Phased Behavior and Its Impact on Program Optimization

Sylvan Clarke, Eric Feigin, Weicon Conan Yuan, and Michael D. Smith
Harvard University

Abstract

Run-time optimization systems are gaining in popularity because they can automatically restructure an executable based on the current program behavior and specifics of the underlying machine. Software-based versions of these systems have several benefits over hardware-based versions: they can support more sophisticated optimizations; they can apply optimizations to larger program regions; and they, like any other software system, can be upgraded as new optimization techniques become available. These benefits, however, come at the expense of a higher run-time overhead. Unlike a hardware-based system which can run continuously during a program's execution, we must selectively apply the power of a software-based system so that its higher overhead is appropriately amortized. This paper investigates *phased behavior*, an aspect of program behavior that software-based systems can use to make them cost effective. This paper illustrates some types of phased behavior that occur in (non-trivial) applications, and it argues why such behavior might be interesting to an optimization system. We view this as an important first step in learning how to identify phased behavior automatically and build systems based on this behavior.

1 Introduction

The goal of code optimization is to streamline applications in ways either difficult or impossible for programmers to accomplish.[1] Programmers usually write applications in high-level languages so that the applications are general (work for a wide range of inputs) and portable (run on a wide range of machines). Yet, when a user runs one of these applications, she wants the application to perform the least amount of work possible for her input set on her current machine. When the execution of an application achieves this desired effect, we say that we have maximized that application's *run-time efficiency.* Generality and portability are characteristics that typically decrease run-time efficiency. Feedback-directed optimization systems are an increasingly-popular way to restructure an application to increase run-time efficiency.

As Table 1 indicates, there exists a wide range of feedback-directed approaches. Each uses feedback to restructure an application's code to take advantage of the target machine features. More importantly with respect to the topic of this paper, each removes redundant work and speculates on the common program behaviors *associated with the computation of the application's result*. Each can, for example, identify highly biased branches and perform path-based optimizations to exploit this knowledge.

The run-time optimization approaches in Table 1 however do more at run time than just compute the application's result. In particular, these approaches collect information summarizing the dominant behavior of

1. Although there are many metrics for code optimization, this paper focuses on optimization systems that make applications run faster. We do believe, however, that the ideas in this paper are applicable to systems that optimize for metrics other than performance, e.g. low power.

Existing Approach	Example Systems
Profile-driven compilation	IMPACT [4]
Off-line optimization with continuous profiling	FX!32 [14], Morph [5]
Run-time code generation (RTG)	‘C [16], DyC [12], Tempo [6]
Run-time optimization in software	DAISY [9], Dynamo [3], Tinker’s DR [7]
Run-time optimization in hardware	out-of-order machines, trace processors [18]

Table 1: Taxonomy of existing feedback-directed approaches. The approaches differ in the timeliness of the feedback information collected and the run-time cost of the feedback-directed optimization performed. For example, profile-driven compilation incurs no run-time costs, and the off-line approaches pay only for the small costs of statistical sampling. However, the benefits of both these approaches are limited by how well past program executions predict the behavior of future executions. As Grant et al. [12] states, all of the RTG systems rely on programmer annotations, and thus these approaches do not incur any costs related to the collection of feedback information. When RTG systems are driven by automatically collected feedback information, they will incur costs similar to those associated with run-time optimization in software.

the application during its recent execution, analyze this information to determine if re-optimization is necessary, and restructure the executable based on this information (assuming the analysis indicated it was necessary). Run-time optimization in hardware is an especially popular approach because exact information about the program’s behavior is available and the ever-increasing VLSI transistor budgets have made this a cost-effective approach. The software-based, run-time optimization approaches attempt to obtain the benefits of feedback information specific to the current run while minimizing the cost of the collection, analysis, and use of this information.

