A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots

Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes,
Wen-mei W. Hwu
Coordinated Science Lab
1308 West Main Street, MC-228
Urbana, IL 61801

{merten,atrick,nystrom,rdbarnes,hwu}@crhc.uiuc.edu

ABSTRACT

This paper presents a new mechanism for collecting and deploying runtime optimized code. The code-collecting component resides in the instruction retirement stage and lays out hot execution paths to improve instruction fetch rate as well as enable further code optimization. The code deployment component uses an extension to the Branch Target Buffer to migrate execution into the new code without modifying the original code. No significant delay is added to the total execution of the program due to these components. The code collection scheme enables safe runtime optimization along paths that span function boundaries. This technique provides a better platform for runtime optimization than trace caches, because the traces are longer and persist in main memory across context switches. Additionally, these traces are not as susceptible to transient behavior because they are restricted to frequently executed code. Empirical results show that on average this mechanism can achieve better instruction fetch rates using only 12KB of hardware than a trace cache requiring 15KB of hardware, while producing long, persistent traces more suited to optimization.

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 optimally. To this point in time, such optimization decisions have been limited in scope and have typically been made on-the-fly without any persistent record. This paper presents a framework for further dynamic optimization with persistent code transformations, and demonstrates a dynamic optimization targeting high instruction issue throughput.

Many optimizations rely on accurate profile information to profitably transform code. While many compilers support the use of profile information, software vendors have been reluctant to add the profiling step to their development cycles. Not only is it difficult to determine a profile that is representative of the way the program will actually be used, but in the presence of profiling, the behavior of certain programs may change. For these reasons, an automatic, transparent mechanism for profiling and reoptimizing the code based on current usage would be advantageous.

ISCA 00 Vancouver, British Columbia Canada

An automatic system could improve performance in ways that a static compiler cannot. As program behavior changes over time, it could reoptimize code for the current behavior, taking advantage of temporal relationships, whereas a typical static compiler optimizes only for the average behavior across the entire execution. Such an automatic, hardware-based system could lead to more targeted optimizations.

One such hardware profiler is called the Hot Spot Detector [7]. At runtime it determines the most frequently executed branch instructions while collecting a profile of their behavior. A snapshot of this profile is taken when the Hot Spot Detector determines that execution is primarily confined to the collected branches. This set of branches is then considered a *hot spot*. Because this process is performed very quickly at runtime, a unique opportunity exists to optimize the currently executing code while leaving sufficient time to gain benefit from executing in the optimized code. The general characteristics of hot spots are considered in Section 3, and the Hot Spot Detector mechanism is discussed more throughly in Section 4.2. This hardware serves as the basis for our code collection mechanism.

Using the Hot Spot Detector, we have developed a hardware runtime optimization that performs code straightening, loop unrolling, partial function inlining, and branch promotion to improve instruction fetch performance. We propose to enable wide fetch by collecting instructions and placing them in *memory* in executed order, thus enabling a traditional instruction cache to fetch multiple blocks per cycle. Unlike many other schemes for fetching multiple basic blocks per cycle, ours requires no extra hardware on the critical path of the microprocessor, but instead uses a modest 12KB of hardware plus control logic located in the instruction retirement stage of the processor pipeline.

2. RELATED WORK

Runtime optimization of applications promises to deliver higher performance than is currently available with static

techniques. A number of software-based, dynamic reoptimization systems, such as Dynamo [1], Daisy [3], and FX!32 [5], have emerged that optimize running applications, storing the optimized sequences into a portion of memory for extended execution. However, such systems may suffer from significant software overhead related to code profiling and optimization. Our mechanism similarly uses a memory-based code storage technique, but utilizes a hardware structure for rapid profiling, analysis, and optimization. The earlier systems also require that a software executive monitor and control the reoptimization process, whereas in our system, the hardware itself manages this process.

The mechanisms and algorithms presented in this paper utilize runtime optimization techniques to attack a problem that traditionally has been managed through either special-purpose hardware or compiler techniques. In order to achieve higher levels of instruction-level parallelism, the fetch unit is required to supply multiple basic blocks per cycle to the execution units. Early designs include the sequential instruction caches and collapsing buffer [2], which were designed to fetch several contiguous blocks across cache boundaries and non-sequential intra-cache line blocks, respectively, in a single cycle. However, the complexity of the shift logic in the collapsing buffer may cause an undesirable increase in fetch latency. More recent solutions to the wide fetch problem involve reordering the code blocks into executed order and storing them into a special cache called the trace cache [11]. This cache organization has been shown to perform well, as instructions normally separated by taken branches can be fetched in a single cycle.

Most compilers improve fetch performance by reordering a program's blocks sequentially along expected frequent paths. Compilers often use profile information [9] [6] to organize the functions and blocks within functions, performing function inlining where appropriate. The Software Trace Cache [10] technique constructs traces at compile time that are similar to the traces constructed at runtime by a trace cache. Unlike most static techniques, our system optimizes across library and DLL boundaries.

The goal of our work is to allow the processor to transform the frequently executed code dynamically so that it better fits the current usage profile and better matches the available resources in the microarchitecture. Other venues for dynamic optimization are being explored by others, such as the transformation of traces in the trace cache [4]. However, the optimizations are limited by the relatively short length of the traces, and the fact that the traces are not persistent over a long time period.

3. HOTSPOT CHARACTERISTICS

Prior work has shown that a majority of dynamic execution takes place inside program hot spots [7]. For many of the hot spots, 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 no 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, which together average about 15% of the dynamic control flow in the examined benchmarks, indicates that hot spots have complex control flow. 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 target may eliminate the effects of a miss on the fetch, and also 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.

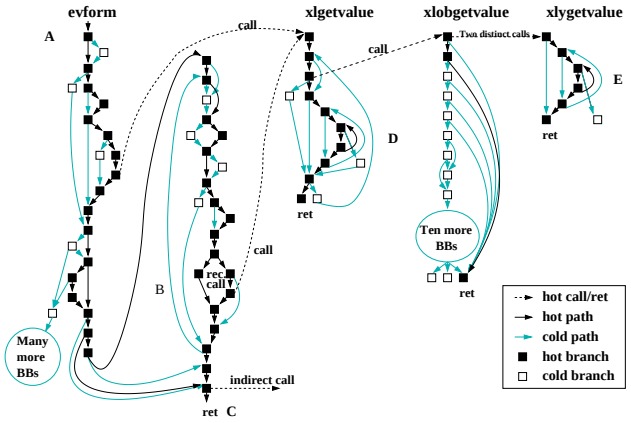


Figure 1: Basic block flow graph for selected functions in a dominant hot spot in 130.li.

