

# 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

Center for Reliable and High-Performance Computing

Department of Electrical and Computer Engineering

University of Illinois

Urbana, IL 61801

Email: {merten, atrick, c-george, gyllen, hwu}@crhc.uiuc.edu

## Abstract

*This paper presents a novel hardware-based approach for identifying, profiling, and monitoring hot spots in order to support runtime optimization of general-purpose programs. The proposed approach consists of a set of tightly coupled hardware tables and control logic modules that are placed in the retirement stage of a processor pipeline removed from the critical path. The features of the proposed design include rapid detection of program hot spots after changes in execution behavior, runtime-tunable selection criteria for hot spot detection, and negligible overhead during application execution. Experiments using several SPEC95 benchmarks, as well as several large WindowsNT applications, demonstrate the promise of the proposed design.*

## 1 Introduction

Optimizing compilers can gain significant performance benefits by performing code transformations based on a program's runtime profile. Traditionally, profiles are collected by running an instrumented version of the executable. However, because this profiling technique incurs a large overhead, applications are only profiled prior to distribution on a set of sample inputs. Consequently, they cannot be adaptively optimized in order to account for changes in program behavior or to take advantage of variations in the production system. More recently, low-overhead methods of profiling have been developed based on statistical sampling [1] [2] [3] [4]. The basic approach, however, remains the same: profile information for the program's entire execution is averaged into a large database and later fed back into a static compiler. This approach is undesirable

for three reasons. 1) The entire profile of each application must be continuously maintained at runtime on the production system. 2) The profile represents only average behavior across an extended period of time. 3) A significant length of time may pass before variations in the program's behavior are detected. This paper addresses these problems by presenting a new method for rapidly, accurately, and transparently collecting profile information with minimal runtime overhead. The proposed hardware approach provides a strong foundation for a runtime optimizing system.

Although static optimizations are essential to application performance, additional opportunities can be exploited by continuously profiling and reoptimizing the code. For instance, many types of optimizations cannot be performed without more specific runtime information. These include optimizing code based on value invariance [5] [6] and inlining dynamically linked library functions [7]. The traditional, static approach also has the disadvantage that aggressive optimization can only be applied selectively, based on average profile weights and other criteria [8] [9]. Optimizations that cause excessive code growth, for example, must be applied conservatively by the static compiler [10]. However, a runtime optimizer can more aggressively apply these optimizations by targeting small regions of the program that represent the critical execution path at a particular time. To support targeted optimization, our runtime profiler extends the traditional role of profilers and captures temporal information as well as reporting the relative importance of basic blocks. The profiler identifies code that can be optimized quickly compared with amount of execution time that will subsequently be spent in code. Additionally, code that yields a significant short-term benefit is given priority since a runtime optimizer may not have enough memory at its

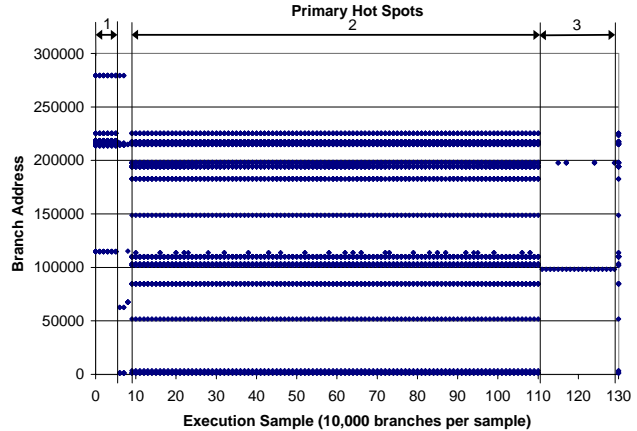
disposal to retain all optimized code indefinitely.

We have observed that many applications exhibit behavior conducive to runtime profiling and optimization. For example, program execution often occurs in distinct phases, where each phase consists of a set of code blocks that are executed with a high degree of temporal locality. When a collection of intensively executed blocks also has a small static footprint, it represents a highly favorable opportunity for runtime optimization. We will refer to such sets of blocks and their corresponding periods of execution as *hot spots*. A runtime optimizer can take advantage of execution phases by isolating a group of hot spots that are active for each phase. Ideally, aggressively optimized code would be deployed early in the phase and the optimized code used until execution shifts to another phase. Optimized hot spots that are no longer active may then be discarded, if necessary, to reclaim memory space for newly optimized code.

Several common types of programs exhibit this execution phase behavior [11], such as compilers, graphics packages, and scripting engines. Because these applications typically implement a wide range of functionality, it is often intractable to aggressively optimize for every possible runtime scenario. Yet, these programs should benefit from runtime optimization because they are divided into focused tasks, or phases of execution. Compilers, for example, allow the user to select a variety of optimizations and other behavior at runtime, and they will often perform multiple passes across the input, exercising different functionality each time. Graphics packages also allow a wide variety of tasks to be selected at runtime, each of which may perform an intensive transformation on the data. Furthermore, it is likely that these transformations can be highly tuned when the data set is available during optimization. A similar situation exists in many scripting engines because a sizeable number of high-level commands are available that perform intensive operations on the data. Yet, for any single script, only a small selection of these routines may be utilized.

A concrete example of this behavior can be seen in 134.perl running the jumble training input from the SPEC95 benchmark suite. As shown in Figure 1, this benchmark contains three primary, distinct phases of execution with one hot spot per phase. Hot spot 1 runs for 72 million instructions, hot spot 2 for 1.35 billion, and hot spot 3 for 200 million. The first hot spot consists of reading in a dictionary and storing it in an internal data structure. The second hot spot processes each word in the dictionary, and the third scrambles a selected set of words in the dictionary.

