TFP: Time-sensitive, Flow-specific Profiling at Runtime

Sagnik Nandy Xiaofeng Gao Jeanne Ferrante

Department of Computer Science and Engineering
University of California at San Diego
{snandy, xgao, ferrante}@cs.ucsd.edu

Abstract. *Program profiling can help performance prediction and compiler optimization. This paper describes the initial work behind TFP, a new profiling strategy that can gather and verify a range of flow-specific information at runtime. While TFP can collect more refined information than block, edge or path profiling, it is only 5.75% slower than a very fast runtime path-profiling technique. Statistics collected using TFP over the SPEC2000 benchmarks reveal possibilities for further flow-specific runtime optimizations. We also show how TFP can improve the overall performance of a real application.*

Key Words : Profiling, dynamic compilation, run-time optimization.

1 Introduction

Profiling a program can be used to predict the program's performance [1], identify heavily executed code regions [9, 23, 11], perform additional code optimizations [12, 10], and locate data access patterns [13]. Traditionally, profiling has been used to gather information on one execution of the program, which is then used to improve its performance on subsequent runs. In the context of dynamic compilation and run time optimizations, profiling information gathered in the *same* run itself can be used to improve the program's performance. This creates a greater need for efficient profiling, since the runtime overheads might exceed any possible benefit achieved from its use. In addition, the information gathered by profiling must be relevant for runtime optimizations and should remain true while the optimized code is executed. In this paper we propose a new profiling framework, *TFP (Time-Sensitive, Flow-Specific Profiling)*, that extracts temporal control flow patterns from the code at runtime which are *persistent* in nature i.e., these patterns hold true for a given, selectable period of time. This information can then be used to guide possible optimizations from a dynamic perspective. This paper makes the following contributions:

1. Proposes a new profiling strategy that is both flow-specific and time-sensitive.
2. Provides a comparison of the profiling overheads of TFP with the dynamic path profiling of [6]. On the SPEC 2000 benchmarks, we show that TFP is on average only 5.75% slower than the technique of [6] (which is well suited fora dynamic environment), while collecting a wider range of information.
3. Provides a case study of RNAfold [21] that demonstrates that TFP can be used to improve overall performance of an application.

The rest of this paper is organized as follows. Section 2 describes the background and motivation for our work. Section 3 discusses our framework in detail and how it can be used to collect a range of runtime information. In Section 4 we discuss some implementation details and how they can be changed to meet specific requirements. Section 5 presents experimental results and a case study using our framework. We conclude in Section 6 with possible future research directions.

2 Background and Motivation

Profiling code to gather information about the flow of control has received considerable attention over the years. Most existing profiling techniques are meant for off-line program analysis. However, with the advent of dynamic compilation and runtime optimizations, the use of profile data generated for runtime use has increased [19, 16, 13, 3, 2, 5, 4]. Recently [7] has shown that runtime flow-specific information can be used to improve code performance significantly. In [15, 13], a technique called Bursty Tracing is introduced that facilitates the use of runtime profiling further. This technique allows the programmer to skip between profiled and un-profiled versions of a code as well as control the duration spent in either version. Such a technique will allow the user to control the overheads involved in running profiled code to a far greater extent. Some of these techniques require hardware support while others rely completely on software. Our work falls in the latter category.

Some of the more popular flow profiling techniques include block profiling, edge profiling [17], [8] and path profiling [9], [18]. These techniques differ in the granularity of the information they collect. Figure 1 illustrates the differences, where block profiling collects basic block frequencies, edge profiling collects edge execution frequencies and path profiling collects path frequencies.

In [6], Bala developed a profiling technique well suited for finding path profiles in a dynamic environment. This technique instruments each edge of a code segment with a 0 or 1, and represents each path as a starting block followed by a bit sequence of 0's and 1's. The easy implementation and simplicity of this technique makes it an attractive choice for runtime path profiling. With adequate support from the compiler and hardware this technique can provide *near-zero* overhead profiling and forms the basis of comparison for the work we develop here.

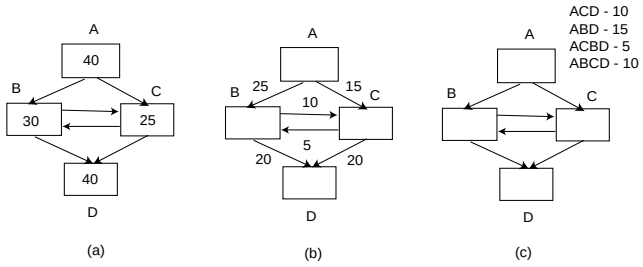

Fig. 1. (a) Block Profile (b) Edge Profile (c) Path Profile of the same run for a given CFG

It must be mentioned that *path profiling* $\supseteq$ *edge profiling* $\supseteq$ *block profiling*, i.e. all the information gathered by block profiling can be gathered by edge profiling, while all the information gathered by edge profiling can be collected by path profiling. However, retrieving this information comes at a greater cost in terms of overheads since one needs to maintain data structures to save this information and often require multiple passes of these data structures to get the necessary granularity of information.