In an effort to better understand the composition of hot spots, an important hot spot in the 130.li benchmark is closely examined. The hot spot represents about 45% of the dynamic execution for the training input. A portion of this hot spot is depicted in the control-flow graph in Figure 1, in which the black boxes and arrows represent the branches and paths collected as part of the hot spot. This entire hot spot consists of 81 branches spanning approximately 368 static instructions. The hot spot is not simply a tight loop; rather, control proceeds through a number of functions with minimal inner loops. The `evform` function also makes an indirect call to a wide variety of other functions, none of which qualify for inclusion in this hot spot.

Control enters this portion of the hot spot in function `evform` at point A and proceeds through `xlgetvalue`, `xlobgetvalue`, and `xlygetvalue` via calls and exits at point C. A majority of the branches are biased, indicating a primary path through the functions. Likewise, the loops marked by back-edges B, D, and E only iterate a few times, if at all, during each invocation of the hot spot. Execution in this hot spot is largely consistent, 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).

A number of layout optimization opportunities exist for this example code. Specifically, a code-straightening optimization could eliminate many of the taken conditional branches, and remove many of the cold blocks from the trace. Furthermore, function inlining may be performed to remove fetch and optimization barriers caused by the calls and returns.

4. ARCHITECTURE

This section describes a hardware-driven mechanism that is capable of automatically extracting frequently executed regions of code and remapping the code so that it closely matches the dynamic behavior of the program. The additional system requirements imposed by this mechanism include a small (less than 12KB) table, some associated registers and control logic, and a few pages of reserved virtual address space for each process. The remapping hardware involved is not sensitive to latency (it may lag behind actual instruction retirement) and should have little effect on the processor's critical path.

To understand the functionality of this system, it is useful to first consider a high-level view of system operation. First, the system must identify and profile suitable hot spots. This is referred to as *profile mode*. Upon hot spot detection, *scan mode* is entered, in which code executes as normal while the system searches for a suitable branch to begin a trace.

Once a suitable entry branch is found, the system will toggle between *fill mode* and *pending mode*, in which the original code is executed and remapped into a reserved area of virtual memory called the *memory-based code cache*. Fill mode requires that the processor write remapped code to memory; this reduces execution speed while it is taking place. However, fill mode is very infrequent (often less than 0.005% of execution, as shown in Table 7), making this overhead insignificant. When the remapping process is complete, the mechanism returns to scan mode to look for additional traces within the hot spot. When the time period allowed for remapping the code for the detected hot spot expires, the system returns to the profiling state, and begins searching for new hot spots.

4.1 Memory-Based Code Cache

As in all dynamic optimizing systems, the optimized code must be saved in some location for future execution. Some systems utilize specific hardware caches that contain the optimized code and automatically substitute it for the original code. Other systems, including the one presented in this paper, use a region of memory commonly called a *code cache* to contain the translated code.

In our system, standard paging mechanisms are used to manage a set of virtual pages reserved for the code cache. Because virtual memory is used to contain the optimized code, standard paging and instruction caching mechanisms allow the 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. Of course, the remapping hardware must be allowed to write into this reserved region of memory.

One limitation of this simple system is that the code cache cannot grow beyond its initial size. Therefore, a sufficiently large code cache must be allocated at process initialization. For our initial experiments, we assume a large code cache, and never remove remapped code from the cache. A code cache replacement policy would be useful in a production system, but is outside the scope of this paper.

4.2 Hot Spot Detection

The first step in the process of remapping hot spots is detection of the frequently executed blocks. By employing a hardware scheme, it is possible to eliminate the overhead of profiling and make hot spot detection completely transparent to the system. The heart of the hardware profiling mechanism is a structure called the Branch Behavior Buffer (BBB), which is shown in Figure 2. While the program executes, the BBB collects profile data on individual branches; this will eventually provide sufficient information to allow reconstruction of the hot paths. The BBB is indexed on the branch address and contains several fields that enable branch profiling: tag (or branch address), branch execution count, branch taken count, and branch candidate flag. The BBB, as described in [7], has been extended from its basic design to facilitate the process of trace formation. The additional fields are not used during hot spot detection and will be explained in subsequent sections.

To collect hot spot information, the BBB monitors retired branch instructions, allocating an entry for each new branch as long as an unused entry is available. Once an entry has been acquired by a branch, its execution counter is incremented whenever that branch is subsequently retired; if the branch was taken, the taken counter is also incremented. If the execution counter reaches its maximum value, both counters are locked to preserve the profile bias. Periodically, a refresh timer invalidates all BBB entries whose execution counter does not exceed a preset "candidate" branch threshold. This effectively implements a "most frequently used" policy. Although unlikely, it is still possible for conflicts to prevent an important branch from entering the BBB. Consequently, the proposed remapping algorithm is designed to easily tolerate an occasional missing branch profile.

Attached to the BBB is a hot spot detection counter that monitors the percentage of executed branches that fall within the current set of candidate branches. The hot spot detection counter increments upon execution of a non-candidate branch and decrements upon execution of a candidate branch by controlled values such that when execution remains within an intensely executed set of branches for a sufficient length of time, the counter eventually reaches zero. When the counter hits zero it signals a hot spot detection, at which time the current BBB state is frozen long enough for the trace generation unit to construct traces for the hot spot before reentering profile mode. To avoid redetecting hot spots that have already been optimized, branches in the memory-based code cache are not placed in the BBB and do not cause the detection counter to move in either direction. Code cache instructions may be identified simply by their instruction address.

4.3 Trace Generation Overview

Once a hot spot has been identified, the processor enters scan mode. In this mode, the program continues to execute in the normal manner while the Trace Generation Unit

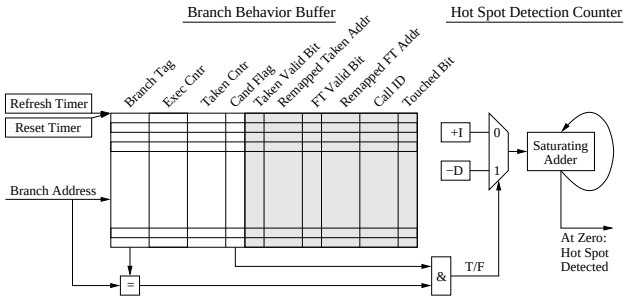


Figure 2: Branch Behavior Buffer with new fields shaded in gray.

