*In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept. 2005.*

# Variational Path Profiling

Erez Perelman[†] Trishul Chilimbi[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Microsoft Corporation

**Abstract**

*Current profiling techniques are good at identifying where time is being spent during program execution. These techniques are not as good at pinpointing exactly where in the execution there are definite opportunities a programmer can exploit with optimization.*

*In this paper we present a new type of profiling analysis called* Variational Path Profiling *(VPP). VPP pinpoints exactly where in the program there are potentially significant optimization opportunities for speedup. VPP finds the acyclic control flow paths that vary the most in execution time (the time it takes to execute each occurrence of the path). This is calculated by sampling the time it takes to execute frequent paths using hardware performance counters. The motivation for concentrating on a path with a high net variation in its execution time is that it can potentially be optimized so that most or all executions of that path have the minimal execution time seen during profiling.*

*We present a profiling and analysis approach to find these variational paths, so that they can be communicated back to a programmer to guide optimization. Our results show that this variation accounts for a significant fraction of overall program execution time and a small number of paths account for a large fraction of this variation. By applying straight forward prefetching optimizations to these variational paths we see 8.5% speedups on average.*

## 1 Introduction

Current profiling techniques are good at informing programmers where time is being spent during program execution. Many programmers use hierarchical and path profiling [13, 10, 15, 7, 16, 19] tools to identify the basic blocks and procedures that account for most of the program's execution time. In addition, hardware performance counter based profiling tools like VTune [5] are used to determine program regions that incur a large number of cache misses and branch mispredictions. These tools allow programmers to attempt to speedup program execution by focusing on the frequently executed loops as well as those that account for the most misses. Unfortunately, these profiling techniques lack the ability to direct a programmer to the program section that has the largest speedup opportunity for optimization. For example, a profiling tool may report that the largest fraction of execution time is spent in an inner loop of a matrix multiply routine. This information is unlikely to be very useful for speeding up the program as it is likely that this routine has already been heavily optimized.

To address this shortcoming, we focus on identifying acyclic program paths that have the highest net variation in their execution through what we call *Variational Path Profiling* (VPP). To find the paths with variation, VPP records the execution time it takes to execute frequent acyclic control flow paths using hardware counters. The hardware cycle counter is recorded at the start of the path's execution and again at the end of its execution. This is accomplished with minimal overhead using the Bursty Tracing framework [14], which is configured to sample paths as is done in [9].

Variational Path Profiling takes these timing measurements and finds the control flow paths that have significant net variation in execution time across the different time samples. The *net variation* is determined by finding the smallest execution time for a path, and then summing up the additional time it took to execute each invocation of that path over its smallest execution time. A path with high net time variation implies that there are certain executions of that path that took significantly more time to execute than others. The key insight of our approach is that this variance in execution time potentially represents significant opportunity for speedup. This comes from the idea that every dynamic invocation of that path should be able to only take the minimal sampled execution time assuming the stalls on the time consuming invocations can be removed with optimization. We observed that this variation in execution time of the same path can arise from differences in the micro-architectural state of the machine during different dynamic path invocations. For example, during one execution of a path, all of the load accesses can hit in the cache, and another execution of the same path can have loads that miss in the cache.

Part of the motivation for this research came from looking at highly optimized commercial applications. For these applications, some of the most frequently executed (hot) paths had very little variation in execution time. They had little variation because these paths had already been highly optimized and there was little to be gained by trying to further optimize them. To gain additional speedups we instead had to look at finding the paths with a high net variation. Our variational

path profiling approach pinpoints exactly these paths with net high variation so a programmer can focus on optimizing paths with a large speedup potential. VPP is intended to complement conventional program profilers, not replace them. For example, VPP will not report program paths that are uniformly slow across all executions, which a programmer may also be interested in.

The main contributions of this paper are:

- We present a low overhead variational path profiling approach to pinpoint paths in a program's execution that have large net variation in their execution time.
- We show that this variation accounts for a large fraction of overall program execution time. In addition, we show that a small number of paths (top 10) account for most of the variation and these same paths are identified under different program inputs and system load conditions.
- We provide examples that use this variational path information to perform very simple optimizations for a handful of the SPEC benchmarks resulting in execution time improvements of 8.5% on average.
- We provide a comparison to other common hot path profiling techniques, and demonstrate the advantage of VPP in finding high net variation paths for three commercial programs that have already been heavily optimized.

The rest of the paper is as follows. First, a summary of the related work is presented in Section 2. Section 3 describes the methodology used in this research and Section 4 presents the profiling approach used to collect the variational path profiles. The analysis of these profiles, optimization results and a comparison to other path profiling techniques are presented in Section 5, and the paper is summarized in Section 6.

## 2 Related Work

The work most closely related to ours, is the correlation profiling approach of Mowry and Luk [17]. Correlation profiling was motivated by the desire to apply latency tolerating techniques to the specific set of dynamic load instructions that suffer cache misses. They look at individual static load instructions and attempt to identify a dynamic context in terms of control flow, or past history of cache misses that correlates well with a load instruction incurring a cache miss. They found that for some applications the dynamic calling-context leading up to the load helped distinguish dynamic instances of the same load instruction that miss, from those that hit in the cache.

Variational profiling can be viewed as somewhat orthogonal to correlation profiling. Our focus is on the variation in execution time of a program path whereas their focus is on variation in contexts leading up to a load instruction and predictably correlating a specific context with an outcome (i.e., cache miss). Their work suggests contexts that may be useful in distinguishing slow instances of our variational paths from fast ones. Another difference arises from their focus on cache misses. While cache misses are certainly a very important source of variation, many others exist, such as branch mispredictions, pipeline stalls, operating system calls, lock contention, and network events. Finally, our focus on program paths rather than individual load instructions offers benefits in the context of a profiling tool. For example, we show that focusing on the top 10 variational paths concentrates on the bulk of the potential benefit, whereas the number of load instructions that incur cache misses is likely to be significantly higher in a reasonably sized program.