However, several possible runtime optimizations such as dependence analysis and loop unrolling can benefit from block and edge profiling alone, and often do not require more refined information. Even though this information might be retrieved from path profiles, it could require considerable additional processing (to store the blocks and edges a path corresponds to, and then scan through the paths again to retrieve the necessary information). A fundamental question to be addressed by our research is whether there is additional advantage in using more powerful profiling information at runtime.

We also question whether a *detailed* analysis of the programs execution pattern is useful for *online* analysis. For example, one might want to detect whether a single path is being executed persistently (thereby making it a possible target of optimizations [7]) or observe if certain pathological cases *never occur* [20]. This paper seeks to combine several benefits of block, edge and path profiling in a single unified profiling framework - providing easy and efficient access to a range of information at runtime.

3 The TFP Approach to Profiling

TFP can not only count frequencies of flow-patterns but is capable of capturing a variety of temporal trends in the code too. These trends can then be used to guide runtime optimizations. To capture this idea we make use of *persistence* i.e. flow patterns and information that continuously holds true for a period of time. We define persistent flow patterns as follows:

A K***-Persistent Flow Property (K-PFP)*** *of a program segment is a property which holds true for the control flow of that segment for* K *consecutive executions of the segment.*

The motivation for such a technique lies in the assumption that if a *PFP* holds for a period K, it may continue for some additional time. Additional optimizations could then be made assuming the trend would remain persistent to the benefit of the dynamic compiler. For example consider the two code snippets in Figure 2.

Traditional frequency based profilers will find both the paths along *f()* and *g()* to be equally hot [9]. The code in 2(a) is not suitable for runtime optimization, since the path in the loop body only lasts for one iteration. On the other hand in 2(b) an optimization that is valid for only one path in the loop body would remain valid longer, making it worthwhile to perform the optimization. This shows that frequency is not the only parameter for locating hot paths, but *persistence* should also be considered (similar distinctions about access patterns can also be found in

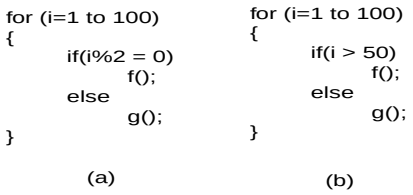

Fig. 2. Sample Code Snippets where the code in (b) has a 50-PFP but the one in (a) does not

[14] for the purpose of code layout). When using a *PFP* guided approach, the code snippet in 2(b) will qualify as a *50-PFP* but the one in 2(a) will not, allowing us to differentiate between them.

Even if a sequence of code does not have a persistent path, we might still be interested in finding other *PFP*s. Each *PFP* might lead to different kind and granularity of optimization. Listed below are some other possible *PFP*s and examples of runtime optimizations that can be based on them.

1. *Persistently taken paths*: This information can help the compiler identify a possibly smaller segment of code on which runtime path-specific optimizations might be conducted.
2. *Basic blocks which are persistently not taken*: This information would allow us to form a smaller CFG of the code region by eliminating these blocks from the original CFG. As a result, we might eliminate dependences, loops, variables etc. that might promote additional runtime optimizations.
3. *Persistently taken path segments*: Even if persistent paths do not exist we might have *sub-paths* that are persistent. This can help in eliminating certain dependences and code regions.
4. *Whether a given set of edges are ever taken*: This information can be used to remove possible dependences at runtime.
5. *The minimum/maximum level of persistence during a time period*: Even if no path meets the persistence parameter of K, such information can be useful to determine the extent to which loops can be unrolled without causing inter-iteration dependences.

Though some of the existing profiling techniques can be modified to incorporate persistence, they are aimed at gathering one kind of information efficiently. While path profiling can do a good job of *PFP*s 1 and 5, block profiling can perform 2 and edge profiling can collect 3, 4 and 5 efficiently. Path profiling techniques like [9] and [6] can also be used to detect 2, 3 and 4 but that would require maintaining additional data structures, storing additional data, and making multiple passes of the profiled information. TFP provides a unified framework that collects all the above mentioned *PFP*s with a small amount of instrumentation. The following section describes TFP in detail.

3.1 Detailed Description of TFP

Like most control-flow profiling techniques, our targets for profiling are acyclic code regions (we later describe in Section 4 how we can include nested loops). Thus any region of code that begins with a block that is the target of a back-edge, and ends with the block from which the back edge originated, with no intermediate back edges, is a valid candidate for our profiling strategy.

TFP is a hybrid between Bala's method [6] of path profiling and block profiling. Instead of assigning each edge a 0 or 1 (as in Bala's method), we represent each (profiled) block by a single bit position in a bit string. Conceptually, each block represents an integer which is a unique power of two (i.e. $block_i$ is represented by the value 2^i). The initial block always sets the value of this register to 0. At the end of each profiled block an instruction is inserted to perform a mathematical ADD (or bitwise OR) of this number to a register r. The value of this register identifies the path taken, and *Bookkeeping* code is inserted in the exit block of the instrumented region. For acyclic code regions with multiple exit blocks we add the Bookkeeping code in each of the exit blocks. The Bookkeeping code can vary with the kind of PFP(s) we wish to track, as illustrated in the sections to come. Figure 3 gives an example of our profiling method, showing a sample code region, the inserted instrumentation code and the register values associated with the various paths.

