# An Architectural Framework for Run-Time Optimization

Matthew C. Merten, Andrew R. Trick, Ronald D. Barnes
Erik M. Nystrom, Christopher N. George,
John C. Gyllenhaal, Wen-mei W. Hwu

Center for Reliable and High-Performance Computing

1308 West Main Street, MC-228, Urbana, IL 61801

{merten,atrick,rdbarnes,nystrom,c-george,gyllen,hwu}@crhc.uiuc.edu

**Abstract**

Wide-issue processors continue to achieve higher performance by exploiting greater instruction-level parallelism. Dynamic techniques such as out-of-order execution and hardware speculation have proven effective at increasing instruction throughput. Run-time optimization promises to provide an even higher level of performance by adaptively applying aggressive code transformations on a larger scope. This paper presents a new hardware mechanism for generating and deploying run-time optimized code. The mechanism can be viewed as a filtering system, that resides in the retirement stage of the processor pipeline, accepts an instruction execution stream as input, and produces instruction profiles and sets of linked, optimized traces as output. The code deployment mechanism uses an extension to the branch prediction mechanism to migrate execution into the new code without modifying the original code. These new components do not add delay to the execution of the program except during short bursts of reoptimization. This technique provides a strong platform for run-time optimization because the hot execution regions are extracted, optimized, and written to main memory for execution and because these regions persist across context switches. The current design of the framework supports a suite of optimizations including partial function inlining (even into shared libraries), code straightening optimizations, loop unrolling, and peephole optimizations.

## 1 Introduction

The development of out-of-order execution and automatic dynamic speculation has led to dramatic improvements in the performance of modern microprocessors. These techniques were the first steps toward allowing the microprocessor *itself* to determine how to execute code efficiently. Thus far, such hardware optimizations have been limited in scope and have typically been made on the fly without any persistent record. More aggressive and persistent optimizations have instead been accomplished in software through the use of optimizing compilers.

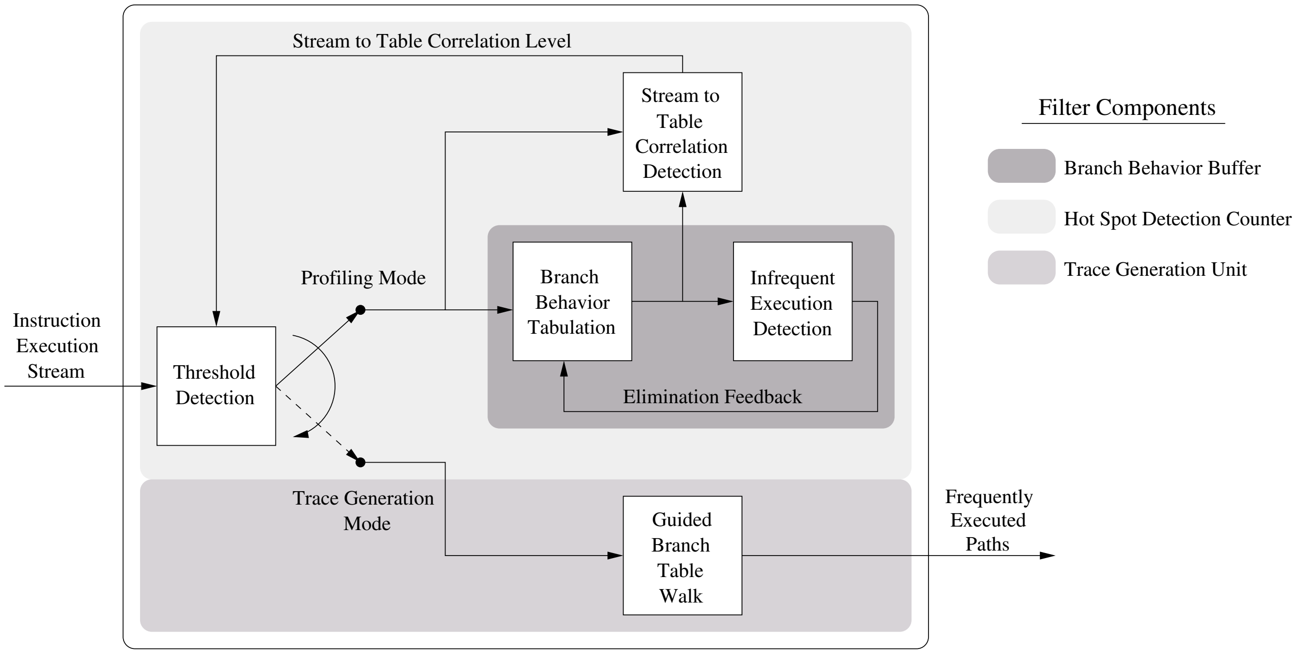



Figure 1: Hot Spot Detector and Trace Generation Unit depicted as an on-line filter.

However, many of these optimizations rely on accurate profile information to profitably transform code. While compilers often support the use of profile information, software vendors have been reluctant to add the profiling step to their development cycles. Not only is determining a representative profile difficult, but when being profiled, the behavior of certain programs may change. For these reasons, an automatic, transparent mechanism for profiling and reoptimizing code based on current usage would be advantageous. Furthermore, a dynamic system could improve performance in ways that a static optimizer cannot. As program behavior changes over time, a dynamic system could reoptimize code to take advantage of temporal relationships. While a typical static compiler optimizes for the average behavior across the entire execution, a dynamic system could lead to more directed optimizations. In addition, since optimization is performed on the same machine that runs the application, the optimizer can specifically target that system.

Hardware support for dynamic profiling and reoptimization reduces the reliance on software systems to perform these tasks. The use of dedicated, background hardware mechanisms allows for better limited run-time overhead. Our hardware system is designed to automatically and transparently profile applications while generating sets of traces that cover the frequently executed paths of an application. These traces can then be optimized and executed in place of the original blocks to improve application performance. Located off the back-end of the processor pipeline, the proposed system incurs negligible overhead and minimally impacts the design of the processor pipeline.

The mechanism itself can be viewed as a filtering system that accepts an instruction execution stream as input and produces instruction profiles and sets of traces as output. As shown in Figure 1, the filter consists of three primary components responsible for: collecting profile information, determining the execution coverage of the profiled instructions, and generating traces for the frequently executed paths.

During application execution, the *Branch Behavior Buffer* component determines the most frequently executed branch instructions while collecting a profile of their behavior. *The Infrequent Branch Detection* process eliminates infrequently executed branches from the table to filter out those that only spuriously execute. Meanwhile, the *Hot Spot Detection Counter* component is responsible for determining when execution is primarily confined to the collected branches. The *Stream to Table Correlation Detection* process compares the execution stream to entries in the table and forces a transition to *Trace Generation Mode* from *Profile Mode* when a comparison threshold is met. The set of branches in the table when the transition occurs serves as a skeleton of the important region of code, and is called a program *hot spot*. Because the detection process is performed very quickly at run time, a unique opportunity exists to optimize the currently executing code while leaving sufficient time to gain benefit from execution in the newly optimized code. Program hot spots provide an excellent opportunity because they can be quickly identified and contain only truly important code. Section 3 provides evidence that programs often exhibit phased hot spot execution behavior. The Branch Behavior Buffer and Hot Spot Detection Counter components are collectively referred to as the *Hot Spot Detector* [1], and are more thoroughly discussed in Section 5.

Using the profiles gathered by the Hot Spot Detector, a hardware component called the *Trace Generation Unit* [2] produces sets of traces that cover the frequently executed paths in the code. The *Guided Branch Table Walk* process utilizes the execution stream to walk the hot spot skeleton stored in the branch table to produce the traces. The use of the stream to step through the table entries eliminates the need to store individual instructions during the monitor phase and prevents entries in the table that were short-lived and no longer active from being included. Meanwhile, the use of the skeleton ensures that the traces are representative of the frequently executed instructions. The instructions in the output traces are generated in execution order, thus providing an inherent code-straightening optimization. The traces are linked together by their exit branches to provide continuous execution within the optimized hot spot. The output of the detector can be used to feed and direct a hardware-based reoptimization mechanism, or can serve as a profiling platform for a software-based reoptimizer. One direct application of this mechanism is to perform code optimizations that improve instruction fetch performance such as loop unrolling, partial function inlining, and branch promotion. The output traces are stored in a memory-based code cache, thus enabling a traditional instruction cache to fetch multiple blocks per cycle, and preserving optimized hot spots for continued execution. The Trace Generation Unit is discussed more throughly in Section 6.

## 2 Related Work

Control-flow profiles have been shown to provide invaluable information to an optimizer because they give insight into the frequently executed paths in the program. While profile-driven optimizations such as superblock scheduling [3] have demonstrated significant improvements in performance, software vendors have been reluctant to add a profiling step to their compilation path. In some cases, compilers have employed static branch prediction techniques to form estimated profiles with moderate success [4] [5]. More recently, a number of post-link-time optimization systems have been proposed that transparently and automatically profile and reoptimize programs. However, even when the cost of profiling is minimized, instrumentation-based profiling can still incur significant overhead [6]. Low-overhead methods of transparent profiling have been developed based on statistical sampling [7] [8] [9]. These approaches suffer from three primary drawbacks. First, the entire profile of each application must be continuously maintained at run time on the production system. Second, the profile represents only average behavior over an extended period of time. And third, the latency of detecting variations in the program's behavior can be great. The proposed Hot Spot Detector addresses these drawbacks by profiling only the most frequently executed instructions, those most likely to benefit from post-link optimization. The detector also thoroughly tracks instruction behavior over a short time window to provide an accurate and timely relative profile for each phase of execution.

Several more comprehensive profiling mechanisms have also been proposed such as the Profile Buffer [10]. This hardware table resides in the retirement stage of the microprocessor and consists of a small number of entries used to thoroughly track branch behavior. This buffer only requires servicing by the operating system when it becomes full, thus minimizing its run-time overhead. The *ProfileMe* [11] system introduced two primary improvements over sampling-based methods. First, the system provides a means for attributing a variety of events to specific instructions, whereas previous event counter mechanisms could only pinpoint offending instructions within a handful of cycles. The ability to correlate events to instructions is a key factor for performing microarchitecture-specific optimization. Second, the system allows for comprehensive random profiling of pairs of instructions in order to better understand their pipeline interaction. Like these systems, the Hot Spot Detector is able to correlate branch behavior with specific instructions, but it also has the ability to automatically filter out the unimportant branches. Furthermore, the Hot Spot Detector only requires servicing by either a hardware or software optimizer when a run-time optimization opportunity has been detected.

Run-time profiling and optimization of applications promises to deliver higher performance than is currently available with static techniques. A number of software- or firmware-based, dynamic optimization systems have emerged that optimize running applications, storing the optimized sequences into a portion of memory for extended execution. DAISY [12], FX!32 [13], and more recently Transmeta [14] and BOA [15], are systems designed to perform dynamic code translation from one architecture to another. Early versions of DAISY and FX!32 were primarily concerned with providing architectural compatibility with subsequent enhancements targeting performance, while the Transmeta processors specifically utilize dynamic translation as a means for improving performance. Dynamo [16], Wiggens/Redstone [17], and Mojo [18] are systems designed to improve application performance on the same architecture. Similarly, just-in-time compilers [19] have been introduced to generate optimized code at run time from an intermediate representation such as Java bytecode. However, these systems often suffer from significant software overhead. Many of them utilize interpretation to comprehensively profile applications before generating optimized code sequences. Others generate poorly optimized versions with embedded profiling counters to accomplish the same goal. However, time spent in an interpreter or spent executing probed code is overhead that must be reclaimed when executing the optimized code in order to see a performance benefit. These systems also require a software executive to monitor and control the reoptimization process. While our mechanism uses a similar memory-based code storage technique, it uses a hardware structure that operates in parallel with the execution of the application for rapid profiling, analysis, optimization, and management.

Similar to Dynamo, the proposed Trace Generation Unit utilizes the real execution stream to select traces for optimization [20]. In Dynamo, instructions are initially interpreted while potential trace starting points are thoroughly profiled. Once an execution threshold for a starting point is reached, a trace is formed beginning at the starting point following the current execution stream. Our proposed mechanism improves upon Dynamo's method by eliminating the overhead associated with interpretation and profiling through performing the profiling in hardware. Because the profiling overhead is low, all branches can be efficiently profiled and used to guide the trace formation process.

