Vacuum Packing: Extracting Hardware-Detected Program Phases for Post-Link Optimization

Ronald D. Barnes Erik M. Nystrom Matthew C. Merten[†]
Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electical and Computer Engineering
University of Illinois at Urbana-Champaign
{rdbarnes,nystrom,merten,hwu}@crhc.uiuc.edu

Abstract

This paper presents Vacuum Packing, *a new approach to profile-based program optimization. Instead of using traditional aggregate or summarized execution profile weights, this approach uses a transparent hardware profiler to automatically detect execution phases and record branch profile information for each new phase. The code extraction algorithm then produces code packages that are specially formed for their corresponding phases. The algorithm compensates for the incomplete and often incoherent branch profile information that arises due to the nature of hardware profilers. The technique avoids unnecessary code replication by focusing on hot code, making efficient connections between the original code and the new code, linking code packages at select points to facilitate phase transitions, and providing a platform for efficient optimization. We demonstrate that using a concise set of profile information from a hardware profiler, we can generate code packages, specialized for each phase of execution, that capture more than 80% of the average total program execution. We further show 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.*

1 Introduction

Modern computer systems are increasingly reliant on optimization techniques that can be applied in the end user's environment. For instance, the Crusoe Processor [13] dynamically translates and optimizes binaries from one instruction set to a different, underlying architecture. Tools such as Spike [6] and Vulcan [20] are *static post-link optimizers* that can statically optimize application binaries in response to user input and data patterns. Just-In-Time compilation systems, such as Java [21], rely on dynamic code generation and optimization to produce native code sequences that alleviate the burden of bytecode emulation. All of these techniques improve upon static compilation by adapting applications in response to specific information about the execution environment or usage patterns. For example, in the end user environment, the exact dynamically linked libraries become available, the exact processor model and features become known, and the important control-flow paths can be discerned. Each of these can be exploited by an optimizer, leading to further customized applications.

In order to effectively exploit this specific information, the optimizers require a mechanism for gathering the profile and environmental information, a platform for transforming the code, and a mechanism for deploying the code. An application profiler is a key component because it identifies important code segments and usage patterns that may benefit from optimization. Clearly, heavily executed code must be selected so that the potential benefit can be realized. Focusing on heavily executed code also helps control the quantity of code produced so that instruction cache, branch prediction, and paging resources are not taxed. Furthermore, instruction-level-parallelism (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 because it must be powerful enough to enable beneficial transformations while limiting the time and resources required. For many run-time systems, the benefits must directly outweigh the transformation costs because the processor may have to stall the application itself to perform the transformation. Many modern systems rely on instruction trace-based approaches because they offer a platform that requires little analysis and overhead and yet achieves reasonable performance gains. While these systems allow for significant local customization, they lack the scope for broader optimizations, such as loop-level transformations.

[†]Currently with Intel Corporation in Hillsboro, Oregon.

Last, the optimized code segments must be deployed. Many modern run-time optimization systems deploy the optimized code as a mass collection of traces, either in a hardware structure or in a dedicated memory region. Static post-link optimizers typically generate a new monolithic binary.

This paper presents a new technique for detecting and packaging regions of performance-critical code that includes a swift profiling mechanism, an efficient platform for transformation, and a phase approach to optimized code deployment. The technique, called *Vacuum Packing*, is based upon region-based optimization [12] which enables the optimizer to focus on only the hot code blocks, even when control-flow crosses function boundaries. Rather than using aggregate profile information to form static regions, as is done within a traditional compiler, our strategy forms a small set of inter-procedural regions, called *packages*, for each phase of program activity. Each package contains the code that is responsible for the particular activities inside a particular program phase. Using these encapsulated regions, an optimizer can focus on the code that is responsible for a vast majority of execution during each phase. Several packages may in fact contain copies of the same segment of code if the segment is utilized in several phases. Phases with overlapping packages are linked together to facilitate transitioning between packages upon a program phase change. Aggressive inlining, ILP optimization, and scheduling can be applied to each package to construct new tight and modular code units. The strategy is an improvement over previous post-link 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.

2 Related work

The Vacuum Packing technique is designed to treat the code within a package (code from a single phase which can span multiple functions) as a single unit. Optimization of our *package* is analogous to the optimization of *regions* in [12]. In a way similar to Hot Cold Optimization (HCO) [7], the heavily executed blocks are extracted and colocated during region formation and become the focus of further optimization. The original program, including the infrequent (cold) blocks, is often left largely untouched and off to the side, reachable only through cold exits from the extracted packages. By extracting the packages of hot code, only a very small subset of the original code must be manipulated (unlike typical compiler approaches which integrate the optimized hot code within the original code). To form a package, a partial-function inlining [23] technique is employed to expand a hot seed function by growing it to deeper calling contexts of hot callee functions. Packages may additionally span across library-call boundaries to achieve a broad scope without the explosive code growth of whole-function inlining.

Systems that employ the Vacuum Packing technique rely on intensive profiling mechanisms, in their efforts to form dense packages of code to match particular phases of program execution. Vacuum Packing has been designed to exploit such phases by providing aggressively optimized code for each point in time of program execution. A number of 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. They often rely on program counter sampling over long periods of time to produce whole-execution aggregate profiles [1], [2], [9]. Basic Block Distribution Analysis [19] combines intense, periodic sample-based profiling to determine the composition of repetitive phases. Typically, however, specialized hardware components have been proposed to accurately detect arbitrary phases. Described in Section 3.1, the Hot Spot Detector [17] (somewhat similar to the profile buffer [8]) is utilized in this work to perform intensive profiling and to analyze the detected blocks for stability. All of these hardware approaches, however, are lossy in that the collected data is incomplete due to the randomness of sampling or the limitations of hardware structures.

Recent work examined several units of optimization for complexity and effectiveness in dynamic optimization systems [5] and concluded that loops provided the greatest performance opportunities, followed by opportunities provided by the much simpler traces. Regions provide a convenient platform for near-global scheduling techniques by focusing optimization efforts on a modestly sized code base that represents a significant portion of execution, often an outer loop. Combined with efficient, global scheduling algorithms, region-based approaches in off-line systems (where optimization resources are more readily available) are likely to boost future performance results of post-link optimization systems beyond that of current trace-based capabilities. Many systems, from off-line optimizers such as Spike [6] and Vulcan [20] to dynamic run-time optimizers such as Dynamo [3], Daisy [11], Crusoe [13], ROAR [16], rePLay [18] and UQDBT [22] could all benefit from the enhanced, focused scope provided through the Vacuum Packing method.