In particular, we can characterize (in general) the software-based approaches as *look-once* and the hardware-based approaches as *look-always* schemes. Both schemes observe the program behavior whenever an application begins executing a piece of its text segment for the first time. A look-once scheme uses this initial observation as the basis for run-time optimization. After this initial look, the system assumes that the this initial behavior is the dominant behavior and no longer observes the behavior of this code segment. In this way, the system can amortize the cost of software-based, run-time optimization on the first (few) executions of each piece of the code across all executions of that code. The initial implementation of DAISY [9] is a good example of this kind of system.[2] An out-of-order machine, on the other hand, is an example of a look-always scheme; the hardware reschedules a section of code every time it is encountered. No effort is made to eliminate the redundant work done when the parameters to the hardware scheduling algorithm remained unchanged from execution to execution. Though not all existing run-time optimization approaches fit neatly into this general categorization, the software-based and hardware-based approaches have not moved too far from look-once and look-always, respectively, as discussed further in Section 4.

In the Deco project at Harvard, we are interested in developing a feedback-directed optimization approach that is able to obtain and exploit timely feedback information from any point in an application’s execution

2. For the purposes of this paper, we view binary translation as simply another kind of code optimization.

and is able to do so while performing a minimal amount of redundant work in the run-time optimization system. Such a system should be able to automatically identify repetitious program behavior and dynamically optimize the appropriate code segment for that behavior. We currently have pieces of a prototype of such a system running at Harvard [11]. It is a software-based approach that uses extremely lightweight mechanisms to identify when the dominant program behavior changes. More sophisticated (and thus more run-time expensive) observation and optimization techniques are invoked only when the lightweight mechanisms deem these more extensive measures appropriate. This paper however is not about the prototype, but about an aspect of program behavior that enables such an optimization approach to be effective.

Our prototype exploits an aspect of program behavior that we call *phased behavior*. We define phased behavior as the tendency for a piece of code to exhibit a sequence of behaviors, each for an extended period of time. For the purposes of this definition, the time period may correspond to a single program run, or it may span multiple runs. In an application that displays phased behavior, there exists at least one code region where optimizing for the aggregate behavior is sub-optimal. We can obtain better performance if we apply one optimization strategy to that region for one phase of the program's execution and another strategy during another phase. Phased behavior within a program run is problematic for the look-once schemes, while phased behavior across program runs is problematic for the off-line approaches.

This paper is organized as follows: Sections 2 and 3 show that interesting phased behavior occurs in real programs, and these sections provide some insights into why each example behavior might be interesting to an optimization system. The examples in these sections are not meant as an exhaustive listing of the interesting kinds of phased behavior. In fact, our goal is quite the opposite: to encourage researchers to look beyond the simple maxims of program behavior that currently dictate many of industry's design decisions (e.g., the 90/10 rule [13]). Section 4 discusses this line of thinking in more detail, and it describes how some of the existing systems in Table 1 indirectly deal with phased behavior. Section 5 concludes with some of our beliefs about the potential of this work in optimization and other research areas.

2 Branch-based phased behavior

We begin our review with examples of phased behavior associated with the execution of conditional branches. The run-time handling of conditional branches has a tremendous effect on the performance of modern systems, and thus any new insights that we can gain in this area will likely have an important impact on the industry.

This section opens with obvious examples of phased behavior in the execution of individual branches and then moves to less obvious (but probably more common) examples. We conclude this section with a discussion of some thoughts on the effect of these observations on program optimization. The interesting issue here is that all of the examples involve branches that are weakly biased; current optimization strategies concerned with conditional branches are tailored toward the identification and exploitation of highly-

biased branches. Except for code transformations like static correlated branch prediction [19], weakly biased branches are largely ignored by modern optimization techniques.

