, a large percentage of our loop heads would be detected as hot if the threshold were set at 100 executions, and few of their counters would be evicted. The *parser* graph shows two areas of concentration between 10 and 100 executions and between 100 and 10000 executions. The counters are clearly more evenly dispersed in parser, but show the same general areas of concentration as gcc, which supports our claim as well.

5 Future Work/Conclusions

Our hardware execution counters were extremely efficient in detecting hot spots, showing up to 12% gain in instructions per program over a simplistic software method. At a one time cost of 11 instructions (plus one-time cost of removal), these counters seem to be an extremely efficient method of hot spot detection. In addition, we were able to see some groupings that led us to clear conclusions about possible threshold levels for initialization of speculative execution profiling. In the future, we would like to collect more data for groupings for more executions with high numbers of loops. Since several of the benchmarks required few counters, we were not able to gauge the usefulness of a higher threshold on a macro level.

However, the extreme "hotness" of some of the program areas of the benchmarks lead us to conclude that this simple method of hot spot detection would lead to reasonably beneficial results when the program areas are optimized at run-time. In the future, we would like to implement an optimizer and measure our instruction savings on each benchmark. We would like to see our system's performance when combined with other systems such as speculative profilers, for collecting most frequently executed paths within program hot spots, or more importantly, the dynamic optimizer itself. However, since the average number of executions per monitored address was high for each benchmark, we can conclude that an average loop would not have to be highly optimized to show gain in real execution time.

In the future as well, we would like to be able to simulate benchmarks using a *floating threshold*, which could feasibly modify the counter structure such that the thresholds were dynamic. This could help if a loop was executing quite frequently but had few instructions per loop. (i.e. its counter was getting incremented quite frequently) Tight loops are harder to optimize and should require more executions to qualify as a speculative profiling candidate. This area of work was somewhat addressed by the solution in [2], but at a high cost.

In summary, we conclude that simple, low hardware cost instruction counters seem to be an efficient and worthwhile method of *hot spot detection.* The numbers we have collected here, especially the average number of executions of monitored addresses, show that cooperative hardware/software methods of hot spot detection and speculative profiling warrant further study.

References

[1] **Ebcioglu, Kemal, et. al**. "Dynamic Binary Translation and Optimization" *IEEE Transactions on Computers*, Vol. 50, No. 6, June 2001

[2] **Merten, Matthew C., et. al.** "An Architectural Framework for Runtime Optimization" *IEEE Transactions on Computers*, Vol. 50, No. 6, June 2001

[3] **Kistler, Thomas and Franz, Michael.** "Continuous Program Optimization: Design and Evaluation" *IEEE Transactions on Computers*, Vol. 50, No. 6, June 2001

[4] **Patel, Sanjay J. and Lumetta, Steven S.** "rePLay: A Hardware Framework for Dynamic Optimization" *IEEE Transactions on Computers*, Vol. 50, No. 6, June 2001

[5] **Voss. Michael J. and Eigenmann, Rudolf.** "Adapt: Automated De-Coupled Adaptive Program Transformation" *IEEE Transactions on Computers*, Vol. 50, No. 6, June 2001

[6] **Merten, M.C.; Trick, A.R.; George, C.N.; Gyllenhaal, J.C.; Hwu, W.W.**
"A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization " *Proceedings of the 26th International Symposium on Computer Architecture*, 1999. Page(s): 136 -148

Appendix A
Strata Architecture

Strata is an architectural framework for dynamic optimization that is based upon a *fragment cache*, where code from a running program is stored awaiting execution or optimization. When *Strata* begins, control is transferred to the fragment builder, which pulls code out of the program's original text segment and copies it into a fragment until it sees a conditional branch. At a conditional branch, two *trampolines* are built which transfer control back to the fragment builder. One trampoline tells the fragment builder to begin at the taken address of the branch, and one trampoline tells the builder to begin at the not taken address. The fragment cache layout is detailed in Figure A.

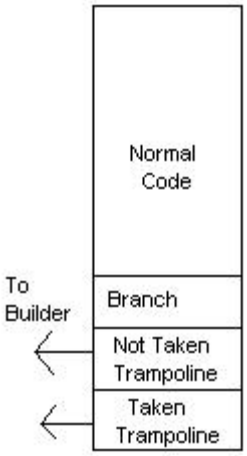


Figure A
Fragment Cache Layout

The function of the trampolines initially is just to pass the taken or not taken address to the builder, so the builder may begin pulling code from that address in text segment to start building the new fragment. However, the fragment cache is advanced enough that we know if a fragment for that text address has already been built. If a fragment for that address has already been built, then the trampoline is just a jump to that fragment. This is shown in Figure B.

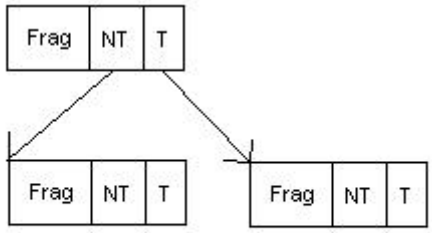


Figure B
Control Change on a Fragment Cache Hit

In addition to being able to build direct jumps from fragment to fragment, *Strata* can also update fragments built when the branch target was not in the cache. The trampolines of those fragments initially point to the builder, but once the branch target is built into the fragment cache, the builder updates those trampolines to point directly to the target fragment. This reduces overhead for fragments executed more than once, because calls to the builder are expensive.