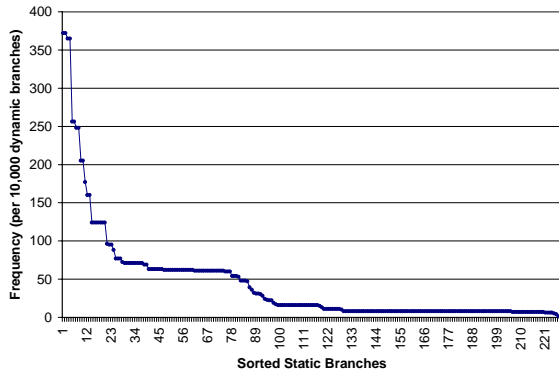
The second of these hot spots serves as an excel-



**Figure 1. Important branches executed in each execution sample for 134.perl. Each data point represents a branch that executed at least 40 times within the sample duration of 10,000 branches. The sample period is 2,000,000 branches.**

lent example of why runtime optimization is needed. The input script causes perl to call *split* which breaks up an input word into individual letters, then to call *sort* which sorts those letters. The first function, *split*, calls a complicated regular expression matcher with an empty regular expression. This region of code would benefit from partial inlining and code layout, followed by classical optimization of the few exercised blocks in the regular expression matcher and split. A static compiler could perform this optimization, but the larger code size and compile time would not benefit most input scripts. The second function, *sort*, calls the library function *qsort* which then calls a perl-specific comparison function. Less than half of the code in the comparison function is ever executed because only single characters are actually sorted. This is another example where inlining is important because of the very frequent calls to the comparison function. However, a link time optimizer or runtime optimizer is needed to support inlining across library and application boundaries. Figure 2 shows a branch profile for a typical 10,000 branch sample of this second hot spot and clearly demonstrates that a small number of static branches account for the vast majority of the dynamic instances in the sample.

Our strategy for detecting runtime optimization opportunities consists of two stages. Initially, our profiler must detect hot spots as they emerge during execution. To do this, we track the execution frequency of code blocks across a relatively short period of time. When a set of blocks fit the characteristics of a hot spot, the profiler notifies the runtime optimizer. During the detection stage, arc weights are also accumulated for the



**Figure 2. Profile distribution for the second primary hot spot in 134.perl. Branches sorted from most to least frequent.**

hot spot to aid profile-driven optimizations. Once a hot spot has been detected, we add the hot spot to a monitor table and disable the profiling hardware. During the second stage, we monitor the program and compare the executing code against the table of currently active hot spots. When execution strays from the monitored hot spots, the profiler is enabled to allow detection of new hot spots.

The rest of this paper is organized as follows: Section 2 explores the mechanisms for performing hot spot detection during runtime; Section 3 details our experimental environment and results; Section 4 examines research related to our endeavors; and Section 5 summarizes our efforts and future directions.

## 2 System Components

Our proposed hot spot detection scheme uses three criteria to classify a region of code as a hot spot. First, the region must have a small static code size to facilitate rapid optimization. Second, the hot spot must be active over a certain minimum time interval so that it is likely to have an opportunity to benefit from runtime optimization. Finally, the instructions in the selected region of code must account for a large majority of the dynamic execution during its active time interval. These three criteria are sufficient to detect code regions that can benefit most from runtime optimization without placing unnecessary restrictions on the type of hot spots that can be identified.

Our proposed scheme must both detect when the execution is in a hot spot and also gather profile data for that hot spot. When a valid hot spot has been discovered, the operating system is notified so that it can invoke the runtime optimizer. Additional hardware is used to ensure that further notifications occur

only when significant new hot spots are detected. Our proposed implementation consists of the following components:

**Hot Spot Detector** Hardware that collects control flow profiles and identifies a collection of important blocks that comprise a hot spot. The Hot Spot Detector contains a Branch Behavior Buffer (BBB) to store the branches and profile data and a Hot Spot Detection Counter (HDC) to track the percentage of dynamic execution accounted for by the hot branches.

**Monitor Table** Hardware that maintains a collection of previously discovered hot spots that the runtime optimizer has already examined. It monitors program execution, noting when execution strays from previously detected hot spots so that profiling can be restarted.

**Operating System Support** Software that reads the hot spot information from the BBB and adds the hot spot blocks to the Monitor Table. It can also assemble the blocks into a region in order to call the runtime optimizer.

The Hot Spot Detector and Monitor Table can easily tolerate a rather large latency when recording information about program execution. Therefore, our proposed hardware can be deeply pipelined and located off the critical path so that it does not affect processor performance. While invoking the operating system does incur a penalty, our experimental data shows that the number of operating system interrupts is insignificant relative to the total execution time. This is because the operating system is typically invoked only when a new optimization opportunity is detected.

The following subsections describe each of the three main components in the context of a single process system. This section concludes with a description of the extensions necessary for a multiprocess system and other enhancements.

### 2.1 Hot Spot Detector

The first step in the process of identifying hot spots is to detect the frequently executed blocks. By employing a hardware scheme, operating system overhead can be eliminated from this portion of the process and detection can be completely transparent to the system. Such blocks can be easily collected in hardware by gathering the branches that define their boundaries. By examining branch execution and direction information, a control flow graph with arc weights can be constructed. Through this hardware scheme, reasonably accurate

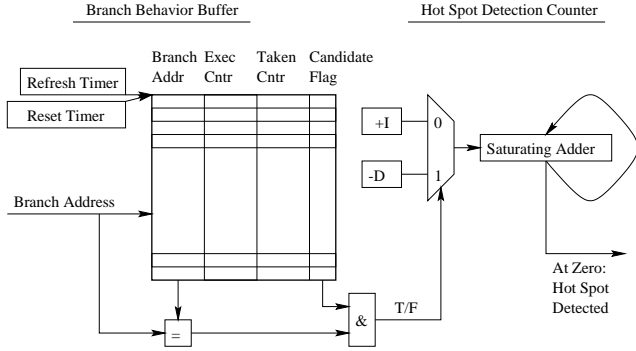


Figure 3. Hot Spot Detector hardware.

profile information will be collected which can benefit the runtime optimization effort.

