EXTRACTING HARDWARE-DETECTED PROGRAM PHASES FOR POST-LINK OPTIMIZATION

BY

RONALD DALE BARNES, JR.

B.S., University of Oklahoma, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my wife Susan. I would be lost without the encouragement that I receive through her patience and persistence. The dedication to my academic endeavor is as much to her credit as it is to mine.

I would like to thank my advisor, Professor Wen-mei Hwu, for giving me the opportunity to develop as an independent researcher under his guidance. He has been an invaluable source of insight and direction. In addition, as an inspiring teacher, he serves as a role model for my future endeavors. Everyone in the IMPACT research group has been a source of both great assistance, support, and friendship. In particular, Matthew Merten has been an influential leader of the post-link optimization efforts within our research group. He has been a helpful project manager and an valuable mentor. I would also like to thank Erik Nystrom for his work on the post-link project and his constant source of ideas. I would like to thank John Sias for his help. He has taught me more about the IMPACT compiler than anyone else. David August and Dan Connors additionally deserve my appreciation for their mentoring and support when I first joined this research group.

Lastly, I would like to thank our group's corporate research partners. In particular, Hewlett-Packard's support has made the post-link optimization research within our group possible. I would also like thank Microsoft Research and, in particular David Gillies and Ronnie Chaiken, for two internship opportunities with MSR and for going out of their way to support my binary optimization work.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1

2 RELATED WORK 5
2.1 Code Placement and Optimization Mechanisms 5
2.2 Post-link Optimization Systems 6
2.2.1 Dynamic optimization systems 7
2.2.2 Static optimization systems 7
2.2.3 Units of optimization 8
2.3 Hardware Profiling Mechanisms 8
2.3.1 Statistical profiling mechanisms 9
2.3.2 Phase profiling mechanisms 9
2.3.3 Hardware profile buffers 10

3 HOT SPOT DETECTION AND MONITORING 11
3.1 Hot Spot Detector 11
3.2 Previous Hot Spot Monitoring Mechanisms 13
3.3 Enhanced BBB for Continuous Profiling 15
3.4 Hot Spot Phase Signatures 24

4 HOT REGION FORMATION 27
4.1 Step 1: Program Phase Detection 28
4.2 Step 2: Region Identification 29
4.2.1 Hot spot blocks 31
4.2.2 Inferred hot and cold blocks 33
4.2.3 Heuristic hot region growth 36
4.3 Step 3: Region Construction and Optimization Support 36
4.3.1 Locating root functions and entry blocks 37
4.3.2 Maintaining dataflow 38
4.3.3 Partial inlining 39
4.3.4 Calculation of profile weights 40
4.3.5 Region transitions 44

5 RESULTS 47
5.1 Experimental Setup 47
5.2 Evaluation 49

6 FUTURE WORK 56

7 CONCLUSION 58

REFERENCES 60

LIST OF TABLES

Table Page

3.1 Hot spot detections with enhanced BBB. 25

5.1 Benchmarks and inputs used in experiments. 48
5.2 Simulated EPIC machine model. 49
5.3 Code expansion in vacuum-packed regions. 51

LIST OF FIGURES

Figure Page

2.1 Branch bias phase shift. 10

3.1 Hot Spot Detector. 12
3.2 Monitor Table. 14
3.3 Enhanced Branch Behavior Buffer. 16
3.4 Gradual hot spot transition. 21
3.5 Effect of hot branch threshold on unique hot spot detections. 22
3.6 Effect of branch bias threshold on unique hot spot detections. 23
3.7 Calculation of proposed hot spot signature. 26

4.1 Region formation process over functions A-F, shown in call tree order. (a) Hot branches, with hot and cold directions. (b) Formed hot region. (c) Extracted and optimized region Z with cold links back to original code. 28
4.2 Region formation. (a) Branch Behavior Buffer profile. (b) Initialization of hot spot branches and their blocks. (c) Propagation of the cold arc information. (d) Propagation of the hot arc and block information. 30
4.3 Rules for selecting vacuum-packed region. 32
4.4 Initialization of hot spot branches and their blocks. 33
4.5 Inference rules. (a)-(c) Propagation to flows. (d)-(f) Propagation to blocks. 34
4.6 Inference growth of vacuum-packed region. 35
4.7 Extraction of vacuum-packed region. (a) BBB for hot spot detection for phase 1. (b) BBB for hot spot detection for phase 2. (c) Extraction of phase 1's region (formed in Figure 4.2). (d) Linking of phase 1's region with phase 2's region. 38
4.8 Maintaining dataflow. (a) Original code sequence. (b) Region after hot instructions have been extracted. 39
4.9 Inferring profile weight from surrounding blocks. 41
4.10 Deriving profile weight from branch probabilities. (a) Extracted hot spot region with branch probabilities. (b) Derivation of **Q** and **M** matricies. (c) Hot spot with derived block weights. 43
4.11 Invalid linking of hot spots inside different contexts. 46

5.1 Linterpret: an interpretation-based, cycle-by-cycle simulation. 48
5.2 Percent of dynamic instructions from within extracted hot spots. 50
5.3 Categorization of hot spot branch behavior across benchmarks. 52
5.4 Performance speedup from basic rescheduling of hot spot regions. 54
5.5 Effect of BBB size on Vacuum Packing effectiveness. 55

CHAPTER 1

INTRODUCTION

Modern computer systems continue to achieve greater performance by exploiting higher levels of instruction-level parallelism (ILP). Such systems are increasingly dependent upon code optimizations to maximize execution efficiency and to reveal the ILP to the underlying computer architecture. Unfortunately, the trend toward increasingly modular software has dwindled the optimization scope of the compiler. Because of such trends, *post-link optimization*, defined as program optimization that occurs after binary creation, is becoming an important computer system component. It provides new ways to improve application performance by availing its whole-application scope and by exploiting information that was unavailable at compile-time.

Such code improvements can be performed both statically between program executions and dynamically while a program is running. Tools like Spike [1], [2] and Vulcan [3] perform static post-link optimization of applications using profile-feedback information. These tools perform optimizations on binary executables much like those performed during compilation. Dynamic post-link optimization systems include Transmeta's Crusoe processors [4], which use low-level software and hardware support and the Aries Translator [5], which is a purely software system.

Post-link technologies improve upon static compilation techniques by exploiting key information about the run-time environment. Such information is gained both when the application is installed and when the application is executed on the end user's machine. At install-time, the exact processor model is known. The model information includes the precise latency of instructions, the number and types of functional units, and the size of caches. Additionally, the exact dynamically-linked libraries (DLLs) that will be employed become available at run time, and the actual usage patterns can be discerned through profiling. Because none of this information was available at compile time, this knowledge represents significant opportunity to further customize applications.

To take advantage of these post-link opportunities, an optimizer requires both a detailed profile-gathering mechanism and an effective optimization platform. Clearly, optimization focus should be upon heavily executed code to maximize the potential benefit, therefore application profiling is a key factor. The quantity of code produced must also be constrained so that application instruction cache, branch prediction, and paging resources are not overly taxed. Furthermore, ILP optimization often favors one usage path at the expense of another, which places a burden on the code profiling and selection mechanisms to make wise choices.

The transformation platform is also critical. It must be powerful enough to enable beneficial transformations while limiting its consumption of time and resources. Many run-time systems have to stall the application while performing transformations, thus requiring the benefits to directly outweigh the transformation costs. Many post-link systems rely on an instruction-trace-based platform because trace-based transformations generate little analysis and overhead and yet achieve reasonable performance gains [6]. While these systems allow for significant local customization, they lack the scope for broader optimizations, such as loop-level transformations.

This thesis introduces *Vacuum Packing*, a new post-link optimization technique for detecting and packaging regions of performance-critical code that includes both a swift profiling mechanism and a solid platform for transformation. The technique is founded in region-based optimization [7], which enables the optimizer to focus on heavily executed code blocks, even when control-flow crosses function boundaries. Rather than using traditional aggregate or summarized execution profile weights to form static regions, as is done within a compiler, this approach uses a transparent hardware profiler to automatically detect execution phases and record branch profile information for each new phase. A small set of interprocedural regions is formed for each phase of program activity. By focusing on these encapsulated regions, an optimizer focuses on the code that is responsible for an overwhelming majority of execution during each phase. Cache locality could be increased, and ILP optimization could be applied to favor these regions. Several regions may in fact contain copies of the same segment of code if the segment is utilized in several phases. The region formation process preserves exit semantics from the new region in case nonregion functions are accessed, but will focus the optimizer's efforts on the truly hot code. Further customization of these functions can also be performed to pare down the included functions to just the hot blocks, inline the region functions, and apply aggressive ILP optimization and scheduling to construct new tight and modular code units. The strategy is an improvement over other dynamic optimization systems because it provides a much larger scope for optimization than those that operate on traces and exploits specific execution characteristics present in each distinct phase.

The algorithms used utilize efficient information propagation and estimation techniques to compensate for the incomplete and often incoherent branch profile information that arises due to the nature of hardware profilers. The technique minimizes unnecessary code replication by making efficient connections between the original code and the new code and linking code regions at select points to facilitate phase transitions. By using a small set of profile information, code regions can be generated which are specialized for each phase of execution and that capture an average of more than 80% of total program execution. Further it is shown that the approach is very effective in extracting code regions that capture the phasing behavior of programs, that the code size increase is moderate, and that the code regions benefit from sample optimizations to improve the performance of programs.

Vacuum Packing is a three-step process for utilizing hardware profiling to detect program phases, extracting code regions that correspond to these phases, and providing a platform for post-link optimization by conditioning these regions and connecting them to the original program. The implementation presented for this technique uses the hot spot detection process [8] to provide control-flow profile information for each phase of program execution. Three algorithms needed to construct the code modules for optimization are described. The first algorithm utilizes the results of the hardware profiler, the *Hot Spot Detector*, to identify initial regions. A second algorithm describes a technique for growing the code regions to mitigate both the effects of missing profile information due to hardware table contention and the effects of spotty profile information due to temporarily dispersed execution. Finally, a third algorithm is presented that describes how execution can be transferred to the proper region even when multiple phase-based regions are rooted at the same function and many contain largely the same blocks. While this thesis evaluates the method using a static post-link optimizer, the technique could be extended for use in a dynamic system as well.

CHAPTER 2

RELATED WORK

The Vacuum Packing technique exploits an intensive profiling mechanism in an effort to form densely executed, packaged code regions customized to match particular phases of program execution. The technique is designed to treat the code that comprises a program phase as a single unit, called a *region* [7], that contains the instructions frequently executed during the phase. A great deal of previous work has similarly focused on the problem of using profile-feedback to bring important code together and optimize it, and on ways of inexpensively collecting such feedback information.

2.1 Code Placement and Optimization Mechanisms

Like Vacuum Packing, interprocedural code placement optimizations [9] [10] aim to colocate instructions with high temporal locality to improve instruction cache performance, but for the most part these attempts have focused on the movement of entire functions and have aimed only to improve caching or paging performance. Like the Hot Cold optimization [1], the presented, region-based approach uses a static post-link optimization tool to colocate heavily executed blocks that then also become the focus of optimization. The infrequent, or cold, blocks are left away from the important code blocks and are largely untouched.

To form the hot regions, this work utilizes a partial-function inlining technique [11] to expand regions by growing them to deeper calling contexts while replicating only the important blocks of the functions being inlined. The strategy has been adapted for post-link optimization systems by inlining only into regions extracted from the original code, forming a separate entity with links to and from the original code. This limits modification to the bulk of the application compared to a strategy that operated on regions left integrated in their original code locations. These new regions may be formed across library boundaries to achieve a broad scope without the explosive code growth of whole function inlining. Furthermore, within a region, interprocedural optimizations that were not available at compile-time may be employed. In the past, such global optimizations in post-link optimizers have mostly focused on peephole optimizations such as those that eliminate function-boundary overhead by breaking calling conventions [12]. Since Vacuum Packing focuses interprocedural optimization on a reasonably sized region, more aggressive transformations might be feasible.

2.2 Post-link Optimization Systems

Virtually all dynamic, post-link systems use execution profiles to focus optimization on traces of execution, much as early compiler scheduling techniques used the profiles to focus on individual traces within the application code [13], [14]. Such traces limit optimization scope but allow for very inexpensive transformations.

More recent compiler techniques have relied upon code motion beyond mere traces. For example, wavefront scheduling [15] provides an efficient method for improving performance over trace scheduling approaches through global scheduling within a function. Similarly, region-based approaches are likely to boost future post-link optimization systems beyond that of current trace-based capabilities of dynamic optimizers.

Regions provide a convenient alternative platform to traces for near-global optimization techniques without unnecessarily increasing the complexity of performing code improvements. This is achieved by focusing optimization efforts on a modestly sized code base that represents a significant portion of execution.

2.2.1 Dynamic optimization systems

Dynamo [16] is a dynamic software optimization system which generates optimized code traces in a *code cache*, a memory space allocated for storage of optimized code. The traces formed follow the path of current execution. The software overhead of such a system likely limits its practical scope to such traces. A run-time hardware optimization system, the rePLay [17] framework, uses a cache structure for storing optimized traces of execution. Since the number of stored traces is limited by the storage of this cache, replacement of traces is frequent and thus very time-consuming trace transformation is not possible. Another system, ROAR [8], [18] uses hardware profiling and a software-based cache to increase the persistence of the optimized traces that are generated.

2.2.2 Static optimization systems

Off-line optimizers such as Spike [1], [2] and Vulcan [3] are not as confined to traces as dynamic systems. In such static systems, optimization time and space requirements are less of a concern as the overhead of optimization does not directly impact the running time of the application. However, they both could benefit from this technique's ability to statically capture code that represents a dynamic phase of execution. By selecting phases of execution as the optimization regions, Vacuum Packing further enables such systems by allowing for specialization that specifically targets these phases.

2.2.3 Units of optimization

Previous work examined several units of optimization for a dynamic optimization system [6]. It considered units including traces, loops, and functions, and concluded that loops provided the best performance while the performance of much simpler traces followed second. The technique presented in this work is targeted to an off-line optimizer and for this reason is designed to capture outer loops for maximum performance. Vacuum Packing provides an further opportunity for phase-specific optimization as its transformation units are phases of execution.