3 Hot region identification

The Vacuum Package region formation process is designed to identify important code regions associated with program phases and extract them for the purpose of code optimization. As shown in Figure 1, the formation process conceptually consists of three steps. In the first step, pro-

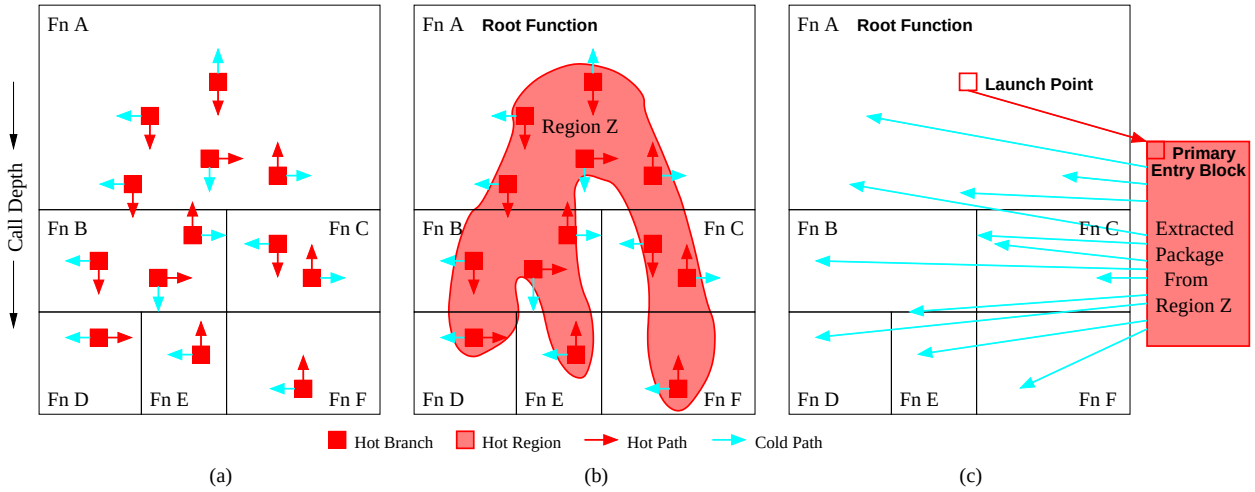

Figure 1. The Vacuum Packing region formation process overview for functions A-F, shown in call tree order. (a) Hot branches with hot and cold directions. (b) Identified hot region named Z. (c) Extracted and optimized region Z with launch points into the new region and cold side exit links back to original code.

filing, a mechanism monitors the execution of a program and assembles a collection of static branches referred to as *hot spot branches* as shown in Figure 1(a). These instructions are the hot branches associated with the current phase of program execution. Upon the detection of a new phase, information accumulated in this monitoring mechanism, including hot spot 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 paper, the profiled program runs to completion before any of the phases are further processed by the software. The monitoring mechanism is implemented in hardware and runs in the background, incurring minimal overhead until a phase transition is detected.

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 identification algorithm. The algorithm maps the hot spot detection information to the code to select inclusion into the hot region and can leverage the branch information to generate estimated execution frequencies of all instructions in the region. Often these hot regions span function boundaries; in Figure 1(b), the hot region spans functions *A*, *B*, *C*, *D*, *E*, and *F*.

In the third step, an extraction algorithm assembles pieces of the hot region into a new, localized code package that can be conveniently handled by an optimizer. One physical region is identified for each program phase from which multiple packages may be constructed. Control transitions are established between the original program and the extracted packages. Finally, control transitions are also established between the packages themselves. The new code packages are structured much like a function body so that optimization algorithms can easily process them. Unimportant code around the hot region is excluded from the extracted package which can then be tightly optimized and scheduled. For this reason, the presented technique is known as Vacuum Packing.

3.1 Step 1: Program phase detection

During the first step of the Vacuum Packing process, the Hot Spot Detector (HSD), as shown in Figure 2, is the monitoring mechanism responsible for detection of hot branches in each of the phases of execution [17]. 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 that provides an on-the-fly analysis of the execution coverage provided by the branches 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 its dynamic behavior is tabulated. The details of the operation of the HSD are found in [17].

Upon the detection of a hot spot, the BBB contains the set of hot spot branches and their executed and taken counts. The counts together minimally provide the taken fraction 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.

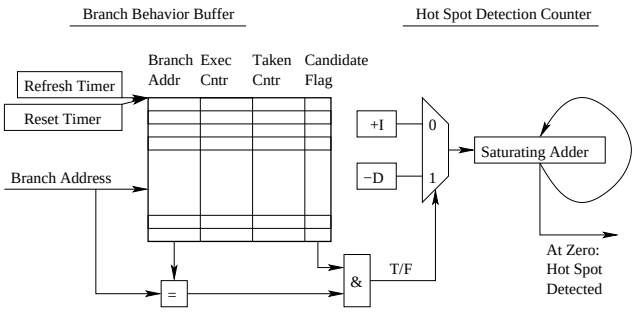

Figure 2. The Hot Spot Detector.

However, contention for table entries 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 [17]. In addition, the hardware counters tracking each branch saturate when the execute count reaches its maximum value. However, at saturation, the taken fraction for the branch is preserved.

Figure 3 illustrates the hot region identification process. The profiled program consists of two functions, as shown in Figure 3(b). Basic blocks A_1 through A_{10} belong in function A and basic blocks B_1 through B_6 belong in function B. Figure 3(a) 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.

After the branch information is stored, the execution of the profiled program resumes under the watch of the hot spot detector. Without any other support, the same hot spot would be repeatedly detected and recorded until a phase transition occurs. While the detection itself has little overhead, the recording of the resultant profile is comparatively expensive. Recording hot spot profiles only at phase boundaries eliminates consecutive recording of the same hot spot.