### 2.1.1 Branch Behavior Buffer

To achieve these collection goals in hardware, a cache structure reminiscent of a *Branch Target Buffer* (BTB) will be used. As depicted in Figure 3, the structure, called a *Branch Behavior Buffer* (BBB), will be indexed on branch address and will contain several fields: tag (or branch address), branch execution count, branch taken count, and branch candidate flag. When the processor retires a branch instruction, the instruction address will provide the index into the BBB. Each time a branch address is found in the BBB, its execution counter is incremented. If the branch is taken, the taken counter is also incremented. The combination of these two values constitutes an arc weight profile of the recently executed code. Note, it is possible for a branch to execute more times than can be represented by the execution counter. When the execution counter reaches its maximum value, the hardware stops incrementing both the execution and taken counters to preserve their ratio. As long as the number of branches that reach saturation is small, the profile will still accurately represent the relative importance of the branches.

The purpose of the BBB is to collect and profile only frequently executed branches whose corresponding blocks account for a vast majority of the dynamically executing instructions. Branches that execute frequently during profiling may be part of a potential hot spot, and we refer to such branches as *candidate branches*. In order to make this determination, an executed branch is temporarily allocated an entry in the BBB and monitored over a short interval. If the execution of that branch is frequent enough during that interval, its execution counter will surpass a predefined *candidate threshold*. The candidate threshold is associated with a single bit in the execution counter, as shown in Figure 4. When this bit is triggered, the can-

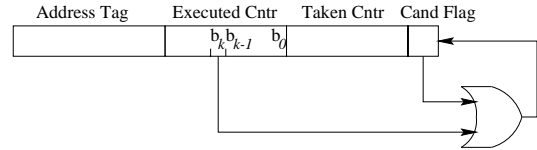


Figure 4. Fields in each Branch Behavior Buffer entry.

didate flag is set, and the entry remains marked as a candidate while the execution counter continues to collect profile data. Thus, the BBB only needs to be large enough to hold the candidate branches and potential candidate branches for a given interval.

To ensure that only frequently executing branches are marked as candidates, a *refresh timer* periodically clears entries from the BBB that have not surpassed the candidate threshold. The refresh timer is simply a global counter that increments each time a branch instruction is executed and triggers a refresh of the BBB when it reaches a certain value. Refreshing the BBB flushes the insignificant entries and ensures that each branch marked as a candidate accounts for at least a minimum percentage of the total dynamic branches during a fixed interval. The minimum percentage of execution required of candidate branches can be expressed as a *candidate ratio*. Thus,

$$CANDIDATE\_RATIO = 2^k / 2^n, \quad (1)$$

given that the size of the refresh timer is  $n$  bits, and the candidate threshold is represented by bit  $b_k$  of the execution counter.

To accurately represent a hot spot, the BBB must be able to allocate entries for most of the branches that are important in the hot spot. The BBB size should be equal to or larger than the total number of branches present within a hot spot. If the BBB is too small, then insignificant branches may prevent important branches from entering during the first few refresh intervals. Statistically, the important branches will eventually get entries in subsequent refresh intervals, but the profile accuracy may be somewhat compromised and the reporting of hot spots may be delayed.

Another problem is the possibility of indexing conflicts that are inherent in any cache structure. As has been seen in the BTB or I-cache, making the structures set associative eliminates many of the indexing conflicts.

Although organized like a set associative cache, the BBB's behavior differs in terms of replacement policy. The BTB and I-cache often use a least recently used (LRU) policy in order to reduce cache misses. For example, suppose X and Y are instructions that map to the same I-cache location, and X is currently in the

cache. Y is accessed three times followed by a long stream of X's. If X is not replaced in the cache by Y, then three I-cache misses will result, one for each access to Y. An LRU policy predicts that the next access will be like the previous one and will replace X with Y in order to save a miss. After the third Y, X will again be accessed, resulting in another miss. In this case, the LRU policy was able to reduce three misses down to two but the replacement of X was required.

The BBB, however, must accurately record the most frequently executed blocks and their profiles rather than the most recently accessed. Therefore, implementing an LRU replacement policy and allowing a rare block to replace a frequently executed block is unacceptable. Instead, branches that conflict with existing BBB entries are simply discarded. The function of branch replacement is controlled by the aforementioned refresh timer. A more serious conflict occurs when two branches are important and map into the same cache location. As long as these conflicts are relatively rare, the runtime optimizer can still recover by inferring the profile value of the missing branches. Using Kirchhoff's First Law and other heuristics [12], the input and output weights of the block can be estimated based on the branches that were accurately profiled.

While the I-cache and BTB collect information about important instructions and branches, respectively, it is infeasible to combine the BBB with either of these two structures. In addition to the previously discussed replacement policy differences, the BTB and I-cache are both potentially on the critical path, and adding complexity in these two structures may adversely affect cycle time. In addition, since only true branch behavior should affect the BBB counters, incorrectly speculated accesses must not be allowed to affect the BBB entries. Although the BBB counters in these structures could be updated after the branches are resolved, the strict timing constraints and complex hardware required to interface with the BTB and I-cache warrant the cost of storing the BBB entries separately. Thus, the BBB is best implemented as a stand-alone structure in the retirement phase because of its simplicity and inherent accuracy.