*Path profiling* (PP) associates hardware performance counter metrics with acyclic program paths and provides context information for these paths in the form of a data structure called a calling-context tree [6]. It uses the Ball-Larus technique to efficiently profile paths [10]. Metrics are summarized per acyclic path by sampling the hardware counters at path begin and end and accumulating them in a 64 bit path counter. The calling context tree provides contextual information for the path that is intermediate in precision between a dynamic calling tree and a dynamic call graph. Like PP, we sample hardware counters (cycle counter) at path begin and end. However, we maintain and compare these metrics for individual path invocations rather than summarize them for all invocations of a path as PP does. This fine-grain profiling allows us to identify paths with high net variation. We use Bursty Tracing [14] to lower the overhead of our technique. Associating calling-context trees with variational paths may help better identify opportunity, and this is left for future research.

DCPI [8] and ProfileMe [11] are extremely low-overhead profiling tools developed at Digital's research labs that associate micro-architectural events such as data cache misses, branch mispredicts, etc., with individual instructions. DCPI uses periodic hardware cycle counter interrupts and static binary analysis to provide instruction-level information on in-order processors. ProfileMe uses similar techniques along with modest hardware modifications to provide identical information for out-of-order processors. These tools provide detailed information but still summarize the information at the instruction level. Consequently, they accurately point to instructions responsible for performance bottlenecks, but are unable to distinguish those that are possible to optimize from those that are impossible or that would require significant effort. Since variational profiling focuses on paths that include at least one fast instance, there is a strong likelihood that some of the slower instances can be optimized.

VTune [5] is a performance analyzer developed at Intel that collects time and event-based samples. Sampling interrupts the processor after a certain number of events and records the execution information in a buffer area. When the buffer is full, the information is copied to a file. After saving the information, the program resumes operation. In this way, VTune maintains very low overhead (about one percent) while

sampling. For this work we require a specialized sampling regiment that can handle fine grained observations with timing information at the start and end of the path to be sampled. This sampling approach is currently not available in VTune and other standard hardware sampling based profilers.

# 3 Methodology

We performed our analysis for the SPEC2000 programs `ammp`, `art`, `bzip`, `equake`, `gcc`, `mcf`, `parser`, `twolf`, `vortex` and `vpr`. We chose these programs because they are challenging and provide a good representation of the suite. Each of the programs also exhibit interesting behavior that is unique.

In addition to the SPEC2000 programs, we analyzed three Microsoft programs: `foxpro` [1] a database tool, a PC game and a multimedia application. Unlike the SPEC2000 programs, these programs are highly optimized and provide an interesting case study with VPP. We do not provide optimization results for the Microsoft programs, only variational analysis, since we did not have access to the source.

Our platform for experimentation is a single CPU-Pentium 4, 2.60 GHz with 1 GB of RAM. It is running Windows XP Professional with Service Pack 2. The programs are compiled with the Visual Studios .Net compiler [2] with full optimizations enabled. Vulcan [18], a binary modification tool, was used to instrument the binaries after they were compiled. All performance results are measured from the hardware cycle counter.

# 4 Variational Path Profiling

In this section we describe the sampling approach used to gather the variational path profiles.

## 4.1 Bursty Tracing

Variability in program execution can be observed across many metrics- memory miss rates, branch misprediction, power and performance. In this work we observe variations in execution time for code paths. To measure these variations we employ the sampling technique described by Arnold and Ryder [9] using the Bursty Tracing framework [14]. Bursty Tracing enables periodic sampling of execution intervals with low overhead. The rate of sampling and size of samples can be tuned, and provides a detailed view of execution at a minimal cost.

Bursty Tracing minimizes profiling overhead by executing a program with very light instrumentation the majority of the time, and occasionally transitioning into a fully instrumented version to do profiling. Figure 1 shows an example of the Bursty Tracing framework. The original procedure is cloned, and the clone is embedded with instrumentation to do the desired profiling. A dispatch check is inserted at the back edges to manage execution transition between the two versions represented as the diamond in the figure. The dispatch check is managed with two counters: *nCheck* and *nInstr*. Initially

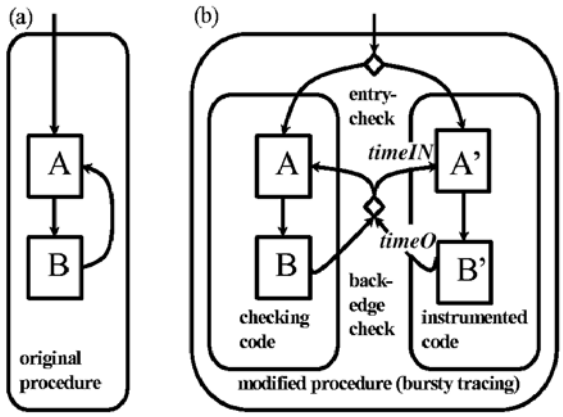


*Figure 1: An example of how Bursty Tracing is used to collect the variational path profiles.*

*nCheck* is set to a *nCheckThresh* (e.g. 10,000 in our case) and *nInstr* is set to zero. Execution begins in the checking code, and every time it passes through the dispatch check the *nCheck* is decremented. Once *nCheck* reaches zero, *nInstr* is set to *nInstrThresh* and execution transfers to the instrumented code. Each iteration through the back edge *nInstr* is decremented. Once *nInstr* reaches zero, *nCheck* is initialized to its threshold again (10,000), and execution transfers back to the clean code. This process is repeated until the program finishes execution. Bursty Tracing applies this process to the entire program by duplicating all the procedures and inserting dispatch checks along all back edges and procedure calls.