Enhancements to the BBB, as described in [4], provide a history of one hot spot and records a phase only when it is different than the previous phase. This history could be extended to more than one to greatly reduce the number of hot spots recorded by not rerecording hot spots held within the history window. Working set signatures [10] could be extended to hot spot signatures to allow inexpensive comparisons between a detected hot spot and a history of previously recorded hot spots. When the hot spot records are processed at optimization time, additional filtering can also be done to further remove redundant hot spots. For this paper we assume software filtering eliminates all redundant hot spot detections, since this work focuses on taking advantage of the hot spots rather than minimizing the amount of data transferred at detection time.

In determining the similarity between two hot spots, two criteria are used. First, given a hot spot *A* and hot spot *B*, if 30% or more of *A*'s branches are missing from *B* (or vice versa) then *A* and *B* are different hot spots. Second, if a single biased branch that is common to both *A* and *B* has a different bias (taken vs. not-taken) between *A* and *B*, then *A* and *B* are different hot spots. As described in [4], the threshold of varying biased branches could be increased to more than one, yielding fewer unique hot spots.

3.2 Step 2: Region identification

The second step of the hot region formation algorithm identifies the hot 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 is the only information used to select a region's instructions and can also be used to determine 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. During the identification and growth of a region, a control-flow graph (CFG) for each function in the region is marked with the hot and cold information. A call graph representing function call relationships within the region is also constructed. Each region is then expanded to include additional blocks and their corresponding control-flow arcs for several reasons:

- Even though exits from a region inferred from the Hot Spot Detector are infrequently traversed, we want to minimize the number of them by opportunistically including infrequent paths when inclusion is associated with little or no cost.
- A hot path may 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 our regions.
- Techniques using hardware counters[1] to determine profile weights provide only an approximation of the actual profile. We must tolerate a certain amount of missing information.

3.2.1 Hot spot block and branch identification

To identify the region of code for Vacuum Packing, the CFG is constructed with instructions divided into basic blocks, where each block contains no more than one branch or subroutine call, which is always the last instruction in the block. Each block and arc in the CFG is augmented with *weight*

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

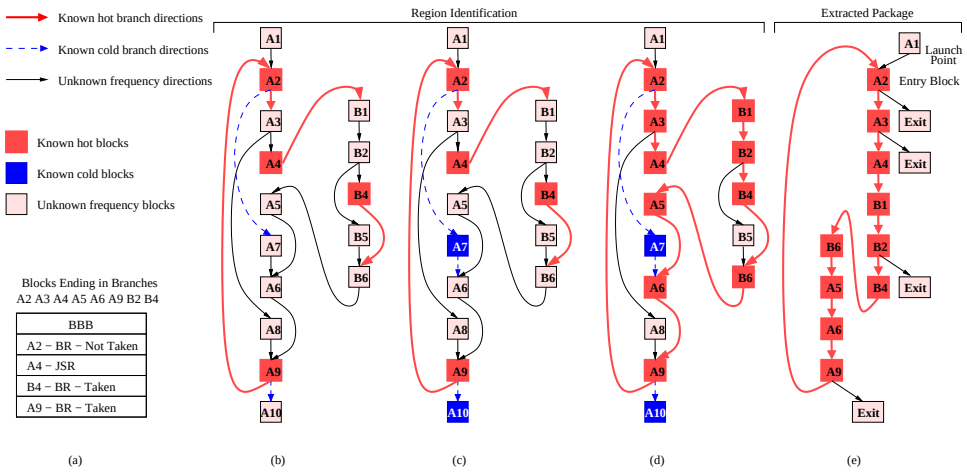

Figure 3. Region identification example functions. (a) Branch Behavior Buffer profile. (b) Profile superimposed on a control-flow graph. (c) After propagation of the cold arc information. (d) After propagation of the hot arc and block information. (e) Resultant Package.

and *temperature* fields, along with an additional *taken probability* field for each block ending in a branch.

At hot spot detection time, the HSD stores records for all hot spot branches containing their taken and executed counts, which provides an outline of the particular execution phase. Each basic block containing a hot spot branch is assigned a weight corresponding to the executed count of the hot spot branch and is assigned the `Hot` temperature. Each block is additionally assigned a taken probability equal to $\frac{taken\ count\ of\ its\ hot\ spot\ branch}{executed\ count\ of\ its\ hot\ spot\ branch}$. The CFG arcs that correspond to the taken and fall-through directions of each hot spot branch are assigned a weight based on the taken and executed counters of that branch. Each arc is assigned the `Hot` temperature if that direction accounts for either a minimum of 25% of the branch's flow of control or if it has a weight greater than the HSD's hot spot branch execution threshold. If a direction accounts for a smaller amount of a branch's flow of control, that arc is given the `Cold` temperature. After this initialization, blocks can have a temperature that is either `Hot` or `Unknown`, while the temperature of CFG arcs can be `Hot`, `Cold`, or `Unknown`.

3.2.2 Block and arc temperature inference

Once region selection is initialized, the region is expanded by inferring additional `Hot` (and `Cold`) blocks using the algorithm presented in Figure 4. The temperature inference process seeks to add blocks which should have been included in the hot spot, but either do not contain branches, or contain branches that were missing from the HSD at detection time. By iterating over the CFG, the algorithm applies the inference rules to blocks and arcs, as depicted in the example in Figure 5. For instance, Statement 3 of Figure 4 employs inference rule *a* of Figure 5 which states that a block can be inferred to have a `Cold` temperature if all control-flow arcs into or out of that block have a `Cold` temperature. Similarly, rule *d* of Figure 5 is performed by Statement 6 of Figure 4 to set the temperature of all incoming and outgoing control-flow arcs of a `Cold` block to `Cold`. Finally, Statement 9 of Figure 4 provides for the inference of a `Hot` temperature for the target block of a subroutine call from a `Hot` block. Once no additional inferences can be applied, some blocks will remain `Unknown` if the algorithm was unable to propagate a temperature of either `Hot` or `Cold` to them. Some `Unknown` blocks will be added to the regions by the heuristic growth process explained in Section 3.2.3.