### 2.1.2 Hot Spot Detection Counter

Once candidate branches have been identified in the Branch Behavior Buffer, they must be monitored to determine whether the corresponding blocks may be considered a hot spot and, thus, useful candidates for optimization. We have found two criteria that should be satisfied before a group of candidate blocks is dubbed a hot spot. First, the candidate blocks should be active

for a specified minimum amount of time. Second, the candidate branches should account for at least a certain percentage of the total branches executed during this time. We define the minimum percentage over the time interval to be the *threshold execution percentage* and the actual dynamic percentage over the interval to be the *candidate execution percentage*.

In order to minimize disruption of the system during hot spot detection, we perform the detection in hardware using a *Hot Spot Detection Counter* (HDC), shown in Figure 3. The Hot Spot Detection Counter is an up/down counter used to detect when the set of candidate branches reaches the threshold execution percentage. The counter is implemented as a saturating adder that is initialized to the maximum value. It counts down by  $D$  for each candidate branch executed or counts up by  $I$  for each non-candidate branch executed, where the determination of  $D$  and  $I$  will be discussed later. When the candidate execution percentage exceeds the threshold percentage, the counter begins to move down. If the candidate execution percentage remains higher than the threshold for a long enough period of time, the counter will decrement to zero. At this point, the operating system will be triggered either via an interrupt or by setting a flag that is checked the next time the operating system is invoked.

The difference between the candidate execution percentage and the threshold execution percentage determines the rate at which the counter decrements (i.e., the rate at which the hardware identifies the hot spot). This corresponds to our observation that hot spots become more desirable as they either account for a larger percentage of total execution or run for a longer period of time. It is assumed that hot spots which have been active over a longer period of time are less likely to be spurious in their execution and are more likely to continue to run after optimization has been complete.

Our experiments have shown this approach to work quite well and to detect all of the major hot spots in our benchmarks. There are three primary scenarios where there is no hot spot to be found, and thus the HDC will never reach zero:

1. Few branches execute with sufficient frequency to be marked as candidates, and collectively, they do not constitute a large percentage of the total execution. Thus, even if they were classified as a hot spot and optimized, only a small benefit is likely.
2. The number of branches that execute frequently enough to be considered candidates is too large to fit into the BBB. If the branches that are able to enter the BBB do not account for a large enough

percentage of execution, they will not be identified as a hot spot. This may happen if the execution profile of the region is very flat. Although some benefit may be gained by optimizing all the frequent branches, the overhead of optimizing such a large region would most likely be prohibitive.

3. The execution profile is not consistent. In this case, a small set of branches may account for a large percentage of execution over a short time, but the execution shifts to a different region of code before the Hot Spot Detection Counter saturates. Optimizing a region of code that only executes spuriously is unlikely to yield much benefit.

As in these three scenarios, branches that collectively do not constitute a hot spot may be locked into the BBB. Since the HDC has not identified these branches as a hot spot, it is possible that the execution has shifted to a new, potentially more important region of code or is not an attractive optimization opportunity. Therefore the BBB will be periodically purged by the *reset timer* to make room for new branches. This timer is similar to the refresh timer BBB but clears all entries in the table, rather than only the non-candidate branches. The reset interval should be large enough to allow the HDC to saturate on valid hot spots but small enough to allow quick identification of a new phase of execution.

Once the threshold execution percentage  $X_t$  required for a hot spot has been selected, the HDC increment and decrement values should be chosen.  $D$  is the decrement value when a candidate branch is encountered (*candidate hit*), and  $I$  is the increment value for a branch that is not in the table or is not yet marked as a candidate (*candidate miss*). Let  $X$  be the actual candidate execution percentage. For a given  $D$  and  $I$ , the counter will decrease when the candidate execution percentage multiplied by the decrement value is greater than the percentage of non-candidates multiplied by the increment value. This is represented by the equation:

$$X * (-D) + (1 - X) * (I) \leq 0 \quad (2)$$

Rearranging the terms and solving for  $X$  yields the formula for minimum percentage:

$$X \geq \frac{I}{D + I} = X_t \quad (3)$$

Equation 3 shows that the counter decreases when the percentage of execution is above the threshold, as determined by  $I$  and  $D$ .

Given the increment and decrement values, the size of the HDC can be chosen to achieve a minimum detection latency. Let  $N$  be the minimum number of branches executed before a hot spot is detected. For detection to occur, the following inequality must hold:

$$N * X * (-D) + N * (1 - X) * (I) \leq -HDC\_MAX\_VAL \quad (4)$$

The latency for detecting a hot spot is determined by the following equation:

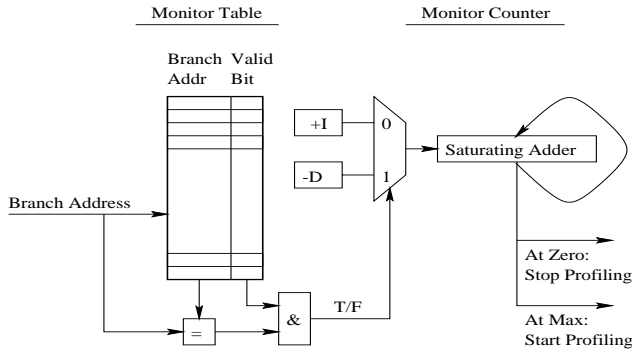
$$N = \frac{HDC\_MAX\_VAL}{(D + I) * (X - X_t)} \quad (5)$$

As the candidate execution percentage further surpasses the threshold, the detection latency decreases. The latency can also be decreased independently of the candidate execution percentage by increasing  $I$  and  $D$  such that  $X_t$  remains constant.