By taking advantage of optimization opportunities chosen at compile time, dynamic compilation [21] [22] has been used to perform optimized code generation at execution time. Run-time information, particularly the consistency of values, is used by a software dynamic compiler to generate optimized code for regions which are annotated during compilation. These regions can be selected automatically through use of code analysis and profile information [23]. By utilizing hardware support, a similar technique, called Compiler-directed Computation Reuse [24], takes advantage of run-time consistent values to eliminate entire regions of redundant computation through result memoization. However, these techniques rely upon regions chosen at compile-time and are limited by the effectiveness of the programmer or automated annotation mechanisms.

Dynamic scheduling of instructions in hardware has been used to improve instruction-level parallelism at run time [25]. By reordering instructions, execution times ideally can be reduced to the computation height of the code sequence. Furthermore, the cost of long latency operations can be hidden by the concurrent execution of other instructions. However, the window from which instructions can be selected is typically small, and no record is kept of failed reordering attempts to prevent them in the future. In addition, since this form of dynamic scheduling is located in the processor's critical path, there is limited opportunity for more advanced optimizations.

One potential hardware platform for dynamic optimization is the trace cache [26]. This caching structure stores dynamically local basic blocks together into traces, allowing for the fetch of multiple blocks per cycle. Optimizations beyond block reordering in a traditional trace cache have been proposed [27], but these transformations have been limited to classical optimizations. Since all the instructions within a trace are likely to execute together, architectures where traces serve as the fundamental unit of work have been proposed. In the Trace Processor architecture [28], sequences of traces are predicted and dispatched as a unit. Since they are cached, fetched, and executed as a unit, they provide an opportunity for more aggressive optimization such as instruction rescheduling [29].

Because traces in the trace cache are short and often have brief lifetimes, the trace cache is a limited framework for dynamic optimization. However, a new mechanism called the Frame Cache [30] has been proposed that forms much longer, atomic traces by replacing highly biased branches with *assertions* [31]. Whenever an assertion fails, the result of the entire trace is thrown away and the architectural state is reverted to the state prior to execution of the trace. Execution resumes at the first instruction in the trace, but in unoptimized code. Essentially, the optimized traces are speculatively executed in similar fashion to those in the Trace Processor. Since the architectural state prior to execution of a trace can be restored upon a failed assertion, aggressive optimization ignoring infrequent obstacles, such as side exits, can be performed. While these frames are much larger than traces in a traditional trace cache, frames are still limited by the length of the cache line and by the number of unbiased branches allowed in the trace because the frames are stored in a hardware structure.

# 3 Program Execution Characteristics

Many applications exhibit behavior conducive to run-time 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

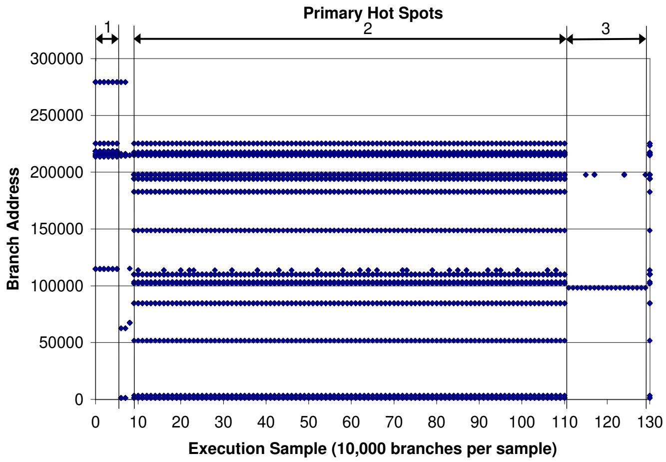



Figure 2: Important branches executed in each execution sample for *134.perl*. Each data point represents a branch that executed at least 40 times within the sample duration of 10,000 branches. These samples of 10,000 contiguous branches are taken once every 2,000,000 branches.

with a high degree of temporal locality. When a collection of intensively executed blocks also has a small static footprint, a highly favorable opportunity for run-time optimization exists. A set of branches that define a collection of intensely executed blocks is referred to as a program *hot spot*. A run-time optimizer can take advantage of execution phases by isolating and optimizing the corresponding instruction blocks for a group of hot spots. Ideally, aggressively optimized code would be deployed early in its phase for use until execution shifts to another phase. Optimized hot spots that are no longer active may be discarded, if necessary, to reclaim memory space for newly optimized code. They may also be saved for later use for re-occurring hot spots.

## 3.1 Program Phasing

An example of phasing behavior can be seen in a perl interpreter (*134.perl* from the SPEC95 benchmark suite) running a word jumble script. As shown in Figure 2, 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 involves 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. Figure 2 also indicates that, within each hot spot, the addresses of the intensely executed instructions are not typically clustered in one small address range, but instead have components widely scattered throughout the program.

The second hot spot serves as an excellent example of why run-time optimization is needed. The input script exercises perl's `split` routine, which breaks up an input word into individual letters, and `sort`, which sorts

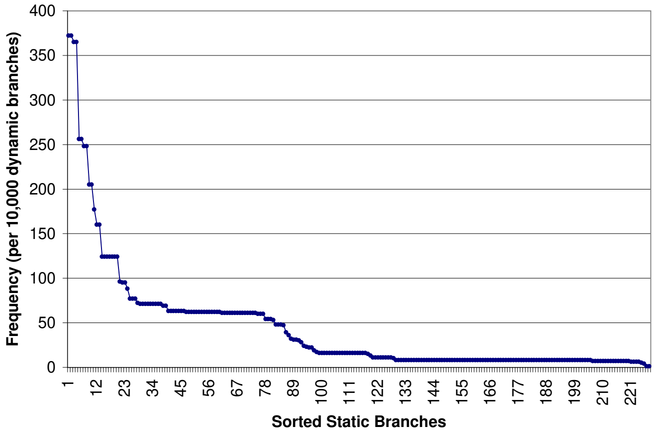



Figure 3: Profile distribution for the second primary hot spot in *134.perl*. Branches sorted from most to least frequent.

those letters. The first function, `split`, calls a complicated regular expression matching algorithm with an empty regular expression. Because execution consistently traverses a small number of paths within the functions that comprise the algorithm, this region of code would benefit from partial inlining and code layout, followed by classical optimization. A static compiler could perform these optimizations, but the larger code size and compile time would be wasted for 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 or run-time optimizer is needed to support inlining across library and application boundaries. In addition, if dynamically-linked libraries are used, inlining would have to be delayed until application load or run time. Figure 3 shows a branch profile for a typical 10,000 branch sample of perl's second hot spot. All of the branches that execute in the phase comprise its *working set*. However, the data reveals that an even smaller number of static branches account for the vast majority of the dynamic instances in the sample, for example branches 0 through 122. These intensely executed branches comprise the hot spot, and effective optimization can be limited to that portion of the working set.

### 3.2 Hot Spot Characteristics

Programs often contain some functions that appear in multiple hot spots. This commonly occurs with internal library functions that are called from different locations within the program. Naturally, the behavior of these functions may vary depending on the calling context. One such example is the function `str_new` from the

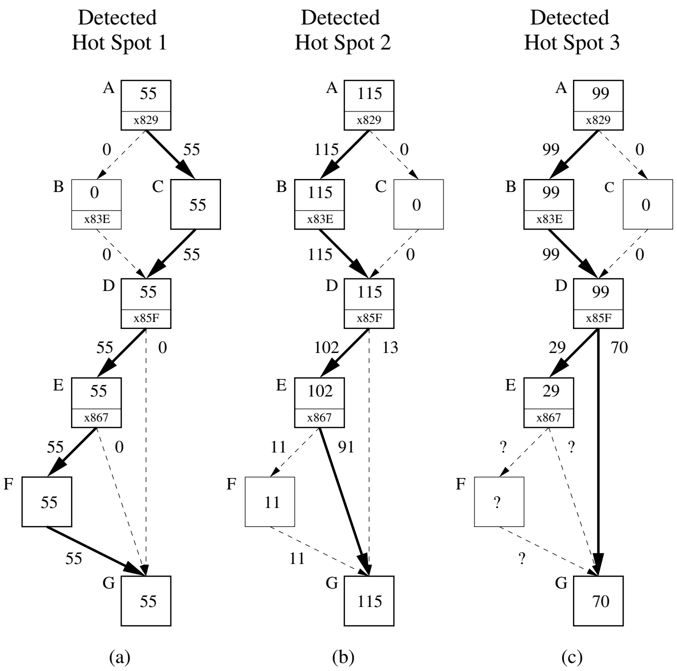



Figure 4: Different detected profiles for the `str_new` function from three different hot spots within *134.perl*.

perl interpreter. This function is present in three main hot spots and has three different profiles, each of which were gathered by the mechanism described in the next section. Figures 4a-c depict these three versions and are annotated by profile weights collected from the Hot Spot Detector. The dark arrows and blocks indicate the important edges and basic blocks as determined by the profile weights. Note that the `x867` branch from block E is missing from the profile for Hot Spot 3. This situation results from contention for resources within the hardware detector. For blocks that end in a branch, the block weight is the branch execution weight, while for other blocks, the weight is derived from the known input arcs. The profile of Hot Spot 1 reveals that branch `x829` in block A always branches to block C. Branch `x829` decides whether a previously freed string is available from the string free list, or whether a new string must be created. For Hot Spot 1, the free string list is empty each time the function is called. The input script begins by reading a dictionary file and creating new strings for the words. This operation requires no string deletions and hence no strings are added to the free list. The opposite is true for Hot Spot 2, where a free string is always available from the free list.

In an effort to better understand the composition of hot spots, another important hot spot was dissected, exposing its control flow structure. A portion of the primary hot spot from a lisp interpreter (*130.li* from SPEC95) running the training input script is depicted in the control-flow graph in Figure 5. This hot spot represents 45% of the program's dynamic execution. Control enters the shown portion of the hot spot at point A in function

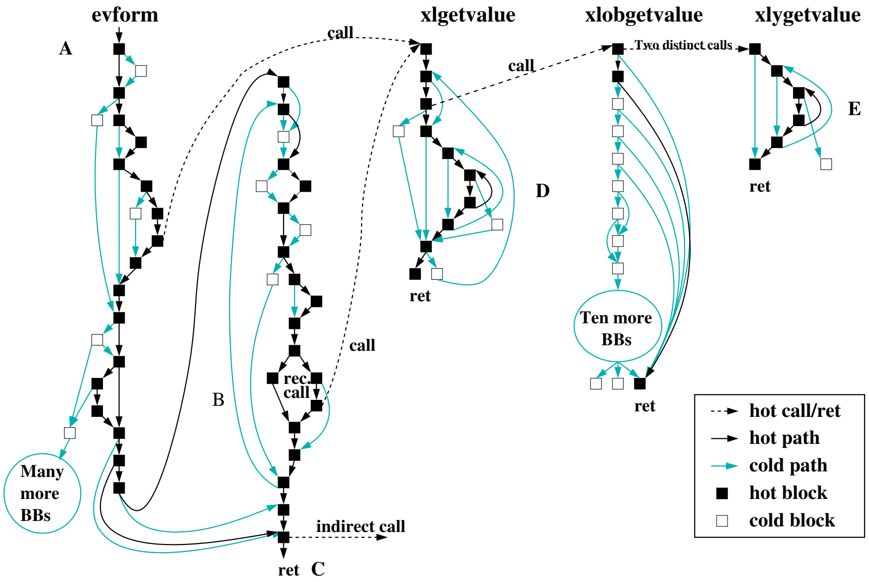



Figure 5: Basic block flow graph for selected functions in a dominant hot spot in *130.li*.

`evform`, then performs nested calls to `xlgetvalue`, `xlobgetvalue`, and `xlygetvalue` before exiting the hot spot at point C. Black Boxes and arrows indicate the intensely executed paths (those that are part of the hot spot), while lighter boxes and arrows indicate less frequently executed, or cold, blocks and paths. This entire hot spot consists of 81 branches spanning approximately 368 static instructions. The branch profile indicates a primary path through the hot spot, as 47 of the 56 static branches (9.2M of 10.9M dynamic branches) have highly consistent dynamic branch direction (greater than 90% in one direction). The hot spot is not simply a tight loop; rather, control proceeds through a number of functions with minimal inner loops. Notice, for instance, that the loops marked by back-edges B, D, and E iterate only a few times, if at all, during each invocation of the hot spot.