2.3 Hardware Profiling Mechanisms

The Vacuum Packing strategy has been designed to exploit program phases because each phase naturally corresponds to a particular code region. A hardware profiling mechanism is utilized to capture these phases and the relative profile weights of control-flow within these regions.

2.3.1 Statistical profiling mechanisms

A number of previous strategies exist to analyze the phasing properties of applications, but most rely on statistical sampling of executing instructions. These samples are subsequently analyzed in software to determine phase composition. Hardware sampling mechanisms are often used to gather low-overhead profiles, but generally do not have the resolution required to separate one phase from another. These rely on program counter sampling over long periods of time to produce whole-execution aggregate profiles [19], [20], [21].

The Digital Continuous Profiling Infrastructure utilizes the ProfileMe [22] mechanism to collect detailed information, such as branch directions, about randomly sampled instructions. Basic Block Distribution Analysis [23] combines intense, periodic sample-based profiling to determine the composition of repetitive phases.

2.3.2 Phase profiling mechanisms

Specialized hardware components have been proposed to accurately detect phases. The dynamic working set signature detector [24] intensively monitors the executing blocks for pattern shifts and could be used to trigger a profiling phase as described in Section 3.4. However, these phases are based on instruction cache block working sets rather than the control-flow phases targeted by Vacuum Packing. While there is likely some correlation between a cache block working set phase and a phase defined by branch behavior, such a technique could not distinguish between different phases that, while having different control-flow behavior, share significant amounts of code. This distinction is evident in Figure 2.1. Though the directional

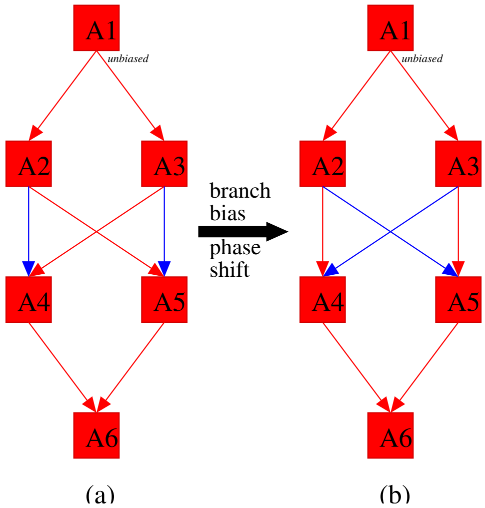


Figure 2.1 Branch bias phase shift.

biases of two hot branches are reversed, Figure 2.1(a) and (b) have the working set since they contain identical blocks. This issue is discussed further in Section 3.4.

2.3.3 Hardware profile buffers

A mechanism closer to that used in this work, the profile buffer [25] is a small table that interacts with the reorder buffer to record branch directions over a short time window to garner profile weights. The operating system periodically samples this table and gradually builds an approximate arc-weight profile. As described in Section 4.1, Vacuum Packing utilizes the Hot Spot Detector [8] detailed in Section 3.1 to perform intensive profiling and to analyze the detected blocks for stability. All of these hardware profiling approaches, however, are “lossy” in that the collected data is imperfect and incomplete due to the randomness of sampling or the limitations of hardware structures. Vacuum Packing overcomes this missing information by using inference and heuristic to select the hot region and estimate profile weights.

CHAPTER 3

HOT SPOT DETECTION AND MONITORING

3.1 Hot Spot Detector

During the first step of the Vacuum Packing process, the Hot Spot Detector (HSD), Figure 3.1, is the monitoring mechanism responsible for the detection of hot branches in each of the phases of execution [8]. The Hot Spot Detector consists of two components: the Branch Behavior Buffer (BBB) which is a table for profiling the executing branches, and the Hot Spot Detection Counter (HDC) which is a simple counting mechanism for estimating the execution coverage of the branches currently tracked in the BBB. As a branch retires from the processor, a record of its execution is passed to the detector. The static address of the branch is used to locate a table entry where the branch's dynamic behavior is recorded. This tabulation is done by incrementing the *executed counter* for every branch and the *taken counter* for every taken branch.

The Branch Behavior Buffer maintains its contents using an approximated *most frequently used* retention policy instead of a more traditional, most recently used cache retention policy. If free entries exist, each new static branch at retirement is provisionally given an empty entry and granted a window of time, referred to as a *refresh interval*, to accumulate dynamic profile information. If no entry is available, the retired branch is simply discarded. A static branch that

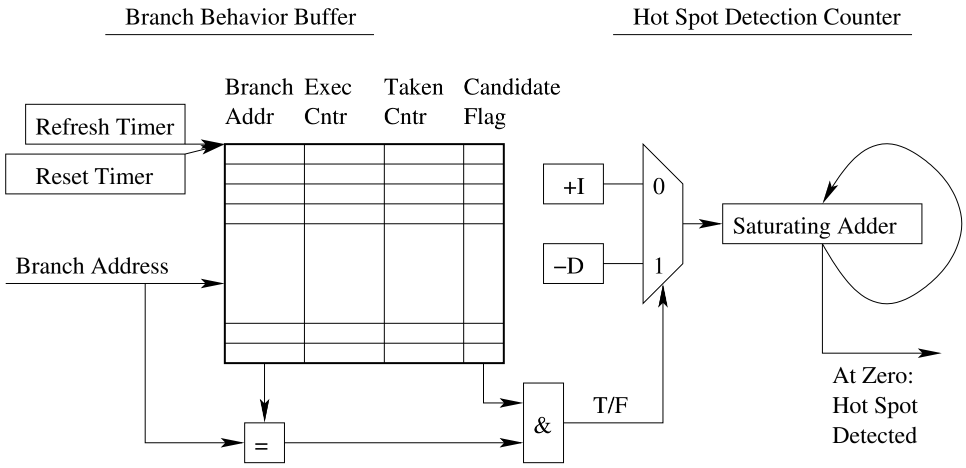


Figure 3.1 Hot Spot Detector.

is either missing from the BBB or in a BBB entry on a provisional basis is called a *noncandidate branch*. If a new static branch executes frequently enough before the end of the refresh interval, it is marked as a *candidate branch*. At the end of each refresh interval, noncandidate branches have either been promoted to candidate branches or are eliminated from the table by the refresh to make room for other branches that may be more important. Once a branch becomes a candidate branch, it is no longer reviewed at the refresh intervals, and is locked into the table for longer-term profiling. After several refresh intervals, the entries should be populated by the most frequently executed branches.

The Hot Spot Detection Counter is designed as a simple up/down saturating counter that counts up by I when noncandidate branches are executed and down by D for candidates. By choosing the I and D value, one determines the cumulative level of importance required before a set of candidate branches will trigger a hot spot detection [8]. For example, with I set to 2 and D set to 1, at least two thirds of the retired branches must be candidate branches for the HDC to proceed downward. If this happens consistently, HDC will quickly reach zero, indicating that the candidate branches have consistently accounted for the required minimum percentage of the dynamic branches. Thus, a hot spot detection is signaled. All branches which were candidate branches at the time of detection are chosen as *hot spot branches*. The hot code region defined by these branches is called a *hot spot*.

All of the branches (candidates and noncandidates) in the BBB are examined at the end of a second, much longer review period referred to as the *reset interval*. If the reset interval is reached without triggering a detection, the BBB is flushed and all branches become noncandidates once again. This condition indicates that control-flow is not currently constricted to a consistently behaving set of branches.

3.2 Previous Hot Spot Monitoring Mechanisms

In order to monitor and control profiling by the Hot Spot Detector proposed in [26], an additional hardware structure was used, called the *Monitor Table*, shown in Figure 3.2. The operation of this table was similar to that of the BBB; however, the monitor table considered only of a tag array of branches with one entry for every branch in every hot spot that was thus far detected. The Monitor Table enabled the profiling system to detect when a high percentage of the recently executed branches were outside of all of the previously detected hot spots. In this way, it determined whether a new hot spot detection was necessary.

In the Monitor Table, the addresses of all hot branches are stored in its tag array. Each retiring branch's address is looked up in this array to determine if the current execution is

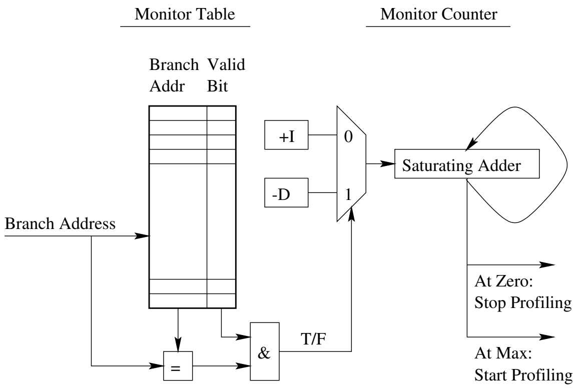


Figure 3.2 Monitor Table.

within a hot spot. If a valid entry for a retiring branch is found, a saturating counter called the *Monitor Counter* is decreased by the decrement value *D*. Otherwise, if no valid entry is found, this counter is increased by the increment value *I*. As in the HDC described in Section 3.1, the values for *D* and *I* determine the threshold ratio of hot spot to non-hot spot branches. Merten et al. found that by using this Monitor Table hardware, very infrequent but intense profiling by the BBB could find a small number of hot spots that encompassed the overwhelming majority of execution of the program.

In subsequent work, the HSD was used to drive run-time optimization [8],[18],[27]. Like the other dynamic optimization systems mentioned in Section 2.2, optimization was performed on individual traces. Using the HSD, a set of traces were formed from the execution seen within hot spots. The detection process in the HSD was enabled by excessive execution outside of these traces, transitioning between traces from different hot spots, or transitioning between the traces and original code.

To meet the goals of the system detailed in this work, it is not sufficient to determine only that execution is mostly limited to instructions that have been detected in *some* previously detected hot spot. Since this thesis targets an off-line transformation fed by the profiling of an unaltered program, monitoring the transitions from extracted hot spots for the purpose of controlling future detection is also not a practical solution. Instead, the detection of the next hot spot should be enabled whenever program execution has transitioned into a new phase of execution. Different phases of execution may actually share the same static instructions, and such phases may be missed by controlling hot spot detection using the Monitor Table. This work desires to detect a new hot spot whenever the control-flow behavior of a program has changed, even if the working set of branches for the program has not changed. To accomplish this and to better support use of the BBB for off-line post-link optimization, an extension to the BBB is proposed.

3.3 Enhanced BBB for Continuous Profiling

The enhanced BBB proposed in this work is presented in Figure 3.3. Two additional flags, the *previous candidate taken* flag and the *previous candidate not taken* flag, are shaded. An added counter, the *evicted candidate counter*, is also shaded. By utilizing these single-bit flags and the additional counter, the BBB can easily monitor whether the currently detected hot spot differs from the last hot spot detected, and will signal a real detection only when a transition between phases has occurred.

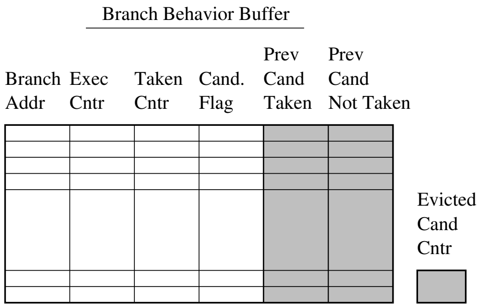


Figure 3.3 Enhanced Branch Behavior Buffer.

Using this extended BBB, profiling for hot spots proceeds for the most part as proposed by Merten [26]. Each time a hot spot is detected that differs from the previously detected hot spot, the previous candidate taken and not taken flags are set accordingly for each BBB entry that corresponds to a hot spot branch. For heavily taken branches (those whose taken counter is at least 70% of their executed counter), the previous candidate taken flag is set, and for strongly not taken branches (those whose taken counter is less than 30% of their executed counter), the previous candidate not taken is set. For the other unbiased hot spot branches, both the taken and not taken flags are set. Finally, the hot spot branches for the current detection are recorded by operating system software. Then, all of the entries for noncandidate branches are cleared, and the executed and taken counters and the candidate flag for all entries are cleared, leaving the candidate branches in the table, now marked as a noncandidate branch. The process of detection is then restarted, this time with the buffer primed with the addresses of the hot spot branches from the last hot spot detection. These entries remain flagged as previous candidates using the two new flags. If they are subsequently removed from the BBB under the most frequently used policy, the evicted candidate counter is incremented. Such branches instead could be locked into the BBB, but doing so might increase contention for entries and might prevent the profiling of an important branch. If the reset interval elapses before a hot spot is detected, the entire BBB is cleared, as in the original scheme, forcing the next hot spot detection to be different than the previous detection.

While an application continues within the same phase, the HSD will continue to repeatedly detect that phase. Branches that were part of the set of hot spot branches will quickly become candidates during the next detection, indicating that no phase transition has occurred. Once a phase change does occur, however, it will be detected in one of three ways:

- Several previous candidate branches will not become candidates
- Several entries which were not previously allocated to candidates branches will become candidates
- The direction bias of one or more branch will change

All of these conditions can be distinguished by comparing the sets of candidate branches and previous candidate branches. The first three conditions can be determined by the percentage change in the hot spot branches, which is determined as

$$percent\ change\ in\ hot\ branches = \frac{number\ of\ changed\ candidates}{number\ of\ previous\ candidates} \tag{3.1}$$

The number of changed candidates includes those branches which were previously hot spot branches but are not candidate branches at the current detection and those branches which were not previously hot spot branches but are now candidate branches. The number of changed candidates which still have entries in the BBB is computed as

$$BBB\ changed\ candidates = \sum_{all\ entries} (candidate\ flag \oplus previous\ candidate) \tag{3.2}$$

Where each entry can be determined to be a previous candidate by

$$previous\ candidate = (prev.\ cand.\ taken\ flag \vee previous\ cand.\ not\ taken\ flag) \tag{3.3}$$

However, the changed candidates in the BBB do not account for hot branches which have been evicted from the BBB through the least-frequently used replacement policy. The total number of changed candidates includes these evicted candidates as well:

$$num.\ of\ changed\ candidates = BBB\ changed\ cand. + evicted\ cand.\ count \tag{3.4}$$

Finally, in the calculation of the percent change of hot branches, the number of previous candidates is

$$\begin{aligned} &number\ of\ previous\ candidates = \\ &\qquad \sum_{all\ entries} (previous\ candidate) + evicted\ candidate\ count \end{aligned} \tag{3.5}$$

If the percent change in hot branches is greater than a set threshold, a new hot spot detection is triggered; if the difference is too small, the hot spot currently detected has not changed from the previous detection. Note that because of the way the percent change in hot branches was defined, a 100% change means that, for each branch in the original hot spot, the current hot spot either includes a unique, additional branch or does not include one of the original branches. Under this definition, the difference between two hot spots can be greater than 100%.