## 2.2 Monitor Table

The purpose of the *Monitor Table* is to determine when hot spot profiling is necessary. This hardware mechanism is continuously running, watching program execution and comparing the executing branches to those already determined to be in hot spots. When the program is executing in the known hot spots, the system is said to be in *monitor mode*, which is the steady-state mode of execution. The system enters *profile mode*, and the BBB is enabled when the Monitor Table determines that execution has strayed from the known set of hot spots. Note that the Monitor Table continues to operate during profile mode, watching for execution to return to the set of known hot spots. If this situation were to occur, the BBB would be deactivated, since it is unnecessary and costly to extract and possibly optimize a hot spot that has already been processed. If a new hot spot is found, the operating system is notified which then extracts the hot spot from the BBB; the new hot spot is entered into the Monitor Table; and the BBB is deactivated. When new, optimized code is ready for deployment, the branches in the new code will be added to the table just prior to deployment to avoid hot spot detection in the optimized code.

In order for the system to detect when execution strays from the known set of hot spots, the hardware must be aware of those hot spots which have already been identified. In an ideal Monitor Table, the addresses of all branches in all known hot spots would be placed into a tag array as shown in Figure 5. When a branch is executed, the instruction address is looked up in the tag array. Its presence in the array indicates execution in a hot spot. An up/down counter called the *Monitor Counter* is used to track long-term execution trends and operates much like the HDC. It counts



**Figure 5. Monitor Table hardware.**

down when a hot spot branch is executed and counts up when a non-hot spot branch is executed. When the Monitor Counter saturates at the maximum value in monitor mode, a high percentage of recent branches outside of the known hot spots have been executed, indicating a possible transition to a new hot spot. At this time, profile mode is resumed. Similarly, when the system is in profile mode and the Monitor Counter reaches zero, program execution must have returned to the set of known hot spots. At this time, the BBB is deactivated, and monitor mode is resumed. When monitor mode is entered from profile mode, the Monitor Counter is initialized to zero, indicating that execution is in a hot spot. This is a valid assumption because either the Monitor Counter just saturated at zero to end profile mode, or a hot spot was just detected and execution is likely to continue in the hot spot.

As in the HDC, the increment and decrement values for the Monitor Counter determine the threshold ratio of hot spot to non-hot spot branches. Although a minimum ratio of hot spot to non-hot spot branches must be maintained to remain in monitor mode, this ratio should not be as high as in the HDC. A lower ratio is used for the monitor hardware to allow the behavior of the hot spots to vary slightly without reentering profile mode. Once this ratio is determined, the same formula used for the HDC can be used to derive suitable increment and decrement values for the Monitor Counter.

### 2.3 Operating System Support

The interface between the BBB and the operating system was designed to be simple: the BBB signals the OS only when a hot spot has been detected, and the operating system copies the BBB table contents into system memory upon hot spot detection. This interface minimizes operating system involvement with hot spot detection. Specifically, cycles are not stolen away by the OS from the user application unless a hot spot is

actually detected.

In order for the OS to read the BBB contents, the BBB must be either hardware addressable or readable through special instructions. Once extracted, the branches are stored internally for access by the runtime optimizer. They are also processed and written to the Monitor Table, again via hardware addressing or via special instructions. Since the table is a cache, conflicts over entries in the Monitor Table can arise. In this situation, the OS must either remove an old hot spot that contains a conflicting branch, thus accurately preserving a smaller set of hot spots, or simply eliminate one of the conflicting instructions from the table, thus slightly compromising the recognition of one of the known hot spots. Both of these situations are suboptimal because code previously detected as being part of a hot spot may now trigger the profiling system. The OS is also responsible for deploying optimized code and entering it into the Monitor Table. Furthermore, the operating system has control over several system parameters (BBB increment and decrement values, etc.) and can adjust them based on quality of the hot spots being collected. The actual operating system policies are beyond the scope of this paper.

### 2.4 Multiprocess support

Thus far, the hardware design has assumed single process execution. Operation becomes slightly more complicated when considering the context switching abilities of microprocessors. It is the responsibility of the operating system to correctly maintain the state of the proposed hardware in a multiprocess environment as it is required to do for traditional hardware components.

Because of the expense of swapping out even a subset of the BBB during context switches, the hot spot detection hardware is designed to operate in single process mode, persisting across context switches. Because the BBB responds to hot spots quickly, it is only active during a small percentage of an application's execution. Because of this low utilization, a single BBB can be shared among multiple processes. The BBB could be configured to search for hot spots associated with a particular process ID, thread ID, or code segment. This ID would be stored in a control register.

Unlike the BBB, the Monitor Table is always in use by each process. Since swapping a table in and out at each context switch would be extremely costly, all processes could share a single table. In order to accomplish this, an up/down Monitor Counter is necessary for each process or active subset to effectively track hot spot behavior. Furthermore, each entry in the Monitor Table must also be tagged with its process ID. This ID serves

as a tag for comparison purposes when determining a hit or miss and for determining which Monitor Counter to update.

When a Monitor Counter saturates indicating that profiling is necessary, the BBB must first be allocated to that particular process. A simple check of the BBB process ID control register can be made to determine BBB ownership. If the requesting process owns the BBB, profiling can continue without delay. Otherwise, arbitration must occur between all of the processes requesting use of the BBB. This arbitration may be implemented in the hardware itself or within the OS. Although processes may be denied use of the BBB for a short time, the BBB may be acquired by the next waiting process as soon as the BBB resets or the HDC saturates.

## 2.5 Enhancements to the base hardware

Several enhancements might be made to the base hardware. The first enhancement would be to reduce the size of the BBB by considering only conditional and indirect branches. The execution and taken profile weights of blocks with unconditional branches or direct calls can be determined by inference via Kirchhoff’s First Law and need not be profiled. However, this approach requires that the runtime optimizer spend more time analyzing and constructing the control flow graph.