## 4.2 Collecting Path Timings

We use Bursty Tracing to measure execution time of acyclic paths. Our profiling is very time sensitive, and we want to accurately measure how much time an acyclic path executes for. To gather the path timings we use the hardware cycle counter to record a time stamp before and after the execution of the path. These points are denoted in Figure 1 along the edges leading in and out of the path as *timeIn* and *timeO* respectively.

We achieve this by setting the Bursty Tracing parameter *nInstrThres* to 1. This causes a sample to be collected for a single iteration of an acyclic path in the instrumented code. Since timing profiles are collected only for acyclic execution paths, we do not measure aggregate timings for consecutive iterations of acyclic paths. This is because instrumentation overhead can bias the timing profiles, and we want this bias on each sample to be consistent. Timing consecutive iterations through the instrumentation code will make it harder to keep a consistent bias and to apply our analysis.

Dispatch checks are the transition points between the two copies of the binary. All acyclic path boundaries are demarcated by dispatch checks. Since we insert dispatch checks only at procedure entries and loop backward branches, all of our acyclic paths either start at a procedure entry or a loop backward branch and also end at one of these points.

Each profiling sample records in a data structure the execution time from entry to exit of a single acyclic path of execu-

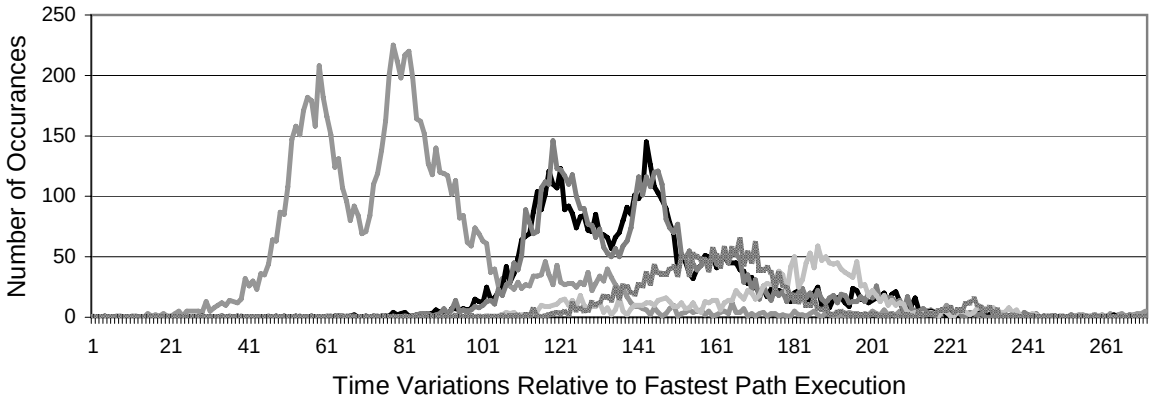


*Figure 2: Bzip2 variational path histogram shows the top 5 ranked paths with variation from the basetime. The x-axis measures time variations relative to the basetime, where the origin denotes the basetime. The further along the x-axis the larger the time delta is for a path sample from the basetime. The y-axis measures the number of paths that had that much variation (x-axis) from the minimum timed basetime path.*

tion, the branch history for the path that was executed, and the PC at the entry point to the path. At the end of program execution all of these samples, which is the contents of the profile, are stored to disk to be post processed as described in the next section to find the paths with high net variation. The branch history is stored as a series of bits indicating if the branch was taken or not.

There are two Heisenberg effects which we consider significant: (1) uncertainty in the timing of a path in its natural state vs. its instrumented state, and (2) if there is a bias, is it consistent across all profile instances?

To address the first uncertainty, we reduced the bias introduced from instrumentation by carefully measuring time only for executing code that is in the original binary and not for the instrumentation code. We read the cycle counter right before path execution begins and again right after it exits. This will provide the most accurate time for how long execution took for the path. Still, we believe there is a small bias introduced by the call to the TimeStamp counter and the additional branch instructions. We have collected results for bias analysis, and discovered that this bias is extremely consistent across all profile instantiations.

This addresses the second point, in that the bias is systematic and can be removed from the measurements. Since we are looking at timing variations between iterations of a path, the actual size of the bias is not as important as the consistency of the bias. We found that the bias is small and systematic. In addition, we found it is sufficient to sample once every 10,000 path iterations, which carries a small overhead of less than 5% slowdown.

Profiling an execution to completion produces a file with all the sampled paths' timings, branch histories, and PCs. A timing for a path consists of two time stamps: one before the path executes and one after exiting the path. To compute the path execution time we subtract the entry time from the exit time. This will provide a precise duration in cycles of how long the path executed. We describe in the following section how we use this data to analyze the path variation.

# 5 Path Variational Analysis and Optimization

In this section we describe how we use the data profiled with Bursty Tracing to analyze path variation and select paths with the highest net variation for optimization.

## 5.1 Path Variability and Picking the Top N Variable Paths

Bursty Tracing collects profiles for all paths that execute more than the sampling rate- ( 1 sample for every 10,000 path iterations). Each path profile contains three data types that we use in analyzing path variability:

1. Path entry point: PC
2. Path branch history as a string of bits
3. A cycle count of how long it took to execute the path

We combine the path PC and branch history bits to form a path signature as a PC-branch history. This signature precisely indicates where in the code the path begins, and a mapping through all the branches until the path exits. It also captures path length in terms of number of basic blocks. For example, the path signature, `0x0040211F-110`, means the path starts at PC=0x0040211F, executes through the first two taken branches and a third non-taken branch, and is 3 basic blocks long.

We use the path signature to filter for each unique path's time samples. We keep only the paths for which we have two or more time samples; the minimum required for measuring variability. We now describe the analysis to compute the path variation from a unique path's execution times.

