Impact of Path Profile Estimation on Superblock Formation

Jeffrey Pang
jeffpang@cs.cmu.edu

Jimeng Sun
jimeng@cs.cmu.edu

Abstract

Many features of modern architectures, such as Performance Monitering Units (PMUs), can be leveraged for efficient program profiling. We present several simple heuristics to estimate path profiles from sampled partial paths which can be collected at runtime. Like recent work [13], we show that we can find 80-90% of the hot paths within a program using these techniques. However, we also find that despite this high accuracy in identifying hot paths, using estimated paths to perform *superblock* formation does not necessarily yield an equivalent boost in performance. We evaluate how estimated path profiles impact superblock formation in the CASH compiler. In addition, we compare superblocks to hyperblocks and demonstrate that predicate calculation overhead can cause maximal hyperblocks to have inferior performance.

1 Introduction and Motivation

Profiles of a program's runtime behavior, such as instruction, edge, or path counts, provide valuable information that enables a number of optimizations, such as trace scheduling [8], superblock formation [9], code positioning [12], and improved function inlining [6]. Moreover, a number of dynamic optimization systems [3, 7] use execution profiles to continuously optimize programs in the face of changing or phase-oriented workloads.

Path profiles are especially useful because they capture correlation among different branches in program traces. Traditionally, accurate path profiles are collected by explicitly instrumenting program code, such as in [4]. However, instrumentation usually degrades performance too much to include in production code, sometimes limiting path profiles to unrealistic or unpredictable workloads. For example, Ball and Larus [4] noted that even efficient path profiling incurred an overhead of 31%.

Nonetheless, Anderson, *et al.* [2] showed that by taking advantage of specific microprocessor features like interrupts, *sampling* in a real production system can be used to obtain relative instruction counts and other profiling information with negligible overhead. Nonetheless, their statistics were primarily designed to be used by humans rather than compilers. Given the high cost of human labor (when compared to automatic optimization) and the advent dynamic optimization systems that could benefit from more accurate online profiles, we ask the following question: Can we estimate path profiles well enough to be used profitably in profile-based compiler optimizations?

We explore one specific scenario in this paper to begin answering this question. We assume that modern microprocessors, like the Intel Pentium 4, Intel Itanium, and IBM PowerPC 970, provide a rich set of performance counters. In particular, we describe how we can leverage branch samples from a *Performance Monitoring Unit* (PMU) to estimate path profiles. Like [13], we find that we can accurately reconstruct profiles for 80-90% of *hot* paths within programs, despite the simplicity of our estimation heuristics. However, our evaluation shows that capturing a large fraction of the hot paths is sometimes *not enough* to capture a similar amount of performance gain from optimizations like *superblock formation* [9], which we implemented for evaluation. Although programs optimized using our estimated path profiles (using superblock formation) beat out locally scheduled basic blocks by a large margin, they are still non-trivially worse off than if they had been optimized using perfect path profiles. This is partially due to inaccuracies in the *relative ranks* of the hot paths themselves.

In addition, because we evaluate our techniques in the context of the CASH compiler [5], which compiles C to asynchronous circuits, we also show that using path profiles to form superblocks does not cause a significant performance degradation when compared to using maximal sized *hyperblocks* [11], despite the reduction in parallelism (due to lack of predication). In fact, in certain cases, we show that profile-based superblocks perform better.

The remainder of this paper is organized as follows. Section 2 describes related research efforts. Section 3 describes our heuristics for estimating path profiles. Section 4 briefly describes our algorithm for superblock formation. Section 5 presents a short evaluation of path profile estimation and its impact on superblock formation. Finally, Section 6 concludes.

2 Related Work

Traditional path profiling efforts [4, 10] focus on using instrumentation to perform profiling. Path profile estimation by Chen, et al. [7] and Shye, et al. [13] attempt to leverage performance counters to sample branches taken at runtime and extrapolate profiles from them, and hence are most similar to our efforts. Chen, et al. [7] focused on trace selection for dynamic optimization and did not try to predict path profiles directly. Shye, et al. [13] used sampled branch to extrapolate path profiles. Our results on estimated path profile accuracy are similar. However, our heuristics are much simpler. In addition, their estimator only considered simple paths while we also consider general paths, which can guide loop unrolling in superblock formation, as described in Section 4. Moreover, in addition to quantifying the accuracy of estimated paths, we also evaluate the impact of estimation on an actual optimization and show that capturing a large fraction of the hot paths is not always good enough to capture a similar fraction of the performance gain.