A second enhancement would be to index into the BBB with a combination of the branch’s address and its direction. Thus, the taken and not taken paths of a particular branch would be recorded in separated entries allowing for the elimination of the taken counter in each BBB entry. While this approach would result in either an increase in the size of the BBB or a reduction in the number of distinct branch addresses in the BBB, it would allow the system to detect finer changes in program, and hot spot, behavior. For example, control flow changes within a particular set of blocks would now be detected and possibly reoptimized.

A third enhancement would be to include support for profiling the arc weights of indirect branches. Currently, profiling determines only the branch weights of these branches. An additional table indexed on a combination of the branch and target addresses could be used to store the actual profile. The BBB entry for that branch would still gather the branch execution count and determine branch candidacy.

A fourth enhancement would be to add a secondary, coarse-grained component to the Monitor Table. This secondary table would track ranges of addresses rather than single branches. Each range table entry would consist of a base address and size, and any address falling inside the range would be considered part of a

Benchmark	Num. Insts.	Actions Traced
099.go	89.5M	2stone9.in training input
124.m88ksim	120M	clt.in training input
126.gcc	1.18B	amptjp.i training input
129.compress	2.88B	test.in training input <i>count</i> enlarged to 800k
130.li	151M	train.lsp training input (6 queens)
132.jpeg	1.56B	vigo.ppm training input
134.perl	2.34B	jumble.pl training input
147.vortex	2.20B	vortex.in training input
MSWord(A)	325M	opened 16.0 MB .doc file, searched for number, then closed
MSWord(B)	912M	loaded 25 page .doc file, repaginated ran built-in word count, selected entire document, changed font to Arial, undo change midway through, closed file
MSEXcel	160M	generated silicon diffusion data from VB script & graphs from data
Adobe Photo-Deluxe(A)	390M	loaded detailed tiff image, brightened, increased contrast, and saved as PhotoDeluxe image
Adobe Photo-Deluxe(B)	108M	exported detailed tiff image to encapsulated postscript
Ghostview	3.17B	loaded ghostview, loaded 9 page ps file, viewed and zoomed in on 4 pages, performed text extract

**Table 1. Benchmarks used for hot spot detection experiments.**

hot spot. Since a hot spot would be returned from the optimizer as a single, contiguous region with a larger code size due to aggressive optimizations, the region could be added to the range table and thus help avoid conflicts among entries in the fine-grained table. Clearly, this enhancement would require a higher degree of operating system support to manage.

## 3 Experimental Evaluation

Trace-driven simulations were performed for a number of benchmarks in order to explore hot spot characteristics and to establish the effectiveness of the proposed hot spot detection scheme. Both *SPECINT95* and common *WindowsNT* applications were simulated to provide a broad spectrum of typical programs. These benchmarks are summarized in Table 1. The eight applications from the SPECINT95 benchmark suite were compiled from source code using the *Microsoft VC++ 6.0* compiler with the *optimize for speed* and *inline where suitable* settings. Several WindowsNT applications executing a variety of tasks were also simulated. These applications are the general distribution versions, and thus were compiled by their respective software vendors. In order to ensure examination of all executed user instructions, sampling was not used

Parameter	Setting
Num BBB entries	2048
BBB associativity	2-way
Exec and taken cntr size	9 bits
Candidate branch thresh	16
Refresh timer interval	4096 branches
Clear timer interval	65535 branches
Global cand cntr size	13 bits
Global cand cntr inc	2
Global cand cntr dec	1
Global mon cntr size	12 bits
Global mon cntr inc	1
Global mon cntr dec	1

**Table 2. Hardware parameter settings.**

during trace acquisition or simulation.

The experiments were performed using the inputs shown in Table 1. The hot spot detection hardware was then simulated on an instruction-by-instruction basis for the entire execution. In order to extract complete execution traces of these applications (all user code, including statically- and dynamically-linked libraries), we employed special hardware capable of capturing dynamic instruction traces on an AMD K6 platform. This unit, known as SpeedTracer, was donated by AMD for our research.

### 3.1 Hardware Parameters

Because the design space is large, experimentally evaluating the individual effect of each hardware parameter was infeasible. We, therefore, selected initial parameters that attempted to match the observed hot spot behavior and then further refined them, resulting in parameters that exhibit desirable hot spot collection behavior. These parameters were used in the experiments presented in this section and are shown in Table 2. The BBB hardware was configured to allow branches with a dynamic execution percentage of .4% (16 executions/4096 branches) or higher to become candidates (the candidate ratio). The HDC was configured to require that the candidate branches total more than 66% (2:1) of the execution to become a hot spot (the threshold execution percentage). The BBB was allowed 16 refreshes (totaling 65535 branches) to detect a hot spot before it was reset. To collect these results, we used a Monitor Table large enough to contain all hot spot branches in order to examine the hot spot behavior over the course of the whole execution.

### 3.2 Results

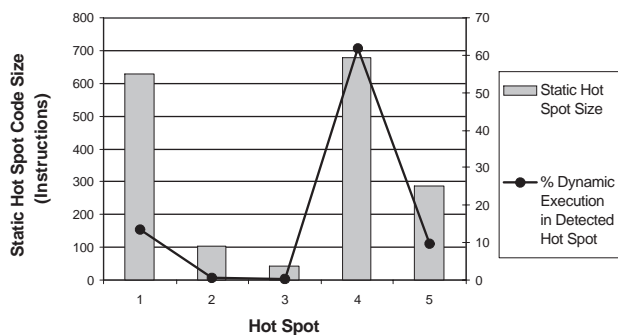
Table 3 summarizes the effectiveness of our proposed hardware at detecting runtime optimization opportu-