The variation analysis on a path computes a measurement of variability: how much time variation the path exhibits which has potential for optimization. We first compute a path's *basetime*, which is the minimum execution time we

observe for a path. Next, we compute a time delta for each of the other path times by subtracting the basetime from it. These time deltas signify how much extra time the path executed for. The net path variability is then computed by summing up all the time deltas.

We define the *path net variation time* for a single path *i* as follows:

*path net variation time(i) = total path execution time(i) –* (*path frequency(i)* x (*path basetime(i)*))

This path net variation time represents the potential execution time savings if all executions of that path took the basetime to execute. This is an ideal potential, but the ramifications of optimizing a portion of the path variation can be substantial. We use this path net variation time metric to rank paths.

### 5.2 Path Variability Analysis

Figure 2 is a histogram of time deltas for `bzip2` for the top five paths with highest net variability (one line for each path). The x-axis measures time variations relative to the basetime, where the origin denotes the *basetime*. The further along the x-axis the larger the time delta is for a path sample from the basetime. The y-axis measures the frequency of paths that occur for each time delta. We can see that for each of the paths illustrated, a substantial portion of execution time is spent with a high time delta (distance from origin). The path net variation time can be measured in this graph by computing the area under a curve. In addition, the spikes likely correspond to accesses missing at different levels of the memory hierarchy. Since paths include multiple load instructions, some of which may hit while others miss, these spikes are not always sharply defined.

Figure 3 shows the net variation time (total execution time above path basetime) for the benchmarks examined executing the reference inputs. For each of the benchmarks the net variation time for the top ten paths are plotted, where the path with the largest amount of variation is on the bottom. The y-axis measures percent of path net variation time relative to the total program execution time that is spent in the top 10 path variations. The percent execution time shown here represents the optimization opportunity and speedup that exists if we could make each invocation of the path execute in the shortest time that we observed during sampling (basetime). Consider the top variation path in vortex. It's net variation consumes more than 30% of execution time. If we can optimize this path to remove half the variation, we would achieve a 15% speedup. On average, the net variation in the top ten paths consume more than 43% of execution time and only the dominant path's net variation accounts for more than 17% of execution time. This is encouraging for developers, since the majority of variation optimization potential can be exploited by focusing on a few of the top paths.

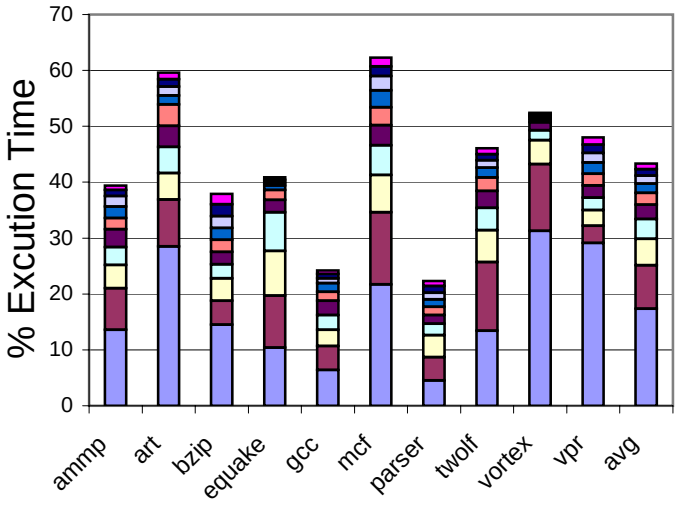


*Figure 3: Path variation for the top ten paths. Results show the path net variation time for the benchmarks examined executing the ref inputs. For each of the benchmarks the path net variation time for the top ten paths are plotted, where the path with the largest amount of net variation is on the bottom. The y-axis measures percent of execution time relative to the total program execution time that is spent in path variations. The percent execution time shown here represents the optimization opportunity and speedup that exists if we could make each invocation of the path execute in the smallest amount of time that we found during sampling (basetime).*

### 5.3 Path Variation Stability

The previous section showed the potential savings from the net variations of the top ten paths. If we apply an optimization to a path with high net variation it may achieve significant speedup under conditions in which the path variations were collected, but we also want the optimization to be effective across inputs and system conditions. This entails measuring the stability of paths with high net variation across these different conditions. In this section we show that the paths with highest net variation are stable across different input and system loads.

#### 5.3.1 System Load Stability

System load measures how strained the resources are during program execution. Heavy system load would mean CPU contention and the strain on memory structures will cause more misses and stalls during execution. These misses may cause significant variations to appear that are non-existent under light system load. On the other hand, variations experienced with a light-system load may drown out from higher variations caused by heavy system load.

We conducted an experiment to find if path variations are stable across system load. We computed path variations with a light system load and also with a heavy system load. For the light system load no applications were executing during experimentation except for a few essential OS background services. For the heavy system load we strained the system with a 3-D graphics game, Unreal Tournament [4], during program path