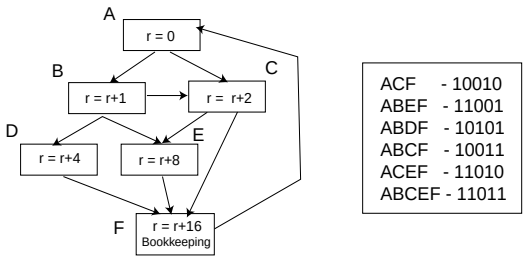

Fig. 3. An example of our profiling technique (the values of r corresponding to the paths is also given)

The basic idea behind our approach is that each path will produce a unique value in the register, as well as give all the information about the blocks that form that path. Thus we get the benefits of both block and path profiling simultaneously (and some benefits of edge profiling as shown in Section 3.5). This idea is embodied in the following:

Theorem 1. With the register assignments inserted as described above, each different value of the register r corresponds to a unique path.
Proof. Since each basic block is represented by a bit in the register, the only way in which the register can get a value is by traversing all the blocks that correspond to a 1 in the bitwise representation of the value. Thus given a value in the register, we can determine the basic blocks in the corresponding path. To complete the proof, we have to show that no two paths can have the exact same set of blocks in them.

The proof is by contradiction. Assume that $X = x_1, x_2, ..., x_N$ are the basic blocks that were traversed and $Y = y_1, y_2, ..., y_N$ and $Z = z_1, z_2, ..., z_N$ are two different paths using X, i.e., Y and Z are two different permutations of X. All the elements of X must be unique, else X would have a loop, and thus an associated back edge, contradicting our assumption. Now let k be the position where Y and Z first differ i.e. $y_i = z_i$ for $i = 1, ..(k-1)$ and $y_k \neq z_k$. Obviously $k < N$ or Y and Z would be identical. Now since $y_k \neq z_k$ there is some value z_j, $j \epsilon (k+1, ..., N)$ for which $z_j = y_k$ (since both Y and Z have the same set of elements). Thus there exists an edge from z_{j-1} to y_k in the path Z. Now z_{j-1} has to appear in Y as well and it can only appear after or at the k^{th} position. Thus in Y there is a path from y_k to z_{k-1} and we also know that there is an edge from z_{k-1} to y_k. Thus this edge is actually a back edge, contradicting our assumption for profiling candidates. Hence Y and Z cannot be different. □

TFP provides two major benefits when compared to traditional path profiling techniques. Firstly it collects a much wider range of information as a by-product of path profiling. Other path profiling techniques would require additional data structures (TFP uses just a few variables), and multiple passes over these data structures to find this information. The second advantage that TFP provides is that most other path profiling techniques instrument the edges which can result in additional branches in the program, affecting the overall performance. TFP instruments at the block level and though this requires instrumenting every block of the region it does not add further checks in the code.

We now describe how TFP can be used to detect some of the *PFP*s mentioned earlier in this section. We first consider the parameter *Persistence Factor* (K) which represents a lower bound (threshold) on the persistence of interest. To gather various *K-PFP*s the TFP instrumented code is executed for K iterations (this can be achieved using [15]). The values of the Bookkeeping variables at the end of these iterations reveal the various *K-PFP*s observed.

3.2 Persistent Paths

The following Bookkeeping is needed to track persistent paths using TFP.

```
bblock1 = bblock1 AND r;
bblock2 = bblock2 OR r;
```

Bookkeeping for persistent paths

After running the TFP instrumented code for K iterations if $bblock_1$ and $bblock_2$ are equal then we know that we have a K-persistent path (which the compiler might then wish to optimize). This follows from the fact that each path produces a unique value of r (from Theorem 1) and the only way $bblock_1$ and $bblock_2$ will be equal is if r remained unchanged for the K iterations ($bblock_1$ and $bblock_2$ are initially set to -1 and 0 respectively). If we detect a persistent path then we can expect the code to remain in the same path for a while and make further optimizations based on this assumption. As an example assume that for the CFG shown in Figure 3, the path

ABEF is persistently taken for 10 iterations. It is easy to see that at the end of the 10 iterations both $bblock_1$ and $bblock_2$ will be equal to *11001*, denoting the existence of a 10-persistent path.

3.3 Path Segments that are always taken

Even if we do not find persistent paths using the method given in Section 3.2, we might still want to find the set of *path segments* or sub-paths that are always taken. This kind of information can be used to form a smaller CFG from the code by eliminating untaken blocks and paths. To get this information using TFP, we use the *same Bookkeeping code as in Section 3.2* but assign the numbers ADD/ORed to r in each basic block in a *topologically sorted* manner (this need not be done at runtime if one uses a framework like [15] or if all regions of possible interest are instrumented at compile time itself). Thus if each instrumented basic block b_i ADD/ORs the value v_i to r, then $v_i < v_j$ if b_i comes before b_j in the topologically sorted order of the blocks.