nities for each benchmark. The *number of hot spots* column lists the number of times that the HDC saturated at zero, indicating the detection of a new hot spot. This number also represents the number of times that the operating system was interrupted by the BBB. The *number of static instructions in hot spots* is a close estimate of the total number of instructions that will be delivered to the optimizer over the entire execution. The next column, *percent static executed instructions in hot spots*, relates the number of static instructions in hot spots to the total number of static instructions executed. In other words, out of all the static instructions executed by the microprocessor, only a small percentage lie within hot spots. The portion of total dynamic instructions represented by these hot spots is shown in the next column, *percent total execution in hot spots*. Because this hardware cannot detect hot spots instantly, some time that could be spent executing in optimized hot spots is spent during detection. The time spent in hot spots after they are detected is shown in the *percent total execution in detected hot spots*, and the time lost to detection can be found by taking the difference between this column and the previous column. Finally, the last column, *dynamic instructions in hot spots after detection*, shows the number of dynamic instructions that could benefit from runtime optimization. This number reflects any subsequent reuses of detected hot spots. For the purposes of our experimentation, when an instruction is part of several hot spots, we heuristically attribute the instruction to the most recently accessed of the relevant hot spots.

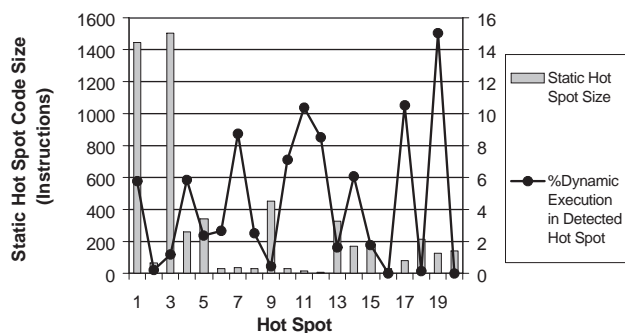
Analysis of the results shows that only a small percentage, usually less than 3%, of the static code seen by the microprocessor executes intensively enough to become hot spots. Since a large percentage of the dynamic execution is represented by a small set of instructions, often nearly 90% of the program’s execution, a runtime optimizer could easily focus on this small set with the potential for significant performance increase. In addition, only about 1% of the possible time spent in optimized hot spots is lost due to detection. For example, in 130.li, the number of hot spot static instructions comprise only 3.00% of the total static instructions, yielding a total hot spot code size of 1447 instructions. Furthermore, 90.88% of the entire execution is spent in detected hot spots. Our analysis shows that ideally the hot spots account for 91.28%, and, thus, only .40% is lost during the detection process. This indicates that our Hot Spot Detector makes the identification so swiftly that the execution of hot spot regions falls almost entirely within potentially runtime optimized code.

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.jpeg	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

**Table 3. Summary of the hot spots found in the benchmarks.**



(a) 134.perl.



(b) PhotoDeluxe(A).

**Figure 6. Detailed hot spot statistics.**

Examining individual hot spots reveals interesting characteristics of program behavior. Figure 6(a) details the detected hot spots from the 134.perl benchmark. For each hot spot, the bar graph shows the static

code size of the hot spot, while the line graph shows the percentage of the execution spent in that hot spot after detection. This benchmark consists of three primary hot spots: hot spots 1, 4, and 5 on the graph. These correspond to the three hot spots in Figure 1 in the introduction (note that in the histogram, 134.perl was compiled for the *IMPACT* architecture without inlining). From the histogram, code can be seen executing between the first and second primary hot spots. Hand analysis did not classify code in that region to be hot spots, but the hardware did as actual hot spots 2 and 3. Upon further hand examination of actual hot spot 3, an intensely executed region of code is found, thus verifying the hardware’s determination. In hot spot 3, the *cmd\_exec* function loops 117k times calling *str\_free* in each iteration. The 9 blocks, totaling 43 static instructions, contribute 7.3M dynamic instructions to the program’s total execution. Because of the intense nature of these blocks, they serve as good candidates for runtime optimization. Analysis of this benchmark also shows that one hot spot is much more dominant than the others in terms of dynamic execution. In this case, optimizing only hot spot 4 could benefit over 58% of the dynamic instructions executed.

Similar characteristics were observed in the other benchmarks. Figure 6(b) shows an example from one of the pre-compiled WindowsNT applications. For this benchmark, there are a few hot spots that each represent 8% or more of the total execution which together represent more than 50%. We also see quite a few hot spots with small static code sizes, indicating tight, in-

tensely executed code. In fact, for this benchmark, the smaller-sized hot spots are also those with high total execution percentages, indicating excellent opportunities for runtime optimization.

The benchmark `099.go` is a notable example of a benchmark without obvious hot spots. While this game simulation repetitively executes players' moves, each move touches a large amount of static code with little temporal repetition. The hardware was still able to detect six hot spots representing 35% of the execution. There is one primary hot spot that represents 28% of the execution with a static code size of 1170 instructions.

Our data has shown that the static sizes of the detected hot spots vary significantly, from tens of instructions to the low thousands. At this point in the development of runtime optimization technology, the maximum static code size that a runtime optimizer could handle is an open question. The goal of our work is to quickly and accurately identify the hot spots, without placing too many artificial restrictions on the characteristics of the hot spots detected. We believe that it is better to submit a larger region of code to the runtime optimizer and rely on the optimizer to pare down the code size using all available information that has been gathered about the hot spot. However, by adjusting the existing hardware parameters it is easy to limit the size of the hot spots detected.

The dynamic execution lengths of the individual hot spots also vary significantly. In some cases, the execution lengths of the hot spots are fairly small but still account for hundreds of thousands of instructions. These lengths may be too short to benefit from runtime optimization, especially when the performance gains earned from the optimized code must offset the cost of detection and reoptimization. Unfortunately, the total execution time is not known at runtime. The operating system will have to make further decisions about which hot spots to actually optimize.