2.1 Phases in branch execution

Our first few examples focus on two conditional branches that occur in the `shade` procedure from `ray.c` of *rayshade*, the C ray-tracer developed by Peter Holst Andersen [2]. Figure 1 lists the code associated with these two branches, which we label b35 and b36. This code looks much like the code associated with the other conditional branches in `ray.c`, and thus there is not any obvious syntactic clue alerting us to the following interesting program behavior.

```
if (flat_object(isect.object)) {    // b35
  if (n_dot_v < 0.0) {              // b36
    // ...
  }
```

Figure 1. Code snippet from shade() *in* ray.c.

Figure 2 plots the execution trace of b36 when *rayshade* is run on the input *scene1*. In the aggregate, b36 is a weakly-biased branch, a branch taken 44.0% of the time. As Figure 2 illustrates however, b36 is taken exclusively in its first 123,090 executions, and it is never taken after that. This is a clear example of phased behavior; b36 exhibits one behavior and then another, each for a long period of time.

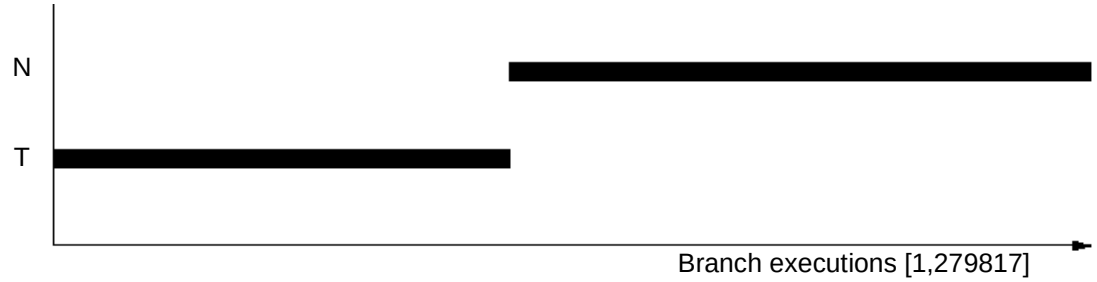

Figure 2. Execution trace of b36 on the input scene1. *The horizontal axis plots time from left to right as measured in executions of this conditional branch. This branch executed a total of 279,817 times. For each branch execution, we mark in the vertical dimension whether the branch was taken (T) or not taken (N).*

Figure 3 plots the execution trace of b35 during the same run. This figure displays an interleaving of taken and not-taken executions (non-shaded segments) that does not appear in Figure 2. Even so, the execution of b35 contains a long period of time where it is always taken and two smaller periods where it is always not taken. Overall, nearly 50% (49.1%) of this branch's execution is spent in the large taken segment—fairly amazing for a branch that is taken 66.9% of the time in the aggregate.

Figure 4 shows that the phasing behavior in Figure 2 is not something that we could exploit using traditional profile-driven optimization. Figure 4 plots the execution trace of b36, but this time when *rayshade* is run on the input *scene24*. Comparing Figures 2 and 4, we see that b36's execution is now more "random"

Figure 3. Execution trace of b35 on the input scene1. *We use shading to distinguish contiguous segments of taken (or not-taken) executions from those segments that have an interleaving of taken and not-taken executions. For example, b35 is only taken between executions 183272 and 595513, inclusive.*

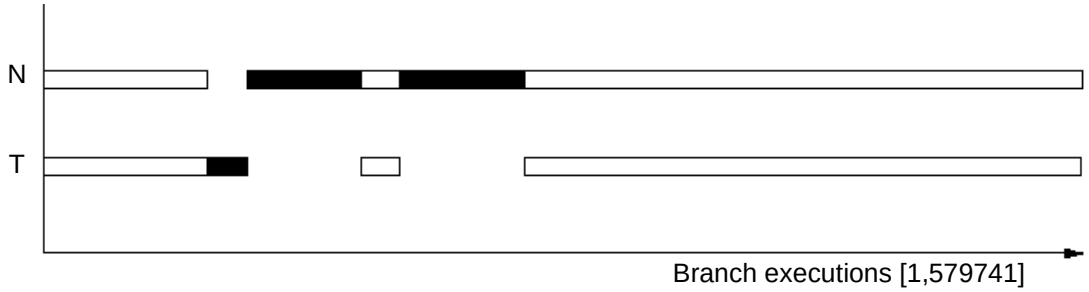

Figure 4. Execution trace of b36 on the input scene24.

than it was under *scene1*. The phased behavior of this branch is influenced by the input data set. However, the aggregate behavior of this branch has not changed noticeably; b36 is taken 39.8% of the time under *scene24* and 44.0% of the time under *scene1*.