To gather the information about path segments that are always taken (during the K iterations of the TFP instrumented code), we need to scan through $bblock_1$ (from left to right or right to left) and join blocks that correspond to adjacent 1's in $bblock_1$, unless there is some other block in between the two blocks in $bblock_2$ that is a 1. Thus if $x_{i1}, x_{i2}, ..., x_{iK}$ where $(i_1 < i_2 ... < i_K)$ is a persistent path segment then the bit locations $i_1, i_2, ..., i_k$ will be 1 in $bblock_1$ and the only bits which are 1 in $bblock_2$ between i_1 and i_k will be those at $(i_1, i_2, ..., i_k)$. This follows from the fact that since the blocks are topologically sorted, then the edge (x_{i1}, x_{i2}) is always taken if no block between x_{i1} and x_{i2} is ever taken.

As an example let us refer to Figure 3 again. Assume that during a profiled run only paths *ACEF* and *ABCEF* are taken. At the end of the profiled run $bblock_1$ will contain *11010* and $bblock_2$ will contain *11011*. Following the technique given above we see that bit positions 1,3 and 4 are set to 1 in $bblock_1$ and none of the other intermediate positions (position 2) are set to 1 in $bblock_2$. Thus we can conclude that the path segment connecting bit positions 1,3 and 4 (i.e *CEF*) is always taken - which is the case.

3.4 Basic Blocks that are not taken

To determine the basic blocks that are not taken, we could use block profiling and check each counter of the basic blocks to see if they are 0. However, for TFP, we do not need counters to gather this information, which saves us memory. Using our method this information can be easily obtained using the following code for Bookkeeping.

```
bblock = bblock OR r;
```

Bookkeeping for blocks persistently not taken

On executing the instrumented code for K iterations, the variable *bblock* has a 1 for all the blocks that get taken at any time during the execution of the instrumented

code, and all bit positions that have 0's correspond to basic blocks that are not executed even once, in those K iterations. Note that TFP doesn't gather the exact frequency of the blocks that are taken.

It can be observed that the Bookkeeping for this *PFP* is a subset of the ones described in Sections 3.2 and 3.3, and need not be additionally inserted in case we are also instrumenting for persistently paths or sub-paths.

3.5 Tracking if specific edges are taken

Several useful optimizations are impossible to verify statically because of possible dependences along different control flow paths. Our framework provides an easy way of tracking whether a specific set of edges is ever executed (or persistently not executed). The compiler can use this information to eliminate false dependences at runtime, enabling several optimizations (such as constant propagation, loop unrolling, code compaction etc). To achieve this using our framework, we assign blocks their additive values based on a topological sort as described in Section 3.3. Thereafter if we want to test if an edge between blocks i and j is ever taken, we add a test in the Bookkeeping code to check if the bit positions i and j are ever simultaneously 1 with no other 1's between them. This can be done by assigning two variables having the initial values of r_1, an integer with *all bits between positions i and j as 1*, and r_2, an *integer with only bit positions i and j set as 1*. These variables can be defined at compile time with their corresponding values. At runtime, the following code needs to get executed for Bookkeeping:

```
if ( (r AND r1) == r2)
        inform OPTIMIZER that edge (i, j) is taken;
```

Bookkeeping needed to track if specific edges are taken

It is easy to see why this works. If the edge $i \rightarrow j$ is ever taken, then the bit positions i and j of r will be 1 (by definition of our profiling technique). Moreover all the intermediate bit positions between i and j will be 0 (otherwise the edge $i \rightarrow j$ could not have been taken since the blocks are topologically sorted). Thus when r is ANDed with r_1, the only bit positions which will be 1 are i and j, making the profiled code call the optimizer. If after executing the TFP instrumented code for K iterations the *OPTIMIZER* is not informed (we need not necessarily inform the *OPTIMIZER* but can just set a *flag* to true as well) we can conclude that the monitored edge is not taken *persistently*.

3.6 TFP for normal path profiles

It must be mentioned that TFP can be used to measure normal path frequencies as well. We have already seen that each path produces a unique value in r. This value can be hashed into a counter array at the end of the loop (back-edge) to maintain an exact count of the path frequencies. However, path profiling techniques like [9] will do a better job of maintaining such frequencies alone. The range of the path identifiers used by this technique is exactly equal to the total number of paths, making direct indexing into the counter array possible. Both TFP and Bala's

method [6] use path identifiers that do not reflect the actual number of paths in the instrumented region, thereby requiring hashing. To summarize, we claim that several dynamic optimizations might not need the "exact" frequency of paths. However, if needed, TFP can easily be modified to maintain these frequencies without adding to the overheads significantly ([6] required ≈ 3 cycles for their hashing phase).