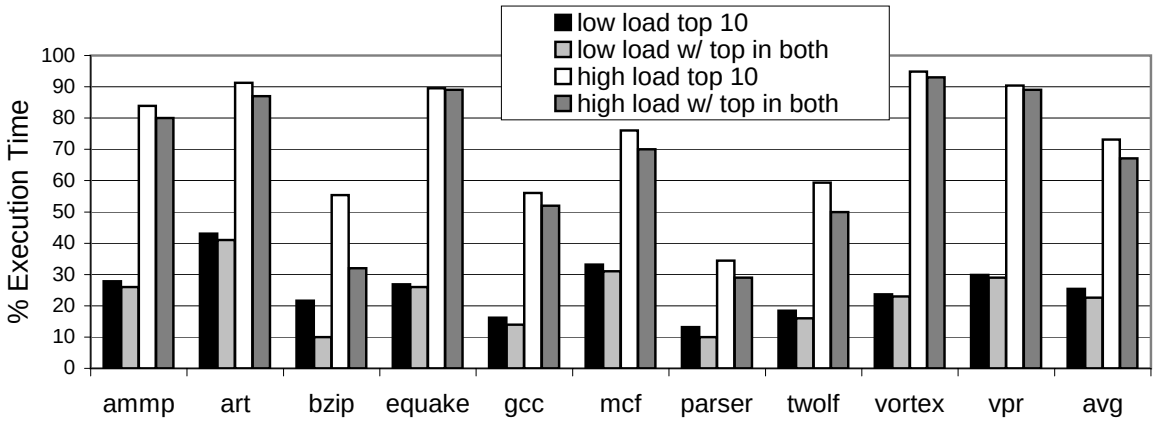


*Figure 4: Stability of variational paths found under different system loads. The y-axis measures percent of execution time. The first and third bars show the net path variations of the top ten paths under light and heavy system load respectively. The second bar shows the net path variations under light system load, for the 10 paths found under heavy system load. The fourth bar is like the second bar, but the net variation in execution time shown is for the top ten paths found under light load.*

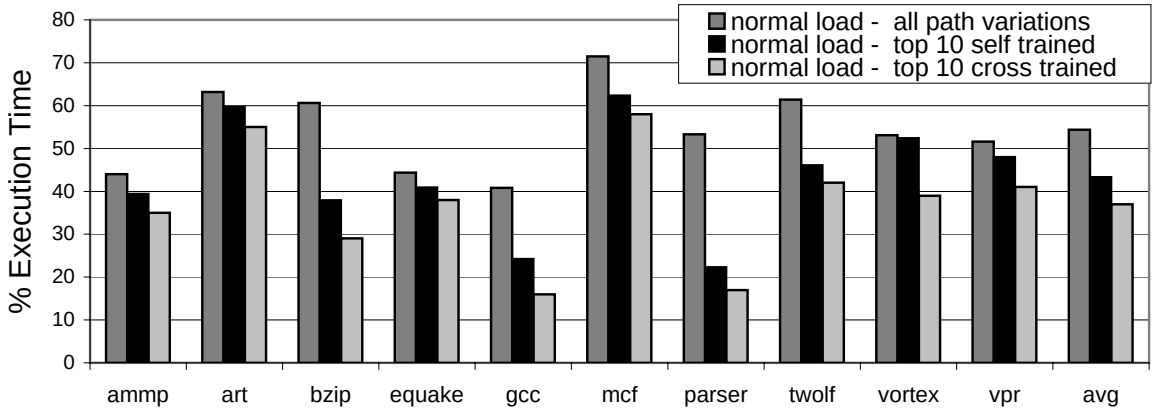


*Figure 5: Stability of variational paths across inputs. The first bar shows the net variation in execution time for all paths in each program. The second bar shows the self-trained top ten path net variation for the ref input. The third bar shows the cross-trained net path variation for the ref input using the training input to pick the top 10 paths.*

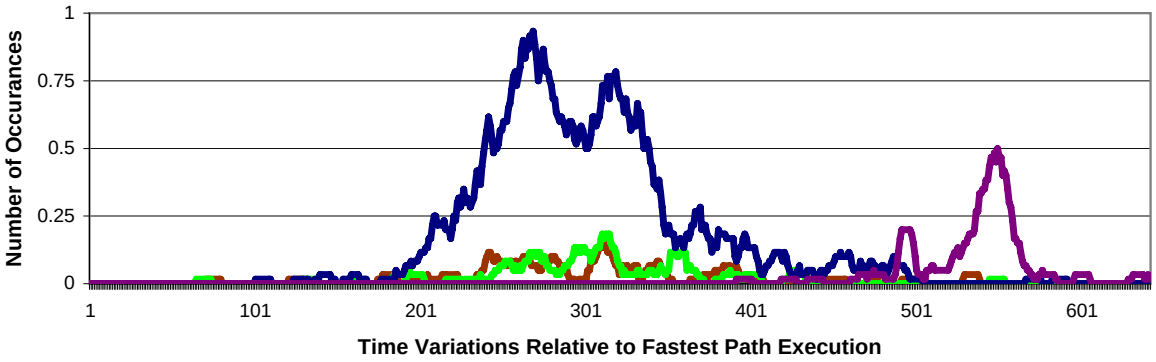


*Figure 6: Foxpro variational path histogram. This shows the top 5 ranked paths with variation from the basetime. The x-axis measures time variations relative to the basetime, where the origin denotes the basetime. The further along the x-axis the larger the time delta is for a path sample from the basetime. The y-axis measures the normalized number of paths that had that much variation (x-axis) from the minimum timed basetime path.*

profiling. The game running during profiling cause the program to execute between 2 and 5 times slower.

Figure 4 shows results of the experiment for the 10 benchmarks. The y-axis measures percent of execution time. The first and third bars show the path net variations of the top ten paths under light and heavy system load, respectively. The second bar shows the path net variations under light system load for the top ten variational paths found from profiles collected under heavy system load. This measures how stable the paths found under heavy system load are when the program is run under light system load. The fourth bar is like the second bar, but the top ten variational paths were chosen using the light load before calculating the net variation using the heavy load samples. These results show that the same paths exhibit high net variation under light and heavy system load conditions. Another interesting observation is the significant difference in net variation time consumed under the different loads. On average we see that the net variation in the top ten paths consume about 25% of execution under light system load, but over 70% under heavy system load. The second and fourth bar results are encouraging, since it implies that the top varying paths are consistent across the different loads.