2.2 Phases in branch bias

As one might expect, the execution trace of a weakly-biased branch does not always contain long periods of time where the branch executes in only one direction. As illustrated in Figure 5 however, some of these weakly-biased branches still exhibit phased behavior that is interesting to an optimization system. In particular, Figure 5 looks at phased behavior in the bias of a conditional branch (labeled b99) in the SPECint92 benchmark *compress*.

Figure 5 shows that the bias of b99 oscillates between periods of low and high bias over time. In the aggregate, this branch would be classified as a weakly-taken branch, based on its average taken percentage of 78.0%. Clearly, this is the wrong classification for periods, like the beginning of the program, where the branch is nearly always not taken.

The actual shape of the curve in the graph plotting bias against time depends upon the characteristics of the input data set. To understand this, we first must explain a bit more about the functioning of the code in this section of *compress*. The branch b99 is part of the hashing code for the code table in *compress*. Specifically, b99 checks to see if the probed slot in the code table is empty after we found that we did not have a hash hit. This explains the initial increase in the bias of this branch; the table is initially empty and slowly

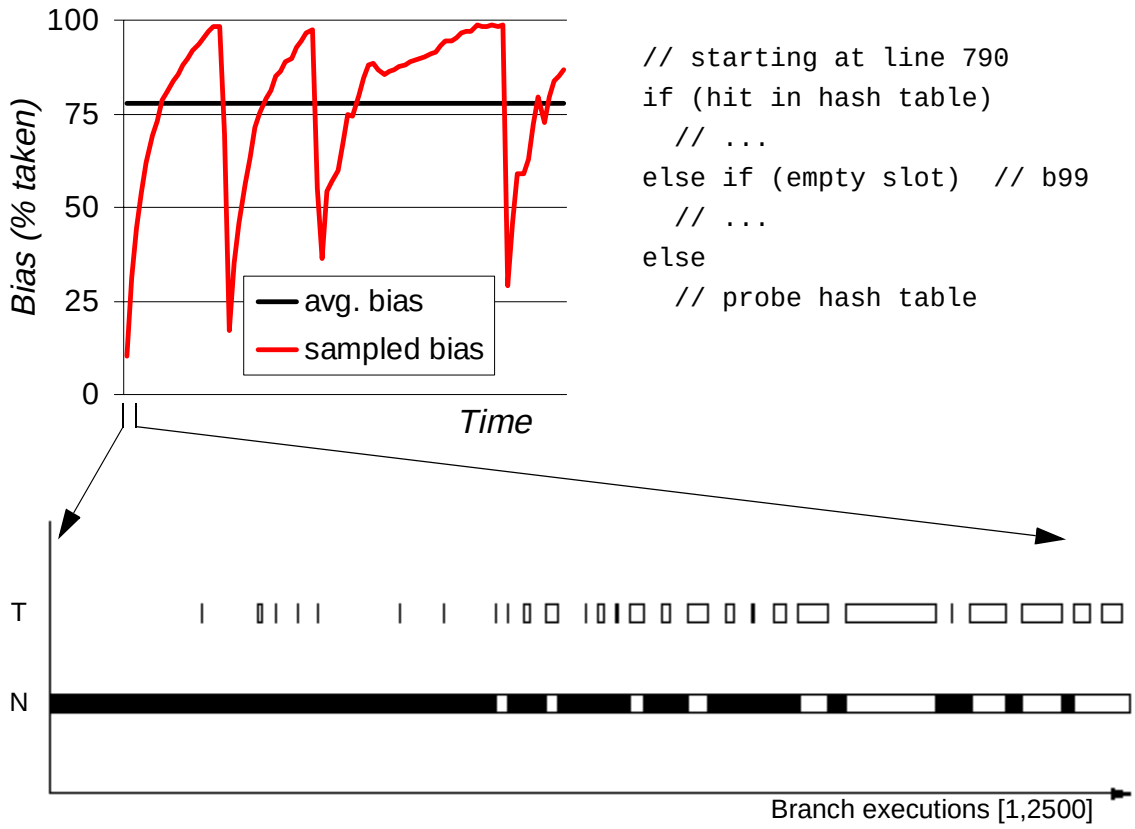

Figure 5. Execution trace of b99 in compress.c. *The graph at the upper left plots the bias of this branch at a granularity of 5000 executions. This branch executed a total of 433,223 times. The graph at the bottom of this figure illustrates the pattern of taken and not-taken outcomes in the first 2500 executions.*

fills changing the branch's bias. The later abrupt changes from a high to a low bias are due to the fact that *compress* implements block compression with an adaptive reset, as described in the comments in the source code of *compress*. If the compression ratio begins to decrease after the code table reaches a predefined fill threshold, the program will clear the code table, output a special CLEAR code, and begin a new encoding. Since this criterion depends upon the characteristics of the input, the shape of the plotted curve also depends upon the characteristics of the input.

2.3 Discussion

The vast majority of optimizations concerned with branching behavior focus their attention on the aggregate behavior of each branch. For the branches discussed in Section 2.1, this would result in lost optimization opportunities. The clear phased behavior in Figure 2 leads us to consider one optimization strategy for the first 123,090 executions of b36 and a different strategy for the last 156,727. Furthermore, given the dependence of b36's behavior on the input data set, we would probably want to implement this flexibility in a run-time optimizer. Considering b35, adaptation between an optimization strategy involving speculation (to take advantage of long execution runs in one direction) and predication (for those execution runs