We implemented path-profile based superblock formation to evaluate the impact of path-profile estimation error. Our algorithm is based on work by Hwu, et al. [9] and Young and Smith [15].

3 Path Estimator

The profiler is responsible for producing profile information based on the execution environment. A path profile records the access frequencies of the hot paths usually accurately reflects the true execution ordering. However, substantial overhead is required to collect the path profile perfectly. Unlike a path profile, an edge profile, which only records the access frequencies of the edges, is much cheaper to collect (or at least sample). But trying to estimate a path profile from an edge profile is inaccurate due to the independence assumption between branches when in reality there may be correlations.

The goal is to have a profiler that can provide accurate estimation and requires only a small overhead. An observation is that we can collect something that has more information than an edge profile but still requires very low overhead, such as a partial path profile [13], which can be collected by sampling a small number of consecution branches. Compared to an edge profile, a partial path profile partly captures the dependency across different branches. Moreover, modern hardware allows efficient collection of such partial paths. For example, the BTB registers in the Intel Itanium microprocessor (part of the PMU) effectively act as a four branch circular buffer. Hence, with a PMU, we can sample length four paths in the execution sequence with negligible overhead [13].

However, partial paths are still incomplete when compared to a perfect path profile. Based on the partial paths, the we want to estimate the full path information. More formally, given the program control flow graph (CFG) and the set of partial paths $(p_1, \ldots, p_n)$ of length k and their respective counts $(c_1, \ldots, c_n)$, we want to estimate the set of paths of length $\leq m$ (where $m > k$) and their relative counts. To perform this estimation we use the following heuristics:

Join: Given two paths p_1 and p_2, if the tail of p_1 is the same as the head of p_2, we join p_1 and p_2 to produce a new, longer path with the count equal to min(p_1, p_2). For example, we have $p_1 =$ ABCD with count c_1 and $p_2 =$ CDEF with count c_2 (note that our paths here are sequences of edges, not basic blocks). The join result is ABCDEF with count min(c_1,c_2).

However the join on the original paths can only provide limited number of additional paths. If two paths are disjoint, it is not possible to perform direct join on them. The following heuristics provides methods to extend these paths.

One-step Path Extension: Given a path p and its count c, we can extend p to include one edge following p. The new count is the product between c and the access probability of the edge. For example, suppose we have $p =$ AB and $c = 100$, as in the CFG shown in Figure 1. We know that p can be extended into ABC or ABD. The counts for ABC and ABD are 40 and 60 respectively, because the probability for taking C and D are 0.4 and 0.6, respectively. Note that

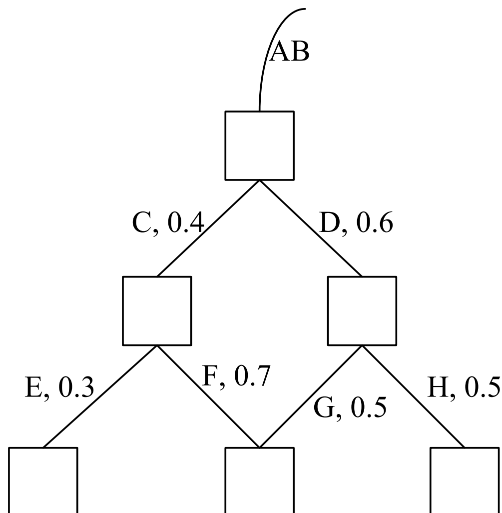

Figure 1: Example: part of a CFG

if there were multiple paths leading into B, then this extension heuristic suffers from the assumption that branches are independent. However, because sampling will likely give us information for popular, correlated paths, this assumption is more likely to be correct for paths we are missing and is less likely to affect the accuracy of popular paths.