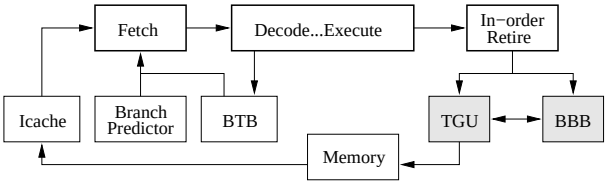


Figure 3: Instruction supply portion of the processor with the remapping hardware.

(TGU) examines the retired instructions. Guided by profile information collected by the BBB, the trace generation unit detects instructions that follow a hot path and stores these instructions in the code cache. Figure 3 depicts the interaction between the TGU and other components.

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 these 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 of code for a significant length of time. Second, the TGU automatically performs code relayout along the most frequently executed paths in the hot spot.

The TGU writes instructions into the code cache in sequential blocks or *traces*. Each trace is associated with an entry point, which is the point at which control can enter the trace from the original code. The mechanism used to activate trace entry points is discussed in the next section. For each hot spot, a collection of traces, or a *trace set*, is created for the code that comprises the hot spot. 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, each remapped hot spot is a self-contained region of code that can be independently optimized and deployed.

4.4 Code Deployment

Transfer of execution to the optimized code is handled by the Branch Target Buffer. Because of code self-checks, a criterion for our system is that the *original code cannot be altered in any way*. After each new trace is constructed in the code cache, an entry point for the trace is written to an array located in the first page of the code cache. After a preset interval, a timer signals the end of remapping for the current hot spot. At that time, a routine is initiated to install the array 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. During fill or pending mode, which are described in the next section, 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 those modes, the processor must ignore the entry point bit. The additional number of branch mispredictions incurred due to locked but ignored entry branches is small, because very little time is spent executing in the fill or pending modes.

4.5 Trace Generation Algorithm

Figure 4 illustrates the decision logic used by the TGU. As traces are remapped into the *memory-based code cache*, the TGU transitions between the following three modes of operation:

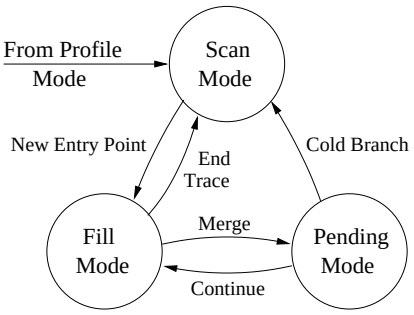
- Scan Mode: Search for a trace entry point. This is the initial mode following hot spot detection.
- Fill Mode: Construct a trace by writing each retired instruction into the code cache.
- Pending Mode: Pause trace construction until a new path is executed.

To assist on-the-fly code remapping, additional fields have been added to the BBB, as shown in Figure 2. The BBB.taken_target and BBB.fall-through_target fields are used to hold offsets into the code cache, in which the code following the corresponding *original branch direction* has been laid out. Valid bits for each target indicate whether or not that path has already been remapped. A callID field also supports remapping by tagging the target fields to a particular calling context. This prevents code from different contexts from being linked together, a problem that is discussed in Section 4.7. Finally, a touched bit is added to support the rollback operation described in Section 4.6.2. In addition to the BBB, the TGU uses the set of registers listed in Table 1.

In Figure 4, each of the rules that cause a state transition is listed next to the condition that triggers the rule. In this section the rules that are associated with the same operation (e.g. End Trace) are listed together, followed by a list of actions that are taken when any of the conditions are satisfied.

- New Entry Point (Scan Mode)

  Rule 1: A taken conditional branch or unconditional jump is retired corresponding to a candidate BBB entry in which neither direction has been remapped.



Rule	Condition
New Entry Point:	
1	→ (jcc || jmp) && candidate && taken
End Trace:	
2	→ jcc && candidate && both_dir_remapped
3	→ jcc && !candidate && off_path_branch_cnt > max_off_path_allowed
4	→ ret && ret_addr_mismatch
5	→ jcc && candidate && recursive_call
Merge:	
6	→ jcc && candidate && other_dir_not_remapped && exec_dir_remapped
Continue:	
7	→ jcc && addr_matches_pending_target && exec_dir_not_remapped
Cold Branch:	
8	→ (jcc || jmp) && !candidate

Figure 4: Trace generation modes.

CodeCacheOffset	Offset to next available memory location in code cache. Updated each time the current trace is filled with an instruction.
CurrEntryBranch	Original address of branch instruction used as the entry point to the current trace.
CurrEntryTarget	Code cache target of *CurrEntryBranch*.
PendingTarget	Original address of the instruction that that follows the current trace. Used in Pending mode to determine when to continue filling a trace.
OffPathBranches	Number of non-candidate branches executed while filling the current trace.
NextCallID	Integer value for the next unused call ID. A call ID estimates calling context.
CallIDStack	Stack representing the current call-chain during Fill mode. Each entry contains a call ID and a return address. The top entry is for the current function, below it are entries for a fixed number of parents.

Table 1: TGU registers used for code remapping.

- Align *CodeCacheOffset* with cache line boundary
- Set BBB.taken_target = *CodeCacheOffset*
- Set *CurrEntryBranch* = branch PC
- Set *CurrEntryTarget* = *CodeCacheOffset*
- Reset *CallIDStack* so that it contains no entries
- Transition: Scan Mode → Fill Mode

- End Trace (Fill Mode)

  Rule 2: A conditional branch is retired that has a candidate BBB entry for which both directions are remapped.

  Rule 3: A conditional branch is retired that does not have a candidate BBB entry and *OffPathBranches* exceeds a maximum off-path threshold.

  Rule 4: A return instruction is retired and the next PC does not match the return address on the top of *CallIDStack*. This is usually the result of executing a return instruction without having inlined its corresponding call instruction.

  Rule 5: A conditional branch is retired that has a candidate BBB entry whose BBB.CallID is on the *CallIDStack* below the top entry. This indicates recursion.