with no obvious predominant direction) is a potentially fruitful avenue of attack for the branch execution pattern in Figure 3.

Section 2.2 broadened our view of branch-based phased behavior. This broadening is interesting because some optimizations, like those based on speculative execution, do not require long runs of a single branch outcome. Adaptation in the optimization strategy can still occur over time to reflect changes in the coarser view of the branch's execution, such as those in Figure 5.

Finally, the existence of sizable periods of high bias in the execution history of weakly-biased branches implies a change in the importance of individual program paths over time. This observation may affect how we build path-based optimizers. For example, if a branch within a program path transitions from mostly taken to mostly not-taken (or vice versa), it is likely that the common paths before and after this transition have many code blocks in common. To get the most benefit, a path-based optimizer would duplicate the common blocks. Code duplication impacts performance by stressing the memory hierarchy. If we can identify the phase transitions, we could dynamically generate and cache the appropriate path-based optimization—potentially improving performance by never having both versions in memory at the same time and by linking each in as if the other had never existed. To accomplish this, we must be able to not only identify the phase change, but also be able to link the change to the affected optimization region.

3 Value-based phased behavior

This section provides examples of phased behavior in the execution of load instructions. In particular, we look for phased behavior in the values returned by the load instructions. Techniques like value prediction have increased the research community’s interest in the values loaded by programs. Like conditional branches, the handling of memory loads has a large impact on the overall performance of applications.

Figure 6 plots the value trace of a load (called d2542) from the procedure `BMT_CommitParts` in `bmt10.c` of the SPECint95 benchmark *vortex*. This instruction loads the base address of the array `Vlists` from the global pointer table. As Figure 6 shows, the program initially accesses one array in memory through this pointer variable and then switches to another array for the remainder of the run. Even though this load returns more than one value, a run-time optimizer could take advantage of the fact that the values are not interleaved in the value trace.