3.2.3 Heuristic hot region growth

For reasons previously detailed, important blocks might not be initially selected using hot spot block identification or inference. In order to additionally reduce the frequency of transitions from our optimized code to the original code, two additional steps of heuristic expansion of the selected region are performed. First, any arc with an `Unknown` temperature between two `Hot` blocks is included in the selected region. Since nothing is known about the arc and since its target is already selected as part of the region, it is eliminated as an exit. `Cold` temperature arcs between two `Hot` blocks continue to be excluded from the region in keeping with the goal of producing code packages specialized to the behavior of a phase. Second, in an attempt to find a single launch point for each package, the selected region is expanded into adjacent predecessor blocks from each entry block until another `Hot` temperature block is reached. Such

1. Iterate through blocks in CFG to attempt to solve unknown temperatures:
2. Propagate control−flow arc temperatures to blocks:
3. Set block temperature to Cold if all arcs in or out have known, Cold temperature
4. Set block temperature to Hot if any one arc in or out has a Hot temperature
5. Propagate block temperatures to control−flows arcs:
6. Set temperature of all arcs in and out of blocks with Cold temperature to Cold
7. Set temperature of arc in or out of a Hot block if all other arcs in or out (respectively) of block have a known, Cold temperature
8. Propagate temperature through Hot calls:
9. Set temperature of callee's prologue block to Hot

Figure 4. Algorithm for inferring additional hot blocks.

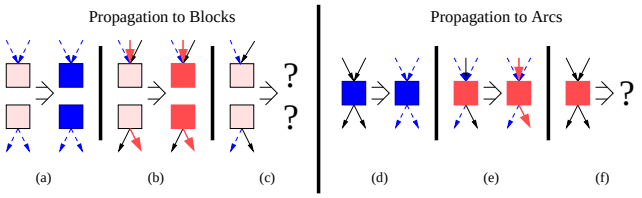

Figure 5. Inference Rules; Legend in Figure 3. (a)-(c) Propagation to Blocks. (d)-(f) Propagation to Flows.

growth avoids all `Cold` arcs and blocks, and is limited to **MAX_BLOCKS** additional blocks. Since we wish to limit the size of the selected regions **MAX_BLOCKS** should be chosen as a small number of blocks. For the experiments in this work, **MAX_BLOCKS** is chosen to be 1.

3.2.4 Region identification example

In Figure 3(a), the profile information for the detected hot spot covers only four of the eight hot branches due to limited number of BBB entries. 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 our algorithm must be tolerant of some branches missing from the buffer.

In Figure 3, information about several additional branches can be derived from the HSD branch profile using the algorithm in Figure 4. Since A_2's branch is strongly not-taken, the flow to A_7 is identified as `Cold`. The flow from A_9 to A_{10} is similarly identified as `Cold`. Since the flow from A_2 to A_7 is `Cold`, block A_7 must be `Cold` *(Statement 3)*. In Figure 3, the propagation of `Cold` and `Hot` blocks is shown separated in (c) and (d). Some additional flows can be positively identified as `Hot` based on the HSD profile. Since A_2 is `Hot` and is also strongly not-taken, the flow to A_3 is `Hot`. The temperature of this flow is propagated to block A_3 by Statement 4 even though it was missing from the hot branch profile. The fact that B_4 is `Hot` implies that B_2 and B_6 are `Hot` *(Statements 7 and 4)*.

The algorithm identifies the hot basic blocks from this program phase in spite of the lack of profile information for half of the branches. The result is shown in Figure 3(d). Note that in general, however, there will be `Hot` blocks and `Cold` blocks that cannot be positively identified due to incomplete profile information. In those cases, heuristic growth will be relied upon to determine whether or not the blocks are included in the region.

3.3 Step 3: Package construction and optimization support

Once a hot region of code has been detected, identified, and grown, the package construction and optimization step begins. A *package* is a connected piece of code derived from a region that may include instructions from multiple functions and may have multiple entrances and exits. A single region would potentially generate multiple disjoint packages.

3.3.1 Function pruning and maintaining data-flow

For each hot region, copies of the marked functions are reduced to include only the blocks and control-flow arcs declared important (`Hot`) for that region. This can eliminate a large fraction of the blocks and flows within a function, particularly eliminating merge points between hot and cold paths. However, control-flow paths from hot to cold blocks cannot be ignored as they may occasionally be traversed, transferring execution out of the package. The live registers at these exit points are maintained in the optimizer by creating a new basic block, called an *exit block*, along each exit path and by placing dummy consumer instructions for each register that is live across the exit. This allows the removal of the cold instructions without corrupting or complicating the formal data-flow analysis process. Figure 6(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 a representative exit block is shown in Figure 6(b).

3.3.2 Locating root functions and entry blocks

The call graph for the region is examined to find *root functions*. These root functions will serve as seeds for the partial inlining process and will be the containers for the generated

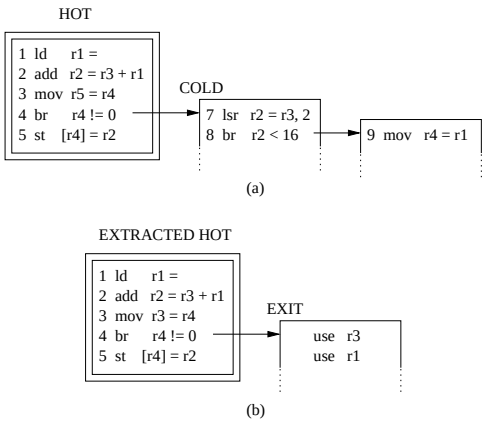

Figure 6. Maintaining data-flow. (a) Original code sequence. (b) After hot instructions have been extracted.

packages. A function will be chosen as a root for one of three reasons. First, any function without any callers in the region (ignoring back edges in the call graph) will be a root since it will be encountered first during execution. Second, any function that will not be inlined into any callers will be marked a root function. Section 3.3.3 describes how this can occur when the hot parts of a function lack a prologue, epilogue, or a path in between the prologue and epilogue. Last, any self-recursive function will be chosen as a root and a single copy will be allowed to be partially inlined into itself. This provides a package into which recursive calls can enter if their call depth goes beyond what was explicitly inlined.