The hot spot behavior witnessed in the lisp interpreter generalizes to other programs as well. In many of the hot spots detected across a variety of benchmarks, the dynamic number of taken branches does not heavily outweigh the number of fall-through branches, as would be the case for inner loops with little internal control flow. While some hot spots do contain such loops, many also have much more complex control flow. The number of call and return instructions together average about 15% of the dynamic control flow in the examined programs. While these control transfers are often highly predictable, their frequency presents a barrier to wide fetch and optimization. Many of the benchmarks have a fair number of unconditional jumps, representing, on average, about 5% of dynamic control flow instructions. These instructions perform no control decisions and are obvious candidates for optimization. Indirect jumps and indirect calls do not make up a large portion of the branches, but are frequent enough to become hazards to long trace formation for some benchmarks. Inlining the potential

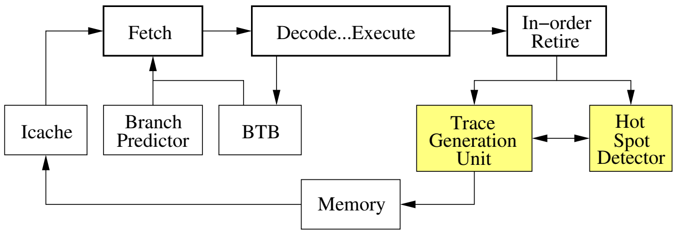



Figure 6: Instruction supply and retirement portions of the processor with the hot spot detection and trace generation hardware.

target may allow for wide fetch across the indirect jump or call. Finally, only about 35% of the dynamic control flow instructions fall-through to sequential instruction addresses. Consequently, traditional fetch architectures that break fetches at taken branches will often be limited to one basic block per fetch.

# 4 Architecture Overview

Our proposed architecture, presented in Figure 6, consists of two primary hardware components called the Hot Spot Detector and the Trace Generation Unit. Both of these hardware components are located off the critical path of the processor core in the retirement stage of the pipeline. The Hot Spot Detector monitors retiring instructions, searching for a hot spot. Once a hot spot is found, the Trace Generation Unit tracks the retirement of subsequent instructions, utilizing the retirement stream to construct traces that fall within the bounds of the detected hot spot. These generated traces collectively contain a vast majority of the frequently execution instructions for a particular phase of program execution. Once generated, traces can be optimized and are written into a code cache for execution in lieu of the original program. The code cache resides in instruction memory so that program execution can occur through the traditional fetch and execute mechanisms.

The optimization process begins as the Hot Spot Detector searches for program hot spots as they emerge during execution. Hot spot detection consists of two parallel tasks which operate on a hardware profiling table. First, retired branches are examined and entries are maintained in the profiling table for the most frequently executed branches. Often, an infrequently executed branch will retire. Such branches, however, will be entered into the table for a short time while their importance is monitored, but will later be evicted from the table. The profiler retains the frequently executed branches, which are called *candidate branches*.

The second task is to determine how closely the collected behavior stored in the table matches the actual instruction stream. This is accomplished by tracking the ratio of candidate branches to non-candidate branches in the stream. As more candidate branches are detected and placed into the profiling table, the ratio of candidate

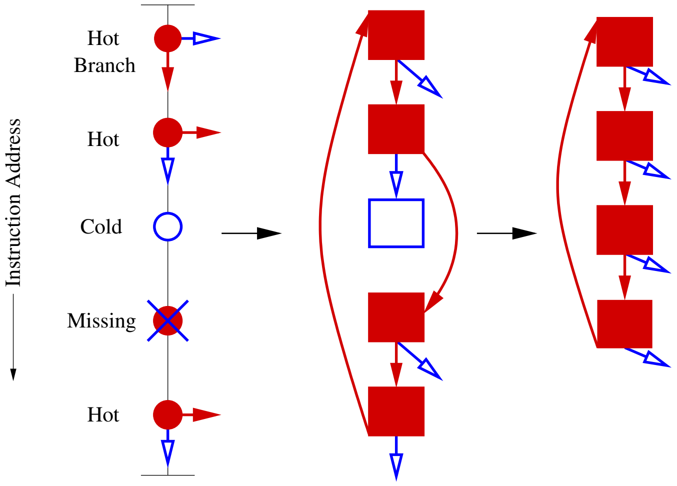



Figure 7: Process for using branch profile information to form traces.

branches to non-candidates increases. When the ratio exceeds a set threshold and the ratio is sustained for a set period of time, the detector declares that a *hot spot* has been identified and freezes all of the entries in the profiling table. Upon detection, the Trace Generation Unit is activated to construct traces based on the contents of the profiling table. However, if the ratio does not exceed the threshold quickly enough, the entire profiling table will be flushed and the detection process will be restarted. During a flush, a number of threshold values and parameters in the detector could be dynamically updated to alter the hot spot search criteria.

As described, the detector is designed to find, track, and profile the most frequently executed branches. Figure 7a depicts branch entries stored in the profiling table as gray circles. While profiling, the branch execution and direction weights were accumulated to provide behavior information about the branches during the current execution phase. The important, or hot, branches and directions are shaded and comprise a skeleton of the frequently executed region of code. The task of the Trace Generation Unit is to use the skeleton to determine the instruction blocks associated with the branches, as shown in figure 7b, and to construct a new set of blocks covering the frequently executed ones, as shown in figure 7c. While a compiler can perform control-flow analysis over an entire code region to identify and extract traces, it is impractical for a hardware mechanism to do this because of time and space requirements. Instead, the Trace Generation Unit watches the sequences of subsequently retired instructions and constructs traces from those sequences that conform to the bounds of the detected hot spot. Furthermore, the Trace Generation Unit must be able to form good quality traces even though some of the important branches may be missing from the table. Because the detector can only profile a finite number of branches, a few important branches can be missing from the profile due to contention for table entries.

Traces can be optimized either during or following the trace generation process. This work specifically uses the detected hot spots to guide a code straightening and layout optimization that targets high-performance instruction issue.

# 5 Hot Spot Detector Architecture

The Hot Spot Detector is designed to be a general mechanism for finding code regions that would derive the most benefit from run-time optimization. The region selection process is guided by three criteria. First, the region must have a small static code size to facilitate rapid optimization. Second, the instructions in the code region must be active over a minimum time interval so an opportunity exists to benefit from run-time optimization. Third, the instructions in the code region must account for a large majority of the total executed instructions during its active time interval.

The first step in the process of identifying hot spots is to detect the frequently executing blocks of code. Employment of a hardware scheme eliminates the time overhead and allows detection to be transparent to the system. The Hot Spot Detector collects the frequently executing blocks by gathering the branches that define their boundaries as well as their relative execution frequency and bias direction. Though not explicitly constructed, a control flow graph with edge profile weights can be inferred from the collected branch execution and direction information.

Relative to latencies within the processor core, the Hot Spot Detector can tolerate a large latency before recording information about program execution. For this reason, our proposed hardware is off the critical path and gathers required information from the retirement stage. This serves both the purpose of limiting adverse affects on the processor's timing while also preventing the need to handle updates from speculative instructions.

## 5.1 Branch Behavior Buffer

To implement hot spot detection in hardware, we use a cache structure called a *Branch Behavior Buffer* (BBB). The purpose of the BBB is to collect and profile frequently executed branches whose corresponding blocks account for a vast majority of the dynamically executing instructions. Depicted in Figure 8, the BBB is indexed on branch address and contains several fields (detector fields shown in white): *tag* (or branch address), *branch execution count*, *branch taken count*, and *branch candidate flag*.

When the processor retires a branch instruction, the BBB is indexed by the branch's instruction address. If the branch address is found in the BBB, its execution counter is incremented. If the branch is taken, the taken counter is also incremented. To prevent the execution counter from rolling over, the counter saturates. When

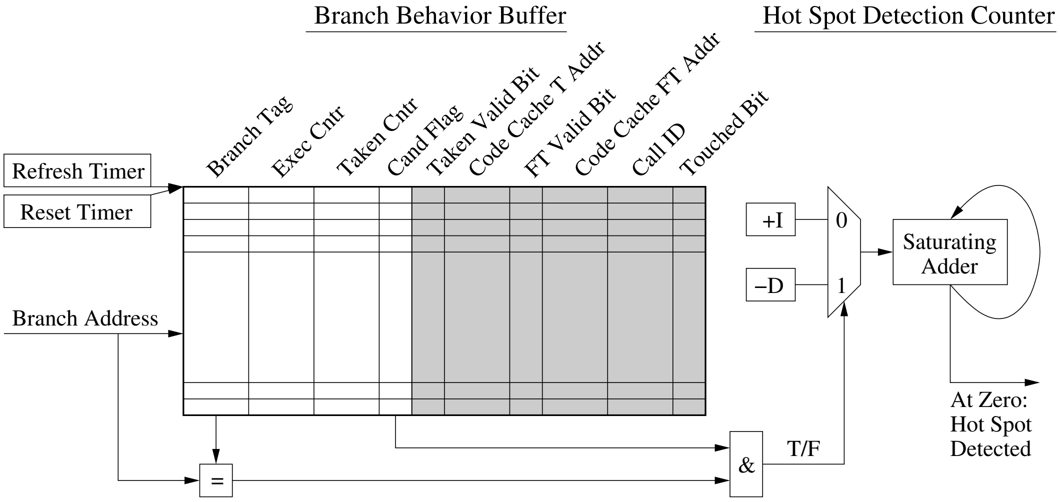



Figure 8: Hot Spot Detector hardware. Hot Spot Detector fields shown in white; additional fields for trace layout shown in gray.

the execution counter saturates, the taken counter is no longer incremented in order to preserve the *taken branch counter* to *executed branch counter* ratio. As long as the number of branches that reach saturation is small, the profiles will still reflect the relative importance of the branches collected.

A replacement policy must exist for when a branch address is not found in the BBB and the entry to which the branch indexes is currently in use. Since the BBB must accurately record the most frequently executed branches and their profiles rather than the most recently accessed, it would be unacceptable to implement an LRU replacement policy and allow a rare branch to replace a frequently executed branch. Instead, branches that conflict with existing BBB entries are simply discarded. The function of branch replacement is controlled by periodically invalidating some entries.

The entry of branches into the BBB proceeds as follows. When a branch is seen for the first time its behavior is unknown, making it necessary to give the branch a trial period, gather an initial profile, and determine its likely importance. If an entry at its index is available, the branch is temporarily allocated into the BBB and profiled over a short interval called a *refresh interval*. Its execution counter must surpass a threshold called the *candidate threshold* to avoid having its BBB entry invalidated at the next refresh. A branch that surpasses the candidate threshold is called a *candidate branch*, for which the candidate flag in its BBB entry is set, and its entry will not be invalidated at the next refresh.

The refresh interval is implemented using a simple global counter called a *refresh timer* that increments each time a branch instruction is executed. When the refresh timer reaches a preset value, all BBB entries for branches that have not yet surpassed the candidate threshold are invalidated. 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,

$$CANDIDATE\_RATIO = 2^k/2^n, \tag{1}$$

given that the size of the refresh timer is *n* bits, and that the candidate flag is marked when bit $b_k$ of the execution counter is set.

Since entries for low weight branches are invalidated at each refresh, the BBB only needs to be large enough to hold the candidate branches for a hot spot plus a number of potential candidate branches. If the BBB is too small, the initial allocation of entries to insignificant branches will delay the entrance of important branches into the BBB. Statistically, the important branches will eventually get entries after subsequent refresh intervals, but the profile accuracy could be compromised and the reporting of hot spots delayed.

A more serious conflict occurs when two important branches index into the same cache location. Many of these conflicts can be eliminated by making the BBB set-associative. However, some conflicts will still exist. As long as the remaining conflicts are relatively rare, a run-time optimizer can be designed to infer the presence of the missing branches. If required, even the profiles for these branches may be inferred.

### 5.2 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 for optimization. We define a threshold on the minimum percentage of candidate branches that must execute over a time interval as the *threshold execution percentage* and define the actual percentage of candidate branches executed over an interval as the *candidate execution percentage*. Two criteria must be satisfied before a group of candidate blocks is declared a hot spot. First, the candidate execution percentage should equal or surpass the threshold execution percentage. Second, this high candidate execution percentage should be maintained for some minimum amount of time. When the two criteria are met, a detection occurs.

In order to minimize disruption of the system during the hot spot detection process, we track the behavior of the candidate branches in hardware using a *Hot Spot Detection Counter* (HDC), shown in Figure 8. The Hot Spot Detection Counter is implemented as a saturating up/down counter that is initialized to its 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 values of *D* and *I* are determined by the desired threshold, and will be discussed later. When the candidate execution percentage exceeds the threshold execution percentage, the counter begins to move down because the total amount decremented exceeds the amount incremented. 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, a hot spot has been detected and a software run-time optimizer may be invoked via the operating system, or a hardware run-time optimizer, such as the Trace Generation Unit in the next section, enabled.