As we saw in Section 2, interesting phased behavior may occur in traces where there is an interleaving of different results. Figure 7 plots the value trace of a load (called d4531) at the end of procedure `addunsigned` in `dpath.c` of the SPECint95 benchmark *m88ksim*. This load restores the return address register just before it is used in the procedure return.The value trace shows that `addunsigned` is called from just two call sites (named cs1 and cs2 for convenience) during the program run. The interesting aspect of this trace is that, except for two small pieces at the beginning and end of the trace, `addunsigned` is

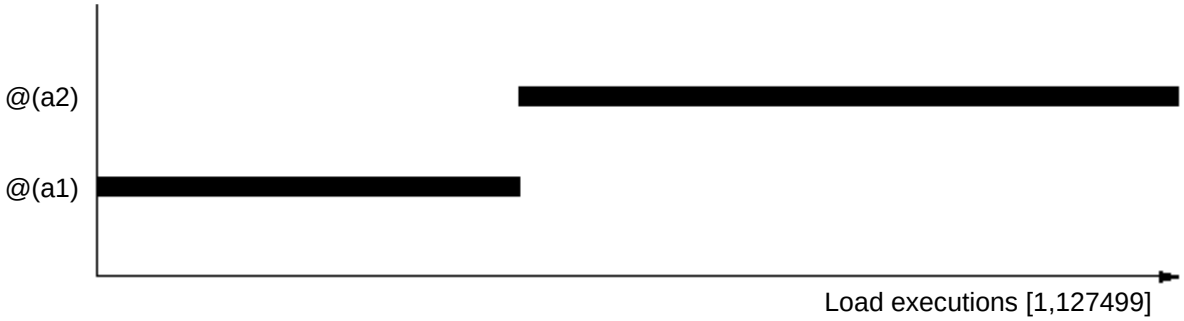

Figure 6. Value trace of load d2542 in vortex. *The horizontal axis plots the executions of this load from left to right. For each load execution, we record in the vertical dimension the value returned. This load returns only two unique values: the base addresses of arrays named (for convenience) a1 and a2.*

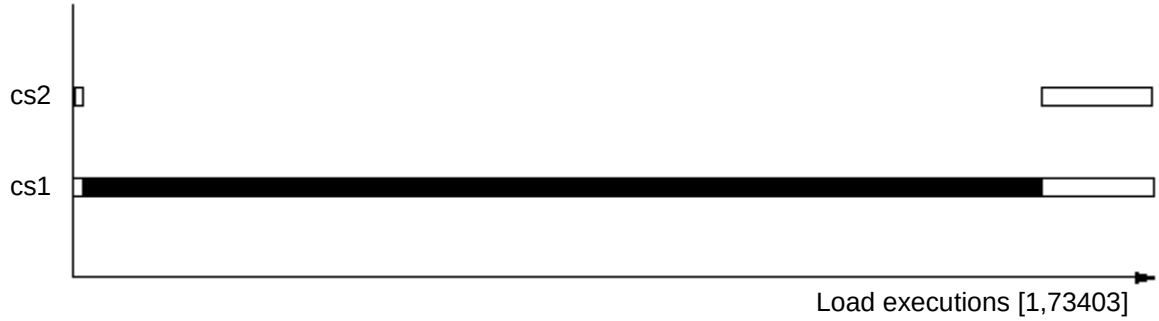

Figure 7. Value trace of load d4531 in m88ksim. *This load restores the return address register. Since the load returns only two unique values,* `addunsigned` *is called from only two call sites during this run.*

called exclusively from cs1. During this extensive time period, a run-time optimizer could elect to inline `addunsigned` at cs1 and do so without increasing the instruction memory footprint.