  - Emit a conditional branch whose taken direction matches the direction followed by the retired branch. Thus, if the retired branch's fall-through direction was executed, its branch sense is inverted. If this direction has already been remapped, the code cache offset from the branch's BBB entry is used as the branch target. Otherwise, the next PC in original code is used.
  - Emit an unconditional jump for the opposite direction. If this direction has been remapped, the jump will transfer control to another point in the code cache; otherwise, it will return control to the original code.
  - Install the current entry point. *CurrEntryBranch* and *CurrEntryTarget* are written to an array of entry points in the code cache for future insertion into the BTB.
  - Transition: Fill Mode → Scan Mode

- Merge (Fill Mode)

  Rule 6: A conditional branch is retired that has a candidate BBB entry in which the current branch direction has already been remapped and the other direction has not been remapped. It may later be possible to continue growing the trace if execution ever follows the other path.

  - Emit a conditional branch with BBB.current_target as the branch target. BBB.current_target refers to the taken_target field if the retired branch was taken, and the fall-through_target field if it was not.
  - Set *PendingTarget* = address in original code of the next instruction opposite the retired branch direction
  - Transition: Fill Mode → Pending Mode

- Continue (Pending Mode)

  Rule 7: A conditional branch is retired whose target address matches *PendingTarget* and whose current direction has not been remapped.

  - Set BBB.current_target = *CodeCacheOffset*
  - Transition: Pending Mode → Fill Mode

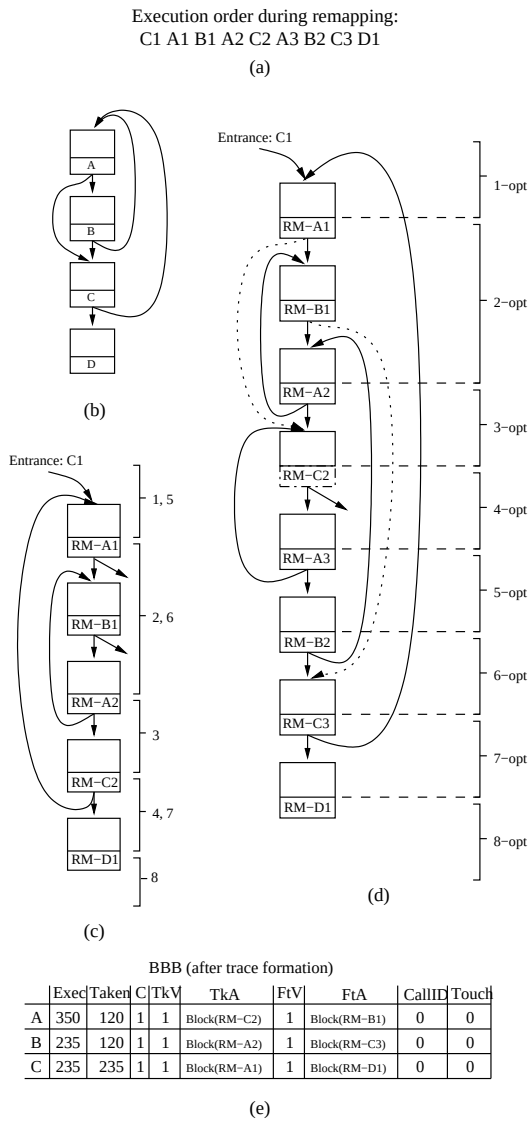
- Cold Branch (Pending Mode)

  Rule 8: A conditional branch is retired for which a candidate BBB entry does not exist.

  - Emit an unconditional jump to *PendingTarget*.
  - Install the current entry point.
  - Transition: Pending Mode → Scan Mode

4.6 Trace Example with Optimizations

This section steps through the remapping process by following a code example. Figure 5b shows the original code layout, and Figure 5a lists the execution sequence seen after entering scan mode. Basic remapping generates the trace shown in Figure 5c. The application of two remapping optimizations, patching and branch replication, results in the trace shown in Figure 5d. Patching reduces premature



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

Figure 5: Trace generation example.

trace exits while branch replication performs more aggressive code straightening, unrolling loops in the process. Steps 1 through 8 refer to the basic algorithm. Steps 1-opt through 8-opt refer to the algorithm with the optimizations. Often the operations performed in the unoptimized algorithm are also performed in the optimized version. Figure 5e depicts the final contents of the BBB after remapping.

To aid in the explanation, the following notation will be used: X refers to static branch X. X_n refers to the nth instance of X dynamically. X-tk is the taken target of the static branch X, and X-ft the fall-through target. X_n-ft indicates that X_n falls-through to X-ft. X_n-tk means that X_n branches to another target, which will be X-tk except in the case of indirect jumps. *Block(X)* is the basic block of instructions terminated by branch X. Finally, RM-X_n is a copy of X in the code cache caused by the retirement of X_n.

Step 1 and 1-opt: RETIRED[C_1-tk $\rightarrow$ Block(A)] As shown in Figure 5a, C_1 is the first candidate branch seen in scan mode. Following Rule 1, *CurrEntryBranch* is set to C and *CurrEntryTarget* to Block(RM-A_1). C's BBB.taken_target is also updated with the offset of Block(RM-A_1). The TGU then enters fill mode, and Block(A) is filled into the trace.

Step 2 and 2-opt: RETIRED[A_1-ft $\rightarrow$ Block(B), B_1-tk $\rightarrow$ Block(A)] A is seen as fall-through branch A_1 and is written into the code cache as RM-A_1. The first time any branch X is remapped as RM-X_n, the branch is emitted such that it falls-through to the target of X_n. In our example RM-A_1 falls-through to A-ft. Thus RM-A_1 and A have the same branch sense (i.e. RM-A_1 is not inverted). Because A's BBB.taken_target is invalid at this point, the taken target of RM-A_1 must point back to A-tk which is Block(C) in the original code. Next, A's BBB.fall-through_target is updated to point to Block(RM-B_1).

Block(B) is filled following RM-A_1. B_1 causes RM-B_1-ft to be set to Block(RM-A_2). Since RM-B_1-ft is equivalent to B_1-tk, RM-B_1 has the opposite branch sense of B_1 and is said to be inverted. RM-B_1-tk points to Block(C) in the original code. B's BBB.taken_target is updated to Block(RM-A_2), and Block(A) is filled again, now following RM-B_1-ft.