The final phase-change condition, that of changes in the direction bias of hot spot branches, is also found from the previous candidate taken and not taken flags. For each entry, a change in direction bias is seen as

$$changed\ bias =$$
$$(taken \wedge prev.\ cand.\ not\ taken\ flag) \vee (not\ taken \wedge prev.\ cand.\ taken\ flag) \quad (3.6)$$

The total change is direction bias is the sum of these changes:

$$total\ changed\ bias = \sum_{all\ entries} changed\ bias \quad (3.7)$$

If the number of branches with a change in bias surpasses a threshold, a new phase is also considered to have been detected.

After each detection of an unchanged hot spot, only the entries marked as previous candidates are retained, while the remaining entries are completely cleared along with the Hot Spot Detection Counter. Again, all of the executed counters, taken counters, and candidate flags are cleared as well. In this way, the BBB is initialized for the next hot spot detection with the candidate branches from the original detection. Additionally, the value of the evicted candidate counter is maintained. If instead, the previous candidate taken and not taken flags were updated to the currently detected set of candidate branches, or if the evicted candidate counter was reset at each detection, a gradual phase shift might be overlooked.

A portion of a six-entry BBB is shown in Figure 3.4 to demonstrate the possibility for this problem. A sequence of four hot spot detections is shown in (a)-(d) assuming instead that the previous candidate flag is upset at each detection. The initial detection of a hot spot, shown in (a), includes four candidate branches, labeled *A*, *B*, *C* and *D*. At the time of the second detection, shown in (b), the BBB includes a fifth candidate branch *E*. The percent change in hot branches for this detection is $1/4 = 20\%$. If the previous candidate flags are then updated to include *E*, the percent change at the next detection shown in (c), with the failure of *A* to become a candidate, is $1/5 = 20\%$. Again, assuming the Evicted Counter were to be cleared, the addition of candidate *F* in (d) results in only a 25% change. However, if the change with respect to the initial detection (a) is instead considered, a $3/4 = 75\%$ change is seen. If the change threshold is set above 25% this large phase shift that occurred gradually over four detections will not be seen.

Such gradual transitions are somewhat infrequent, as previous studies have shown that phase transitions are generally abrupt and pronounced [24],[28]. Figure 3.5 shows the number of unique hot spot detections as the threshold for the change in hot branches is varied. Note that the scale of Figure 3.5 is logarithmic. In general, as the threshold is raised, the number of different hot spots found decreases; however, in a few instances the number of detections does not monotonically decrease with respect to an increasing threshold. The change in this threshold affects both the time at which hot spots are detected and the branches included in

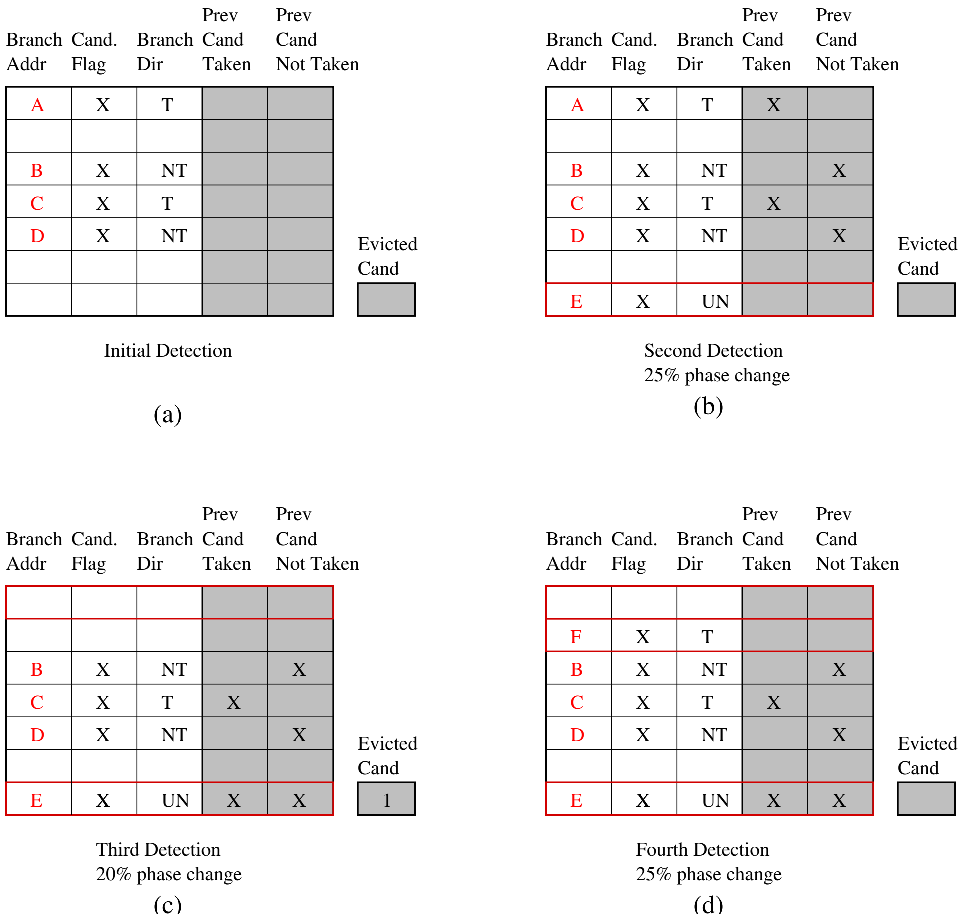


Figure 3.4 Gradual hot spot transition.

them at detection time. These slight differences can cause small variations in the number of hot spots detected.

For a very small threshold, a great number of very similar hot spots are detected. Once this threshold becomes more reasonable, the change in the number of new detections becomes rather small. For almost all benchmarks, increasing the threshold beyond 35% yields little reduction in the number of unique hot spots. This corroborates the claims of previous work that phase detection is largely independent of the phase change threshold.

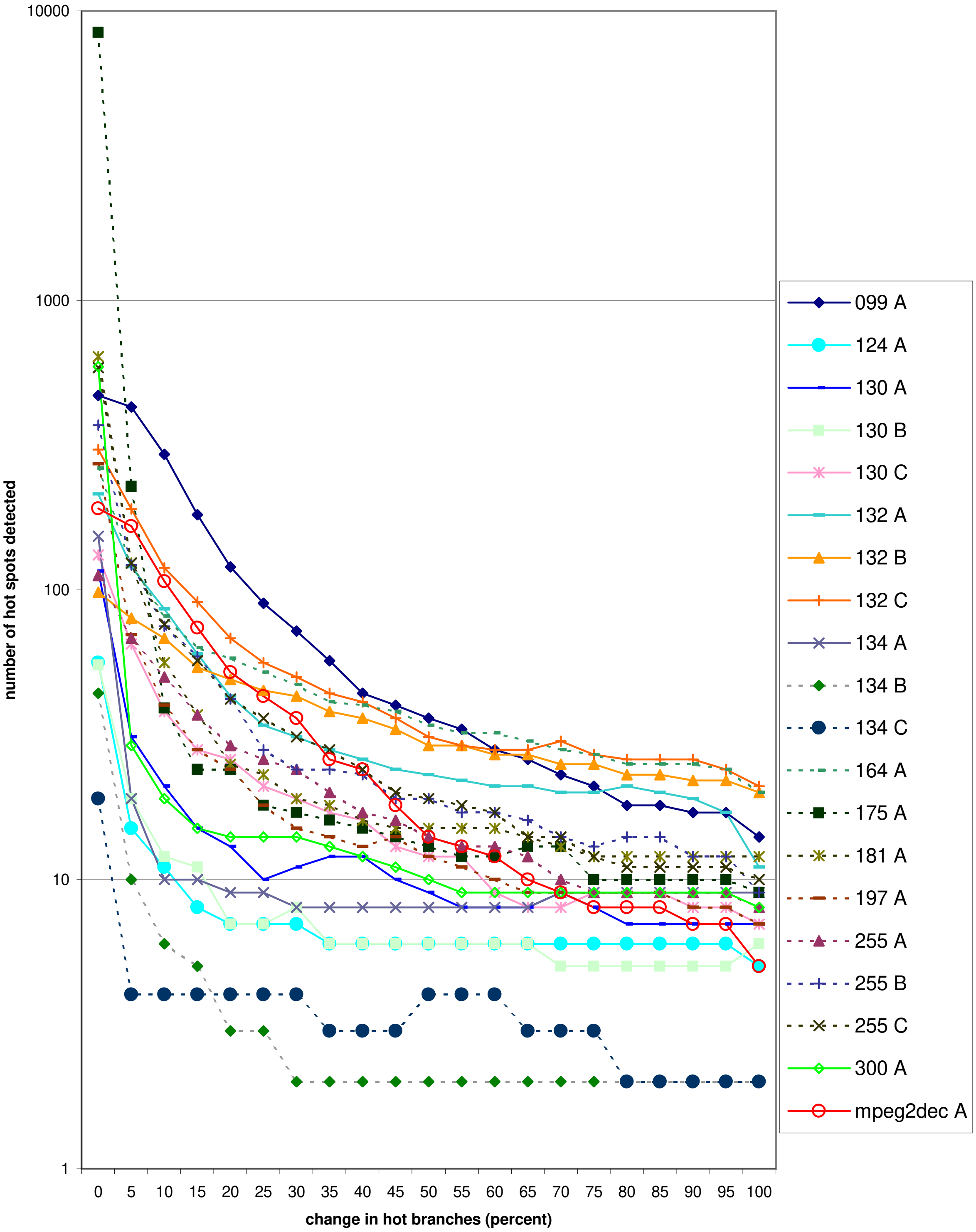


Figure 3.5 Effect of hot branch threshold on unique hot spot detections.

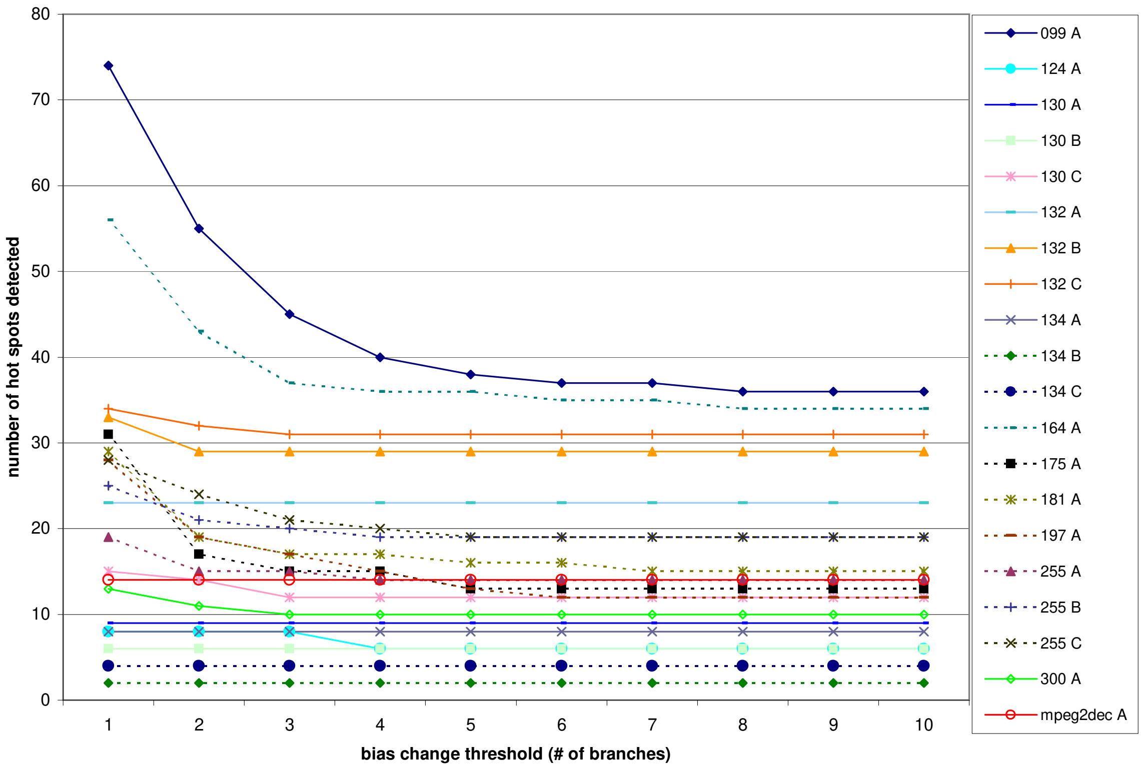


Figure 3.6 Effect of branch bias threshold on unique hot spot detections.

Figure 3.6 demonstrates the number of hot spots found to be unique because of the bias change threshold. These detections were all performed with the threshold for the percent change in hot branches set to 50%. Branch bias only has an effect on differentiating unique hot spots for a threshold of a small number of branches. With such a small threshold, several unique hot spots are distinguished. Thus, for almost all benchmarks, the number of branches that change bias is small (as will be further examined in Section 5.2), but the number of unique phases defined by these changes is significant.