These are just some of the statistics we can gather using our profiling strategy. One can easily change the Bookkeeping segment to calculate further statistics like basic blocks that are always taken, minimum amount of persistence between paths etc. Moreover we have already seen that some part of the bookkeeping needed for different statistics overlap, making the bookkeeping more efficient.

4 Implementation Issues for TFP

In this section we discuss some of the issues involved in implementing TFP and how the strategy can be modified in different situations.

4.1 Use of Variables and Registers

Much of TFP's value relies on the fact that it uses only a few variables to achieve profiling as well as to maintain the information gathered. Traditional profilers can consume large amounts of memory to store profiled data. This use of memory can have adverse effects on performance if used in a runtime scenario. TFP shows that it is possible to maintain a fairly wide and relevant range of runtime information by using only a few variables. This, however is based on the assumption that the number of blocks in the instrumented region is not too large. If the number of blocks instrumented by TFP is small, a single register can be used to represent all the blocks (since each block is represented by a bit in the register). This helps in reducing the overheads of TFP as we avoid reloading values from memory every time profiling occurs, and all the data needed for profiling can be maintained in a single register. The bookkeeping may need a few extra variables (depending on the amount of information we want to gather) but still this would be significantly less than using large arrays to store the frequencies of every path/block/edge.

To test our assumption that a single register is sufficient to store temporal program behavior, we profiled the code regions covered by the most frequently executed back edges in the SPEC 2000 benchmarks[1] to see how many basic blocks they cover. The results are in Figure 4. It is observed that more than 99% of these frequently executed code regions have less than 64 blocks in them. This implies that in nearly all cases a 64 bit register is sufficient to implement TFP efficiently.

To use TFP for code regions having more than 64 blocks, we can use the same technique as long as the number of block instrumented is not too large. We use a new variable every time we finish instrumenting 64 blocks (assuming we are using a 64-bit register) i.e. instead of just using r we use (r_1, r_2, ..., r_n) as needed. At the

[1] We did not consider some trivial two block loops having just a single path. Also for *eon* and some FP benchmarks we considered less than 10 back edges as there was a significant drop in the frequencies of the remaining ones.

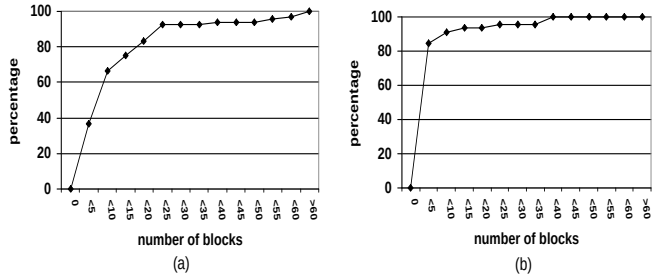

Fig. 4. Cumulative distribution of the number of basic blocks present in the profiled code regions in (a) INT Benchmarks and (b) FP Benchmarks

end in the Bookkeeping section, instead of checking if ($r = prev$) we check if ($r_1 = prev_1$ AND $r_2 = prev_2$ ... AND $r_n = prev_n$) and set all r_is to 0 after that. Thus we make up for not being able to store the bit stream corresponding to a path in a *single* variable by maintaining parts of the bit stream in *separate* variables. However, $n = \lceil$ *(Num of Blocks Instrumented)/64* $\rceil$, is rarely more than 1 (for the 110 code regions we instrumented only one had more than 64 blocks in it). Thus at most we will need a few extra variables, which is still better than using large arrays. Moreover, if we assign blocks the variables they update by traversing the instrumented code region in a depth-first manner, we ensure that most paths do not involve updating more than one variable. This would mean that a variable once loaded in a register is likely to remain there for the whole path, reducing unnecessary *load*s at runtime.

4.2 Nested Loops and Procedure Boundaries

Till now we have discussed how TFP can be applied to an *acyclic* region of code. TFP can also be applied to multiply-nested regions of code. A simple way to achieve this this would be to assign a separate variable to monitor different levels of loops. Figure 5 shows this assignment. Since loops normally don't have more than 2-3 levels of nesting, this should not be a problem.

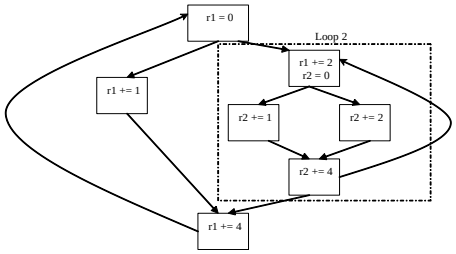

Fig. 5. Effect of nested loops. We use 2 variables r1 and r2 for the 2 levels of the loop

The same technique can also be used to perform inter-procedural profiling using TFP by treating function calls as inner-loops and using separate variables to profile them.

Another trend of interest is to make the tracking of these *PFP*s last across multiple procedure calls. For example one might detect a *K-PFP* in a procedure, even if the K runs of the instrumented code region is spread across multiple calls to the procedure. A simple way of achieving this would be to declare the TFP variables used for profiling the procedure as *static* so that they are persistent across multiple procedure calls.