In a similar fashion, *entry blocks* within the root functions are selected based on their relationships with the function's CFG. Ignoring back edges within the CFG, entry blocks are ones without any predecessors. The entry blocks are the sole points of entry from original code into the final packages. The original code locations that transfer control into package entry blocks are called *launch points*. This can be seen in Figure 7(a) where branch *A0* can jump to the block containing branch *A1*. The block containing *A1* is an entry block of root function *A* and will be included into the constructed packages. *A0* is a launch point and resides only within the original code. In the figure, function *B* is not a root function but will be partially inlined into the package rooted at function *A*.

3.3.3 Partial inlining

The inlining process successively progresses through root functions of the call graph producing individual packages for the region and is *partial* inlining because the parts of the callees were discarded in Section 3.3.1. It proceeds by finding an out-going arc in the call graph from the root function to another function within the same region. Partial inlining of a callee will not be performed unless the callee contains a prologue and an epilogue with a path between them. Inlining of a callee that fails to meet these conditions would separate the region into two pieces, the part that makes the call and the part after the callee returns. Therefore inlining of such disjoint callees is skipped.

When partial inlining is performed, the blocks 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 out-going call graph arcs are merged in with the root function's arcs, and the callee function is removed from the out-going arc set of the root function. The inlining process continues for this root function until its out-going arcs are exhausted.

3.3.4 Package transitions

Program phasing may lead to a situation where a particular function is the root function for several different phase regions. Consider a perl interpreter where the command execution loop may serve as the root function for different packages that are specialized for different types of commands, such as string or numeric processing. Since a launch point can only point to a single package, a means for transferring control to the package that corresponds to the current phase is necessary. The example in Figure 7 will be used to demonstrate many of the package entrance and transition features of Vacuum Packing. Figure 7(a) depicts an original code fragment in which three different execution phases were detected. In the figure, only the branches are depicted for clarity (*A1* through *A4* and *B1*). The three phases cause the formation of three packages shown symbolically in Figure 7(b). For all of the packages, function *A* is the root function and the launch point is from *A0* to the entry block containing *A1*. Each package is designed to contain all of the hot code need for the execution of its phase. As described in Section 3.3.3 partial inlining is used to include customizable copies of code into the packages. Along with copies of the selected pieces of callee functions, selected pieces of the root function are also replicated into each package. In Figure 7(b) **U** (unbiased), **F** (biased fall through), and **T** (biased taken) mark the bias for a particular branch during the respective phase and imply the contents of the corresponding package. For example, *A2* is biased fall through in phase 1 and thus its package skips the second call to *B*. *A2* is biased taken for phases 2 and 3 and thus these packages make a second call to *B*. For phases 1 and 2, branch *A1* is unbiased, **U**, taking and falling through roughly equally and thus the packages include code along both directions.

If all packages have disjoint root functions, then a one-to-one mapping exists between launch points and entry

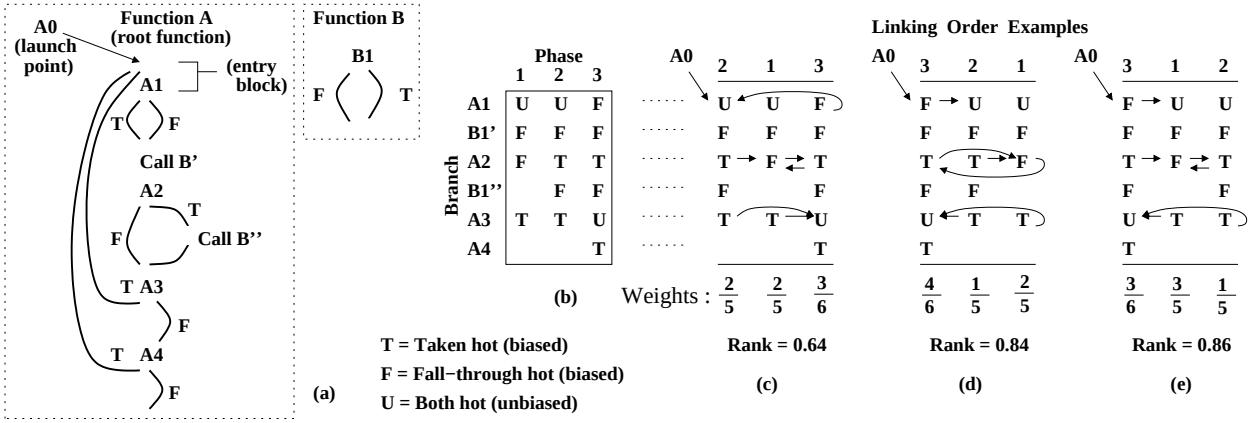

Figure 7. Package construction and linking example.

blocks. However, it is not uncommon for multiple packages to have the same root function. Thus, there may be no definitive location in the original code to launch into each distinct package. This clearly occurs in Figure 7 where there is only a single entry block, *A1*, and correspondingly only one launch point, *A0*. Since there a single launch point, only one of the packages can be entered into directly from the original code.

Package linking provides paths to selectively reach alternate packages rooted at the same point by retargeting cold (exit) paths in one package to their target blocks that are hot in another package. Another solution would have been to dynamically modify the launch point branch to point to the expected best package. However, a mechanism would need to be in place to make the modification. While a monitoring code snippet could be introduced along the exit path to feed a dynamic predictor, an easy, static solution is to simply link the side exit from one package to the corresponding point in the other.

Caution must be exercised to ensure that the calling contexts from the root function to the link sites within both packages are *identical* otherwise execution could traverse into incorrect functions. This is shown in Figure 7 in that branch *B1* from function *B* occurs twice in the packages formed for phases 2 and 3. For these phases the function is partially inlined twice, once at call*B'* and once for call *B"*. Even though *B1'* and *B1"* originate from the same branch *B1*, they are from different contexts. A link between a *B1'* and a *B1"* would clearly be incorrect since it would be the same as creating a path from call *B'* directly to call *B"* (or vice versa) in the original code. Thus in Figure 7(b), *B1'* and *B1"* are listed in different rows and treated as incompatible branches.