After extending the paths, we can try to join the paths again which may lead to more join results. For example, suppose originally we have path p_1 = AB and p_2 = CD where no join can be performed. However, after extending AB to ABC and ABD, we can join ABC with CD to yield ABCD.

Multi-step Path Extension: Naturally, we can extend the method into multiple steps to obtain even longer paths. For example in Figure 1, one-step extension of AB produces ABC and ABD with count 40 and 60, while 2-step extension produces ABCE, ABCF, ABDG, and ABDH with count 12, 28, 30, and 30.

By apply join and path extension iteratively, we can effectively produce longer paths. The termination conditions are either that 1) we reaches the specified maximum path length, or 2) the count for the new path is below a specified threshold.

4 Superblock Formation

In order to evaluate the effectiveness of estimated path profiles on compiler optimizations, we implemented a path-profile based superblock formation optimization, largely based on the heuristics developed by Young [15]. We briefly describe the algorithm in this section.

4.1 Formation Algorithm

Superblock formation attempts to find paths (or *traces*) of a program that frequently executed together and transforms the sequences of code so that they can be scheduled together. Although this expands code size, a scheduler can expose more parallelism in larger regions of code, especially on VLIW microprocessors or when targeting asynchronous circuits, as in our case. More formally, a superblock is a sequence of basic blocks that has a single entry point (the *head*) and, hence, no side entrances, though it can have multiple exits. The dark green traces on the right side of the arrows in Figure 2 are examples of superblocks.

To construct superblocks, we first partition the control flow graph into disjoint traces as follows: We sort the basic blocks in descending order based on execution frequencies, which can be inferred from the path profile. Then we consider each block in turn to be the head of a new superblock if it is not already part of another superblock. Suppose we choose block A to be the head of a superblock, as in the first example in Figure 2. Then we extend the superblock along the most likely successor block (either B or F), determined by examining the execution frequency of that particular path (A B or A F). We continue this process until we reach a block that is part of an existing trace.

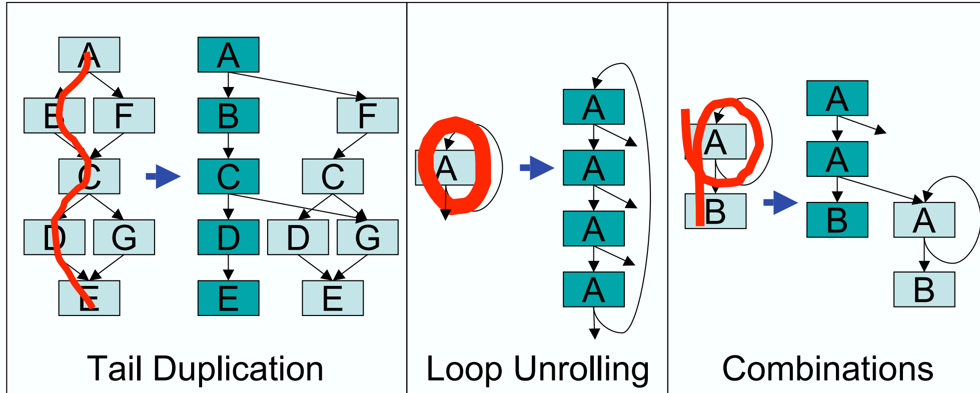

Figure 2: Examples of superblock code transformations.

```
// corr1,corr2,etc. are correlated
if (corr1(a))
    ...
else
    ...
if (corr2(a))
    ...
else
    ...
if (corr3(a))
    ...
else
    ...
...
```