Step 2-opt only: In addition to the above actions, the address of each remapped branch is written to its unused BBB target field. This allows the remapped branch to be modified later so that its target can be redirected. In this case A's BBB.taken_target is set to the address of branch RM-A_1 and B's BBB.fall-through_target is set to the address of RM-B_1. To distinguish this from the case in which both targets of a branch have already been remapped, A's BBB.taken_target and B's BBB.fall-through_target remain marked as invalid.

Step 3 and 3-opt: RETIRED[A_2-tk $\rightarrow$ Block(C)] A_2 is the first time A is seen taken, so RM-A_2-ft is directed along the taken path to Block(RM-C_2). Previously, in step 2, A_1-ft was seen and A's BBB.fall-through_target was made valid. Since RM-A_2 is inverted, RM-A_2-tk is directed to the BBB.fall-through_target, Block(RM-B_1).

Step 3-opt only: Previously, in step 2-opt, A_1-ft was seen and RM-A_1-tk was directed to original code because the taken path of A had not yet been seen. However, A_2-tk now provides the taken path. A simple optimization called *patching* allows RM-A_1-tk to be updated to branch to a target within the code cache.

4.6.1 Patching

Without patching, RM-A_1-tk would cause the trace to exit the code cache prematurely and the system would be forced to execute original code until a new entry point was encountered. Patching can be performed because, in Step 2, the address of RM-A_1 was placed into the currently invalid target field of A's BBB entry (in this case, BBB.taken_target). Now, in Step 3-opt, RM-A_1-tk can be patched to the same target as RM-A_2-ft. This is shown by the dotted arc from RM-A_1 to Block(RM-C_2) in Figure 5. In general, patching can be performed in fill mode whenever the trace encounters a branch that has been remapped only in the opposite direction.

Step 4 only: RETIRED[C_2-tk → Block(A)] Since C-tk has already been seen (it was the entry branch), C_2-tk matches rule 6 and is merged back to Block(RM-A_1). RM-C_2 is written to the code cache with RM-C_2-tk pointing to Block(RM-A_1). Since C-ft has not been seen, the TGU switches from fill to pending mode, where it will wait for either C-ft or a cold branch.

Step 4-opt only: RETIRED[C_2-tk → Block(A)] By merging RM-C_2 to Block(RM-A_1), the loop is unable to take advantage of the rest of the cache line. To improve instruction issue bandwidth, it is desirable to eliminate as many taken branches as possible without reducing instruction cache performance. The TGU can use a second optimization called *branch replication* to avoid the taken jump from RM-C_2 to Block(RM-A_1).

4.6.2 Branch Replication

Branch replication is a general optimization that has the dual effect of both unrolling small loops and tail duplicating blocks that are remapped 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 (Section 4.6.3 contains a more precise treatment of branch replication). Applying this optimization to the example, instead of merging RM-C_2-tk to Block(RM-A_1), RM-C_2 is inverted so that RM-C_2-ft now points to Block(RM-A_3). C's BBB.taken_target is not updated since it already contains a valid target. Because C-ft still has not been seen, RM-C_2-tk must point back to Block(D), which is the fall-through address in the original code.

Step 5 only: RETIRED[A_3-ft → Block(B)] The TGU is still pending on Block(D) and A is not a cold branch, so the TGU stays in pending mode.

Step 5-opt only: RETIRED[A_3-ft → Block(B)] At A_3, both A-ft and A-tk have been seen. Rule 2 could be used to end the trace, in which case RM-A_3-tk would be set to jump to A's BBB.fall-through_target, Block(RM-B_1). To close the trace, an unconditional jump would be used to make RM-A_3-ft go to the taken_target in A's BBB entry, Block(RM-C_2). However, branch replication has a higher priority than Rule 2, so the TGU will continue filling. A's BBB.taken_target is used to point RM-A_3-tk to Block(RM-C_2), and Block(B) is filled after RM-A_3-ft.

Step 6 only: RETIRED[B_2-ft → Block(C)] The TGU is still pending on Block(D) and B is not cold, so the TGU stays in pending mode.

Step 6-opt only: RETIRED[B_2-ft → Block(C)] This is the first time B-ft is seen. Since RM-B_1's address was stored in B's BBB.fall-through_target in Step 2-opt, patching can be done to RM-B_1 as was done for RM-A_1 in Step 3. RM-B_1-tk is patched to Block(RM-C_3). Block(C) is then filled following RM-B_2-ft in the code cache. Since B's BBB.taken_target is valid, RM-B_2-tk points to Block(RM-A_2).

LastReplicatedBranch	Code cache offset of the last replicated branch instruction in the trace.
LastReplicatedEntry	BBB index of the last replicated branch. The target fields of this entry are used when the trace is rolled back.
LastReplicatedSP	Index into the Call ID stack that corresponds to the calling context of the last replicated branch.

Table 2: TGU registers used for branch replication.

Step 7 only: RETIRED[C_3-ft → Block(D)] The TGU is pending on Block(D), which is the target of C_3. The TGU re-enters fill mode and fills Block(D) following RM-C_2-ft.

Step 7-opt only: RETIRED[C_3-ft → Block(D)] C-ft is encountered for the first time at C_3. RM-C_3-tk is set to point to BBB.taken_target, which is Block(RM-A_1), and Block(D) is filled following RM-C_3-ft.

Step 8 and 8-opt: The filling continues through D until it enters into cold code and triggers Rule 3.

4.6.3 Branch Replication Details

Because of the complexity of branch replication, this section presents a more complete picture of the implementation details. To support branch replication, the TGU requires the additional registers listed in Table 2.

Branch replication may be performed any time that the conditions defined above in rules 2 or 6 arise. Normally these conditions result in an End Trace or Merge operation. However, branch replication adds an additional stipulation to these conditions. If the remapped target points to a previous trace (BBB.current_target < *CurrEntryTarget*) or if the remapped target and code cache offset are in the same or sequential cache banks, then branch replication is performed instead of the End Trace or Merge operation. Recall that BBB.current_target refers to BBB.taken_target if the branch is taken, and BBB.fall-through_target otherwise. Likewise, BBB.other_target refers to the target opposite BBB.current_target. If BBB.current_target points to a previous trace, branch replication tail duplicates the next block in the current trace, and the following actions are performed:

- If not valid(BBB.other_target), set BBB.other_target = *CodeCacheOffset*. This updates the address used for patching.
- Emit a conditional branch whose taken target follows the direction opposite of the currently executing branch. If this direction has already been remapped, BBB.other_target is used as the branch target. Otherwise, the original target address is used.
- Set BBB.current_target = *CodeCacheOffset*. This updates the BBB target with an offset into the current trace.