Figures 7(c-e) shows three (of six possible) different ordering options between the packages. Looking at Figure 7(c) in detail shows the entrance into phase 2's package that links from phase 2's *A2* biased **T** to phase 1's *A2* biased **F**. This means that when phase 2's branch falls through it jumps to phase 1's version of the fall-through code. For linking, an **F** can connect to a **T** and to a **U**, while a **T** can connect to an **F** and to a **U**. Given the overlapping entry points and the different formations of the three packages many linking options exist. For our implementation, a link is always formed to the first compatible package to the "right" wrapping around the end to the first package. Additionally, the "left-most" package in the ordering is given precedence for entry points when shared entry points exist. These two rules convert the linking problem into a package ordering problem, though they do not allow for a truly exhaustive search of all linking possibilities. Now the packages must be ordered in a way that provides the highest reachability to the available packages.

While not necessarily optimal, the following method is used to rank a given package ordering, where a higher rank is better. For each package, the number of incoming links is divided by the number of package branches to yield a weight. This rank is shown below each column in Figures 7(c-e). The ratio represents the number of ways of entering the package through links to the size of the package. Using Figure 7(c), the rank is calculated by using the first package's (phase 2's) ratio of 2/5 to initialize both an accumulator and a weight variable. The weight is then multiplied by the second ratio of 2/5 and added to the accumulator. Similarly, the weight is again multiplied by the third ratio of 3/6 and added to the accumulator yielding a final result of 0.64. The ranks provide a rough comparison of the likelihood of staying in the packages formed. The ordering and links shown in Figure 7(e) would be picked for this example.

4 Experimental setup

Listed in Table 1 are benchmarks representing a wide variety of application types selected from SPEC CPU95, SPEC CPU2000 (including shortened reference inputs from

Table 1. Benchmarks and inputs used in experiments.

Benchmark	Input	# of Inst	Benchmark	Input	# of Inst
099.go	A: SPEC Train	338M	134.perl	A: SPEC Train 1 B: SPEC Train 2 C: SPEC Train 3	1512M 28M 8M
124.m88ksim	A: SPEC Train	89M			
130.li	A: SPEC Train B: 6 Queens C: Reduced Ref	122M 32M 362M	255.vortex	A: UMN_sm_red B: UMN_md_red C: UMN_lg_red	63M 315M 886M
132.ijpeg	A: SPEC Train B: Custom Faces C: Custom Scenery	1094M 57M 320M	197.parser	A: UMN_sm_red	178M
			300.twolf	A: UMN_sm_red	167M
164.gzip	A: SPEC Train	1902M	mpeg2dec	A: Media Train	99M
175.vpr	A: SPEC Test	1012M			
181.mcf	A: SPEC Test	105M			

Table 2. Simulated EPIC machine model.

Parameter	Value	Parameter	Value
Instruction issue	8 units	LD/ST buffer size (each)	8 entry
Integer ALU	5 units	BBB associativity	4−way
Floating point unit	3 units	Num BBB sets	512 set
Memory unit	3 units	Candidate branch threshold	16
Branch unit	3 units	Refresh timer interval	8192 br
LI data cache	64 KB	Clear timer interval	65526 br
Unified L2 cache	64 KB	Hot spot detection cntr size	13 bits
LI instruction cache	512 KB	Hot spot detection cntr inc	2
RAS size	32 entry	Hot spot detection cntr dec	1
BTB size	1024 entry	Exec and taken counter size	9 bits
Branch resolution	7 cycles		
Branch predictor	10−bit history gshare, 3 predictions per cycle		

the University of Minnesota (UMN) [14]), and MediaBench [15] to test the performance of our region formation and package extraction strategies. The benchmarks were each compiled with the IMPACT compiler using control-flow profiling information, inlining, classical optimization, pointer aliasing analysis, and instruction scheduling with control speculation. Data was collected across the complete run of each benchmark and input. The number of instructions executed is also listed.

The performance measurements reported in this work were generated by a custom software emulator that performs cycle-by-cycle full-pipeline simulation of each instruction. The architecture modeled consists of a ten stage EPIC pipeline containing five functional unit types (Integer ALU, FP, Long Latency FP, Memory, and Control). The simulations also include a multi-level memory hierarchy, and branch and return-address predictor. The emulator fully accounts for the affects of branch prediction, wrong path execution, cache utilization and pollution, varying memory latency, interlocking, and bypassing. Table 2 reflects the architectural parameters chosen for the evaluation system.

5 Experimental evaluation

5.1 Execution coverage

A primary concern for a post-link optimizer is the amount of program execution spent in the optimized code. This principle is even more important for systems that generate code at run time since performance gains due to execution in optimized code must overcome the losses incurred performing the optimization. For Vacuum Packing, higher quality packages lead to a greater percentage of execution from within the optimized code regions. Our emulator tabulated the number of dynamic instructions executed in the packages and in original code and computed the percentage spent in the packages, which is shown in Figure 8. The experiments vary the use of hot block inference (Section 3.2.3) and inter-package ordering (Section 3.3.4). Four bars are listed for each benchmark input, one without inference or linking, one without inference but with linking, one with inference but without linking, and one with both inference and linking.

Turning off hot block inference makes the region identification process treat the branch data recorded by the HSD as complete. When turned off, additional inference is only performed to blocks that do not contain a branch, but the remainder of the formation algorithm is used in full. Inference helps if a package is missing branches due to conflicts within the BBB but is unlikely to have any effect if a program's phase regions are very small. Though it does not greatly effect the average, individual benchmarks like *175.vpr* and *300.twolf* benefit noticeably. Missing branches can, in some cases, actually aid coverage if they are missing from the root function. In this situation, additional launch points may be created that provide more opportunities for execution to transition into the packages, although the optimization potential may be reduced due to the partitioned region.

As previously described, linking provides a means for execution control to reach multiple packages even if they have overlapping launch points. For example, *124.m88ksim* has two phases for loading a binary, each with the same launch point. Without linking, as is shown in the first and third bars of Figure 8, both packages can not always be reached, thus preventing execution from reaching the matching optimized code region. However, some overlapping packages have launch points that do not overlap with any other packages while still others have unique root functions. In both cases they are reachable without linking. This mitigates some of the loss when linking is turned off. Despite this, *124.m88ksim*, *181.mcf*, *197.parser*, and *300.twolf* all show large gains in coverage from linking.

The benchmark *130.li* exhibits an interesting characteristic where a few weakly executed callers call an important callee. Only one caller is hot enough to be detected and the callee gets inlined into it. This prevents the callee from being a root function and thus 10% of the execution is missed.