### 3.3 Implementation Cost

For our experiments we used a BBB large enough to guarantee accurate results for all of our benchmarks. With 2048 entries available, very few cache conflicts occurred, and 9-bit execution and taken counters were sufficient to capture profile data for most hot spots. For an equivalent hardware implementation, each BBB entry would contain a 22-bit tag field, two 9-bit counters, and a one-bit candidate flag. Thus, the total BBB hardware cost is slightly over 10 kilobytes. Compared to AMD's K7, which is reported to have a 128KB on-chip L1 cache, this size seems justifiable. Although a variety of hardware schemes may be used to implement

the Monitor Table, it is likely to require less hardware than the BBB. A Monitor Table (without a coarse-grained component) that accommodates 3000 branch entries would achieve ideal results for our benchmarks. The Monitor Table, however, does not require profile counters, and, furthermore, a clever implementation may use a single entry to represent several nearby branches.

Next, we considered the cycles lost to reporting the hot spots to the OS. These cycles offset some of the gains made by executing in optimized code. The operating system is only interrupted when a hot spot is reported by the BBB. A rough estimate can be made about the time required to transfer the contents of the BBB to operating system memory. If we conservatively suppose that each of the BBB's 2048 41-bit entries requires three cycles to transfer, then the net cost per hot spot is 6144 cycles. Even for `126.gcc` with 47 hot spots, the net cost is 288K cycles, which is negligible when compared to the number of instructions (617M) spent executing within the hot spots.

## 4 Related work

Several strategies have been proposed to collect profile data, and ideas from those implementations have motivated our efforts to create the Branch Behavior Buffer. Other recent efforts, however, have focused on estimating average program behavior through statistical sampling. Consequently these techniques rely on heuristics used during static analysis to discover important regions of code. To our knowledge only our method addresses the unique requirements of runtime optimization.

Conte, Menezes, and Hirsch [13] also propose the use of dedicated hardware for profiling. Their profile buffer is in the retirement stage and interacts with the reorder buffer to determine whether a branch has been taken. The operating system periodically samples the profile buffer, gradually constructing an arc-based profile for use in static compiler optimizations. Various techniques are explored to reduce contention among branches within the profile buffer, and similar schemes could be used to lower the hardware cost of our Branch Behavior Buffer.

The *ProfileMe* approach taken by Dean et al [14] collects detailed information about a single sampled instruction and the interaction between a pair of sampled concurrent instructions. They capture interesting events such as cache misses and branch mispredictions so that an in-depth performance analysis of the microarchitecture can be performed offline.

The Morph system, developed by Zhang et al [3] uses

operating system and compiler technology to provide a framework for continuous profile-driven optimization. Although our proposed hardware greatly reduces the demand for software support, a low-overhead OS layer is still necessary. Because Morph demonstrates the feasibility of such an OS infrastructure, it is orthogonal to our hot spot detection mechanism.

Although not the focus of this paper, we envision utilizing our binary reoptimization technology and implementing region-based runtime optimizations [15] using our hot spot profile data. Our isolation of program hot spots also lends itself to the hot-cold optimization strategy proposed by Cohn and Lowney [16]. By excluding the infrequently used, or cold, instructions and concentrating on the intensely used, or hot, instructions in a hot spot, we can potentially derive significant performance benefit.

## 5 Conclusion

Traditional profiling focuses on collecting data to be used for compiler optimizations in a compile-run-recompile methodology. This approach restricts aggressive optimizations to a specific workload that is assumed to be representative. We have presented a method that dynamically identifies hot spots within a program and collects sufficient data to provide a framework for timely runtime optimization during the same execution. Our scheme requires minimal operating system support, incurs negligible runtime overhead, and requires a few hardware tables and counters located off of the critical execution path to identify and monitor the hot spots.

Detailed trace-driven simulations were performed for a number of integer programs, including benchmarks from SPEC95 and several WindowsNT applications. The resulting data presented in this paper showed that the hot spot detection algorithm, when applied to many common benchmarks and applications, achieves a high rate of success and provides a promising framework for building a runtime optimizer for general purpose programs. Subsequent efforts will be directed toward implementing several region-based runtime optimizations using the hot spot data and further exploring methods of delineating hot spots in the code.

## 6 Acknowledgments

The authors would like to thank all the members of the IMPACT team for their valuable comments, along with Microsoft for the donation of software tools. This research has been supported by Advanced Micro Devices under the direction of Dr. Dave Christie.

## References

- [1] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?," in *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pp. 1–14, October 1997.
- [2] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, pp. 187–206, April 1996.
- [3] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pp. 15–26, October 1997.
- [4] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM SIGPLAN Notices*, vol. 32, pp. 85–96, June 1997.
- [5] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 259–269, December 1997.
- [6] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, "Fast, effective dynamic compilation," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, vol. 31, pp. 149–159, June 1996.
- [7] T. Kistler, "Dynamic runtime optimization," in *Proceedings of the Joint Modular Languages Conference*, pp. 53–66, 1997.
- [8] B. L. Deitrich, *Static Program Analysis to Enhance Profile Independence in Instruction-Level Parallelism Compilation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.
- [9] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.
- [10] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Transactions on Computers*, vol. 42, pp. 1045–1057, September 1993.
- [11] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad, "Execution characteristics of desktop applications on windows nt," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 27–38, June 1998.
- [12] D. Wall, "Predicting program behavior using real or estimated profiles," Tech. Rep. TN-18, Digital Equipment Corporation WRL Technical Note, Palo Alto, CA, December 1990.
- [13] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 36–45, December 1996.
- [14] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: Hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 292–302, December 1997.
- [15] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 158–168, December 1995.
- [16] R. Cohn and P. G. Lowney, "Hot cold optimization of large windows/nt applications," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 80–89, December 1996.