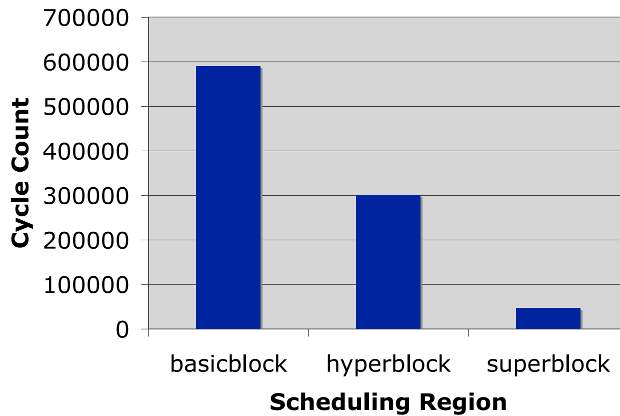

Figure 3: Example of a case when superblocks outperform maximal hyperblocks

Then we try to extend each trace by examining whether considering additional blocks past the tail of a trace would still yield a frequent path (we consider executing more than 1/16 of the time frequent). Note that this extension heuristic captures both extension into another trace and *loop unrolling* or *peeling*, such as shown in the second example of Figure 2. For example, if A A A A is a frequent path in our profile, then this extension technique will extend the trace by "unrolling" the loop containing A. We unroll loops up to 4 times. We also capture the case when a loop executes for a small number of times and proceeds to another block, such as in the third example in Figure 2.

Once the traces are extended, we perform the necessary code duplication of basic blocks to ensure that each trace has no side entrances.

4.2 Comparison with Hyperblocks

On architectures supporting predicated instructions (including CASH), it is possible to schedule multiple targets of a single branch at the same time by speculating on which block will actually execute. These larger code regions, called hyperblocks [11], can expose even more parallelism than path-profile based superblocks since a superblock is limited to a single flow of execution.

However, this does not always imply superior performance. For example, when we have a highly correlated sequence of a large number of branches, such as shown in Figure 3(a), forming a maximal hyperblock (i.e., one that encompasses all of the branches) can incur a substantial overhead due to predicate calculation. Figure 3(b) compares the time to execute 5000 iterations through a similar sequence of 30 branches when compiling using different regions of scheduling granularity in the CASH compiler (or, in the asynchronous circuit context, the granularity of forming hyperblock circuits). `basicblock` schedules locally in each basic block, `hyperblock` schedules the entire region as a single hyperblock, and `superblock` forms superblocks using a path profile. In this extreme case, using path

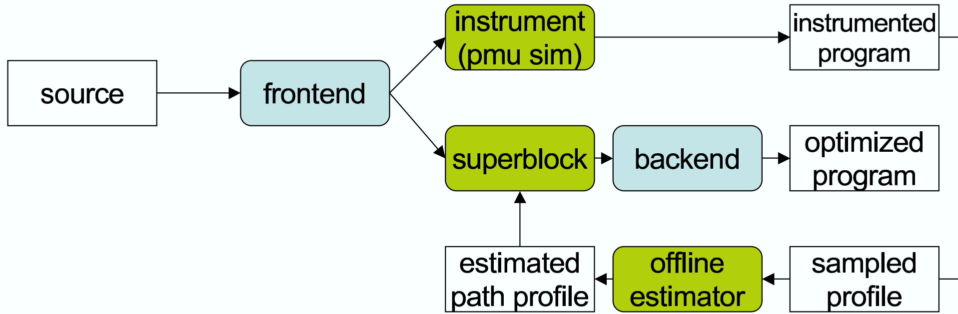

Figure 4: Evaluation framework.

profile based superblocks achieves nearly an order of magnitude improvement over maximal hyperblocks. In addition, we believe that it is likely that the superblock case also reduced energy consumption since there is less speculative execution. However, we were not able to test this hypothesis with the current CASH simulator.

Of course, similar profile information can also be used to guide more intelligent formation of hyperblocks. We believe this topic would be an interesting topic for future research.

5 Evaluation

Figure 4 shows the components in our evaluation framework, which we implemented using the SUIF compiler system [1]. To collect exact and sampled path profiles, we instrumented branches in compiled code using the HALT library [14] and a custom library to simulate sampling at regular intervals.[1] We compute the estimated path profile from the sampled partial paths using an offline tool which implements the heuristics described in Section 3.