#### 5.3.2 Cross Input Stability

Just as system load can cause deviations in a program execution, different inputs can cause execution behavior to change dramatically as shown in [12]. A heavily executed procedure under one input may become dormant with another input. Therefore we need to examine the stability of the top varying paths across inputs.

To measure how stable the top paths are across inputs we collect path variations for the benchmarks with the train and ref inputs under normal system load. Normal system load is similar to light system load, except that special care wasn't taken to minimize the system load and there may be lightweight applications running in the background (e.g. text editor). We compute the net variation for the top ten paths for the ref input, called self-trained. Then we compute the net variation for the ref input using the top 10 paths found using the training input, called cross-trained. These results are seen in Figure 5. The first bar shows the execution time from only the net variation for all paths in each program for the ref input. The second bar shows the self-trained top ten path variations also for the ref input. The third bar shows the cross-trained path net variations for the ref input but using the training input to pick the top 10 paths. The similar heights between the first bar and the second and third bars signifies that using the top ten path variations account for the majority of all path variations for most of the programs. The high correlation between the net variations seen between the second and third bars implies that the top varying paths are stable across the train and ref inputs.

The results show that there are programs where the top ten paths do not capture the majority of all path variation. This is the case with `bzip2` , `gcc` and `parser`. In those programs the top paths exhibit lower variation relative to other programs' top paths. We have found this due to two factors. First, some programs do not have a small group of paths that dominate the execution, so looking at more than the top 10 variational paths is needed to capture most of the potential path variation in the program. Second, some programs have many paths, where no individual path has a high net variation. This occurs more for highly optimized programs, and also for programs where the compiler and hardware do a great job at hiding memory latency, which is the case for `parser`.

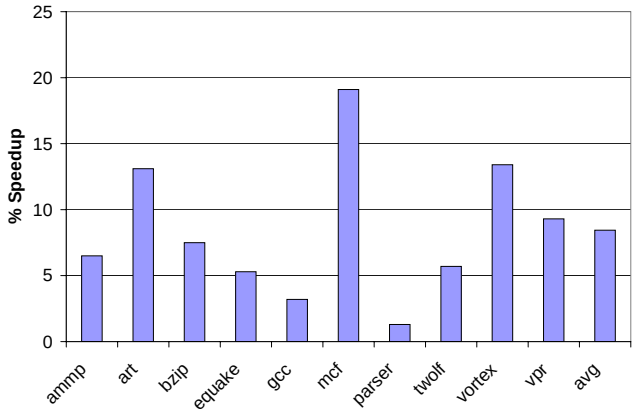


*Figure 7: Speedup results from applying simple prefetching optimizations to the paths with high net variation.*

### 5.4 Optimizing Variational Paths

We now examine the top paths that are found for the benchmarks running on the training input and report the performance speedup from applying very simple optimizations to those paths with the reference input. Cross-trained path rankings are stable as was shown in Section 5.3.2. We examine the top one to four paths in each program, and we found that they typically point to loops. For these, the main explanation for the variation was due to memory stalls. Some of the time the same exact path through the loop would incur no cache misses, whereas other times that path would have cache misses. We applied the following algorithm by hand to perform a prefetching optimization to the top few net variational paths in each of the SPEC programs:

1. Find all data references in the variational path to be optimized. There were usually only a few loads.
2. For each data reference, insert a prefetch to the data to be accessed in the next iteration of the loop, or two iterations in advance. The address to be prefetched was either an array index, which was an easy calculation, or a pointer chain, where the next iteration pointer chain was followed.
3. Insert an if clause before the prefetch and address computation to avoid prefetching beyond the bounds of the data access or following a null pointer.

The above algorithm was applied manually in all of the experiments and we were able to optimize each of these paths

in a matter of minutes applying very simple prefetching optimizations to achieve speedups ranging from 2% to 19%. The optimizations use the Streaming SIMD Extension [3] `_mm_prefetch()` to prefetch a data address into the L2 cache. Figure 7 shows the percent savings in execution time from these optimizations. The variational path profiles from the training input were used to guide the optimization, and performance results are shown for the ref inputs. The results show the power of knowing paths in which there are variations to easily exploit, and how easy it was to use our profiler to find the top paths and optimize them to exploit this variation. Even with this simple optimization strategy we were able to reduce the net variation in the paths that were optimized by 41% on average.

To gain more insight into what we did, Figures 8, 9, and 10 show the optimizations we performed on `vpr`, `art` and `mcf` for the paths with the highest net variation. Figure 8 is source code from `vpr` that forms the path with highest variation. This path variation is responsible for almost 30% of execution time. The optimization code is marked with `**`. In this example we have a heap being traversed in geometric strides, and an element is compared and pointer swapped with ones that have been visited in the previous iteration. A simple prefetching optimization has been inserted for this path, based on the assumption that some accesses to the heap miss and cause the variation in execution times. We prefetch the heap element that will be accessed two iterations in the future. A condition is set for the prefetch to guarantee that we never attempt to access an element that is beyond the bound of the heap. This 2 line optimization we applied to `vpr` achieves nearly 10% in speedup on the ref input.

Another optimization example is shown for `art` in Figure 9. In this example a nested for-loop in another for-loop accesses two elements, `f1_layer[ti].P` and `bus[ti][tj]`, computes their product and stores it in `Y[tj].y`. The main culprit of variation in this path is `bus[ti][tj]`, since after the first iteration of the outer for-loop, all the elements accessed in `f1_layer[ti].P` will be warm in the cache. The optimization code inserted in this code segment is marked with `**`. It bootstraps the outer for-loop to prefetch elements of `bus[ti][tj]` that will be accessed in the next iteration of the outer for-loop. The if-statement in line 4 is inserted to verify that we do not prefetch beyond the bounds determined by the outer for-loop. This condition will hold true for every iteration of the outer loop, except the last one. We do not achieve speedup for the first iteration, since prefetching is done for elements used in the second outer for-loop iteration and up. Line 6 is the prefetching instruction and lines 5 and 7 do the original operation as seen in lines 9 and 10. Similar optimizations are applied to 3 other top paths in `art`. They result in 13% speedup on the ref input.