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 to materialize.

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 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.

In each of these scenarios, some branches were executed frequently enough to warrant candidate status and therefore consideration for inclusion in a hot spot and further profiling. However, if the collection of candidate branches does not materialize into a hot spot after continued tracking, all branches must be cleared in order to begin a fresh detection process. In other words, some branches may have once been important, but now are cluttering the detection attempt. 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 but clears all entries in the BBB, including 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.

It should be noted that the weights of the blocks and edges are only valid within a particular hot spot. Although this profile information is useful for inferring a control flow graph for a particular hot spot, profiles of different hot spots cannot be meaningfully compared. For instance, even if one hot spot has block weights twice that of another hot spot, the first hot spot is not necessarily executed twice as often as the second. These weights are primarily affected by two factors: refresh periods required to detect the hot spot, and frequency of the particular instructions within the hot spot. The greater the number of refreshes required to detect a hot spot, the longer the profiles are allowed to accumulate, and the greater the weights will be.

### 5.3 Hot Spot Detection Parameters

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:

$$X * (-D) + (1 - X) * (I) \leq 0 \tag{2}$$

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

$$X \geq \frac{I}{D + I} \equiv X_t \tag{3}$$

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:

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

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

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

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.

# 6 Trace Generation Unit Architecture

After hot spot detection, the Branch Behavior Buffer contains a set of frequently executed branches whose blocks constitute a large fraction of the current overall instruction execution. This section will detail a hardware-driven mechanism that is capable of automatically extracting trace regions whose dynamic behavior closely matches the hot spot captured within the BBB and forming traces from that region to permit optimization. This hardware, referred to as the *Trace Generation Unit* (TGU), adds additional system requirements: an expanded BBB, some associated registers and control logic, and a few pages of reserved virtual address space for each process. Like the BBB, the TGU is not sensitive to latency (it may lag behind actual instruction retirement) and should have little effect on the processor's critical path.

The TGU remains idle until the hot spot detection hardware detects a suitable program hot spot. Following detection, the TGU is enabled for a short period of time, during which it constructs a set of traces from the stream of instructions retired by the processor. The TGU spends most of this time operating in a passive mode, scanning the retired instructions for branches that match those captured by the BBB during profiling. The TGU actively generates traces in short bursts (experiments show $0.005\%$ of total execution time), writing the instructions to a *code cache* that resides in the virtual memory space of the process being optimized. Other than a potential slowdown during this active phase of trace generation, the TGU is non-intrusive to program execution. Because virtual memory is used to contain the optimized code, standard paging and instruction caching mechanisms allow translations to persist across context switches. The code cache pages can be allocated by the operating system at process initialization time and marked as read-only executable code.

## 6.1 Code Deployment

Because of code self-checks, a criterion for our system is that the *original code cannot be altered in any way*. Therefore any optimizations performed on the program are only performed on the generated hot spot traces. For the same reason, a seamless mechanism for transferring execution into the code cache, instead of changing the branch targets in the original code, is needed. The Branch Target Buffer can be used to facilitate control transfers. This structure associates a branch instruction with its taken target by storing the branch's target address. Similarly, our system utilizes the address field in the BTB to store the taken target's location *in the code cache*. Therefore, all entry point transitions in our system must occur along taken branch paths.

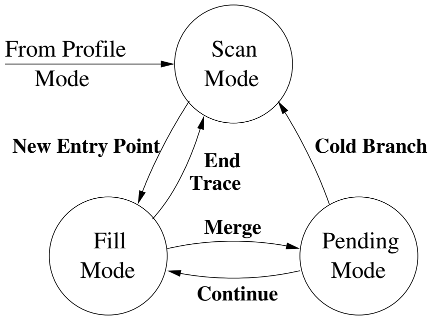
After each new trace is constructed, an entry point record for the trace is written to a list located in the first page of the code cache. The record associates the entry point branch in the original code with its target placed in the code cache. During the trace generation process, a timer signals the end of trace generation for the current hot spot. At that time, a routine is initiated to install the list of entry points into the BTB. For each entry point, the BTB target for the entry point branch is updated with the address of the entry point target in the code cache. An *entry point bit* is also set in the BTB to lock the entry in place until a BTB flush. After a context switch, the same routine can be invoked to reinstall the entry points into the BTB on a per-process basis. No new hardware is required, other than that needed to update BTB entries and to ignore branch address calculations selectively for branches that have the entry point bit set.

Self-modifying code (code that writes into its own code segment) presents a challenge to all dynamic optimization systems. Modifications made to the original code segment must also be reflected in the optimized traces. To prevent optimized code from becoming inconsistent with the behavior of the modified original code, either instructions that have been modified must be updated or traces that contain such instructions must be flushed. Since it is not generally possible to locate all modified instruction locations in the optimized code, systems like Dynamo or those that employ a trace cache flush their entire contents when self-modification is detected. Our system must also immediately flush the contents of the code cache and return to unoptimized code.

## 6.2 Trace Generation Overview

As the TGU writes instructions into the code cache, it performs two important functions. First, it creates connected regions of code that embody the detected hot spot and defines entry points to those regions. It does this in such a way that if program control enters a hot region at a selected entry point, control will likely remain inside the region for a significant length of time. Second, the TGU automatically performs code straightening along the most frequently executed paths.

The process of copying an instruction into the code cache is referred to as *remapping* the instruction. If the copied instruction is a branch, this process can involve changing the target and possibly the sense of the copy within the code cache. Thus *remapped instruction* or *remapped branch* refers to an instruction within the code




(a)

| Rule | Condition | Rule | Condition |
|---|---|---|---|
| **New Entry Point Transition:** | | **Merge Transition:** | |
| 1 | Executed suitable entry–point branch | 6 | Executed direction of candidate branch already placed, but other direction not placed |
| **End Trace Transition:** | | **Continue Transition:** | |
| 2 | Both paths from executed candidate branch already placed | 7 | Executed candidate branch matches pending target |
| 3 | Executed non–candidate branch and maximum allowed off–path branches exceeded | **Cold Branch Transition:** | |
| 4 | Executed return target mismatches call site | 8 | Executed non–candidate branch |
| 5 | Executed candidate branch will link to recursive context | | |

(b)

Figure 9: Trace generation modes with rules for mode-altering transitions.

cache. If a branch has been *remapped in its taken direction*, then an instance of the same static branch has been placed into the code cache and the instance has had its taken target changed to point to a location in the code cache.

The collection of traces created for each hot spot is called a *trace set*. Although an individual trace may contain internal branches as well as branches to other traces in the same set, it never transfers control directly to code in a different trace set. Therefore, traces generated from a hot spot form a self-contained region of code that can be independently optimized, deployed, and removed if necessary.

To assist on-the-fly trace formation, additional fields have been added to the BBB, as shown shaded in gray in Figure 8. The code cache taken address and fall-through address fields are used to hold offsets into the code cache at which code following the corresponding branch direction has been placed. Storing the code cache addresses for the branch targets in the BBB allows the TGU to link important branches from a trace directly to the target of a previously remapped instruction in the same trace set. Valid bits for each target indicate whether or not that path has already been generated. By marking the paths that have already been generated, the TGU avoids creating redundant traces. A *callID* field also aids trace generation by tagging the target fields to a particular calling context. This prevents linking code from different contexts together, a problem that is discussed in Section 6.7. Finally, a *touched* bit is added to support the backtracking operation described in Section 6.6.

## 6.3 Trace Generation Control Logic

This section describes the operation of the state machine in Figure 9a that controls trace generation. More precise treatment of the Trace Generation algorithm is provided in [2]. Figure 9b illustrates the decision rules used for each transition arc in the state machine. The trace formation process consists of the following four modes of

| TGU Mode | Instruction Type | Candidate Branch | Off-Path Branches > Maximum Allowed | Calling Context | Executed Direction Already Remapped | Other Direction Already Remapped | **Trans-ition Rule** | Next State | Action |
|---|---|---|---|---|---|---|---|---|---|
| Scan | Taken branch or jump | Yes | N/A | N/A | No | No | **1** | Fill | Record entry point location in BBB. |
| | Default | | | | | | **Default** | Scan | |
| Fill | Conditional branch | No | Yes | N/A | N/A | N/A | **3** | Scan | End trace by branching original code. |
| | | Yes | N/A | Recursive | N/A | N/A | **5** | Scan | End trace by branching to original code. |
| | | | | Different | N/A | N/A | **9** | Fill | Place inst. Overwrite code cache target location in BBB. |
| | | | | Same | No | N/A | **10** | Fill | Place inst. Record code cache target location in BBB. |
| | | | | | Yes | No | **6** | Pending | Link trace to code cache target. Set pending target to other direction. |
| | | | | | | Yes | **2** | Scan | End trace by linking to code cache targets. |
| | Mismatched return | N/A | N/A | N/A | N/A | N/A | **4** | Scan | End trace with return. |
| | Default | | | | | | **Default** | Fill | Place inst. |
| Pending | Conditional branch | Yes | N/A | N/A | Matches pending target | | **7** | Fill | |
| | | No | N/A | N/A | N/A | N/A | **8** | Scan | End trace by branching to original code. |
| | Default | | | | | | **Default** | Pending | |

Table 1: TGU actions based upon state and BBB entry contents.

operation:

- Profile Mode: Search for and detect a hot spot.
- Scan Mode: Search for a trace entry point. This is the initial mode following hot spot detection.
- Fill Mode: Construct a trace by writing each retired instruction into the code cache.
- Pending Mode: Pause trace construction until a new path is executed.

When the Hot Spot Detector signals that a new hot spot is available, the TGU transitions from its idle Profile Mode to Scan Mode. In Scan Mode, the TGU performs a BBB lookup for each retired taken branch instruction. If the retired branch is listed in the BBB as a candidate branch that has not been used in a trace, the TGU initiates a new trace. This is accomplished by setting the current trace entry point to the next available code cache offset, storing this offset in the code cache taken field in the BBB, and transitioning to Fill Mode (transition Rule 1). Table 1 depicts these actions and transitions taken based on the BBB field contents. For example, the first row represents the transition to Fill Mode when a suitable entry point is found (Rule 1).

During Fill Mode, the TGU writes each retired instruction sequentially into the code cache. All non-branching instructions are written without modification (default fill rule). However, branching instructions require further treatment. The TGU ignores unconditional jump instructions because the block at a jump target will be filled into sequential locations immediately following the copy of the jump's predecessor. Although conditional branch instructions are written to the code cache, the TGU may invert the branch sense if execution proceeds along the taken path. Conditional branches cause the TGU to perform a BBB lookup to determine how trace generation should proceed.

If the BBB lookup fails to locate a candidate branch entry, the TGU increments a small counter that indicates the number of sequential *off-path* branches that have been retired. When this counter exceeds a preset threshold (Rule 3), the TGU signals an End Trace condition and transitions back to Scan Mode. Otherwise, the TGU continues to fill instructions along the executed path. For a reasonably sized BBB, a maximum off-path branch threshold of one allows for an occasional missing branch entry while preventing excessive trace generation down cold paths.

When the BBB lookup returns a candidate branch entry, the TGU checks whether the branch has been remapped in the current direction. In the case that the retired branch was taken, the TGU checks the code cache taken field in the BBB entry; otherwise, it checks the fall-through field. If the retired branch has not yet been remapped in the current direction, the TGU continues to fill instructions from the executed path (Rule 9). Before proceeding, however, the TGU marks the remapped target field valid and stores the current trace address into the remapped address field. When emitting the conditional branch, the TGU uses the remapped address from the BBB for the opposite direction if it is valid. This reduces the number of exit points from the code cache to the original code.

If the BBB lookup reveals that the retired branch has already been remapped in the current direction, the TGU stops filling the trace. If the branch has also been remapped in the opposite direction (Rule 2), then the TGU signals an End Trace condition. At an End Trace condition, the TGU writes the trace's entry point to the code cache for future insertion into the BTB and transitions back to Scan Mode. To close the trace, the TGU emits an unconditional jump using the remapped address for the direction opposite the retired branch direction as the jump target.

If the retired branch's executed direction has been remapped, but not it's opposite direction (Rule 6), the TGU signals a Merge condition. At a Merge condition, the current branch is placed in the code cache so that its taken direction links to the already remapped code. Then, the TGU sets the *pending target* register to the target address of the branch direction that has not yet been remapped and transitions to Pending Mode. In Pending Mode, the

TGU monitors retired conditional branches. If the TGU encounters a retired branch whose target has not been remapped, it compares the branch's target address with the pending target register. If both target addresses are equal (Rule 7), the TGU signals a Continue condition and transitions back to Fill Mode.