5.2 Code expansion

The package construction process causes code expansion due to partial duplication of root functions and successive

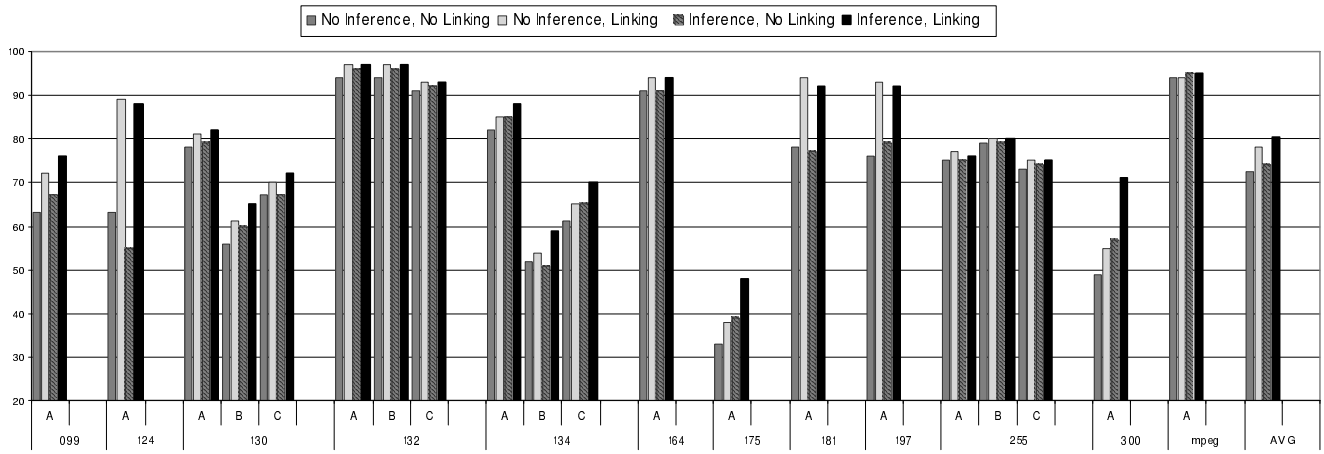

Figure 8. Percent of dynamic instructions from within packages.

Table 3. Code Expansion

Bench	% Incr in size	% Static inst selected
099 A	37.4	10.1
124 A	3.9	2.5
130 A	17.4	7.2
130 B	12.2	7.2
130 C	17.4	7.2
134 A	3.6	1.4
134 B	3.8	1.4
134 C	3.8	1.3
181 A	23.9	7.7
197 A	19.7	3.5

Bench	% Incr in size	% Static inst selected
132 A	7.9	4.2
132 B	7.6	4.4
132 C	9.4	5.7
164 A	9.2	5.8
175 A	6.0	2.7
255 A	15.0	3.0
255 B	15.7	3.2
255 C	16.7	3.1
300 A	7.2	4.0
mpg A	5.8	3.6

inlining of heavily executed callee functions into multiple packages. Table 3 shows the percentage growth of static instructions due to package construction and averages 12% (with a large number of benchmarks substantially below the average). In spite of the specialization required for individual phases, 12% growth covered 81% of the dynamically executed instructions. Table 3 additionally shows the percentage of static instructions that were selected to be a part of at least one package. An average of 4.5% of instructions were selected, yielding an average replication factor for these instructions of approximately 2.6.

5.3 Branch categorization

To explore the value added due to phase-sensitive profiling over traditional aggregate profiling, we categorized the dynamic branches in our experiments. 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 9. The unique branches were then divided into biased and unbiased types, and notably were mostly biased. The biased branches are especially beneficial since the compiler can aggressively assume a particular direction without much risk to performance. Multi branches that show a bias in direction are further subdivided into those that exhibit significant swings in their expected direction. Those that vary between phases ($> 70\%$) are categorized as Multi High, those with more moderate swings, between (40%) and (70%), are Multi Low, while all other biased branches are Multi Same. Any Multi branches that never show a bias are 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 our opportunity for customizing an application for its phases by exploiting the differing behavior in each phase. While only differing by a few percent, the Multi High and Low have significant impact because they now allow the optimizer to wisely choose paths where an ambiguous aggregate profile hampers the decision.

5.4 Initial speedup experiments

Given that high coverage packages have been formed, a number of experiments were conducted to examine the potential for performance benefit achieved by Vacuum Packing. After forming the packages, additional code layout and scheduling passes were applied. Using the method described in [4], block and control-flow arc profile weights were calculated using the taken probabilities of each block in the CFG. For run-time systems, such a calculation may be too computationally expensive and a simpler approximate-weight propagation method may suffice. While not performed in this study, various classic, ILP, and loop optimizations could also be applied to further improve the ap-

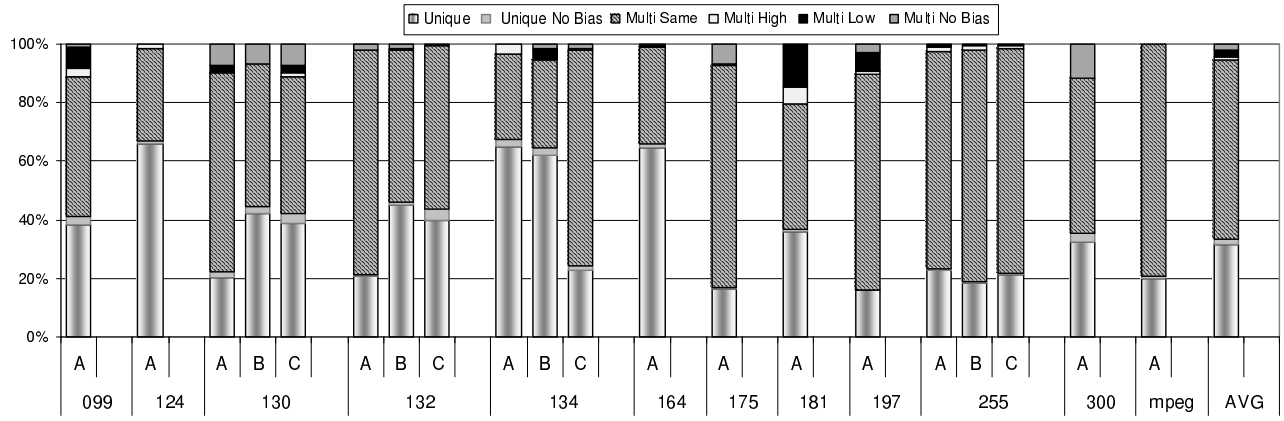

Figure 9. Categorization of hot spot branch behavior across benchmarks.

plication's performance. These optimizations may provide significant benefit since the elimination of cold paths may increase block scope by eliminating side entrances. Further compaction of the code schedule may be achieved by a redundancy-elimination optimization that moves cold instructions (those whose results are not consumed within the hot package) to the side exit block.