If BBB.current_target is in the same trace but is less than two cache banks away, then branch replication effectively unrolls the loop up to the next conditional branch. This heuristic assumes a two-bank cache line, but could be applied to other types of cache configurations. The TGU performs the following actions:

- Set *LastReplicatedBranch* = *CodeCacheOffset*
- Set *LastReplicatedEntry* = current BBB index
- Set *LastReplicatedSP* = current top of *CallIDStack*
- If not valid(BBB.other_target), set BBB.other_target = *CodeCacheOffset*.
- As in the previous case, emit a conditional branch whose taken target follows the direction opposite of the currently executing branch.
- Reset all touched bits in the BBB.

The three "LastReplicated" registers are set during loop unrolling, but are only used in the event that a rollback is performed. Unless the trace ends through one of the other end trace conditions (Rules 3-5), branch replication will continue to duplicate blocks until it finds a loop that crosses two cache banks, at which time the trace is rolled back to the previous replicated branch. Specifically, a rollback takes place whenever *LastReplicatedBranch* contains a valid address and the branch replication condition fails due to a distant branch target in the same trace (the target address precedes the current code cache offset by more than one cache bank).

During a rollback, the code cache offset is reset to the last replicated branch, and the End Trace or Merge operation that was deferred during branch replication is now performed. A rollback triggers the following actions:

- Set *CodeCacheOffset* = *LastReplicatedBranch*
- Set the top of *CallIDStack* to *LastReplicatedSP*
- Invert the branch, overwriting its target with *LastReplicatedEntry*.current_target, advance *CodeCacheOffset*.
- Invalidate BBB target fields with the touched bit set.
- If both targets of *LastReplicatedEntry* are remapped, perform an End Trace operation.
- If only one target is remapped perform a Merge and set *LastReplicatedEntry*.other_target = *CodeCacheOffset*

When the TGU combines both the patching and branch replication optimizations, some care must be taken to ensure correct operation. In particular, a touched bit in the BBB entry is set whenever the entry is modified to indicate that its fields point to the most recently filled block. All touched bits are then reset on a branch replication. During rollback, the touched fields are invalidated by clearing their valid bits so that their values are not used in the future.

4.6.4 Backtracking

Occasionally, during fill mode, actual execution does not follow the most frequently executed path through the hot spot. To avoid the creation of suboptimal traces, an optimization called *backtracking* is used to discard the current trace when execution follows a cold path. The TGU checks for execution down a cold path by comparing the BBB branch bias for the currently executing direction against a preset *cold threshold*. For taken branches, the execution counter in the BBB entry is shifted right by two bits and subtracted from the taken counter. If the result is positive, the branch's taken frequency exceeds a cold threshold of 25%. To perform the check for fall-through branches, the taken counter can first be subtracted from the execution counter. Returning to the example, if execution had fallen-through at branch C rather than iterating at least once, the TGU would have discarded the trace. The TGU performs the following actions during backtracking:

- Reset *CodeCacheOffset* to *CurrentEntryTarget*.
- Invalidate BBB entries with the touched bit set.
- Transition: Fill Mode → Scan Mode

Once patching or branch replication has been performed on a trace, it is no longer safe to backtrack. Therefore a single-bit flag is raised by these operations to suppress backtracking for the remainder of the trace. It is also useful to limit backtracking so that the amount of time spent in fill mode is minimal. This can be done simply with a small counter that counts the number of consecutive backtracks and suppresses backtracking after reaching a threshold.

4.6.5 Branch Promotion

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*. Some trace cache implementations use a Branch Bias Table [8] to track the long-term behavior of the branch, promoting consistent instructions in traces so that they do not require a dynamic prediction. When the static prediction is wrong, however, the processor suffers a branch misprediction penalty, and is likely to cause the promoted instruction to revert back to its dynamically predicted form.

Similarly, wide-issue instruction cache mechanisms suffer from the same limit on branches per cycle. However, the hot spot detection mechanism is well-suited to make promotion decisions during remapping, because a profile of the branches exists in the BBB. Since mispredictions are expensive, we chose to promote only instructions that execute 100% in one direction, according to the BBB. In the example in Figure 5, RM-C_2 is a promoted branch because its taken and executed counters have the same value. Occasionally, branches change their behavior over the course of program execution, causing mispredictions that may negate much of the benefit of their promotion. Thus, we require a means for demoting such misbehaving instructions. We propose a small buffer (empirically chosen to contain 16 entries in our implementation), called the Branch Demotion Buffer, that tracks a typically small number of promoted branches that exhibit mispredictions and marks the misbehaving instructions in the instruction cache to force a dynamic prediction. The buffer is organized as a fully associative cache containing saturating counters. Demotion or re-promotion is performed on the instructions when the counter reaches set threshold values.

4.7 Automatic Inlining Example

Call and return instructions account for a significant portion of control transfers. Therefore, benefit can be gained from inlining calls. Consider the example shown in Figure 6. The caller consists of a single block loop that makes two serial calls to the same callee. The remapping process begins as normal, placing Block(callA) into the code cache. At that point, a direct call is seen, and inlining of the call begins.

The process begins by assigning the callee the next available call ID, 1 in the example, and pushing it onto the hardware

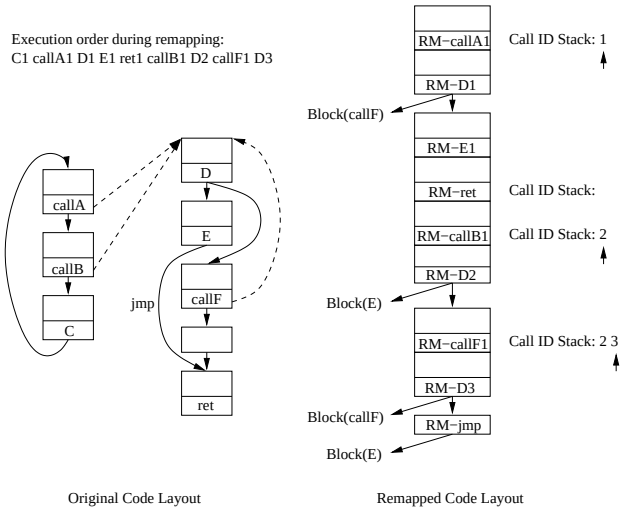


Figure 6: Trace example with inlining.

call stack along with the expected return address. Next, a CALL_INLINE instruction is inserted in place of the original call. When executed, this instruction will push the return address of the original call site, Block(callB), onto the process stack, but will allow execution to fall-through to the next instruction in the trace. Pushing the original return address is a fail-safe that guarantees that control will return to the correct function if execution takes an early exit from the trace. It also hides the existence of the code cache from programs that read the return address value directly.