Yet another optimization example is shown for `mcf` in Figure 10. Here we have a loop that traverses a linked list, and performs operations on the nodes depending on what conditions hold (e.g. line 10, 16, 17, and 22). A few data elements in this loop are being reused across the entire life of the loop, and the elements which cause variations are those dependent on the linked list nodes being traversed. The optimization strategy here is similar to the previous two, where we prefetch data at the start of the loop (lines 4-9) to be used in the next iteration. A condition is imposed on the prefetch operations to check if the next node exists. The data prefetched is used in the if-condition in line 10. A more implicit prefetch happens in line 4, where we check if the next node in the list exists. This access preempts the access that eventually happens in line 26. The potential advantage in doing this access earlier is that it can interleave a miss-penalty with other miss-penalties that can occur during the main body of the loop. Including this optimization, we applied 2 more optimizations to the top 3 paths in `mcf` to achieve a speedup of 19%.

Simple prefetch optimizations like the ones described have been applied to the top few paths in each of the programs. On average we achieve a speedup of 8.5%. This is encouraging since the analysis and optimization can be done on a program in a matter of minutes. For the programs that have high net variation in the top 3 paths (`art, mcf, vortex, vpr`), we see a correspondence with higher speedup achieved (Figure 7). For the programs that exhibit lower top path variability (e.g. `gcc` and `parser`), we find that less speedup is achieved relative to the other programs. This correlation reinforces our claim that Variational Path Profiling determines paths in the program that deserve optimization, as well as indicate potential speedup gains from applying the optimization.

### 5.5 Comparison to Hot Path Techniques

In this section we compare VPP with other hot path profiling techniques. Generally, the critical hot paths in a program are often captured by most hot path profiling techniques. The advantage in VPP is that it identifies the paths that have a high potential for optimization due to execution time variation. In addition, it finds paths that are not always found in the top rankings using other methods, especially once some of the hot paths have already been highly optimized.

A common technique to determine hot paths is based on their frequency of execution [10]. The hot paths are those that execute most often. This approach is good in finding the hot paths in a program, especially the first time the program is to be optimized. If those paths are heavily optimized but still execute the same number of times, this method will still rank these paths as the hottest. A programmer without prior knowledge of these optimizations may concentrate on these paths even though they may no longer have substantial optimization gain. Another method to rank hot paths is based on their net execution time. This method simply aggregates the total time spent in a path throughout the entire execution. It does not involve any variation analysis that is used in VPP. In VPP the ranking of hot paths is not based on frequencies or just total time spent in a path but rather on the net execution time variations. Once a hot path is optimized, it is likely to have a lower net perfor-

```
1 while (ito < heap_tail) {
2    if (heap[ito+1]->cost < heap[ito]->cost)
3       ito++;
4    if (heap[ito]->cost > heap[ifrom]->cost)
5       break;
6**  if (ito*8 < heap_tail)
7**     _mm_prefetch((char*)&heap[ito*8]->cost, 1);
8    temp_ptr = heap[ito];
9    heap[ito] = heap[ifrom];
10   heap[ifrom] = temp_ptr;
11   ifrom = ito;
12   ito = 2*ifrom;
13 }
```

*Figure 8: Optimization example:* `vpr`

```
1 for (tj=0;tj<numf2s;tj++) {
2   Y[tj].y = 0;
3   if ( !Y[tj].reset ) {
4**   if(tj < (numf2s -1)) {
5**     for (ti=0;ti<numf1s;ti++) {
6**       _mm_prefetch((char*)&(bus[ti][tj+1]), 1);
7**       Y[tj].y += f1_layer[ti].P * bus[ti][tj];
8**     }
9     } else
10      for (ti=0;ti<numf1s;ti++)
11        Y[tj].y += f1_layer[ti].P * bus[ti][tj];
12 } }
```

*Figure 9: Optimization example:* `art`

```
1  while( arcin )
2  {
3    tail = arcin->tail;
4**  if ((arc_t*)tail->mark) {
5**    _mm_prefetch((char*)
6**       &(((arc_t*)tail->mark)->tail->time),1);
7**    _mm_prefetch((char*)
8**       &(((arc_t*)tail->mark)->org_cost),1);
9**  }
10   if( tail->time + arcin->org_cost > latest ) {
11     arcin = (arc_t *)tail->mark;
12     continue;
13   }
14   red_cost = compute_red_cost( arc_cost, tail,
15                                head_potential );
16   if( red_cost < 0 ) {
17     if( new_arcs < MAX_NEW_ARCS ) {
18       insert_new_arc( arcnew, new_arcs, tail,
19                       head, arc_cost, red_cost );
20       new_arcs++;
21     }
22     else if((cost_t)arcnew[0].flow > red_cost)
23       replace_weaker_arc( arcnew, tail, head,
24                           arc_cost, red_cost );
25     }
26     arcin = (arc_t *)tail->mark;
27  } }
```

*Figure 10: Optimization example:* `mcf`

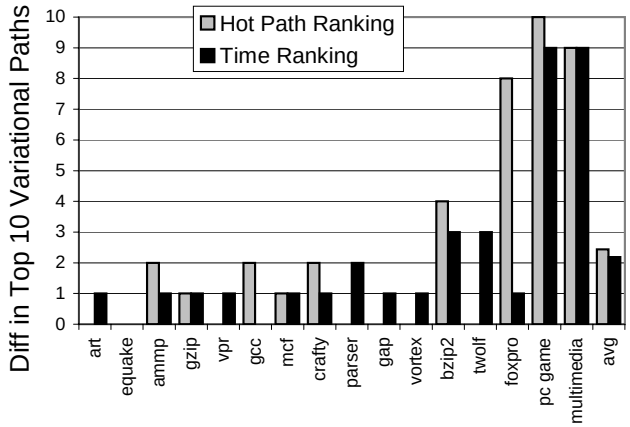


*Figure 11: Difference in paths found with two common hot path methods to those found with VPP. The hot path ranking bar for each program shows how many of the top ten paths are different between the frequency based approach and VPP. The time ranking bar shows how many of the top ten paths are different between the total time based approach and VPP.*

mance variation and thus may no longer be ranked hot. The dynamic nature of the VPP hotness ranking will usually target paths that have high potential for performance improvement.