Though this enhancement to the BBB allows the Hot Spot Detector to be used as a continuous profiling device, the HSD will still be signaling the detection of a new hot spot whenever a phase transition occurs. The majority of applications have several phases which are repeated through out the program's execution. The mechanism that processes each phase detection will have to either record a detection for every phase transition, or eliminate nonunique hot spots when their detection is signaled. The detection results using a percent change in hot branch threshold of 50% and a changed direction bias threshold of 1 branch are shown in Table 3.1. For each benchmark input, the total number of hot spot detections, the total number of hot spots which are repeats of earlier detected hot spots, and the number of detections which are repeats of the last hot spot detected are given. As shown in Table 3.1, the enhanced BBB does eliminate a majority of the nonunique hot spot detections. However, the portion of such detections eliminated varies greatly between benchmarks and the number of nonunique detections for some benchmarks is still large. Further elaboration on the benchmarks evaluated and the experimental setup is found in Chapter 5.

3.4 Hot Spot Phase Signatures

Rather than filtering out nonunique hot spots by comparing every branch within them, as was done in this study, a technique like the *working set signature* [24] could be extended to produce a *hot spot signature*. As previously mentioned in Section 2.3.2, a working set signature is a lossy-compressed working set representation. It is generated by hashing portions of the tags of cache lines included in the working set to bits in vector. This bit-vector serves as a signature for the given working set and can be used to make comparisons against other working sets. In a similar way, a hot spot signature can be generated by hashing the hot spot branches to bits in a

Table 3.1 Hot spot detections with enhanced BBB.

Benchmark	Input	HS Detections	Non-unique Detections	Sequencially Repeated Detections	% of Non-unique Detections
099.go	A	579	505	117	23.2
124.m88ksim	A	1321	1313	1305	99.4
130.li	A	1925	1916	1361	71.0
	B	380	374	210	56.1
	C	6409	6394	3969	62.1
132.ijpeg	A	7058	7035	3649	51.9
	B	539	506	352	69.6
	C	5176	5142	2804	54.5
134.perl	A	18758	18750	17616	94.0
	B	202	200	200	100.0
	C	56	53	44	83.0
164.gzip	A	33370	33314	28233	84.7
175.vpr	A	10381	10351	10213	98.7
181.mcf	A	1747	1718	1123	65.4
197.parser	A	3454	3428	3130	91.3
255.vortex	A	232	210	121	57.6
	B	1488	1460	602	41.2
	C	3471	3443	2180	63.3
300.twolf	A	1473	1460	1216	83.3
mpeg2dec	A	1021	1005	686	68.3

vector. A cheap comparison of these signature vectors will determine if a detected hot spot is in fact unique. To distinguish between phases that differ in branch direction bias, the directional bias of each hot spot branch could also serve as an input to the hashing function. The generation of a hot spot signature is shown in Figure 3.7. As long as a hashing function is effective at mapping different hot spot branches to different bits in the signature, the likelihood of two different hot spots generating the same signature is very low, and therefore the probability of missing a unique hot spot would be low. Instead of the enhanced BBB presented in Section 3.3, such signatures might be used to eliminate sequentially identical hot spot detections as well as the nonsequential ones. However, the overhead of computing a signature for each detection makes the ability to automatically weed out detections of hot spots which have not changed from the last detection a more attractive solution.

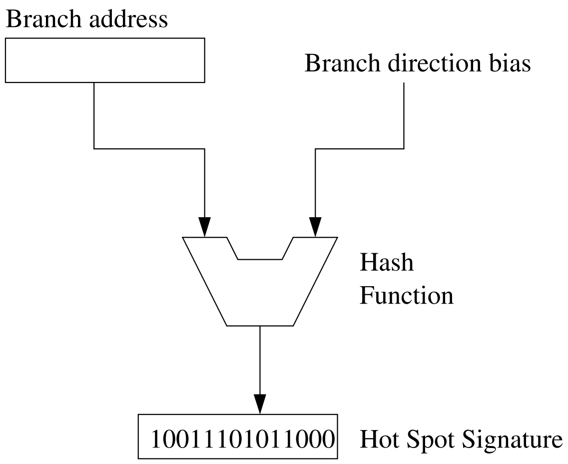


Figure 3.7 Calculation of proposed hot spot signature.

CHAPTER 4

HOT REGION FORMATION

The Vacuum Packing process is designed to use the continuous hot spot profiling described in Chapter 3 to identify important code regions associated with program phases and extract them for the purpose of code optimization. As shown in Figure 4.1, the formation process conceptually consists of three steps, profiling and identification of hot branches, region formation, and region extraction and optimization. During the first step, a profiling mechanism monitors the execution of a program and selects a collection of static branches referred to as hot spot branches as shown in Figure 4.1(a). These instructions are the hot branches associated with a phase of program execution. Information accumulated in the monitoring mechanism, including candidate branch executed and taken weights, is stored away for future processing.

The profiled program continues to execute until another phase is detected, at which point the information on another set of hot branches will be stored away. For evaluation in this work, a complete execution is performed for the purpose of hot spot profiling before any of the detected phases is further optimized. At the completion of the profiled program, a software mechanism processes the stored hot branch information.

In the second step, the stored information is combined with static program representation to form the input to the region formation algorithm. The algorithm selects a region for optimization and leverages the branch information to generate estimated execution frequencies of all

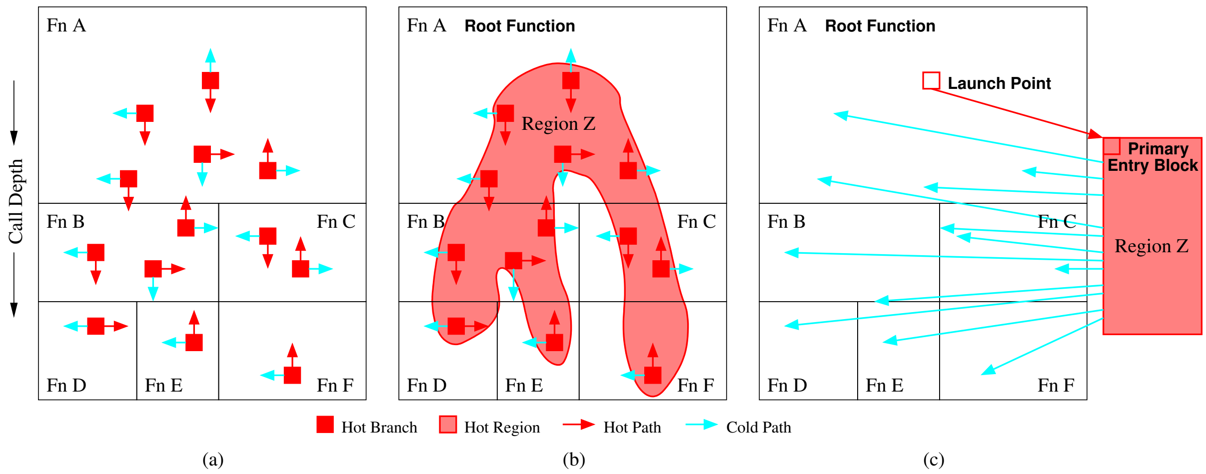


Figure 4.1 Region formation process over functions A-F, shown in call tree order. (a) Hot branches, with hot and cold directions. (b) Formed hot region. (c) Extracted and optimized region Z with cold links back to original code.

instructions in the region. Often these hot regions span function boundaries; in Figure 4.1(b), the hot region spans functions A, B, C, D, E, and F.

In the third step, an extraction algorithm assembles all the pieces of the hot region into a new, localized code region that can be conveniently handled by an optimizer. One physical region is formed for each program phase. Control transitions are established between the original program and the extracted region. Finally, control transitions are also established between extracted regions. The new code regions are packaged much like a function body so that optimization algorithms can process them similarly.

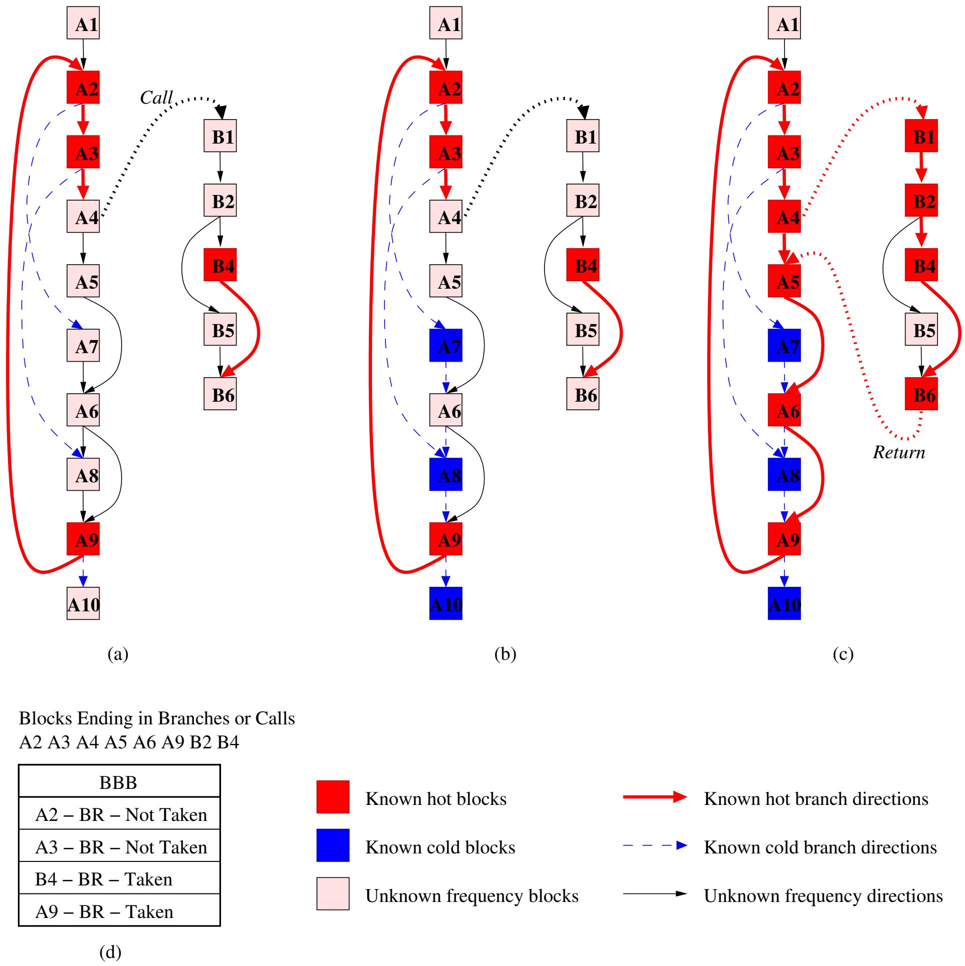
4.1 Step 1: Program Phase Detection

At each detection time, the BBB contains the set of candidate branches along with their executed and taken counts. The counts together minimally provide the taken ratio for the branch during the detection process. The executed weights can also be used to compare the relative significance of different branches within the same hot spot. However, under the approximated most frequently used policy, contention for table entries due to the finite table size and the table's limited associativity may force a static branch to begin profiling later in the detection process. This scenario may cause artificially lower weights compared to other branches in the hot spot, and in the worst case, prevent the branch from being tracked at all. In addition, the hardware counters tracking each branch saturate when the execute count reaches its maximum value. However, at saturation, the taken ratio for the branch is preserved.

Figure 4.2 details the hot region formation process. The profiled program consists of two functions, as shown in Figure 4.2(a). Basic blocks A_1 through A_{10} belong in function A and basic blocks B_1 through B_6 belong in function B. Figure 4.2(d) shows the result of hot spot detection. *For the purpose of this example* a very small, four-entry BBB is used. Since the working set of branches in the phase is much larger than the size of the BBB, only a portion of the branches are captured. In a realistic design, the captured execution of a phase would be expected to be much higher.

4.2 Step 2: Region Identification

The second step of the hot region formation algorithm identifies the hot region instructions of each phase based on the branch profile information provided by the BBB. The profile information available at this point consists only of a set of branches captured during the detected phase of execution along with their executed and taken counts. This information is used to



BBB
A2 – BR – Not Taken
A3 – BR – Not Taken
B4 – BR – Taken
A9 – BR – Taken

Figure 4.2 Region formation. (a) Branch Behavior Buffer profile. (b) Initialization of hot spot branches and their blocks. (c) Propagation of the cold arc information. (d) Propagation of the hot arc and block information.

select the optimization region and will be used later to determine usable profile weights for control flow within the region. In an attempt to provide an optimized piece of code for each important phase of program execution, each hot spot detected is considered separately. The chosen regions are then expanded using *inference* (described in Section 4.2.2) and *heuristics* (described in Section 4.2.3) to include additional blocks and their corresponding flow arcs for several reasons:

- A hot path may temporarily diverge into several paths which do not individually meet the threshold for being hot. If these paths later converge back into hot blocks, including them will improve the connectivity of the selected regions.
- Techniques using hardware counters[1] to determine profile weights provide only an approximation of the actual profile due to the randomness of sampling and the limited number of counters. For this reason, a certain amount of misleading and missing information must be tolerated.
- Even though exits from the region inferred from the Hot Spot Detector profile are uncommon, it is desired to further reduce the number of them by opportunistically including infrequent paths when inclusion is associated with little or no cost.

In Figure 4.2(a), the profile information for the phase of the example covers only four of the eight hot branches due to limited number of BBB entries. As mentioned above, a real design would not detect such a small percentage of hot spot branches; however, a very large program might have a working set of branches that exceeds the available entries. Thus, to be effective the algorithm must be tolerant of some branches missing from the buffer. To achieve this tolerance, a phase of inferring the importance of blocks is followed by a phase of heuristically including other blocks in the region to be optimized.

4.2.1 Hot spot blocks

To begin the region identification process, the hot spot branches and the blocks containing them are initialized. This initialization is performed according to Rule 1 and Rule 2 from Figure 4.3. Under these rules, blocks are assigned temperatures, initial weights, and a taken probability. Flows are assigned a temperature as well. The algorithm for this initialization is given in Figure 4.4.