5 Experimental Results

We performed several experiments using TFP on the SPEC 2000 benchmarks and a real application RNAfold [21]. The objectives behind the experiments were three fold - to study the overheads of TFP for runtime profiling, analyze the information collected by TFP to test for non-trivial *PFP*s in programs, and to study the overall impact of using TFP in a real application.

5.1 Overheads of using TFP

We implemented TFP on 7 SPEC 2000 INT benchmarks and 6 FP benchmarks (the remaining benchmarks were also considered but were left out since most of their dominant back-edges led to trivial single-path regions). Instrumentation was done using ATOM [22]. We first instrumented the programs to detect the most frequently executed back edges, and then instrumented the code regions covered by these back edges. We omitted trivial two-block loops with a single path between them. There remained 4 regions (out of the total 110 regions we instrumented) with only one static control flow path between them. Ideally the compiler would have coalesced them into a single block but it did not do so. The remaining 106 regions had more than one possible static path in them. We did not have a customizable compiler for our use, and ATOM itself adds overheads. Thus we decided to test the overheads of TFP by comparing it with our implementation of [6], one of the fastest runtime path profiling techniques. TFP did not maintain the path frequencies since the primary purpose of our experiments was to study the use of TFP in gathering PFPs. To be fair we did not save the results of [6] as originally done (thus preventing it from making unnecessary *stores*) but just used it to ensure that the same path was persistently taken. TFP on the other hand not only tracked persistent paths but also tracked persistent sub-paths and untaken blocks (Section 3.3 and 3.4). Ideally one would stop running the instrumented code once a *PFP* is detected to take optimization actions. However, for our experiments, we wanted to ensure that the instrumented code kept running for the entire duration of the program (to study its overall overheads) and therefore set a very high value of K. The normalized results are shown in Figure 6.[2]

[2] Actual slowdowns varied from 10% to nearly a factor of 3. Most of the overheads were because of ATOM's use of procedural calls to instrument code. Thus even a single line of instrumented code involved making a procedural call. Compiler support will help in bringing down these overheads considerably.

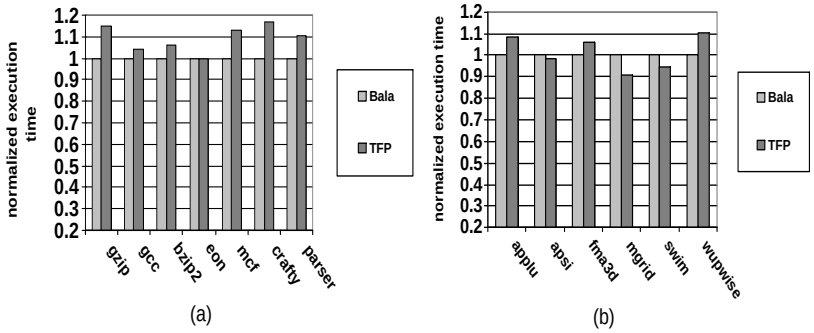

Fig. 6. Comparison of TFP and Bala's technique on (a) INT Benchmarks (b) FP Benchmarks

On average, TFP was only 5.75% slower than Bala's method, even though it gathered a wider range of information (for our experiments TFP tracked persistent paths, persistently untaken blocks and persistent sub-paths while Bala's only gathered persistent paths). For three of the FP benchmarks, TFP outperformed Bala's method. This happens because Bala's method needs two instrumentation statements (a bitwise OR and a register shift) at each *conditional* edge[3], while TFP requires a single instrumentation statement (a bitwise OR) at *every* block. For the FP benchmarks the paths were small and the number of blocks in a path was comparable to the number of conditional edges along the path, making TFP more efficient. For the integer benchmarks we observed that several blocks that could be coalesced together were separate. Since we did not have control over the compiler, we instrumented each of these blocks, though ideally they would have been one block (reducing our overhead). Since there were no conditional edges in these blocks Bala's method did not instrument them. We believe our 5.75% relative slow down is a good result, since Bala's technique achieves nearly zero overhead profiling with adequate compiler support. We thus conclude that TFP is lightweight enough for runtime use on these benchmarks.

5.2 Statistics from TFP

In this section we present some runtime statistics collected by TFP on the SPEC 2000 benchmarks. These statistics reveal the presence of *persistent* trends in programs which can be used for dynamic compilation.

Persistent Paths We ran TFP over the SPEC 2000 INT and FP benchmarks and detected persistent paths with different values of K. The results from these

[3] Often one needs additional conditional statements to instrument conditional edges. TFP instruments at the block level and does not add additional conditional statements in the code.

experiments are shown in Table 1.We used static variables to track the paths as mentioned in Section 4.2. The total number of static paths in these regions and the number of paths that were persistent for a given value of K are given. The percentage of total iterations in the instrumented regions contributed by the *persistent paths* is also provided. Since we have considered the most frequently executed back edges, the instrumented code regions constitute a large fraction of the program's actual running time. In summary, the regions of code we instrumented had 1961 static paths each on an average. Of these a small number of paths ($\approx$ 16 for K=50 and $\approx$ 14 for K=100) account for a fairly large percentage ($\approx$ 61% for K=50 and $\approx$ 59% for K=100) of the total iterations in these regions at runtime.[4] These paths also have the property that the code continuously stays in these paths for at least 50/100 iterations on average without shifting to the other possible paths in the region. Thus it makes sense to perform path-specific runtime optimizations on these paths since *(i)* these paths constitute a fair fraction of the executed code and *(ii)* the path-specific optimizations will hold true for that period, allowing them to be profitable.