By operating in Pending Mode rather than ending a trace, the TGU forms longer, more complete traces that extend beyond loops. Consider forming a trace that contains an inner loop. When the TGU reaches the loop back edge, it enters Pending Mode, because the executed direction of the branch already has a valid remapped target to the top of the loop. When the control finally exits the loop, the TGU continues filling the trace where it left off.

While the TGU operates in Pending Mode, it is possible for program execution to leave the code that has already been encountered without reaching the pending target address. The TGU easily recovers from this situation by exiting Pending Mode if it encounters a non-candidate or cold branch (Rule 8).

## 6.4 Trace Consistency

The traces generated by the TGU are identical to the original code with the exception of control flow instructions. The TGU ensures the correctness of all modified branch instructions because it never emits a branch without providing a valid target address. For each branch in a trace, the TGU either uses the original target address of the branch, or it uses a remapped address that points to a target previously placed into the code cache, which is equivalent to the original code.

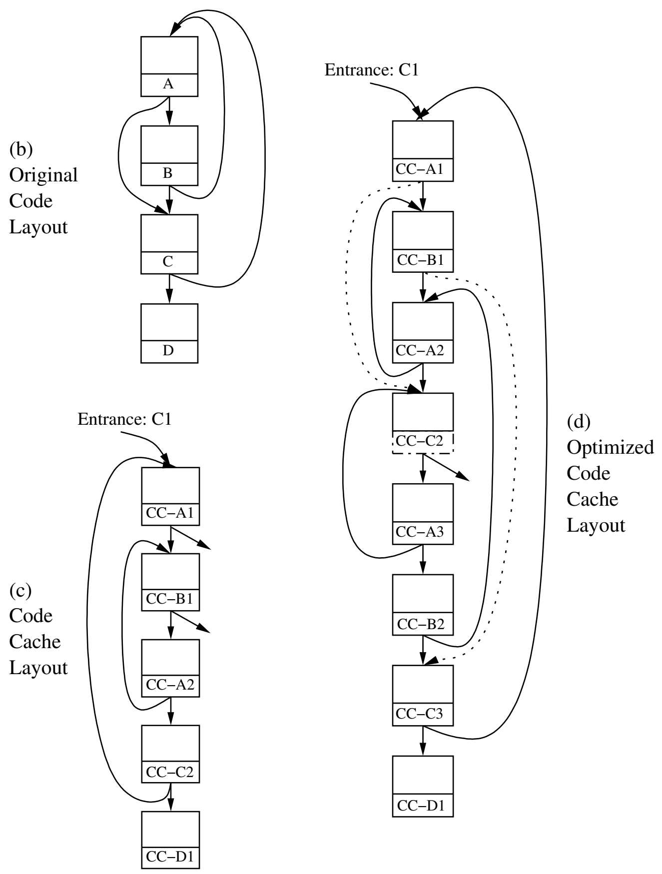
The TGU also guarantees that all traces are *well-formed* before the processor can begin executing them. A trace is well-formed if all possible paths starting at the trace entry point are valid. Recall that no entry points for a trace set are installed into the BTB until all traces in the set have been generated. Furthermore, traces from one set are never linked to traces in another set. Therefore, as long as the current trace is closed before trace generation ends, all traces in the set will be well-formed before they are installed. Note that trace generation can be terminated by an asynchronous event such as a context switch without detriment. The TGU simply emits a single closing jump instruction if it is interrupted while filling a trace.

During Fill Mode or Pending Mode it is desirable for execution to remain in the original code without jumping into the code cache. This prevents new traces from containing copies of code cache instructions and ensures that all exit branches return to the original code. Therefore, while operating in these modes, the processor must ignore BTB entries with the entry point bit set. The time spent in Fill Mode and Pending Mode is small enough that this restriction is inconsequential.

Execution order during trace generation:

C1 A1 B1 A2 C2 A3 B2 C3 D1

(a)




BBB (after trace formation)

| | Exec | Taken | C | TkV | TkA | FtV | FtA | CallID | Touch |
|---|---|---|---|---|---|---|---|---|---|
| A | 350 | 120 | 1 | 1 | Block(CC−C2) | 1 | Block(CC−B1) | 0 | 0 |
| B | 235 | 120 | 1 | 1 | Block(CC−A2) | 1 | Block(CC−C3) | 0 | 0 |
| C | 235 | 235 | 1 | 1 | Block(CC−A1) | 1 | Block(CC−D1) | 0 | 0 |

(e)

Figure 10: Trace generation example.

## 6.5 Trace Generation Example

This section provides an example of the trace generation process. Figure 10b shows the original code layout. The label at the end of each block of instructions represents the static branch instruction that terminates the block. Figure 10a lists the execution sequence seen after entering Scan Mode. The number following each branch label signifies a dynamic occurrence of that branch. The basic trace generation mechanism described in the previous section generates the trace shown in Figure 10c. The static branches in the code cache are denoted by “CC” followed by the label for the dynamic branch that caused the trace branch to be emitted. The application of two fetch optimizations, *patching* and *branch replication*, results in the trace shown in Figure 10d. Patching reduces premature trace exits while branch replication performs more aggressive code straightening, unrolling loops in the process. Figure 10e depicts the final contents of the BBB after trace generation.

The branches $A$, $B$, and $C$ are all candidate branches and, therefore, potential entry points. However, $C$ is most likely to be selected as the entry point because it is the most frequently taken branch. After detecting a new trace entry point at $C$, the TGU remains in Fill Mode until it reaches dynamic branch $C2$. Because the taken target of $C2$ has already been remapped, the TGU transitions to Pending Mode. During this time, the processor executes one loop iteration before reaching $C3$. Execution of the fall-through path of $C$ signals a Continue condition, and the TGU reenters Fill Mode. The TGU continues generating a trace until the number of sequential non-candidate branches (such as $D$) exceeds the off-path threshold (Rule 3). The TGU then closes the trace and returns to Scan Mode to search for another trace.

### 6.6 Enhancements

To further improve the issue bandwidth achieved while executing instructions from the code cache, the TGU was extended with the additional optimizations explained in this section. Implementation details are contained in [2].

Notice from the trace in Figure 10c that executing the taken path of branch *CC-A1* would cause control to exit the code cache. When this happens, the system executes original code until an installed entry point is encountered. The TGU employs a simple optimization called *patching* to prevent such premature trace exits. When the TGU emits branch *CC-A1*, it places the address of *CC-A1* itself in the taken address field of the BBB entry for branch $A$, but leaves the field marked invalid. When the TGU encounters branch $A2$, it reads the address of *CC-A1* from the BBB and patches *CC-A1* so that its branch target points directly to the fall-through path of *CC-A2*, as shown by the dotted line in Figure 10d. Similarly, the TGU patches branch *CC-B1* to the fall-through path of *CC-B2*. In general, patching can be performed in Fill Mode only when the trace encounters a branch whose target has been remapped in the direction opposite from the direction currently executed.

To improve instruction issue bandwidth, it is desirable to eliminate as many taken branches as possible without reducing instruction cache performance. *Branch replication* is a general optimization that has the dual effect of both unrolling small loops and tail-duplicating blocks so they exist in multiple traces. Without branch replication, a trace is filled past a particular branch in the same direction only once. Any subsequent copies of that branch in the same trace set are inverted with respect to the first copy such that their target addresses point to the fall-through address of the first copy. Branch replication, on the other hand, allows traces to continue past the branch multiple times without linking back to the first copy of the branch. Application of this optimization can be seen in Figure 10d. Instead of merging the already remapped taken path of *CC-C2* to Block(CC-A1), *CC-C2* is inverted so that fall-through path now points to Block(CC-A3). *C*'s BBB code cache taken target field is not updated since it already contains a valid target. Because the fall-through path of *C* still has not been seen, the taken path from *CC-C2* must point back to Block(D), which is the fall-through address in the original code.

Occasionally, during Fill Mode, actual execution does not follow the most frequently executed path through the hot spot. To avoid creation of suboptimal traces, an optimization called *backtracking* is used to discard the current trace when execution follows a cold path.

High instruction issue rates are often limited by the number of branches that can be predicted in a single cycle. One method for overcoming this limitation is to mark the instruction with a static prediction via a technique called *branch promotion*. Because mispredictions are expensive, we chose to promote only instructions whose BBB profile indicates 100% execution in one direction.

The Trace Generation Unit is also capable of forming traces across indirect jumps. The new JMP_INDIRECT_INLINE instruction, as described in Section 6.9, positions one of its indirect targets immediately following the remapped copy of the jump. Currently, the selection of the target is based upon the target that was first encountered during the trace formation process. One typical use of indirect jumps is in the calling sequence of a shared library function, where there is only a single target of the indirect instruction. For multi-target jumps, a target profiling mechanism could be be employed to select the most frequent target.

### 6.7 Automatic Inlining

Call and return instructions account for a significant portion of control transfers, and benefit can be gained from inlining frequently executed calls. During trace formation, the TGU inlines subroutines that are part of the current hot spot. Inlining an individual call site requires only minor additions to the trace generation logic. However, the TGU must avoid linking different inlined copies of the same original subroutine to each other. To do this, the TGU attaches a call ID to each inlined call site that is unique within the current trace set. The TGU maintains the next available call ID in a register, which can be reset upon detection of a new hot spot. During trace generation, the TGU also maintains a call ID stack that represents the current calling context.

In addition to branch instructions, the TGU identifies call and return instructions during Fill Mode. When the TGU encounters a retired call instruction, it emits a CALL_INLINE instruction, described in Section 6.9. It then pushes a new call ID onto the call ID stack along with the return address of the call. Similarly, when the TGU encounters a return instruction, it pops the top entry from the call ID stack. If the stack is empty, or if the return address does not match the next retired instruction address, then the TGU signals an End Trace condition (Rule 4). This is usually a result of executing a return instruction without having inlined its corresponding call.

A slight extension to the BBB access performed by the TGU upon retirement of a conditional branch has been made to support inlining. When the TGU updates the remapped target field in the BBB, the TGU also stores the current call ID. Additionally, when the TGU performs a BBB lookup, it compares the recorded call ID with the current call ID. A remapped target is only considered valid if the two call IDs match. This effectively prevents an inlined subroutine from being linked to another copy of the same subroutine in a different calling context. The BBB lookup also compares the call ID field with those on the call ID stack. If the branch entry's call ID is found in a previous stack entry (Rule 5), then a recursive call has been made. In this case, the TGU signals an End Trace condition to avoid recursive inlining.

## 6.8 Automatic Inlining Example

Consider the example shown in Figure 11. The caller consists of a single block loop that makes two serial calls to the same callee. The trace generation process begins as normal, placing the block terminated by $callA$ (Block(callA)) into the code cache. To inline $callA$, the TGU pushes the next available call ID (1) onto the call ID stack along with the expected return address (Block(callB)). Next, the TGU inserts a CALL_INLINE instruction in place of the original call. When executed, this instruction pushes the return address of the original call site, Block(callB), onto the process stack, but allows execution to fall-through to the next instruction in the trace. Pushing the original return address is a safety mechanism that guarantees return of control to the correct function if execution takes an early exit from the trace. This also hides the existence of the code cache from programs that read the return address value directly.

As the TGU fills the body of the inlined subroutine ($D$, $E$, and "$ret$"), the BBB entries for $D$ and $E$ are tagged with call ID 1. When the TGU reaches the return instruction, it pops the top entry from the call ID stack and verifies that the expected return address (Block(callB) matches the address of the next retired instruction. The TGU then emits a RETURN_INLINE instruction and continues filling the trace. Although the RETURN_INLINE normally allows control to fall through to the next instruction in the trace, it still must compare the return address found on the stack with the original address of the next trace instruction. This check is necessary in the event that a program directly modifies its return address. If the comparison fails, the RETURN_INLINE behaves exactly as a normal return instruction.