[1]Similar problems also arise when using software sampling.

Rule 1 *Initialize each basic block containing a hot spot branch:*

- $blocktemperature = Hot$
- $initial\ weight = executed\ count$
- $taken\ probability = \frac{taken\ count}{executed\ count}$

Rule 2 *Initialize for each flow of a hot spot branch:*

- $flowtemperature = \begin{cases} Hot & \textit{(direction > (25\% $\cdot$ executed count))} \bigvee \textit{(direction > cand. threshold)} \\ Cold & \textit{otherwise} \end{cases}$

Rule 3 *Infer using relationships between flows and blocks:*

Rule 3.1 *Infer temperature of flows in and out of blocks (Figure 4.5(a)-(b)):*

- *Any flow into (out of) cold blocks (a)* $\Rightarrow flowtemperature = Cold$
- *Only one unknown flow into (out of) a hot block where there are no hot flows into (out of) the block (b)* $\Rightarrow flowtemperature = Hot$

Rule 3.2 *Infer temperature of blocks (Figure 4.5(d)-(e)):*

- *All flows in or out are cold (d)* $\Rightarrow blocktemperature = Cold$
- *One flow in or out is hot (e)* $\Rightarrow blocktemperature = Hot$

Rule 4 *Expand region from hot blocks which are root basic blocks:*

- *Expand region into adjacent predicessor blocks until a hot block is reached*
  - *Never grow region into cold blocks*
  - *Discard growth which does not reach hot block in MAX_BRANCHES (e.g., 2) additional blocks*

Figure 4.3 Rules for selecting vacuum-packed region.

In the example in Figure 4.2, blocks A_2, A_3, A_9, and B_4 contain hot spot branches according to the BBB record in Figure 4.2(d). These blocks are initialized as hot, and the control flows out of these blocks are initialized as hot or cold according to their directional bias. If a hot spot branch is taken more than 75% of the time, the control-flow arc for that direction is hot. Also, if the weight of a direction exceeds the candidate counter, that direction is additionally marked

```
Initialize_Hot_Spot(hot_spot_branch_list)
{
 1: For each branch ϵ hot_spot_branch_list {
 2:     block = basic block containing branch;
 3:     block.temperature = Hot;
 4:     block.initial_weight = branch.executed_count;
 5:     block.taken_probability = branch.taken_count/branch.executed_count;
 6:     If ((branch.taken_count > 75% · branch.executed_count) and
            (branch.executed_count - branch.taken_count < CANDIDATE_THRESHOLD)) {
 7:         branch.taken_flow.temperature = Hot;
 8:         branch.fall_through_flow.temperature = Cold;
 9:     }
10:     Else If ((branch.taken_count < 25% · branch.executed_count) and
            (branch.taken_count < CANDIDATE_THRESHOLD)) {
11:         branch.taken_flow.temperature = Cold;
11:         branch.fall_through_flow.temperature = Hot;
12:     }
13:     Else {
14:         branch.taken_flow.temperature = Hot;
15:         branch.fall_through_flow.temperature = Cold;
16:     }
17: }
}
```

Figure 4.4 Initialization of hot spot branches and their blocks.

hot. Remaining directions are left as cold. For example, according to the taken and executed counters in the BBB entry for block A_9, the branch ending this block is heavily biased in the taken direction. Lines 7 and 8 of the algorithm in Figure 4.4 mark the taken flow for this branch hot and mark the fall through flow arc cold.

4.2.2 Inferred hot and cold blocks

The temperature of some blocks that do not contain hot spot branches and the temperature of some flows in and out of such blocks are iteratively inferred according to Rule 3.

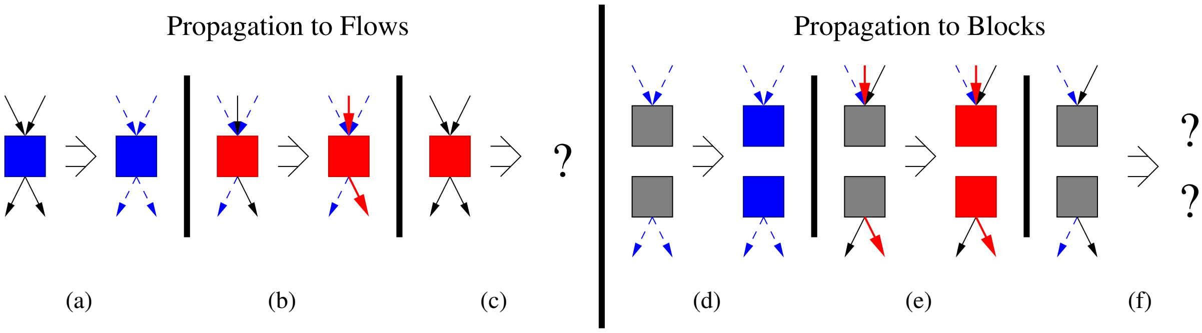


Figure 4.5 Inference rules. (a)-(c) Propagation to flows. (d)-(f) Propagation to blocks.

The algorithm for this inference is given in Figure 4.6. This algorithm assumes that the *flow_work_list* and *block_work_list* have been initialized to the hot flow arcs and hot blocks, and *cold_flow_work_list* and *cold_block_work_list* have been initialized to the cold flow arcs and cold blocks found through the application of the algorithm in Figure 4.4.

From the list of blocks, the temperature of flows in and out of these blocks is inferred as shown in Figure 4.5(a) and 4.5(b). All arcs in and out of cold blocks are also cold. If only one control-flow arc in or out of a hot block is not cold, that flow is inferred to be hot.

From the list of flow arcs, the temperature of blocks at the head or tail of these flows is infered. The cases where this inference can be performed are shown in Figures 4.5(d) and 4.5(e). If every flow in or out of a block is cold, that block must also be cold. If any flow in or out of a block is hot, that block must be hot as well.

In Figure 4.2(b), several blocks are inferred to be cold from the initial list of cold flows. Blocks A_7 and A_{10} are assigned a cold temperature on line 10 of Figure 4.6, since they have only a single incoming arc, and these arcs are all cold.

```
Infer_Hot_region(flow_work_list, block_work_list, cold_flow_work_list, cold_block_work_list)
{
For each flow ϵ flow_work_list {
    flow.source_block.temperature = Hot;
    flow.destination_block.temperature = Hot;
    List_insert(block_work_list, flow.source_block);
    List_insert(block_work_list, flow.destination_block);
    List_remove(flow_work_list, flow);
}
For each flow ϵ cold_flow_work_list {
    If (all flows into flow.destination_block are cold) {
        flow.destination_block.temperature = Cold;
        List_insert(cold_block_work_list, flow.destination_block);
    }
    If (all flows out of flow.source_block are cold) {
        flow.source_block.temperature = Cold;
        List_insert(cold_block_work_list, flow.source_block);
    }
    List_remove(cold_flow_work_list, flow);
}
For each block ϵ block_work_list {
    If (only one flow into (or out of) block is not cold) {
        flow = flow into (or out of) block that isn't cold;
        flow.temperature = Hot;
        List_insert(flow_work_list, flow);
    }
    List_remove(block_work_list, block);
}
For each block ϵ cold_block_work_list {
    For each flow into and out of block {
        flow.temperature = Cold;
        List_insert(cold_flow_work_list, flow);
    }
    List_remove(cold_block_work_list, block);
}
}
```

Figure 4.6 Inference growth of vacuum-packed region.

Figure 4.2(c) shows the further propagation of the inference. From the initial four hot spot branches, several additional blocks have been inferred as hot. Since the flow from A_3 to A_4 is hot, the temperature of A_4 is set to hot on line 3 of the algorithm in Figure 4.6. Similarly, B_6 is also hot because of the hot arc between B_2 and B_4. The flow from A_6 to A_9 is assigned a hot temperature on line 22 of this algorithm, as the only other flow into A_9 is cold. Similarly, the temperature of the flow from B_2 and B_4 is also hot. Because B_1 is the target of a hot call in A_4, it is marked as hot as well. Subsequent iteration of this algorithm will find blocks A_5, A_6, and B_2 to be hot as well.

4.2.3 Heuristic hot region growth

Because of the potential for missing information, heuristic growth of the vacuum-packed regions is performed under Rule 4. *Root blocks* are are first found in the same way entry blocks will be found in Section 4.3.1. Backedges are removed from the control-flow graph, and blocks with no incoming arcs are selected. To elimiate entrances to the region, the heuristic growth aims to eliminate potential entry blocks by growing upward from root blocks, through blocks with no information, finally connecting control to preceeding hot blocks.

4.3 Step 3: Region Construction and Optimization Support

Once a hot region of code has been detected, identified, and grown, the region construction and optimization phase begins. During identification and growth, a control-flow graph for each function in the region is marked with the hot and cold information. A call graph representing function calls relationships within the region iss also constructed.

4.3.1 Locating root functions and entry blocks

The call graph for the region is examined to find functions to serve as *root functions* from which all other functions included in the region can be reached. A call graph can be converted into a tree by braking all cycles in the graph by removing the backedge from each cycle. A root function is a node in such a tree without any callers. These root functions serve as seeds in the partial inlining process.

Some functions can be both root functions and inlinees. If disconnected pieces of a function have been selected as part of the hot region, that function is added to the list of root functions. When that function is used as a root function, these disjoint pieces can be safely included. If that function is used as an inlinee into another root function, the disjoint pieces are excluded to prevent side entrances into the regions from unknown contexts as destribed in Section 4.3.3. All other functions that are part of the phase will be partially inlined into one or more of these root functions. In a similar fashion, entry blocks are chosen for each function's control flow graph *(CFG)*. If CFG cycles are broken by removing a backedge, entry blocks are blocks without any predecessors. The original code locations corresponding to entry blocks into root functions will serve as launch points from original code into the regions.

In the earlier example in Figure 4.2, the root function for the program phase is A. The entry block for phase 1 can be any block in A since all blocks can be reached from any block in loop. As shown in Figure 4.7, a continuation of the earlier example, A_2 has been chosen as the entry block as it is the first block in program order with an incoming arc from outside the region.

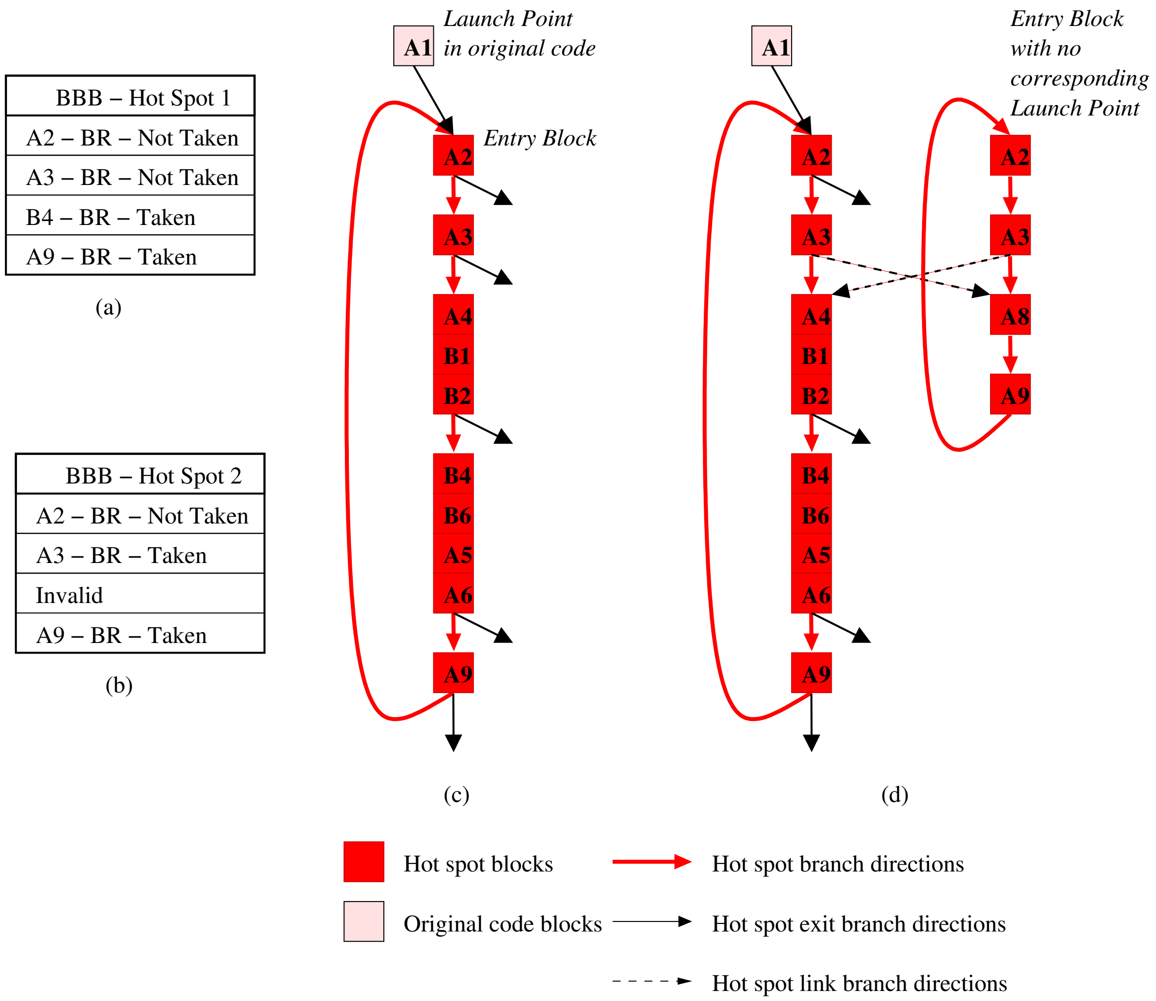


Figure 4.7 Extraction of vacuum-packed region. (a) BBB for hot spot detection for phase 1. (b) BBB for hot spot detection for phase 2. (c) Extraction of phase 1's region (formed in Figure 4.2). (d) Linking of phase 1's region with phase 2's region.