Benchmark Name	*Number of Static Paths Instrumented*	*Persistent Path Statistics*			
		K=50		*K=100*	
		paths	*%*	*paths*	*%*
cc1	33	11	98.14	10	96.67
gzip	3563	15	0.912	10	0.658
bzip2	1057	11	49.11	11	45.71
mcf	130	18	55.81	18	51.91
crafty	826	62	15.54	40	12.13
eon	20	4	3.109	3	3.108
parser	19623	40	45.90	35	41.18
AVG (INT)	**3607**	**23**	**38.36**	**18.14**	**35.91**
swim	9	4	99.99	4	99.99
applu	48	6	85.71	6	85.71
apsi	27	9	99.99	9	99.99
wupwise	18	4	61.54	4	61.54
mgrid	46	10	99.84	10	99.65
fma3d	94	16	75.05	16	75.05
AVG (FP)	**40.33**	**8.16**	**87.02**	**8.16**	**86.98**
AVG (net)	**1961**	**16.15**	**60.81**	**13.53**	**59.48**

Table 1. Runtime *Persistent Paths* detected by TFP in SPEC 2000 INT and FP benchmarks. *Path* denotes the number of unique persistent paths that TFP detected and the percentages denote what fraction of total paths executed at runtime in the instrumented regions were persistent.

[4] Note that 50-PFP $\supseteq$ 100-PFP and 50-PFP $\approx$ 100-PFP implies that most of the PFPs with persistence 50 also had a persistence of 100.

Persistently Untaken Blocks Information that might also be of use is the number of blocks that *do not* get executed *persistently*. One can remove these blocks from the code iterations, which might lead to several subsequent optimizations. We used TFP to detect opportunities for such optimizations. Once again we used static variables for our instrumentation and after every K iterations of that code region we checked to see which were the blocks that were not executed even once during these iterations. A total of the number of such blocks and the total number of blocks for every set of K iterations was maintained and is provided as an average in Table 2. We have also provided the average number of blocks in the code regions we instrumented to give an estimate of how many blocks one might actually eliminate temporarily. Since block-reduction is a smaller sub-set of path-reduction we set higher values of K for these experiments(500,1000). To sum up the results of Table 2 - our instrumented code regions had on an average 10.67 blocks each, of which 29.03% blocks were not executed for at least 500 consecutive runs of these regions and 27.94% of the blocks were not executed for at least 1000 consecutive runs of these regions. One can thus try to eliminate these blocks, and can make further optimizations which are likely to be valid for at least a while.

Benchmark Name	*Average Number of Blocks/instrumented region*	*% of Blocks NOT taken Persistently*	
		K=500	*K=1000*
cc1	6.6	38.97	31.80
gzip	16.4	22.94	21.73
bzip2	13.6	65.10	64.15
mcf	10.5	44.94	44.06
crafty	24.2	24.12	21.59
eon	10.0	0.017	0.017
parser	14.6	19.38	17.93
AVG (INT)	**14.27**	**30.78**	**28.76**
swim	4	0.000	0.000
applu	5	52.47	52.47
apsi	4.4	15.72	15.72
wupwise	7.4	39.18	39.18
mgrid	5.5	0.391	0.300
fma3d	12.5	54.22	54.18
AVG (FP)	**6.47**	**27.00**	**26.98**
AVG (net)	**10.67**	**29.03**	**27.94**

Table 2. Runtime *Persistently Untaken Blocks* detected by TFP in SPEC 2000 INT and FP benchmarks.

5.3 A Case Study: RNAFold

We studied if TFP could lead to improved program performance on RNAfold [21]. This computational biology application folds a given RNA sequence and returns its minimum free energy. The major part of the program is spent in a loop of the form:

```
for (decomp = INFINITY, k = start_value; k <end_value; k++)
        if (decomp > Array1[k]+Array2[k+1][j])
                decomp = Array1[k]+Array2[k+1][j];
```

Though this is a predominantly memory-intensive loop, one can get some benefits by unrolling the loop. However, there is a possibility of a true dependence on *decomp* between successive iterations of the loop. If we implement aggressive unrolling and *decomp* is seldom changed, then we can get a fair amount of additional parallelism. However, we observed that if *decomp* changed frequently (thereby limiting the parallelism), unrolling slowed down the overall execution by consuming additional resources (registers etc.). For RNAfold it is not possible to decide at compile time whether unrolling might be useful, since the decision is dependent on the data values of the input arrays. One can use TFP to detect *PFP*s in the loop (either a persistent path along the dependence-free path OR to see if the edge leading to the dependence is ever taken). If we notice that the path along which *decomp* doesn't change is executed persistently we can decide to unroll the loop.