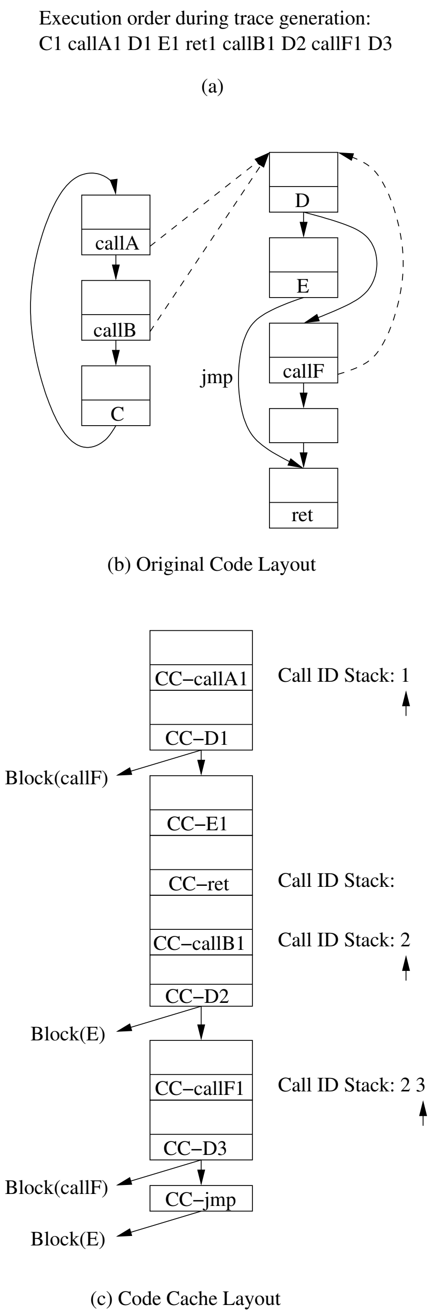



Figure 11: Trace example with inlining.

Following the RETURN_INLINE instruction, a trace is formed through Block(callB), and inlining of $callB$ begins. While inlining the second copy of the subroutine, the call ID stack contains call ID 2. This prevents the TGU from patching the taken target of *CC-D1* to the fall-through target of *CC-D2*.

After Block(callF) is filled, another call site to the same function is encountered. CallF is inlined as before, and the next call ID (3) is pushed onto the call ID stack. Block(CC-D3) is then filled into the trace. When a BBB lookup for $D3$ is performed, the BBB Call ID field value (2) matches one of the call ID stack entries below the current stack pointer. Therefore, the condition for Rule 5 is satisfied, and the TGU ends the trace by emitting a conditional jump to Block(callF) and an unconditional jump to Block(E).

## 6.9 New Instructions

The following are new instructions that are used within the code cache. These instructions are designed to support run-time optimization in hardware but are not visible to the programmer. Although it might be possible to emulate these operations with traditional instructions, they are proposed to be implemented in the microarchitecture for maximum efficiency.

- CALL_INLINE(return addr to *original* code)

  Unlike a normal function call, the program counter is set to the next sequential instruction, and the return address is set to the *operand value* rather than the next PC. The process stack and the branch prediction Return Address Stack (RAS) are properly maintained in case a normal return is executed later.

- RETURN_INLINE(expected return addr)

  Execution speculatively continues with the instructions immediately following the RETURN_INLINE. Meanwhile, the operand, which is the expected return address, is compared to the return address on the stack. If the values do not match, then a misprediction occurs and a normal return is executed.

- CALL_INDIRECT_INLINE(indirect addr, inlined addr, return addr to original code)

  The actual indirect target is calculated normally and compared against the inlined address operand. If there is a mismatch between the actual target and the inlined target, a normal call is made to the calculated target. Otherwise, a CALL_INLINE is performed. In both cases, the original return address provided by an operand is pushed onto the stack.

- JMP_INDIRECT_INLINE(indirect addr, inlined addr)

  The actual indirect target is calculated normally and compared against the inlined address operand. If there is a mismatch between the actual target and the inlined target, a normal jump is made to the calculated target. Otherwise, control passes to the instruction immediately following the inlined jump. This instruction is particularly useful for optimizing across DLL boundaries.

| 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 *count* enlarged to 800k |
| 130.li | 151M | train.lsp training input (6 queens) |
| 132.ijpeg | 1.56B | vigo.ppm training input |
| 134.perl | 2.34B | jumble.pl training input |
| 147.vortex | 2.19B | vortex.in training input |
| MSWord(A) | 325M | open 16.0 MB .doc file, search, then close |
| MSWord(B) | 911M | load 25 page .doc, repaginate, word count, select entire doc, change font, undo, close |
| MSExcel | 168M | VB script generates Si diffusion graphs |
| Adobe Photo-Deluxe(A) | 390M | load detailed tiff image, brighten, increase contrast, and save |
| Adobe Photo-Deluxe(B) | 108M | exported detailed tiff image to encapsulated postscript |
| Ghostview | 1.00B | load gsview and 9 page ps file, view, zoom, and perform text extraction |

Table 2: Benchmarks for detection and trace generation experiments.

# 7 Experimental Evaluation

Trace-driven simulations were performed on a number of applications in order to explore the effectiveness of the Hot Spot Detector and the performance of the Trace Generation Unit. The experiments for the detector were designed to maximize the program coverage of the collected hot spots while minimizing both the number of spurious branches in the hot spots and the latency of detection. Additional experiments examine the ability of the trace generator to produce frequently executed traces optimized for improved instruction fetch performance. Both SPECINT95 and common WindowsNT applications were simulated to provide a broad spectrum of typical programs. These benchmarks are summarized in Table 2. 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 independent software vendors.

The experiments were performed using the inputs shown in Table 2. In order to extract complete execution traces of these applications (all user code, including statically- and dynamically-linked libraries), we used SpeedTracer, special hardware capable of capturing dynamic instruction traces on an AMD K6 platform. Since the traced instructions are from the x86 ISA, variable length instructions are used throughout simulation. To ensure examination of all executed user instructions, sampling was not used during trace acquisition or simulation.

| Parameter | Detection Experiments Setting | Trace Generation Experiments Setting |
|---|---|---|
| Num BBB entries | 2048 | 1024 |
| BBB associativity | 2-way | 4-way |
| Exec and taken cntr size | 9 bits | 9 bits |
| Candidate branch thresh | 16 | 16 |
| Refresh timer interval | 4096 branches | 4096 branches |
| Clear timer interval | 65535 branches | 32768 branches |
| Hot spot detect cntr size | 13 bits | 13 bits |
| Hot spot detect cntr inc | 2 | 2 |
| Hot spot detect cntr dec | 1 | 1 |

Table 3: Hardware parameter settings.

## 7.1 Hot Spot Detector Evaluation

In order to evaluate the performance of the Hot Spot Detector, a number of experiments were conducted to examine the quality of the hot spots produced. Since the detected hot spots provide the basis for trace generation and optimization, maximizing coverage of the dynamic execution is critical. While the trace generation process utilizes heuristics to account for an occasional missing important branch, neglecting branches will often preclude optimization of paths containing those branches. However, providing concise hot spots as swiftly as possible ensures minimal optimization overhead and maximum available time to spend in optimized code.

### 7.1.1 Hot Spot Detection Experimental Setup

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 3. The BBB hardware was configured to allow branches with a dynamic execution percentage of 0.4% (16 executions/4096 branches) or higher to become candidates (the candidate ratio). The HDC was configured with a threshold execution percentage which required that the candidate branches total more than 66% (2:1) of the execution to become a hot spot. The BBB was allowed 16 refreshes (totaling 65535 branches) to detect a hot spot before it was reset. For the trace generation experiments, a table with fewer entries was chosen to mitigate the cost of adding additional fields to each BBB entry.

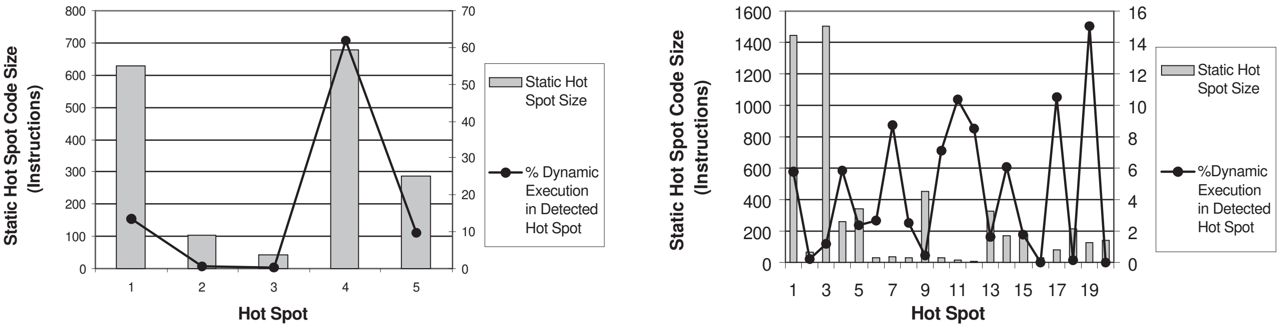
| Benchmark | # hot spots | # static insts. in hot spots | % static executed insts. in hot spots | % total exec. in hot spots | % total exec. in detected hot spots | Dyn. insts. in hot spots after detection |
|---|---|---|---|---|---|---|
| 099.go | 6 | 2398 | 3.46 | 37.84 | 35.39 | 31.7M |
| 124.m88ksim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

Table 4: Summary of the hot spots found in the benchmarks.

#### 7.1.2 Hot Spot Detection Coverage

Table 4 summarizes the effectiveness of our proposed hardware at detecting run-time optimization opportunities for each benchmark. The *number of hot spots* column lists the number of times that the Hot Spot Detection Counter saturated at zero, indicating the detection of a new hot spot. 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 collectively over the entire execution of the program. 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. The results show that, out of all the static instructions executed by the microprocessor, only a small percentage lie within hot spots. Note that some instructions may be present in more than one hot spot and are counted multiple times accordingly. 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 executing original code 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 run-time optimization. This number reflects any subsequent reuses of detected hot spots.

Analysis of the results shows that only a small percentage, usually less than 3%, of the static code seen by




(a) 134.perl.

(b) PhotoDeluxe(A).

Figure 12: Detailed hot spot statistics.

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 run-time optimizer can 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% 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% of execution, and thus only 0.40% 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 run-time optimized code.

Examining individual hot spots reveals interesting characteristics of program behavior. Figure 12a 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 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 of Figure 2 in Section 3 (note that in the histogram, *134.perl* was compiled for the IMPACT [32] architecture without inlining). From this histogram, code can be seen executing between the first and second primary hot spots, namely hot spots 2 and 3. 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 run-time 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.

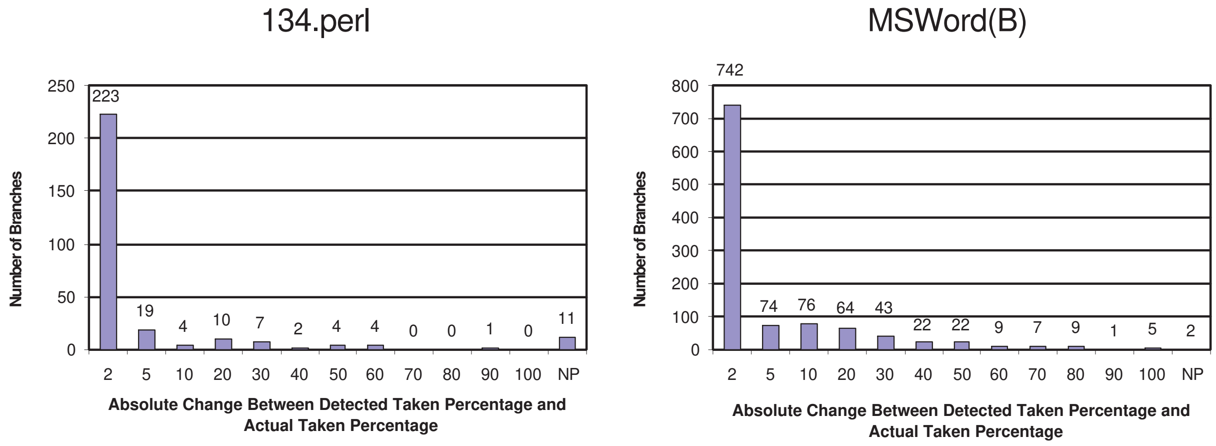



Figure 13: Absolute difference between detected taken percentage and actual taken percentage for all significant detected hot spots.

Similar characteristics were observed in the other benchmarks. Figure 12b shows an example from one of the precompiled WindowsNT applications, *PhotoDeluxe(A)*. For this benchmark, there are a several hot spots that each represent at least 8% of the total execution and together represent more than 50%. We also see quite a few hot spots with small static code sizes, indicating tight, intensely executed code. In fact, for this benchmark, the smaller-sized hot spots are also those with high total execution percentages, indicating excellent opportunities for run-time 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.