4.3.2 Maintaining dataflow

Each marked function is reduced to include only the instructions found in Step 2 to be part of the hot region. The register live ranges are maintained by creating a new basic block along each exit path and placing dummy consumer instructions for each register live across the exit. This new block is called an *exit block*. This allows the removal of the cold instructions without

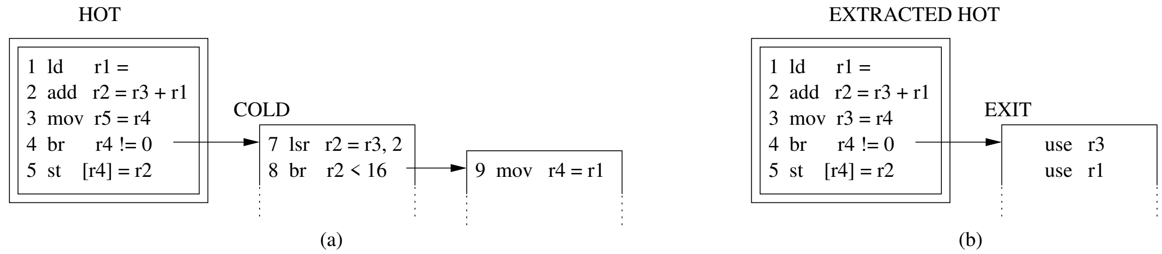


Figure 4.8 Maintaining dataflow. (a) Original code sequence. (b) Region after hot instructions have been extracted.

corrupting or complicating the formal dataflow analysis. Figure 4.8(a) shows a sequence of hot instructions with a branch to a sequence of cold instructions. The result of extracting the hot instructions and inserting representative exit blocks is shown in Figure 4.8(b). These blocks create an opportunity for migration of hot instructions whose results are only consumed along paths through an exit block.

4.3.3 Partial inlining

The inlining process successively progresses through root functions of the call graph producing individual subregions. Just prior to inlining, recursion in the call graph is detected and recursive arcs in the graph are eliminated. However, a copy is made of self-recursive functions to allow a single copy of a self-recursive function to be partially inlined into itself.

Partial inlining proceeds by finding an outgoing arc in the call graph from the root function to another intraregion function. The callee function's hot blocks and exit blocks, described in Section 4.3.2, are copied into the root function. The callee being inlined must have a hot prologue block and at least one contiguous hot path from the prologue block to the return block; otherwise, inlining of the callee is filtered. The parts of the callee reachable from the prologue are inlined as normal into the caller, while any other disjoint segments are discarded to avoid creating side entrances into the inlinee from unknown contexts. Finally, the callee function's outgoing call graph arcs are merged with the root function's arcs, and the callee function is removed from the outgoing arc set of the root function. The inlining process continues for this root function until its region call graph is exhausted.

4.3.4 Calculation of profile weights

Many aggressive optimizations require accurate profile information in order to effectively improve application performance. Unfortunately, while an initial executed count for the basic blocks which contain candidate branches has been determined, these weights are neither complete nor accurate for all of the blocks in the region. The BBB may be missing some important branches or may not have started tracking these branches in a timely manner due to conflicts in the buffer. In addition, since hardware counters are used, heavily executed branches can saturate the executed counter. The problems are not unique to the BBB, as other profiling systems that use techniques like sampling or statically estimated profiles suffer from very similar problems.

4.3.4.1 Inferred block weights

To compute profile weights, Vacuum Packing relies upon the control-flow probability of each branch in the region, as will be detailed in Section 4.3.4.2. For blocks whose weights or taken proabilities are not known, a guess for this probability will have to be made. However, in many cases, a good approximate probability can be determined by examining the flow weights

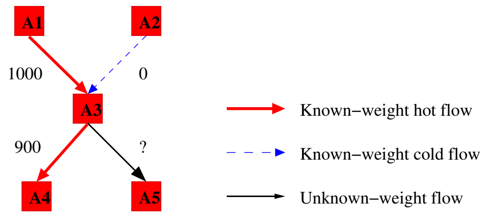


Figure 4.9 Inferring profile weight from surrounding blocks.

coming in and out of a block. If the weight of only one flow in and out of a block is missing, that weight can easily be estimated by conserving flow through the block.

As an example, consider the portion of the control-flow graph in Figure 4.9. It is obvious in this case that the weight of the unknown arc from A_3 to A_5 can be estimated to be 100. Estimates made in this way will be significantly better than the guess otherwise made in Section 4.3.4.2.

4.3.4.2 Derived profile weights

Because of the possible late tracking of branches and the saturation of hardware counters, confidence in the taken probability of hot spot branches is much higher than in comparisons between the absolute execution counts provided by the BBB for such branches. For this reason, the strategy proposed in this thesis is to generate profile weights based primarily on the taken (or not taken) probability of hot spot branches. For candidate branches, their taken probability as computed by Rule 1 is used. Finally, for branches whose flow weights cannot be computed as in Section 4.3.4.1, an assumed weight will be used.

The initial weight of candidate blocks is considered only for the blocks that represent entry points into the region, as explained below. In order to generate complete profile weights for the region, three assumptions are made:

1. for unknown-weight branch with an outgoing hot flow:
   - $hot\ direction\ probability = \begin{cases} .9 & \text{if other flow is unknown} \\ .99 & \text{if other flow is cold} \end{cases}$
2. for noncandidate with no outgoing hot flow:
   - $taken\ probability = .5$
3. flow leaving a block is independent of previous flow up to that point.

By Assumption 3, the control-flow graph is considered to be a Markov chain [29] with the probability of going from any block to its successors determined solely by the taken (or or not taken) probability. A matrix **Q**, describing the probabilities of transition among blocks other than the epilogue block, is found. If basic block *i* has control flow to basic block *j*, then row *i*, column *j* of **Q** will contain the probability of that control flow. **M**, the *fundamental matrix* for this graph is given by $\mathbf{M} = (\mathbf{I} - \mathbf{Q})^{-1}$, where **I** is the identity matrix.

Row *i* of matrix **M** describes the expected number of times each block will execute when control starts at block *i*. The row for each region entry block is scaled using that block's initial weight. Since some IMPACT compiler optimizations are triggered by the absolute weight of a block, these weights are futher scaled so that the total profile weight of all entry point blocks is 10 000. The weights that result are used to drive optimization.

For the region in Figure 4.10(a), the profile weights for each block are less important because only the path through the phase has been included for optimization. As an example, however, consider the exit probabilities shown in in Figure 4.10(a). Since block B_2 does not

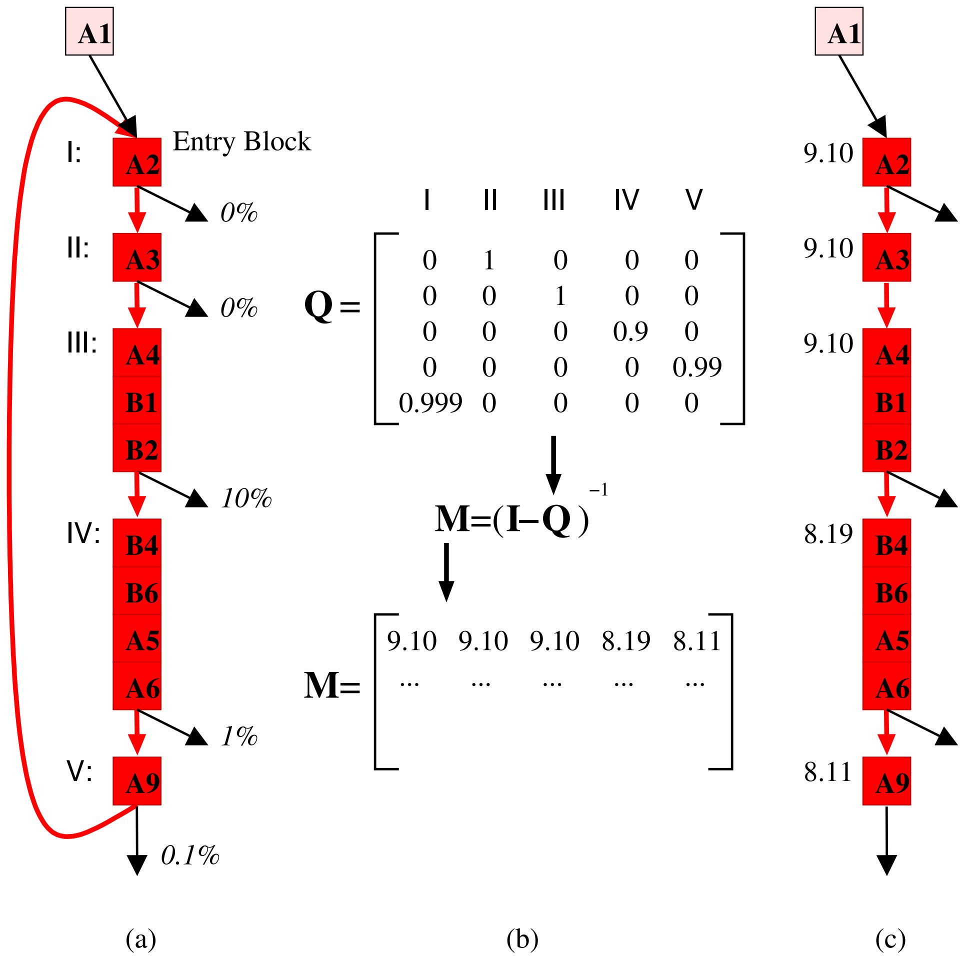


Figure 4.10 Deriving profile weight from branch probabilities. (a) Extracted hot spot region with branch probabilities. (b) Derivation of **Q** and **M** matricies. (c) Hot spot with derived block weights.

contain a candidate branche, the probability of the non-hot direction has been assumed to be 10%. The **Q** matrix coresponding to this graph is shown in Figure 4.10(b). For instance, the probability of going from block A_6 to A_9 is given in element $\mathbf{Q}_{45}$ (II $\Rightarrow$ III) as 0.99. Solving for **M** is shown in Figure 4.10(b), and the resulting profile derivation (before scaling) has been annotated to the corresponding blocks in Figure 4.10(c).

4.3.5 Region transitions

Each region is designed to contain all of the hot code needed for execution in a phase. To accomplish this goal, code replication is used to include customizable copies of this code into the regions. Along with copies of the selected pieces of called functions, the selected pieces of the root function are also replicated into each region. If all regions have disjoint root functions, then a one-to-one mapping exists between launch points and entry blocks. However, it is not uncommon for multiple regions to have the same root function, often due to a main driver outer loop that makes different calls during different phases. Thus, there may be no definitive location in the original code to launch into each distinct region. This scenario is evident from the two extracted phases in Figure 4.7(c) where the main loop in function A has two distinct execution patterns. The launch point is only able to link an original code block with a single region. For example, in *134.perl*, the driver loop `_eval()` makes different service calls based on the types of expressions being evaluated, such as math or string.

In the Vacuum Packing mechanism, a branch is placed at the launch point that redirects execution to the region. However, the target region of this branch may not always be the region formed for the current dynamic phase of execution. In one solution, the launch point branch could be dynamically modified to point to the expected best region. However, some mechanism would need to make the modification based upon some prediction. One indicator of a phase transition is execution along a region exit path that is a hot path in another region. While a monitoring code snippet could be introduced along the exit path to feed some sort of dynamic predictor, an easy, static solution is to link the side exits from one region to corresponding points in another. An example of this linking process can be seen in Figure 4.7(d). Branch A_3 falls through to A_4 in the left region and takes to A_8 in the right region. Instead of A_3 in the left region branching to the *original* A_8, a link to the right region has been installed. If this transition truly represents a phase change, then execution will continue in the right region. The algorithm for detecting and linking along branches with varied behavior based on the BBB profiles is outlined below.

- For each static branch that is in at least one BBB profile:

  Perform pairwise comparison of the static branch in all BBB profiles

  If $abs(difference\ in\ taken\ fractions) > MAX\_VARIATION$ (e.g. 0.1)

  Indicate branch has varied behavior

- When a root function is created, or function is partially inlined, scan instructions. for any indicated branches

  If found, keep pointer to scanned instruction in a list associated with the original indicated branch

- For each root or inlined instance of original indicated branch

  If branch taken (or fall through) direction is a region exit

  Link to taken (or fall through) target of another instance if region calling contexts are identical

Caution must be exercised to ensure that the calling contexts from the root function to the link sites within both regions are *identical* since execution could traverse into incorrect functions along inlined partial returns in the target region. In Figure 4.11, two vacuum-packed regions have been formed and linked. The function *B* has been inlined into both regions, but

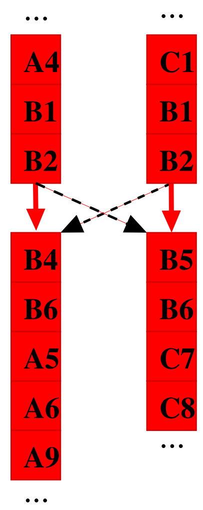


Figure 4.11 Invalid linking of hot spots inside different contexts.

this inlining has been done into different contexts. The two regions are improperly linked because execution could begin in function A on the left, travel into and along the link at the end of block B_2 into the right region, and return incorrectly into function C.

In some cases, branches may be cold and cause exits to original code in one region, but might be hot in several others. Our strategy is to select the next sequential region (based on detection order) that contains the branch in the desired direction. The next sequential region is a logical choice because a change in behavior of this branch may have signaled a new hot spot during the detection process. It also sets up the possibility of a round robin search of the hot spots through several side exits until the matching region is found. This strategy helps execution find the best region as opposed to the most general region. Other options include selecting the region with the most biased version or possibly the most heavily executed version.

CHAPTER 5

RESULTS

5.1 Experimental Setup

Listed in Table 5.1 are benchmarks representing a wide variety of application types selected from SPEC CPU95, SPEC CPU2000 (including shortened reference inputs from the University of Minnesota(UMN) [30]), and MediaBench [31] to test the performance of the region extraction. The benchmarks were each compiled using the IMPACT compiler [32] with control-flow profiling, profile-driven inlining [33], classical optimization, pointer aliasing analysis [34], and instruction scheduling with control speculation.