Our last value trace is again from a load (called d1988) in *vortex*. Figure 8 shows that interesting phased behavior can occur even when the load in question returns more than a pair of values. In this case, d1988 has long periods of time where it returns 0 exclusively. A run-time optimizer would want to identify these time periods and construct a specialized version of the code that is optimized based on this predicted value.

4 Related Work

In general, this paper is about how gaining a better understanding of program behavior can enable us to build better systems. For years, architects and compiler writers have relied upon several simple rules of program behavior to dictate their design decisions. The 90/10 locality rule, which states that nearly 90% of the program's run time is spent in approximately 10% of its code [13], has had an influential impact on the design of modern memory systems, for example. More recently, the realization that a large percentage of conditional branches are highly biased has led to improvements in branch prediction and to the successful development of path-based optimizations.

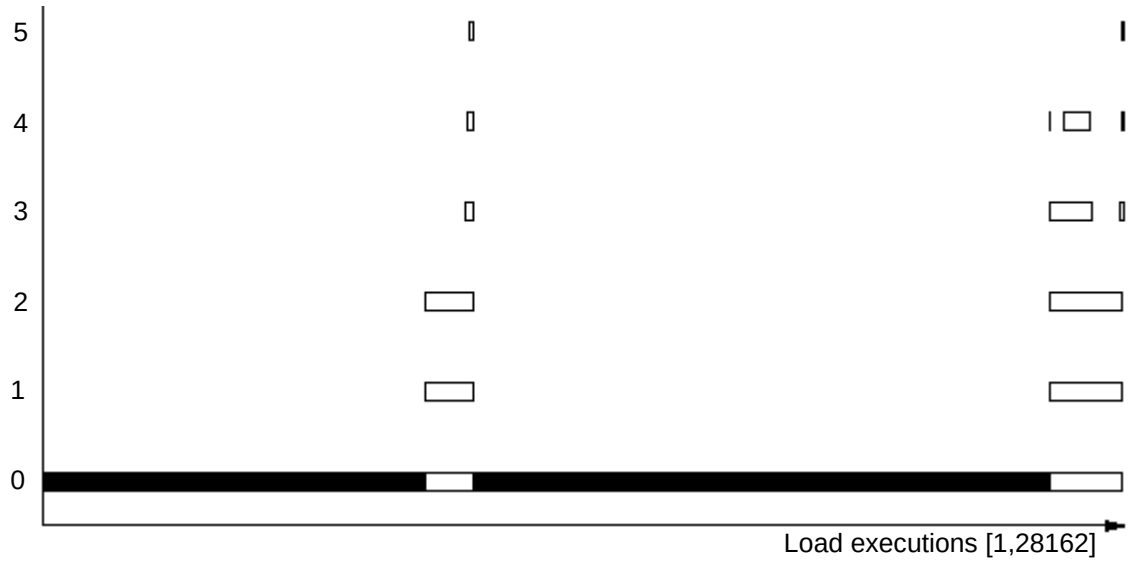

Figure 8. Value trace for load d1988 in vortex.

Several researchers have investigated the time-based behavior of applications with respect to a single metric. For instance, Denning's [8] concept of a working set is often explained using a plot of pages referenced over time. Jouppi [15] examined the distribution of instruction-level parallelism in applications over time, and more recently, Albonesi [1] proposed an adaptive processor architecture based on such observations.

Within the domain of run-time optimizers, we were not able to find any existing system that directly addresses the phased behavior of applications. Several projects, e.g. the IBM DAISY system [10], have proposed a tiered approach to optimization, where a code region is "lightly" optimized at first and then later optimized aggressively when it has been shown to be frequently executed, but this is done more for reasons of amortizing the cost of the aggressive optimizations. The HP Dynamo system [3] implements a policy in its code cache that allows it to react to changes in the application's working set. Since this system executes code only from its code cache, this policy exists primarily to minimize the size of the code cache and not to exploit phased behavior in the application.