### 7.1.3 Hot Spot Detection Accuracy

To evaluate the accuracy of the hot spot profiler, the branch taken ratios collected at hot spot detection-time were compared to their average taken ratios at the completion of the application. The completion-time statistics were gathered by attributing each dynamic branch execution to the appropriate hot spot. Figure 13 shows the comparison between the two taken ratios for two typical benchmarks. In these graphs, only hot spots that represented 1% or more of the program execution were considered. For each hot spot branch, the difference between its detected taken percentage and its accumulated counterpart is computed and tallied. The figure shows that a vast majority of the branches have taken ratios at detection time within 2% of their accumulated values. These results indicate accurate profiling and therefore the potential for using detection profile weights for optimization. However, a few branches also change their behavior dramatically as can be seen by the larger changes. The last category of the graph, `NP`, represents branches that were detected as part of the hot spot but never executed after detection. Including these branches when performing optimization may introduce obstacles to aggressive optimization.

## 7.2 Trace Generation Unit Evaluation

In order to evaluate the performance of the Trace Generation Unit, the quality of the generated traces is examined. Because the generated traces will form the *unit of compilation* for the optimizer, it is critical that they cover the frequently executed paths in the hot spot. Furthermore, program performance will benefit from the formation of longer traces because they provide the optimizer with a larger window on which to operate. Optimizations such as partial inlining, loop unrolling, and branch promotion can be utilized to assist in the formation of longer traces. In addition, a trace optimizer may be employed that performs rescheduling and other optimizations on the traces. For this section's results, optimizations that improve instruction fetch performance on a traditional fetch mechanism were employed, and the resulting fetch performance compared to that of a trace cache fetch mechanism.

Figure 14 depicts an optimized trace taken from the *130.li* benchmark, as shown in Figure 5. This particular trace highlights the use of partial inlining to form a long trace through a complicated calling sequence. The TGU forms a single trace that begins prior to the call of `evform`, continues following the hot branches while inlining both calls to `xlygetvalue`, and returns to `evform`, where the TGU terminates the trace because the maximum number of allowed off-path branches has been exceeded. This trace contains 284 instructions and has 10 inlined function calls. Its execution accounts for 10% of the optimized fetch cycles of the entire application and achieves 15.1 fetch instructions-per-cycle (FIPC) on a 16 instruction issue architecture. Our simulations profiled the execution of the trace, counting the number of trace exits taken. These exits are shown between the blocks in the figure, while a large number of other side exits that are never taken have been eliminated. Clearly, a large number of trace executions proceed at least through the third block, providing evidence that optimizations to reduce the dependence height of that path would increase execution performance. Considering the third branch, the trace appears to have been generated along the less frequently executed path. However, at the time of hot spot detection, that branch flowed 100% of the time to the fourth block. This branch provides evidence that individual branches may also exhibit phased behavior.

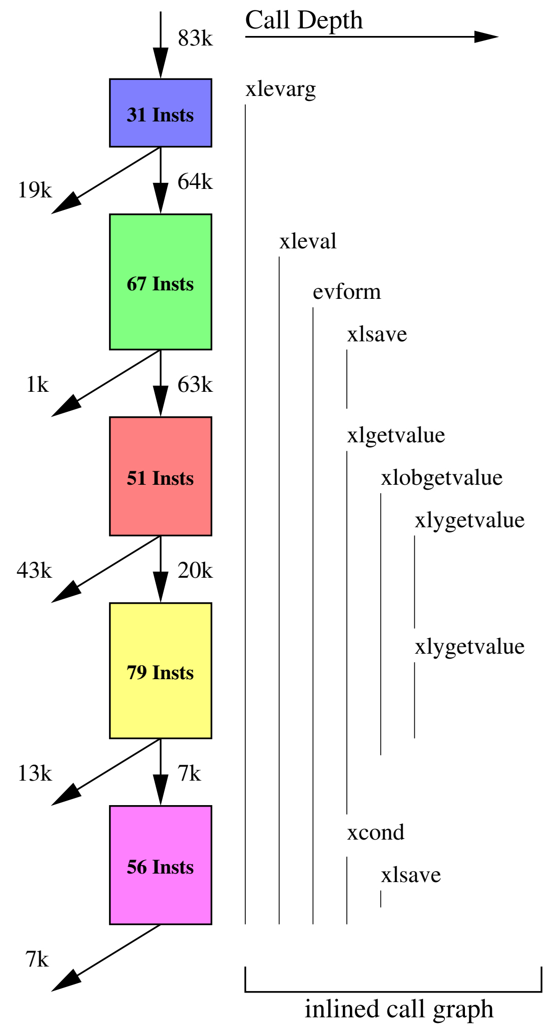



Figure 14: Example generated trace from *130.li* optimized for instruction fetch performance.

#### 7.2.1 Trace Generation Unit Experimental Setup

Instruction-by-instruction simulations of a sequential-block instruction fetch unit were performed featuring a 64KB, 4-way set associative, 128-byte line, split-block, 10-cycle miss penalty L1 ICache. The L2 ICache consists of a 512KB, 2-way set associative, 256-byte line, split-block, 100-cycle miss penalty cache. Some fetch units were also coupled with trace caches featuring either 128 (8KB) or 2048 (128KB), 4-way set associative, 64-byte lines. Table 5 summarizes the various configurations. The trace caches are allowed to form traces containing instructions from the code cache.

The simulated ICache model has a split-block configuration such that each line is divided into two banks. If a request falls into the second bank, the first bank of the subsequent cache line is also returned, if present. The instruction buffer is capable of delivering up to sixteen instructions per cycle to the decoders, but will not issue

| Model | L1 ICache size,way,block | Trace Cache size,way,block,bias table | HSD and TGU |
|---|---|---|---|
| Traditional | 64KB, 4, 128B | none | no |
| with TGU | 64KB, 4, 128B | none | yes |
| Trace Cache | 64KB, 4, 128B | 8KB, 4, 64B, 4KB | no |
| TC with TGU | 64KB, 4, 128B | 8KB, 4, 64B, 4KB | yes |
| Trace Cache | 64KB, 4, 128B | 128KB, 4, 64B, 16KB | no |
| TC with TGU | 64KB, 4, 128B | 128KB, 4, 64B, 16KB | yes |
| Traditional | 128KB, 4, 128B | none | no |

Table 5: Fetch mechanism models.

instructions past a taken branch. Up to three branches may be issued per cycle, and any instructions in the fill buffer that fall after the third branch will not be used until they are verified to be on the predicted path. The ICache assumes predecode information to identify instruction boundaries and branches.

A 14-bit-history gshare branch predictor [33] is modeled with a pattern history table consisting of entries with seven 2-bit counters, together capable of three predictions per cycle. In addition to the conditional branch predictor, a 32-entry return address stack and a 1024-entry indirect address predictor are provided. We model an ideal BTB to isolate the effect of storing entry points in the BTB. The entry point replacement policy has been deferred for future work.

We also model a trace cache that is indexed on the trace's starting address and allows partial matches (it has the ability to fetch the beginning of a trace up to a prediction mismatch). Both trace cache models (8KB, 128KB) are coupled with an ICache and use the same branch predictor as the ICache. When a fetch request is made, both units are accessed in parallel; a trace cache hit always takes precedence over an ICache hit, and only when both caches miss is the L2 ICache accessed. The trace cache is block-based and is modeled after the design in [34]. Each cache line is 64 bytes wide with slots for 16 instructions and up to 3 branches. Four target addresses are stored in the line to provide the next fetch address in case of partial matching. Traces end when the limit on instructions or branches is reached, or when an indirect branch instruction is encountered. The traces are built in basic-block granularity unless more than half of the line will be wasted, in which case partial blocks may be filled. The trace cache also utilizes a Branch Bias Table (BBT) of 1024 or 4096 entries (approximately 4 bytes each) to facilitate branch promotion within traces. Including the additional target addresses and tag stored in each cache line, the combined size for the 8KB trace cache and 1024-entry BBT is approximately 15KB.

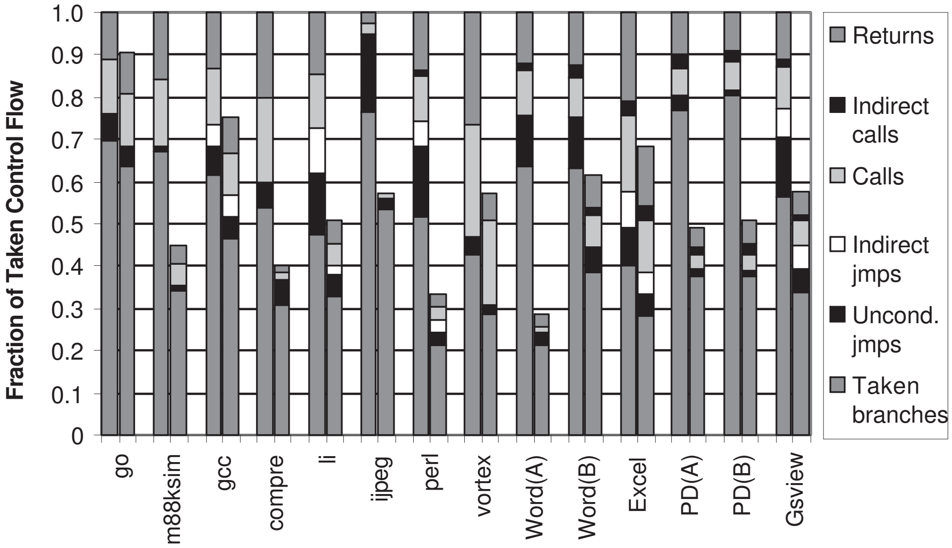



Figure 15: Reduction in taken control-flow instructions in optimized code compared to original code.

### 7.2.2 Performance of Optimized Traces

Figure 15 summarizes the reduction in taken control-transferring instructions due to the code straightening and fetch optimization. Each pair of bars for a benchmark is normalized to 100% of the taken control transfers in the original code. The bars for the optimized applications include taken control transfers from both the code cache and the original code. On average, a 45% reduction is seen across the benchmarks, verifying the effectiveness of the code-straightening techniques. Notice that call and return inlining is particularly effective, removing 25% of the taken control transfers in *147.vortex*, and sizeable amounts in the other benchmarks. Code straightening techniques for the conditional branches yield an average 24% reduction in taken control transfers, and as much as 40% for *MSWord(A)*, *PhotoDeluxe(A)*, and *PhotoDeluxe(B)*. These results show a dramatic reduction in the number of taken branches.

Table 6 presents the results of the hot spot detection and trace generation system with fetch optimizations. A large percentage of dynamic execution occurs in instructions from the code cache. Typically less than one percent of execution is spent looking for traces to form within a hot spot (Scan/Pending Modes), and a very small percentage is spent actually writing to the code cache (Fill Mode), often less than 0.005%. Even if the writing process requires a several cycles per code cache instruction, the total overhead would be well under 0.1%.

To evaluate the effectiveness of the layout optimizations, each benchmark was simulated with several different fetch unit configurations. Figure 16 shows the performance of the various fetch mechanisms. As our optimizations were targeted toward the fetch unit, the *Fetched Instructions Per Cycle* (FIPC) metric was selected

| Benchmark | % Insts. from Code Cache | % Scan/ Pending | % Fill Mode | Code Size(KB) | Entry Points |
|---|---|---|---|---|---|
| go | 10.51 | 1.02 | 0.0051 | 14.8 | 60 |
| m88ksim | 68.91 | 4.98 | 0.0027 | 8.6 | 49 |
| gcc | 32.01 | 1.07 | 0.0063 | 135.1 | 715 |
| compress | 87.05 | 0.84 | 0.0001 | 6.4 | 30 |
| li | 74.32 | 0.61 | 0.0032 | 13.6 | 59 |
| ijpeg | 84.44 | 0.09 | 0.0005 | 22.2 | 57 |
| perl | 72.34 | 0.04 | 0.0002 | 12.4 | 69 |
| vortex | 34.08 | 0.12 | 0.0006 | 26.4 | 103 |
| Word(A) | 78.46 | 0.08 | 0.0014 | 10.2 | 37 |
| Word(B) | 45.66 | 0.29 | 0.0040 | 73.9 | 330 |
| Excel | 30.69 | 3.12 | 0.0271 | 87.6 | 352 |
| PD(A) | 86.38 | 0.58 | 0.0030 | 18.9 | 105 |
| PD(B) | 81.15 | 1.25 | 0.0107 | 19.1 | 101 |
| Gsview | 60.15 | 0.35 | 0.0027 | 61.0 | 336 |
| Average | 60.44 | 1.03 | 0.0048 | 36.4 | 172 |

Table 6: Benchmark detection and trace generation results.

as an appropriate gauge of effectiveness. The bars of the graph compare the FIPC of various processor models to a baseline configuration of an aggressive multiple-block fetch unit operating on the original code. The first bar depicts the FIPC for a processor with hot spot detection and trace generation hardware, which averages 22% improvement over the base case. The improvement achieved by a comparably sized trace cache is 18%. Adding the trace cache in addition to our proposed hardware yields a benefit of 25% over the base case. With a much larger trace cache, approximately 15 times larger in size than the hot spot hardware, the FIPC is improved to 32% over base, and 39% if hot spot hardware is also included. Doubling the size of the traditional instruction cache has a significantly higher hardware cost than the hot spot hardware, and realizes only a 1% improvement. Despite the large reduction in taken control transfers shown in Figure 15, FIPC does not necessarily scale accordingly. This is primarily because branch mispredictions cause a dramatic number of stall cycles, which lessens the effect of improving throughput during useful cycles.