Figure 10 shows the resultant program speedup due to package relayout and rescheduling for each benchmark and input. The same four configurations were examined as with the coverage experiments. The average speedup forms a pattern of improvement over the four experiments that correlates to the improvements in coverage, although there are a few differences for individual benchmarks. The increased coverage due to linking in *197.parser* allows execution to reach more specialized code regions, resulting in an additional 8% performance improvement. While the coverage for *255.vortex* was not very different across configurations, the performance shows that both inference and linking provide for more efficient execution of the code.

6 Conclusion

The Vacuum Packing technique has the potential to significantly improve the strategy employed by a wide variety of post-link optimizers and dynamic code generators. Vacuum Packing statically exploits natural execution behavior that is characteristic of many programs in order to create broad but targeted code regions that will serve as the unit of optimization. This improvement will enable optimizers to have a much larger scope and more manageable, structured deployment method than current trace-based systems while maintaining the adaptive, focused benefit of dynamic systems. Future software systems will continue to grow in size and complexity requiring that optimizers and dynamic code generators minimize the amount of altered code along with the number of alterations to keep the overhead of the transformations in check.

Specifically, this work exploits a hardware profiling mechanism for low overhead profiling which identifies the hot branches for each specific execution pattern, or phase, of program activity. While hardware mechanisms for profiling incur minimal overhead, the resultant profiles often suffer from decreased accuracy compared to complete software instrumentation. Vacuum Packing overcomes this inaccuracy by applying a series of inferences and heuristic growth rules to identify the true core instructions for each program phase. The resulting code packages, each targeted toward a specific phase, represent about 81% of program execution and expand the scope of post-link optimization from traces to an entire phase of execution.

In addition, we demonstrate that the phase nature of programs can be exploited even by a static optimization. Based on phase-sensitive information, Vacuum Packing performs partial inlining of functions into their corresponding phase regions. Such partial inlining would be difficult for a static compiler with only an aggregate profile of program behavior. Furthermore, packages for different phases that share common root code can be be exploited statically by recognizing control-flow differences between the phases and using these differences to allow execution to select the proper package.

Acknowledgments

This research has been supported by the MARCO/DARPA Center for Circuits, Systems and Software under contract 2001-CT-888 and gifts from Hewlett-Packard, Advanced Micro Devices, and Microsoft. Erik Nystrom and Matthew Merten were also supported by Intel Graduate Fellowships.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profil-

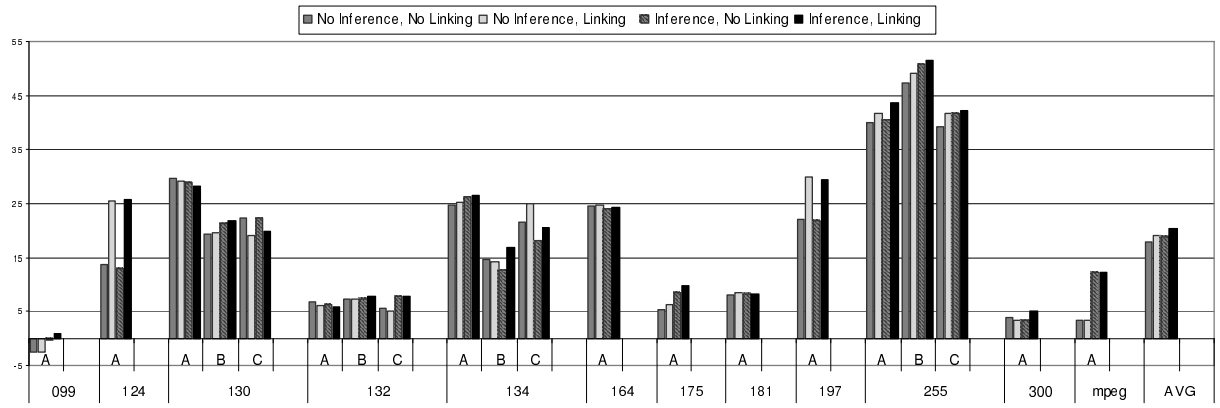

Figure 10. Performance speedup from basic rescheduling of packages.

ing. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, June 1997.

[2] 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*, pages 1–14, October 1997.

[3] 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*, pages 1–12, June 2000.

[4] R. D. Barnes. Extracting hardware-detected program phases for post-link optimization. Master's thesis, University of Illinois, Urbana, IL, 2002.

[5] 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*, pages 13–20, December 2000.

[6] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–19, 1997.

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

[8] 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*, pages 36–45, December 1996.

[9] 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*, pages 292–302, December 1997.

[10] 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*, pages 233–244, May 2002.

[11] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and compilation. *IEEE Transactions on Computers*, 50(6):529–548, June 2001.

[12] 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*, pages 158–168, December 1995.

[13] A. Klaiber. The technology behind Crusoe^{TM} processors. Technical report, Transmeta Corporation, http://www.transmeta.com, January 2000.

[14] AJ KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC 2000 benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Volume 1, May 2002.

[15] 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*, pages 330–335, December 1997.

[16] 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*, 50(6):567–589, June 2001.

[17] 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*, pages 136–147, May 1999.

[18] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[19] 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*, pages 3–14, September 2001.

[20] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary translation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[21] Sun Microsystems. The Java HotSpot^{TM} virtual machine. Technical report, Sun Microsystems, http://java.sun.com, 2001.

[22] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In *Proceedings of the 2000 Workshop on Binary Translation*, pages 55–65. ACM Computer Architecture News, March 2001.

[23] 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*, pages 552–556, June 2002.