To demonstrate the advantage of VPP over the frequency and total time based hot path approaches we compute the hot paths using both techniques on the SPEC benchmarks and some highly optimized industry programs – `foxpro` [1], a pc game, and a multimedia application. These last three programs have been highly optimized at Microsoft. Figure 11 shows the number of top 10 net variation paths that were *not* found in the top 10 hot paths based on path occurrence or total execution time. The hot path ranking bar for each program shows how many of the top 10 VPPs are not found in the top 10 most frequent paths. The time ranking bar shows how many of the top 10 VPPs are not found in the top 10 paths based on time. The results show that for the SPEC programs the hot path ranking and the VPP ranking are similar. For the SPEC2000 benchmarks the biggest difference from the top ten hot paths between the two techniques was `bzip2`, which deviated by 4 paths (4 of the top 10 VPPs were not found in the top 10 hot path ranking).

Unlike the SPEC2000 programs, the top ten hot paths for the highly optimized Microsoft applications found using VPP have only a few paths in common with those found with the frequency and total time based approaches. The only exception is `foxpro` with the total time based approach. This is a significant difference as compared to the number of common hot paths found for the SPEC2000 (on average more than 9 paths in common). The most frequent paths in these programs have been optimized, and as a consequence they have lower net performance variations. VPP finds hot paths that have the greatest net performance variations, regardless of frequency or prior optimizations. For these commercial applications, we could not get access to the source in a timely manner to apply

our optimization. Instead, we looked at the top paths in more detail as shown in Figure 6, which shows a histogram of time deltas for `foxpro` for the top five paths with highest variability (similar to Figure 2). The results show similar ranges of variability as in the SPEC programs top net variational paths.

Another common technique for finding hot paths utilizes the VTune [5] performance profiler. We have extensive experience with using Vtune hot paths to identify hot regions and use this as a basis for optimization, including on the SPEC benchmarks. However, with several days effort we have not been able to obtain the amount of benefit that was obtained using the VPP information in significantly less time. The reason is that the majority of the hot paths have low variation in commercial applications we are examining. The advantage of VPP is that it pinpoints a small subset of hot paths (10 appears sufficient) that have this additional variational property and consequently focuses optimization effort on these high opportunity areas as was demonstrated.

# 6 Summary

In this paper we present a new type of profiling analysis called *Variational Path Profiling* (VPP). VPP pinpoints exactly where in the program there are significant optimization opportunities for speedup. VPP records the execution time it takes to execute frequent acyclic control flow paths using hardware performance counters. From this timing analysis we find that a program path can have significant variation in its execution time across different dynamic traversals in the same program run. This variation (the difference between the fastest execution of that path and slower executions) represents the potential speedup one could achieve if we could optimize away these variations.

We present a profiling and analysis approach to find these variational paths, so that they can be communicated back to a programmer to guide optimization. The goal for concentrating on a path with high net variation in its execution time is that it can potentially be optimized so that all executions of that path have the minimal execution time seen during profiling. In examining the top 10 variational paths, we found that they pointed to loops with memory stalls during some paths of execution and not others. In a matter of minutes, we were able to manually apply a very simple optimization algorithm to these loops to achieve speedups ranging from 2% to 19%. This shows the power of knowing paths in which there are variations to exploit, and how easy it was to use our profiler to find the top paths and optimize them to exploit this variation.

We compare VPP with traditional hot path profiling techniques (e.g. path hotness based on path frequency and total time spent in a path). The advantage of VPP is seen when a program has been heavily optimized and those paths no longer show as much variation. In comparison, the frequency based approach of hot path profiling will keep pointing to the same hot paths even though they may no longer have as much optimization gains, whereas VPP will expose new paths that have the greatest potential for optimization. The key benefit of VPP is that it identifies the paths that have high potential for optimization due to the variation found during profiling.

# 7 Acknowledgments

We would like to thank the anonymous reviewers for their comments. This research was supported by Microsoft, and NSF grants CNS-0311683 and CCF-0311710.

# References

[1] Microsoft visual foxpro 7.0. http://msdn.microsoft.com/vfoxpro/.

[2] Microsoft visual studio .net. http://msdn.microsoft.com/vstudio/.

[3] Streaming simd extension. http://msdn.microsoft.com/library/.

[4] Unreal tournament. http://www.unrealtournament.com/.

[5] Vtune performance analyzer. http://www.intel.com/software/products/vtune/.

[6] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[7] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, 1998.

[8] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 1–14. ACM Press, 1997.

[9] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[10] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[11] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[12] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, 2002.

[13] J. Gosling, B. Joy, and G. Steele. Hiprof advanced code performance analysis through hierarchical profiling.

[14] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. 2001.

[15] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999.

[16] David Melski and Thomas W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.

[17] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *International Symposium on Microarchitecture*, December 1997.

[18] A. Srivastava, A. Edwards, and H. Voi. Vulcan: Binary transformation in a distributed environment. Microsoft Research, 2001. Techical Report MSR-TR-2001-50.

[19] EJ technologies' JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html.