One advantage of our system over a trace cache is its ability to inline returns and indirect branches as was seen in Figure 14. Our traces may also include loops, as was shown in Figure 10. Conveying loop structure potentially allows better optimization than could be performed with simpler traces.

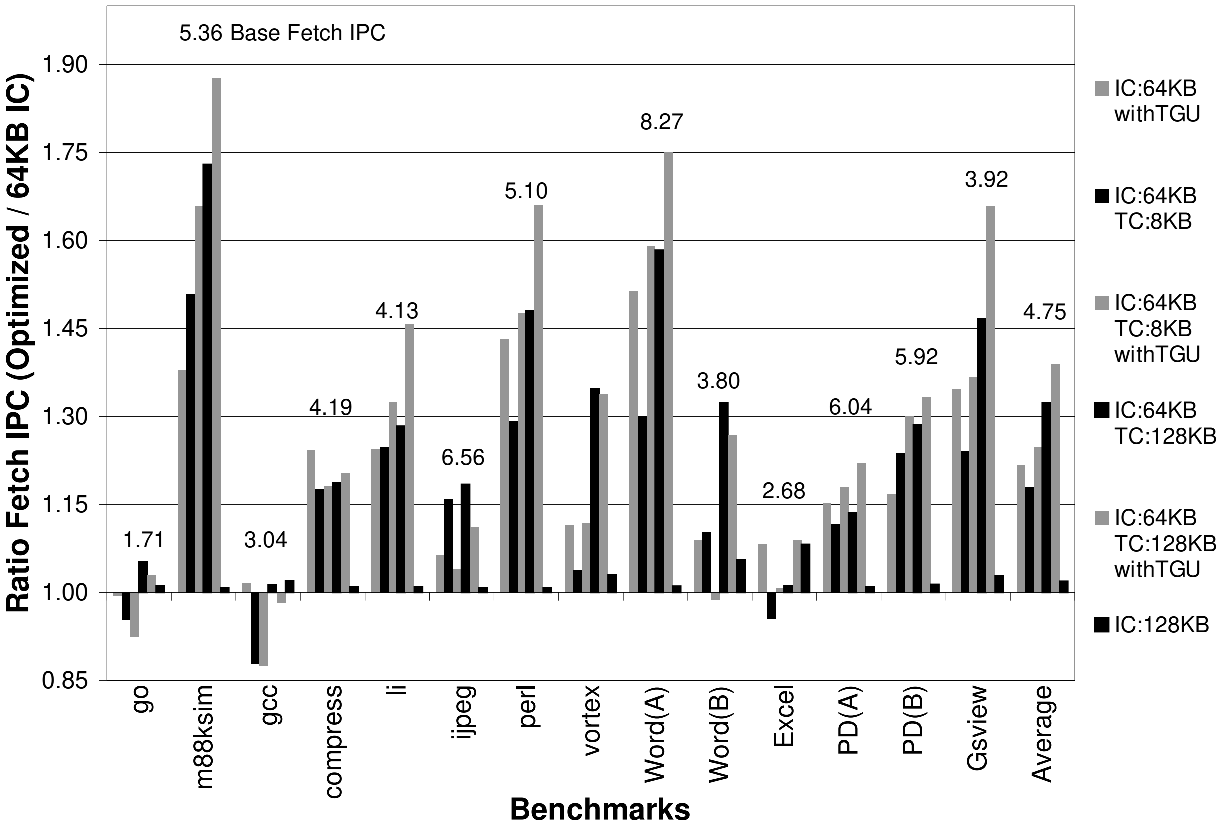



Figure 16: Fetched IPC for various fetch mechanisms.

### 7.3 Benefits of Hardware Support

The allocation of dynamic optimization components to hardware or software impacts the overhead, granularity, and flexibility of the resulting system. The larger the overhead, the longer an optimization must persist to amortize its cost and claim a benefit. If the optimization granularity is kept small, more of a program can be improved. Flexibility on what and when to optimize allows the system to adapt and vary over time and between applications.

Hardware mechanisms promise to control the run-time overheads of dynamic optimization systems by transparently applying profiling and optimization techniques. By using dedicated components, hardware mechanisms typically allow for greater parallelism, often processing in parallel with the running application. Software systems typically implement detailed profiling by utilizing instruction interpretation, or by inserting profiling probes through just-in-time compilation. This leads to frequent transitions between the application and optimizer/operating system, potentially adding a significant overhead.

Experiments have shown minimal overhead due to the hot spot detection and trace generation process. Enhancing the Trace Generation Unit to employ optimization techniques is likely to add minimal overhead as well, since the hardware optimizer will operate in parallel with trace generation and native program execution.

The Hot Spot Detector can continuously monitor program execution at low cost, conducting profiling without degrading the performance of program until a hot spot is detected. In essence, our mechanism enables full-speed native execution of the application with minimal, decisive and surgical optimizations to the code. This combination of continuous profiling and precision allows for a smaller optimization granularity than a software system.

One primary benefit of software reoptimization approaches is their flexibility. Within any profiling and code deployment system, a number of strategies can be employed that can dynamically decide when and what to optimize. While the Hot Spot Detector and Trace Generation Unit are hardware structures, they too contain a number of parameters that can be adjusted dynamically. For example, Hot Spot Detector thresholds and timers can be adjusted to vary the code region size detected. The detector could be configured to identify and optimize critical code first, later broadening its scope to optimize remaining code second.

Hardware mechanisms can also be constructed with accurate knowledge of the underlying microarchitecture. For example, when rescheduling code, the exact latencies of processor instructions can be known. This mechanism provides a means for optimizing code for new microarchitectures since the information required for performing quality optimization will be present in the microarchitecture itself.

# 8 Conclusion

Recent innovations in microprocessor design have given the processor more control over how to execute code optimally. Our system advances the state-of-the-art by allowing the processor to detect the most frequently executed code, to perform code straightening, partial function inlining, and loop unrolling optimizations, and to deploy the code for immediate use, in a manner transparent to the user application. The Hot Spot Detector monitors the retired instruction stream, providing a relative profile of the most frequently executed instructions in the stream. Unlike other hardware profilers, the detector utilizes an on-line analysis algorithm to determine when a suitable region for run-time optimization is found. The Trace Generation Unit extracts traces from the instruction stream, filtering out infrequent paths by using the profiles stored in the detector. The optimized trace sets are written into a memory so that they may utilize the traditional fetch mechanism. The detection and extraction of frequently executed code is done at the retirement stage of the processor, off the timing-critical paths. Preliminary results show that the optimizations applied by our system achieve significant fetch performance improvement at little extra hardware cost. The most important reason for the performance improvement is due to the mechanism's ability to identify hot spots early in their lifetime so that the vast majority of the execution of these hot spots is spent in optimized code. In addition, because the generated code consists of important, persistent traces, our mechanism creates opportunities for more aggressive optimizations. Areas of future investigation include issues involved in performing code optimization in the presence of exceptions and in multiprocessor environments, the application of more aggressive run-time optimizations within our framework, and application of hot spot profiling to static compilers and binary reoptimizers.

## 9 Acknowledgments

The authors would like to thank John Sias, Chris Shannon, Joe Matarazzo, Hillery Hunter, and all of the other members of the IMPACT research group, whose comments and suggestions have helped to improve the quality of this research. We also thank Prof. Sanjay Patel for his assistance in understanding the trace cache, and the anonymous referees for their constructive comments. This research was has been supported by Advanced Micro Devices, Hewlett-Packard, Intel, and Microsoft.

## References

[1] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 136–147, May 1999.

[2] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu, "A hardware mechanism for dynamic extraction and relayout of program hot spots," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 59–70, June 2000.

[3] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[4] 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.

[5] B. L. Deitrich, B. C. Cheng, and W. W. Hwu, "Improving static branch prediction in a compiler," in *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pp. 214–221, October 1998.

[6] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1319–1360, July 1994.

[7] 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 *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, pp. 1–14, October 1997.

[8] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, pp. 15–26, October 1997.

[9] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 85–96, June 1997.

[10] 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.

[11] 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.

[12] K. Ebcioglu and E. R. Altman, “DAISY: Dynamic compilation for 100% architectural compatibility,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 26–37, June 1997.

[13] R. J. Hookway and M. A. Herdeg, “Digital FX!32: Combining emulation and binary translation,” *Digital Technical Journal*, vol. 9, August 1997.

[14] Transmeta, “The technology behind Crusoe processors,” tech. rep., Transmeta, http://www.transmeta.com/crusoe/technology.html, 2000.

[15] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, “Dynamic and transparent binary translation,” *IEEE Computer*, pp. 54–59, March 2000.

[16] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system,” in *Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation*, pp. 1–12, June 2000.

[17] D. Dever, R. Gorton, and N. Rubin, “Wiggens/Redstone: An on-line program specializer,” in *Hot Chips 11*, (Stanford University, Palo Alto, CA), August 1999.

[18] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gilles, “Mojo: A dynamic optimziation system,” in *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[19] A.-R. Adl-Tabatabi, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, “Fast, effective code generation in a just-in-time java compiler,” in *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation*, pp. 280–290, June 1998.

[20] E. Duesterwald and V. Bala, “Software profiling for hot path prediction: Less is more,” in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 202–211, December 2000.

[21] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers, “Annotation-directed run-time specialization in C,” in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pp. 163–178, June 1997.

[22] M. Poletto, D. Engler, and M. Kaashoek, “tcc: A system for fast, flexible, and high-level dynamic code generation,” in *Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pp. 109–121, June 1997.

[23] M. Mock, C. Chambers, and S. J. Eggers, “Calpa: A tool for automating selective dynamic compilation,” in *Proceedings of the 33th International Symposium on Microarchitecture*, pp. 291–302, December 2000.

[24] D. A. Connors and W. W. Hwu, “Compiler-directed computation reuse: Rationale and initial results,” in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 158–169, November 1999.

[25] W. W. Hwu and Y. Patt, “Checkpoint repair for high performance out-of-order execution machines,” *IEEE Transaction on Computers*, vol. C-36, pp. 1496–1514, December 1987.

[26] E. Rotenberg, S. Bennett, and J. E. Smith, “Trace cache: A low latency approach to high bandwidth instruction fetching,” in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24–34, December 1996.

[27] D. H. Friendly, S. J. Patel, and Y. N. Patt, “Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors,” in *Proceedings 31th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 173–181, December 1998.

[28] E. Rotenberg, Y. S. Q. Jacobson, and J. E. Smith, “Trace processors,” in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 138–148, December 1997.

[29] Q. Jacobson and J. E. Smith, “Instruction pre-processing in trace processors,” in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pp. 125–129, January 1999.

[30] S. J. Patel and S. S. Lumetta, “rePLay: A hardware framework for dynamic program optimization,” Tech. Rep. CRHC-99-16, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, December 1999.

[31] S. J. Patel, T. Tung, S. Bose, and M. Crum, “Increasing the size of atomic instruction blocks by using control flow assertions,” in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 303–316, December 2000.

[32] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, “Integrated predicated and speculative execution in the IMPACT EPIC architecture,” in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.

[33] S. McFarling, “Combining branch predictors,” Tech. Rep. TN-36, Digital, WRL, June 1993.

[34] S. J. Patel, D. H. Friendly, and Y. N. Patt, “Evaluation of design options for the trace cache fetch mechanism,” *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*, vol. 48, pp. 193–204, February 1999.