Blocks D, E, and "ret" are then filled into the code cache as normal. When branch D is remapped, its BBB.CallID field is replaced with the current call ID (1). Once the return instruction is reached, the TGU pops the top entry from the call ID stack and compares the expected return address with the address of the next retired instruction. If these addresses do not match (Rule 4), the TGU terminates the trace by emitting a return instruction. A mismatch commonly occurs when a trace has no corresponding inlined call (the call ID stack is empty). However, a mismatch can also occur if, while in pending mode, the program returns from the current function and later enters it from a different call site before continuing to fill the trace. In our example, call ID 1 is popped from the call ID stack and the expected return address matches Block(callB). The TGU continues to fill instructions past the return, but instead of emitting a normal return instruction, it uses a *return inline* instruction, which allows control to fall-through to subsequent instructions in the trace. Although the *return inline* normally falls through to the next instruction, 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.

Following the *return inline* instruction, Block(callB) is remapped, and inlining of callB begins. Recall that RM-D_1-tk points to original code. Without the call ID fields, this branch would be incorrectly patched to point to the current code cache offset. To avoid this problem, each potentially distinct calling context acquires its own unique call ID. If on a BBB lookup during remapping BBB.CallID does not match the call ID on the top of the stack, then both target fields are invalidated. Consequently, when a function is inlined multiple times, targets associated with previous inlined copies are ignored and overwritten with offsets into the most recent copy. In our example, when the second call is remapped, the next call ID, 2, is pushed onto the stack. Because D's BBB.CallID is 1, BBB.fall-through_target is invalidated and BBB.taken_target is set to Block(RM-callF).

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 stack. Block(RM-D_3) is then filled into the trace. When a BBB lookup for D_3 is performed, the BBB.CallID 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).

4.8 New Instructions

The following are new instructions that are used within the remapped code. These instructions are designed to support runtime optimization in hardware but are not visible to the programmer. Although these instructions may be emulated with traditional instructions, they may need to be implemented in the microarchitecture to be maximally efficient.

- CALL_INLINE(return addr to *original* code)

  Unlike a normal 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 data/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 3: Benchmarks for remapping experiments.

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

Table 4: Hardware parameter settings.

5. EXPERIMENTAL EVALUATION

Trace-driven simulations were performed for a number of benchmarks in order to explore hot spot characteristics and to establish the effectiveness of the proposed hot spot remapping scheme. Both *SPECINT95* and common *WindowsNT* applications were simulated to provide a broad spectrum of typical programs. The benchmarks and their input sets are summarized in Table 3. 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 were the general distribution versions, and thus were compiled by their respective independent software vendors. In order to extract complete execution traces of these applications (all user code, including statically- and dynamically-linked libraries), we employed special hardware capable of capturing dynamic instruction traces on an AMD K6 platform. 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.

The hot spot detection hardware and trace generation unit were simulated on an instruction-by-instruction basis for the entire execution. When the new instructions listed in Section 4.8 were being remapped, their extra operand sizes were taken into account. The Hot Spot Detector was configured according to the parameters listed in Table 4. These parameters are similar to those used in [7], which have been determined empirically to detect consistent and concise hot spots. Compared to the previous work, we were able to reduce the BBB table size to 1024 entries without incurring an excessive number of conflicts. We also collect slightly tighter hot spots by reducing the clear timer interval and increasing the hot spot detector counter size. With the additional fields, the BBB occupies approximately 12KB of hardware.

Model	L1 ICache size,way,block	Trace Cache size,way,block,BBT	Remap Code
Traditional	64KB, 4, 128B	none	no
Remap	64KB, 4, 128B	none	yes
Trace Cache	64KB, 4, 128B	8KB, 4, 64B, 4KB	no
TC + Remap	64KB, 4, 128B	8KB, 4, 64B, 4KB	yes
Trace Cache	64KB, 4, 128B	128KB, 4, 64B, 16KB	no
TC + Remap	64KB, 4, 128B	128KB, 4, 64B, 16KB	yes

Table 5: Fetch mechanism models.

5.1 Instruction Fetch Mechanism

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 in remapped code.

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 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 is modeled with a pattern history table consisting of entries with seven 2-bit counters, together capable of three predictions per cycle [8]. In addition to the conditional branch predictor is a 32-entry return address stack and a 1024-entry indirect address predictor. 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 [8]. 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

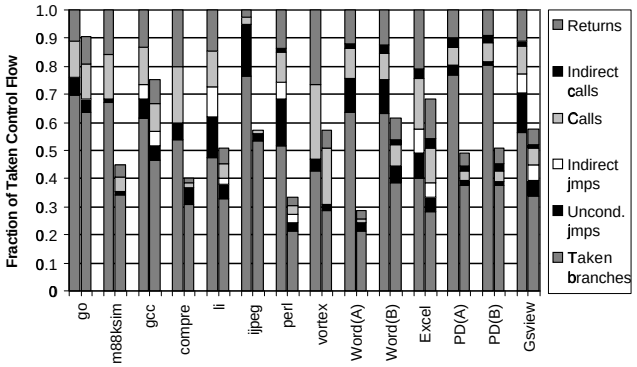


Figure 7: Reduction in taken control-flow instructions in remapped code compared to original code.

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.

The CALL_INLINE and RETURN_INLINE instructions do not require branch prediction and do not consume any of the three branch resources allowed per cycle. CALL_INLINEs are unconditional control flow, and RETURN_INLINEs almost always return to the inlined caller because they are only emitted into the code cache when the respective CALL_INLINE is in the same trace. Since they are predicted statically, functions that modify the return address will result in branch misprediction upon returning. CALL_INDIRECT_INLINE and JMP_INDIRECT_INLINE do not terminate a trace in the trace cache as their non-inlined counterparts do. When fetched from either the ICache or TCache, these instructions are predicted as if they were conditional branches. A fall-through prediction causes the issue of the inlined target, while a taken prediction relies upon an indirect predictor to obtain the new fetch address.