The performance measurements reported in this work were generated by Linterpret, a custom software emulator that performs cycle-by-cycle full-pipeline simulation of each instruction shown in Figure 5.1. This emulator fully accounts for the effects of branch prediction, wrong path execution, cache utilization and pollution, varying memory latency, interlocking, and bypassing. It directly simulates the execution of the IMPACT compiler's machine-level internal representation for the IMPACT EPIC architecture [35]. A complete description of the structure of the IMPACT compiler and its internal representation can be found in [36].

The architecture modeled consists of a 10-stage EPIC pipeline containing five functional unit types (Integer ALU, FP, Long Latency FP, Memory, and Control). The simulations include

Table 5.1 Benchmarks and inputs used in experiments.

Benchmark	Inputs	Instr
099.go	A:SPEC Train	337 M
124.m88ksim	A:SPEC Train	89 M
130.li	A:SPEC Train	122 M
	B:6 Queens	32 M
	C:Reduced Ref.	362 M
132.ijpeg	A:SPEC Train	1094 M
	B:Custom Faces	57 M
	C:Custom Scenery	320 M
134.perl	A:SPEC Train 1	1512 M
	B:SPEC Train 2	28 M
	C:SPEC Train 3	8 M
164.gzip	A:Reduced Train	1902 M
175.vpr	A:SPEC Test	1012 M
181.mcf	A:SPEC Test	105 M
197.parser	A:UMN_sm_red	178 M
255.vortex	A:UMN_sm_red	63 M
	B:UMN_md_red	315 M
	C:UMN_lg_red	886 M
300.twolf	A:UMN_sm_red	167 M
mpeg2dec	A:Media Train	99 M

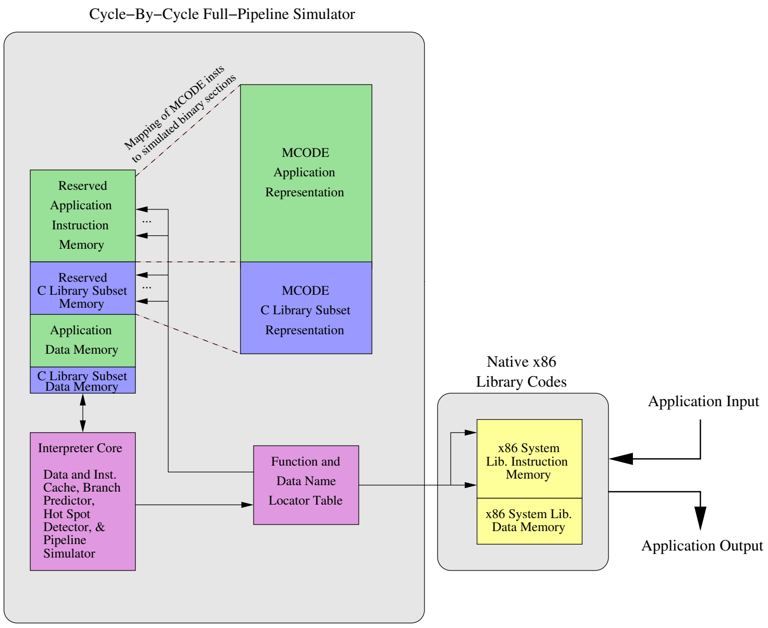


Figure 5.1 Linterpret: an interpretation-based, cycle-by-cycle simulation.

Table 5.2 Simulated EPIC machine model.

Parameter	Setting
Instruction issue	8 units
Integer arithmetic and logic unit	5 units
Floating-point arithmetic unit	3 units
Memory unit	3 units
Branch unit	3 units
Branch predictor	10-bit history gshare 3 predictions per cycle
BTB size	1024 entry
RAS size	32 entry
Branch resolution	7 cycles
LD/ST buffer size	8 entry each
L1 data cache	64KB
L1 instruction cache	64KB
Unified L2 cache	512KB
BBB associativity	4-way
Num BBB sets	512 sets
Candidate branch thresh	16
Refresh timer interval	8192 branches
Clear timer interval	65536 branches
Hot spot detect cntr size	13 bits
Hot spot detect cntr inc	2
Hot spot detect cntr dec	1
Exec and taken cntr size	9 bits

a multilevel memory hierarchy, and branch and return address prediction. Table 5.2 reflects the architectural parameters chosen for the evaluation system.

5.2 Evaluation

A primary concern for a post-link optimizer is the percent of program execution that is spent in the code that is optimized. This is even more important for systems that are generating code at run time. To measure this percentage, the emulator tracked the number of executed instructions that originated from within the extracted regions and those from the original code base. Higher quality regions lead to a greater percentage of execution from the hot spot regions. Figure 5.2 shows the percentage of all dynamic instructions originating from an extracted region. The first (lighter) bar for each experiment shows this percentage when

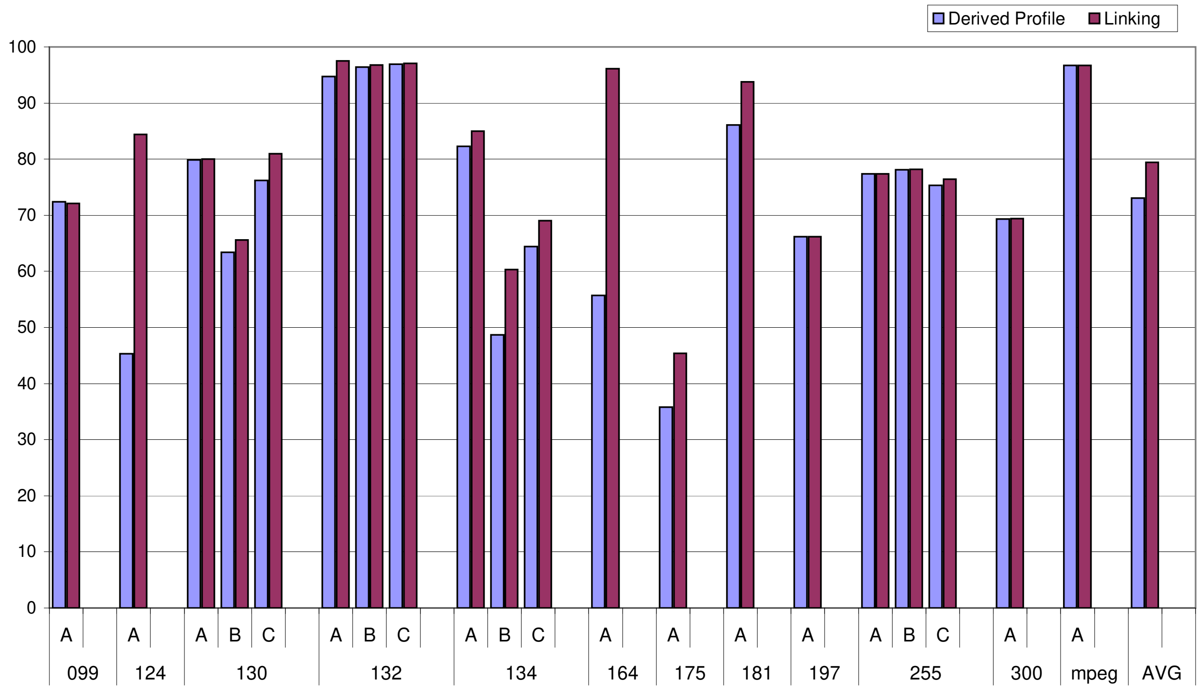


Figure 5.2 Percent of dynamic instructions from within extracted hot spots.

region construction is halted after weight-guided block layout is performed (described in Section 4.3.4). The second (darker) bar shows the benefits of direct linkage between hot regions as described in Section 4.3.5. Without direct linkage, the average dynamic coverage is 73% and with linking the average is 79%. For many benchmarks, especially the ones with very high coverage, like *132.ijpeg*, linking shows little benefit if not a slight degradation. In these cases there is little room for improvement and additional overhead is generated while traversing links between regions. However, for a few benchmarks, substantial gains can be seen. For example, *124.m88ksim* has two specialized load-memory hot spots for loading its text and data. Without linking, only one version can be used. The text version will perform poorly during the data

Table 5.3 Code expansion in vacuum-packed regions.

Bmk	% Incr in Size	Bmk	% Incr in Size	Bmk	% Incr in Size	Bmk	% Incr in Size
099 A	132	A	10	A	3.2	A	24
124 A	3.7	132 B	8.6	134 B	4.2	255 B	27
A	16	C	14	C	7.1	C	28
130 B	16	164 A	15	181 A	48	300 A	9
C	24	175 A	6.2	197 A	27	mpg A	13

loading and vice versa. By linking these two phase regions with conflicting launch points, both specialized regions are effectively utilized.

The region formation process causes code duplication since region entry functions are partially duplicated and since heavily executed callee functions may be successively inlined into multiple regions. Table 5.3 shows the percent growth of static instructions due to region formation. Excluding *099.go*, the average code expansion due to region formation is only 16%. This means that the vacuum-packed regions account for 16% of the instructions from the unoptimized application, but have been selected and connected in a manner that they cover almost 80% of the dynamically executed instructions. In fact, the static number of instructions chosen in this manner is much smaller. *099.go* shows the worst effects, more than doubling the code base. *099.go* has a large number of weakly used hot spots. This excessive growth leads to its poor performance, which will be shown later in the section.

To explore the value added due to phase-sensitive profiling over traditional aggregate profiling, the dynamic branches in the experiments were categorized. First, the branches were separated into two groups, those whose static branch appears in only a single phase (Unique) and those whose static branch appears in multiple phases (Multi), as shown in Figure 5.3. The unique branches were then divided into biased and unbiased types, and notably were mostly

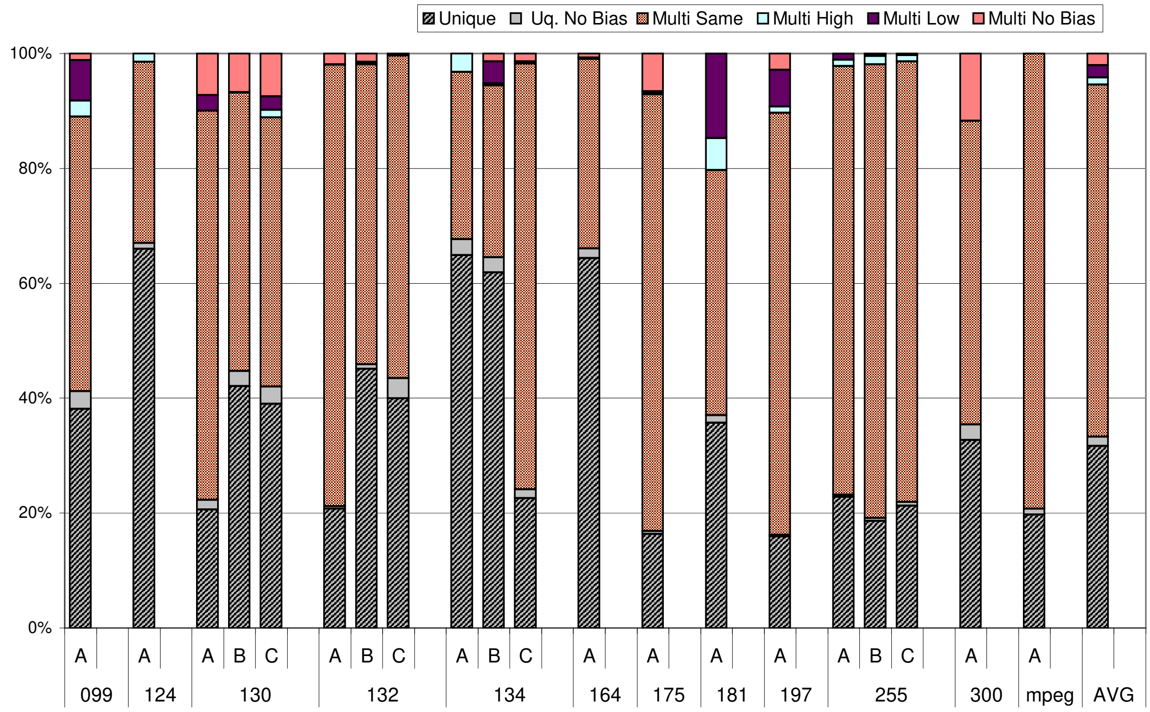


Figure 5.3 Categorization of hot spot branch behavior across benchmarks.

biased. Biased branches are especially beneficial, since optimizers can aggressively assume a particular direction without much risk to performance. The Multi branches are further divided into several categories based on the variation in expected direction. Biased branches that exhibit significant swings in their expected direction between phases, ($> 70\%$) change, are categorized as Multi High, while those with more moderate swings, between (30%) and (70%), are Multi Low. The remainder of biased branches, Multi Same, are considered to have insignificant swings, while branches that exhibit particularly unbiased behavior are further categorized as Multi No Bias. For example, the benchmark *099.go* has about 3% of its dynamic branches whose static branch is shared in multiple phases with a large swing in its behavior between the phases (Multi High). Aggressive static compilers may perform poorly on functions containing such branches as the aggregate profile may differ substantially from the behavior seen in each phase. In addition, it is evident that a significant portion of execution is seen in instructions which occur in multiple phases. The Multi High and Low instructions represent the opportunity for customizing an application for its phases by exploiting the differing behavior in each phase. While only differing by a few percent of all branches, the Multi High and Low branches have significant impact because they now allow the optimizer to wisely choose paths where once an ambiguous aggregate profile hampered the decision. As discussed in Section 3.3, these branches also serve to distinguish between a significant number of different phases.

Given that high coverage regions have been selected, extracted, and internally laid out using derived branch and block weights, a basic block reordering and scheduling pass was performed to further demonstrate the potential for optimization within these regions. Figure 5.4 shows the resultant program speedup due to region rescheduling for each benchmark and input. Three bars are shown for each experiment. The lightest bar shows the speedup when block reordering is performed using the original profile weights as generated for use by the compiler, on average 16.5%. *099.go*, as introduced earlier, loses performance due to the excessive amount of code duplication that occurs.