5 Summary and future directions

We have presented several examples of phased behavior and made arguments for how a run-time optimizer could benefit from the identification of such behavior. We acknowledge that many of the example behaviors identified in this paper may not be amenable to optimization by current feedback-directed systems. This is ok; this is not a "limits" study. We expect that as our understanding of program behavior grows, we will be able to envision new optimization techniques and possibly even new hardware organizations.

The next step is to develop techniques that allow us to identify interesting phased behavior in time and with enough accuracy to allow a run-time optimization system to link the identified behavior to an optimizable region. Even if we are able only to characterize this type of behavior, researchers in areas like simu-

lation [17] and adaptive architectures [1] will be able to use this knowledge (for example) to know when and for how long to simulate and to know when to reconfigure their hardware organizations. Our goal is to use this information to build effective software-based run-time optimizers that will fill a niche in the existing spectrum of optimization approaches.

6 References

[1] D. Albonesi. “Dynamic IPC/Clock Rate Optimization,” Proc. 25th Annual Int. Symp. on Computer Architecture, pp. 282–292, June 1998.

[2] P. Andersen. “Partial Evaluation Applied to Ray Tracing,” unpublished technical report, January 1995.

[3] V. Bala, E. Duesterwald, S. Banerjia. Transparent Dynamic Optimization. Report HPL-1999-77, HP Laboratories Cambridge, June 1999.

[4] P. Chang, S. Mahlke, and W. Hwu. “Using Profile Information to Assist Classic Compiler Code Optimizations,” Software Practice and Experience, 21(12):1301–1321, Dec. 1991.

[5] J. Chen, et al. The Measured Performance Of Personal Computer Operating Systems. ACM Trans. on Computer Systems, 14(1):3-40, Feb. 1996.

[6] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall and J. Noyé. "Tempo: Specializing Systems Applications and Beyond," ACM Computing Surveys, 30(3es): Sept. 1998.

[7] T. M. Conte and S. W. Sathaye. “Dynamic rescheduling: A Technique for Object Code Compatibility in VLIW Architectures,” in Proceedings of the 28th Annual Int. Symp. on Microarchitecture, Nov. 1995.

[8] P. Denning. “The Working Set Model for Program Behavior,” Communications of the ACM, May 1968.

[9] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility, Proc. 24th Annual Int. Symp. on Computer Architecture, pp. 26–37, June 1997.

[10] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind. “Execution Based Scheduling for VLIW Architectures,” Proc. Europar '99, Topic 16: Instruction Level Parallelism and Uniprocessor Architecture, Sept. 1999.

[11] E. Feigin. A Case for Automatic Run-Time Code Optimization. Senior Thesis, Harvard College, Div. of Eng. and Applied Sciences, April 1999.

[12] B. Grant, et al. “An Evaluation of Staged Run-Time Optimizations in DyC,” Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 293–304, May 1999.

[13] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Palo Alto, CA, 1990.

[14] R. Hookway and Mark Herdeg. DIGITAL FX!32: Combining Emulation and Translation. Digital Technical Journal 9(1):3–12, 1997.

[15] N. Jouppi. “The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance,” IEEE Trans. on Computers, 38(12):1645–1658, Dec. 1989.

[16] M. Poletto, D. Engler and M. Kaashoek. "tcc: a System for Fast, Flexible, and High-Level Dynamic Code Generation," Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation, pp. 109–121, June 1997.

[17] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. “Complete Computer Simulation: The SimOS Approach,” IEEE Parallel and Distributed Technology, 3(4):34–43, Winter 1995.

[18] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. “Trace Processors,” Proc. 30th Int. Symp. on Microarchitecture, Dec. 1997.

[19] C. Young and M. Smith. “Improving the Accuracy of Static Branch Prediction Using Branch Correlation," Proc. 6th Int. Conf. on Architectural Support for Programming Lang. and Operating Systems, pp. 232–241, Oct. 1994.