Ideally the instrumentation and optimization would be done in the compiler. However, since we used an existing compiler (gcc-2.96) that we did not have full control over, we hand-coded the optimization. We manually implemented different unrolled versions of the loop (3-level and 4-level). The original loop was instrumented using TFP. The instrumentation searched for certain degrees of persistence along the path where *decomp* did not change and on finding such a trend it passed on control to the corresponding optimized, unrolled version. To test the usefulness of TFP in this experiment we also ran a separate version of the code with just the unrolled version of the loop. We ran the program with four different sizes of input sequences. The results are shown in Figure 7.

The TFP-enabled unrolled version outperforms both the original code and the unrolled version (without TFP). This is because the unrolled version uses up registers and is only useful if it manages to introduce additional parallelism. The TFP enabled version uses the original loop till it finds a persistent trend, and then dynamically transfers control to the unrolled version, making the optimization more profitable. Though the improvements in time are small, these experiments show that time sensitive flow information can be used to improve overall program performance at runtime.

6 Conclusion

In this paper we presented a new profiling strategy, TFP, designed to be used in the context of dynamic compilation and optimization. In such a context, profiling must not only provide information useful in a dynamic setting, but do so with

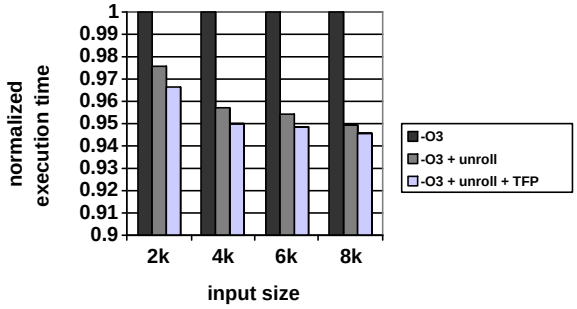

Fig. 7. Normalized execution time for for three different optimized versions of RNAfold

low runtime overhead. Our strategy, TFP, can collect a range of time-sensitive, control-flow-based information which is more detailed than than that collected by block, edge or path profiling. Despite being more powerful, TFP's overhead is on average only 5.75% greater than that of [6]. We used TFP to gather statistics from the SPEC 2000 benchmarks, showing possible opportunities for profile-directed flow specific optimizations at runtime. We also showed a case study that demonstrates the usefulness of the information collected by TFP for optimization at runtime.

TFP is still in its initial stages. We plan on incorporating it in the context of a dynamic compiler to further explore its usefulness and actual overheads. Moreover, the amount of *persistence* (K) needed at runtime to actually produce benefit should be explored. Work is also going on to find efficient ways of using TFP to also gather the exact path frequencies, if needed, at runtime. We plan to study if the definition of persistence can be relaxed (to accommodate a larger range of information) without adding to the overheads. Work is also going on in using TFP for "application profiles" that can help in predicting application performance.

References

1. A. M. Alkindi, D. J. Kerbyson, and G. R. Nudd. Dynamic instrumentation and performance prediction of application execution. *Lecture Notes in Computer Science*, 2110, 2001.
2. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.
3. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM: The controller's analytical model. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.
4. Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Instrumentation and Feedback-Directed Optimization of Java. In *In the proceedings of the ACM Confer-*

ence on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), November 2002.
5. Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *In Proceedings of the Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001.
6. Vasanth Bala. Low overhead path profiling. *Technical Report, Hewlett Packard Labs*, 1996.
7. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 1–12. ACM Press, 2000.
8. Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
9. Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
10. B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. In *Journal of Instruction Level Parallelism*, 1999.
11. P. P. Chang and W. W. Hwu. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, pages 21–29. IEEE Computer Society Press, 1988.
12. Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software - Practice and Experience*, 21(12):1301–1321, 1991.
13. Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02*, pages 199–209. ACM Press, 2002.
14. Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, 1999.
15. Martin Hirzel and Trishul M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Workshop on Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001.
16. Thomas Kistler and Michael Franz. Continuous pogam optimization: Design and analysis. *IEEE Transaction on Computers*, 50(6):549–566, June 2001.
17. D.E. Knuth and F.R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.
18. James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999.
19. M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, , and W.M. HwuB. An architectural framework for runtime optimization, June 2001.
20. Michale Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot server compiler. In *In Proceedings of the USENIX Symposium on Java Virtual Machine Research and Technology*, 2001.
21. Vienna-RNA-Package. http://www.tbi.univie.ac.at/ĩvo/RNA/, 2002.
22. A. Srivastava and A. Eustace. Atom:a system for building customized program analysis tools. In *Proceedings of 1994 ACM Symposium on Programming Language Design and Implementation*, pages 196–205. ACM Press, 2002.
23. C. Young and M. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, 1994.