5.2 Performance of Remapped Code

Figure 7 summarizes the reduction in taken control-transferring instructions due to the remapping 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 remapped applications include taken control transfers both from the code cache and from the original code. On average, a 45% reduction is seen across the benchmarks, signifying 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, on average, about a 24% reduction in taken control transfers, and as much as 40% for MSWord(A), PD(A), and PD(B). These results show a dramatic reduction in the number of taken branches in the benchmarks.

Table 6 summarizes the average static length and useful lifetime of the traces formed by the remapping system. To measure effective trace length, each time a trace is entered, the number of static instructions executed before exiting the

Trace Length - Static Inst. Executed per Entry			
0-16	17-32	33-49	50+
54%	24%	10%	12%
Trace Age - Millions Inst. Executed Since Creation			
0-24	25-49	50-99	100+
4%	3%	4%	89%

Table 6: Aggregate trace characteristics.

Benchmark	% Remap Code	% 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 7: Benchmark remapping results.

trace are counted. The distribution shows that often more than 16 static instructions are executed, indicating that the traces may expose more optimization opportunities than in the trace cache. To measure the useful lifetime traces, each time a trace is entered the number of instructions executed since the installation of that trace are calculated. The age of an executed trace is usually greater than 100 million instructions, which is the entire runtime of some benchmarks and should provide opportunity for further optimization.

Table 7 presents the results of the remapping optimization system. A large percentage, often as much as 80%, of the dynamic execution occurs in the remapped code. 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, often less than .005%, is spent actually remapping the code (fill mode). Even if the remapping process requires a several cycles per remapped 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 8 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 as an appropriate gauge of effectiveness. The first bar in the graph represents the baseline FIPC of the native applications. This is an aggressive multiple-block fetch unit operating on the original code. The second bar depicts the FIPC of the remapped code, which averages 21.5% improvement over the base case. The improvement achieved by a comparably sized trace cache is 18%. Adding the trace cache in addition to the remapping hardware yields a benefit of 24.6% over the base case. With a much larger trace cache, approximately 15 times larger in size than the remapping hardware, the FIPC is improved to 32.3% over base, and 38.4% if remapping hardware is also included. Despite the large reduction in taken control transfers, as shown in Figure 7, FIPC does not necessarily scale

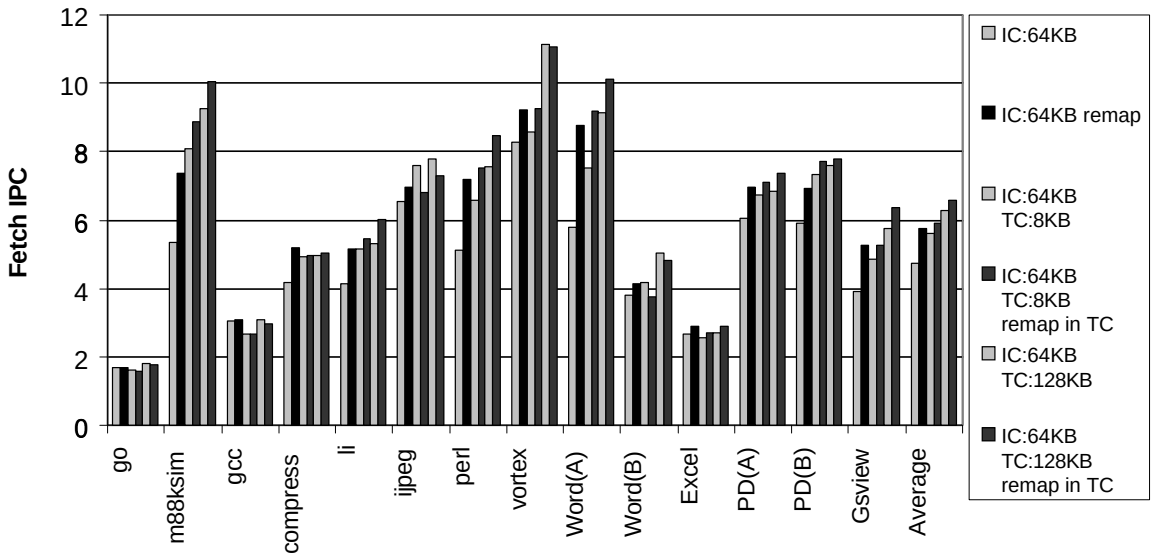


Figure 8: Fetched IPC for various fetch mechanisms.

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. The benefit of inlining returns is evident in a trace constructed for the example hot spot from the li benchmark shown in Figure 1. 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 is 284 instructions long, represents approximately 10% of program execution, and achieves an average of 15.1 FIPC when executing in the trace. In addition, our traces may include loops, as was shown in Figure 5. Conveying loop structure potentially allows better optimization than could be performed with simpler traces.

6. CONCLUSION

Recent innovations in microprocessor design have given the processor itself 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, with all of this transparent to the user application. 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. In addition, because the remapped code consists of important, persistent traces, our mechanism creates opportunities for more aggressive optimizations in the future.

7. ACKNOWLEDGMENTS

The authors would like to thank all the members of the IMPACT research group, especially Joe Matarazzo and John Sias, for their valuable insight and assistance in developing this work. We also thank Prof. Sanjay Patel for his insight into the operation of the trace cache, and the anonymous referees for their constructive comments. Our research has been supported by Advanced Micro Devices, Hewlett-Packard, Intel, and Microsoft.

8. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories Cambridge, June 1999.

[2] T. M. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proc. 22nd Annual Int'l Symp. on Computer Architecture*, pages 333–344, June 1995.

[3] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 26–37, June 1997.

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. 31th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, pages 173–181, December 1998.

[5] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), August 1997.

[6] 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*, 7(1):229–248, January 1993.

[7] 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 *Proc. 1999 Int'l Symp. on Computer Architecture*, pages 136–147, May 1999.

[8] 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*, February 1999.

[9] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, pages 16–27, June 1990.

[10] A. Ramirez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proc. 1999 Int'l Conf. on Supercomputing*, June 1999.

[11] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proc. 29th Int'l Symp. on Microarchitecture*, pages 24–34, December 1996.