The medium grey bar shows the result of deriving the weights from the branch behavior counters using the inference and Markov modeling techniques proposed in Section 4.3.4. Though the BBB may be missing branches and has small saturating counters, it produces profile information that is specific to a phase of program execution. While software profile generation methods have practically unbounded counters in both number and size, the profiling performed is typically an aggregate of the execution of the entire application. Reordering blocks in the

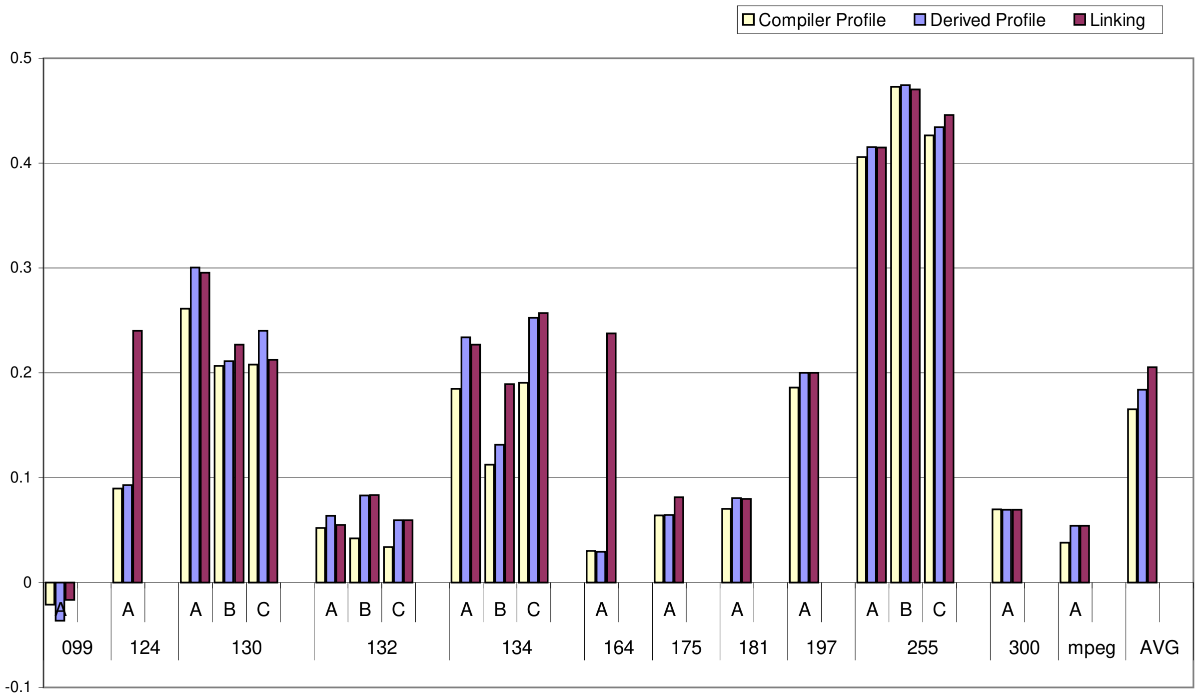


Figure 5.4 Performance speedup from basic rescheduling of hot spot regions.

phase regions yields improved results over reordering based on aggregate profiling with an average of 18.3% improvement over baseline. Lastly, the black bar shows the performance improvement due to linking. Linking helps performance indirectly by facilitating a higher coverage, thus enabling an average speedup of 20.5%.

The previous experiments have been performed with a BBB sized so that it does a good job of capturing hot spots. As the BBB size decreases, the likelihood that conflicts will occur increases, causing branches to be missing. Figure 5.5 demonstrates the effect of the BBB size on the effectiveness of hot spot detection (HS) and Vacuum Packing (VP). For hot spot detection, the percent of executed instructions that are part some detected hot spot for a given BBB size is shown. For Vacuum Packing, the percent of instructions executed within vacuum-

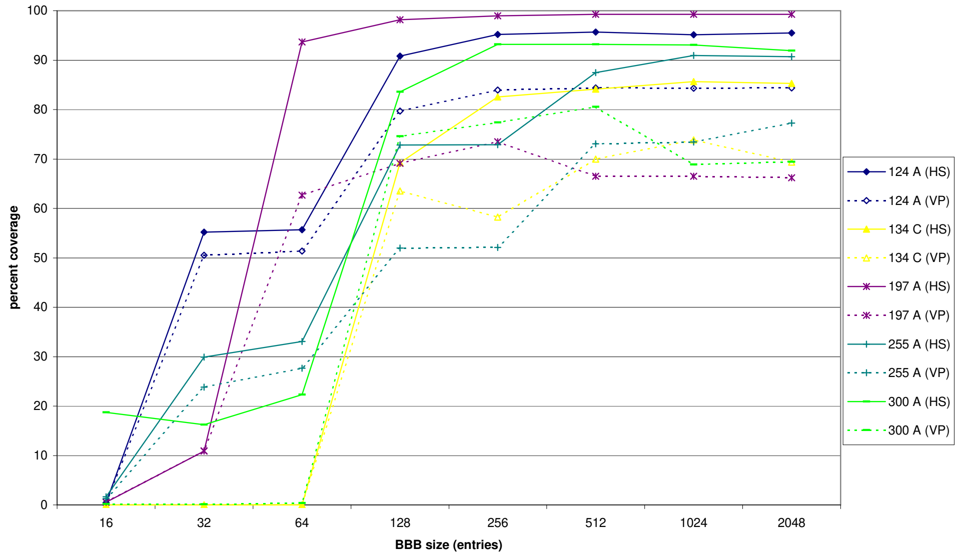


Figure 5.5 Effect of BBB size on Vacuum Packing effectiveness.

packed regions is presented. This percentage is generally less than the hot spot percentage because hot spot detection measurement did not require execution through an entry branch, while the Vacuum Packing measurement had a limited number of entry branches.

Even with a much smaller BBB, and a significant number of missing branches, Vacuum Packing selects similar regions for optimization, and the dynamic instruction coverage remains close to that of the largest BBB size. However, when the BBB becomes so small that it cannot hold most of the hot branches for a hot spot, the Hot Spot Detection Counter may never be able to saturate [26]. The percentages for both hot spot detection and Vacuum Packing declines significantly at BBB sizes less than 128 entries. In these situations, the Hot Spot Detector is often no longer able to successfully detect some how spots at all. At this point it would be impossible for Vacuum Packing to select regions which are representative of these program phases.

CHAPTER 6

FUTURE WORK

There are several opportunities for future research related to the techniques presented in this thesis. While a significant performance benefit from performing block order and instruction scheduling, optimization techniques specifically targeting vacuum-packed regions should be examined. These regions are designed to be a platform for additional optimization.

Since in Vacuum Packing code is collected into interprocedural regions, straightforward transformations like spill code elimination could be applied, further reducing the overhead that existed at function call boundaries. In this work, each inlined function retained its own stack frame and register allocation. For each vacuum-packed region, a merged stack frame could be formed, and register virtualization and reallocation could be performed. Since execution is likely to remain within a Vacuum Packed region during the corresponding phase of execution, code migration could move operations whose results are only needed upon exits from the region into compensation blocks [7] on those side exits. Such targeted, aggressive optimizations might significantly improve performance through increased specialization of the phase regions.

Additionally, the Vacuum Packing technique could be extended for utilization by a run-time optimizer, thus allowing its scope of optimization to approach that of an off-line optimizer. Since each region captures the execution of an application during a particular dynamic phase of execution, these regions might be ideal targets for dynamic optimization. In addition, since the Hot Spot Detector captures outer loops, they would allow a dynamic optimizer to maximize the performance impact of its transformations. The region selection technique in Sectionsec.algorithm.regionidentification is linear in complexity with respect to the number of blocks in the region. While comparing two hot spots is less efficient, and is required both to eliminate redundant hot spot detections and to link similar Vacuum Packed regions together, the use of hot spot signatures described in Section 3.4 could simplify these operations. The most computationally complex step in the Vacuum Packing process is the derivation of profile weights in Section 4.3.4.2; however, a dynamic optimizer might not need the complete, consistent set of profile weight generated using this approach. A combination of the simple propagation of weight described in Section 4.3.4.1 and the use of a heuristic profile guess might be sufficient to provide useful profile information to a dynamic optimizer.

Further characterization of the Vacuum Packed regions would be useful. While these regions were formed as control-flow phases of program execution, it is possible that they represent phases of program behavior with respect to other performance aspects, such as cache misses. The benefit of the redundancy generated in the Vacuum Packing process could be additionally evaluated. It is possible that, by identifying cases where the program phase extraction and partial inlining have not eliminated any optimization constraints, code size could be saved by not doing the unnecessary replication.

Finally, using static profile estimation techniques [37], the heuristic growth could be improved. Paths that are statically determined to be more likely could be favored during region growth. In this way, growth through less likely paths could be avoided without reducing the portion of execution captured by the extracted region.

CHAPTER 7

CONCLUSION

This work presents a technique for exploiting a transparent hardware profiling mechanism in a static post-link optimizer. An enhanced Hot Spot Detector is used to capture an estimated profile of each application's program phases. The presented additions to this profiling mechanism allow it to be used to continually monitor execution throughout an entire run of an application. These enhancements keep the overhead of the operating system interaction reasonable by automatically eliminating the majority of repeated program phase detection. While hardware mechanisms like the Hot Spot Detector incur minimal overhead in profiling, the resultant profile suffers from decreased accuracy compared to complete software instrumentation. Vacuum Packing overcomes this inaccuracy to select critical program regions for optimization.

Taking advantage of the phase-sensitive nature of the Hot Spot Detector, the presented algorithm forms regions that represent the important phases of program execution, thus expanding the scope of post-link optimization from individual traces to entire phases of execution. The application is connected to these optimized code regions through only a handful of select entry points, keeping modifications to the original code to a minimum. The captured and extracted phases are demonstrated to represent the vast majority of program execution. Where dynamic, trace-based systems continually generate new traces to ensure execution in optimized code, Vacuum Packing achieves a very high percentage of execution in statically optimized phase regions.

This work demonstrates that applications have several phases which are made up of very similar working sets of blocks, but have branches with different directional bias in each phase. The Vacuum Packing strategy links such hot spots in a way that allows execution to quickly migrate into the optimized code region that matches the current dynamic phase of execution. This is shown to improve the ability of Vacuum Packing to capture a larger percentage of execution for several benchmarks.

Finally, the benefit of exploitation of the program phase behavior is also examined in this thesis. Based on phase-sensitive profile information, partial inlining of functions into their corresponding phase regions was performed. This allows optimization benefit across the boundary that function calls represent without the code size expense of complete inlining. Such partial inlining would be difficult for a static compiler that has only an aggregate profile of program behavior. Similarly, using the phase profiles derived from the Hot Spot Detector, a modest speedup over using the compile-time profile was achieved for a simple block reordering optimization.

REFERENCES

[1] R. S. Cohn and P. G. Lowney, "Hot cold optimization of large Windows/NT applications," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 80–89.

[2] R. S. Cohn, D. W. Goodwin, and P. G. Lowney, "Optimizing Alpha executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, no. 4, pp. 3–19, 1997.

[3] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary translation in a distributed environment," Microsoft Research, Tech. Rep. MSR-TR-2001-50, April 2001.

[4] A. Klaiber, "The technology behind Crusoe^{TM} processors," Transmeta Corporation, January 2000, http://www.transmeta.com, tech. rep.

[5] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent execution, no recompilation," *IEEE Computer*, pp. 47–52, March 2000.

[6] D. Bruening and E. Duesterwald, "Exploring optimal compilation unit shapes for an embedded just-in-time compiler," in *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000, pp. 13–20.

[7] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 158–168.

[8] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, pp. 567–589, June 2001.

[9] S. McFarling, "Program optimization for instruction caches," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 183–191.

[10] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[11] T. Way and L. L. Pollock, "A region-based partial inlining algorithm for an ilp optimizing compiler," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[12] G. Haber, M. Klausner, B. Mendelson, and V. Eisenberg, "Light weight optimizations for reducing hot saves and restores of callee-saved registers," in *Proceedings of the Fourth Workshop on Feedback-Directed Optimization*, December 2001, pp. 31–42.

[13] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[14] 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.

[15] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: Path based data representation and scheduling of subgraphs," in *Proceedings of 32nd Annual International Symposium on Microarchitecture*, December 1999.

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

[17] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, June 2001.

[18] M. C. Merten, "Run-time optimization architecture," Ph.D. dissertation, University of Illinois, Urbana, IL, 2002.

[19] 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*, October 1997, pp. 1–14.

[20] 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*, October 1997, pp. 15–26.

[21] 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*, June 1997, pp. 85–96.

[22] 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*, December 1997, pp. 292–302.

[23] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001, pp. 3–14.

[24] A. S. Dhodapkar and J. E. Smith, “Managing multi-configuration hardware via dynamic working set analysis,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 233–244.

[25] 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*, December 1996, pp. 36–45.

[26] 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*, May 1999, pp. 136–147.

[27] 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*, June 2000, pp. 59–70.

[28] P. Denning, “Working sets past and present,” *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 64–84, January 1980.

[29] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1982.

[30] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, “Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research,” in *Proceedings of the Workshop on Workload Characterization*, September 2000, pp. 83–100.

[31] C. Lee, M. Potkonjak, and W. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 330–335.

[32] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, “IMPACT: An architectural framework for multiple-instruction-issue processors,” in *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 266–275.

[33] B. C. Cheng, “A profile-driven automatic inliner for the impact compiler,” M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

[34] B. C. Cheng and W. W. Hwu, “Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation,” in *Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation*, June 2000, pp. 57–68.

[35] 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*, June 1998, pp. 227–237.

[36] C. J. Shannon, “The IMPACT SC140 code generator,” M.S. thesis, University of Illinois, Urbana, IL, 2002.

[37] T. Ball and J. R. Larus, “Branch prediction for free,” in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313.