In our evaluation, we do not perform any optimizations before superblock formation, however our backend, the CASH compiler, performs several simple optimizations such as dead code elimination and some trivial constant propagation. We use the CASH simulator to simulate the kernel of select benchmarks.[2]

5.1 Path Profile Accuracy

We collected a sampled path profile consisting of partial paths of length 4 and a perfect path profile consisting of paths of length 20. We used a sampling interval of 10,000 cycles. We use a relatively short period because the benchmarks we used were unusually short. When running longer programs, such as those in the Spec95 benchmark suite, we were able to capture similar accuracy with a sampling interval of several million instructions, which [13] showed could be done with minimal runtime overhead. However, we were unable to run these benchmarks in the current CASH compiler and simulator. Nonetheless, we believe that longer programs are the most likely target for optimization in the real world. We ran the estimator on the sampled profile to compute an estimated profile with estimated paths of length 20.

To evaluate the quality of the estimated path profile, we compare the top k hot paths P' captured in the estimate paths to the actual top k hot paths P from the exact path profile. Like [13], the accuracy metric we use is:

$$\frac{\sum_{p \in (P' \cap P)} c(p)}{\sum_{p \in P} c(p)}$$

where $c(p)$ is the count of path p. The metric computes the ratio between the correct hot paths from the estimation and all the hot paths, weighting each path by its frequency.

[1] Although we do not count the cycles expended in the instrumentation code as part of the sampling period, because the instrumentation inserts a function call at each branch point, our sampling periods are still somewhat inaccurate, especially when considering architectures with better branch prediction, etc. Nonetheless, we note that our accuracy results are comparable to those in [13], which performed a much more thorough simulation of partial path sampling on an Intel Itanium microprocessor.

[2] In particular, we used the new simulator with the `all_one` latency model.

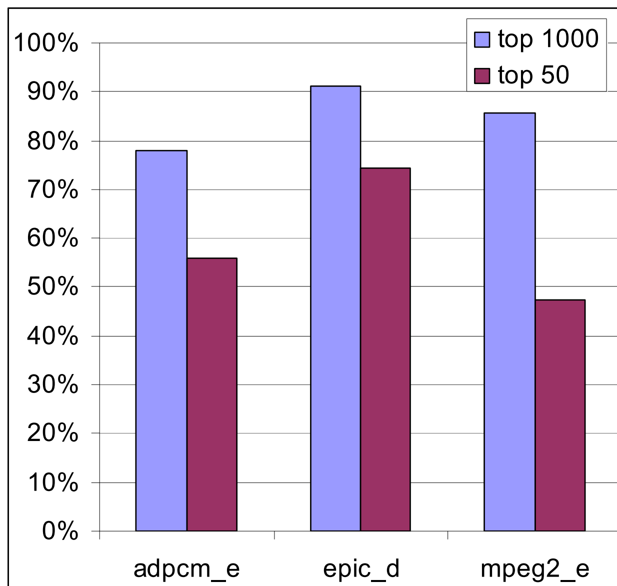

Figure 5: Accuracy of short benchmarks

Figure 5 shows the accuracy over different benchmarks. When looking at the top 1000 paths (out of 4786, 37,334, and 1,675,859 total paths for `adpcm_e`, `epic_d`, and `mpeg2_e`, respectively), we find 80-90% of the actual hot paths. However, the accuracy is much lower with the top 50 paths. This suggests that the estimated profile does not always preserve the relative ranks of the true hot paths, although almost all the hot paths are captured at least once. The error is due to two factors: 1) the sampling error and 2) the estimation error. The sampling error is due to the variation of the random samples when we collect the partial path information. The estimation error results from the independence assumption of the path extension heuristics and the sampling error (since it uses the imperfect partial paths as the input).

Unfortunately, the relative ranking of paths, especially those which may overlap, is important to the superblock formation optimization, which we will see in the next section.

5.2 Superblock Performance

Next, we evaluated how estimated path profiles impacted superblock formation. Figure 6 compares the execution time for the same three media benchmarks using local scheduling (`basic`), maximal hyperblocks (`hyper`), superblock formation using a perfect path profile (`super`), and superblock formation using an estimated top-1000 path profile (`estimate`). Although the last section showed that we can find a large fraction of the hot paths, the relative ranking inaccuracy causes superblock formation to perform worse than we might expect from looking at the accuracy estimated for the top-1000 paths in Figure 5. For example, in the `adpcm_e` and `epic_d` benchmarks, using a perfect path profile, superblock formation was able to produce a program that ran more than 30% faster than local scheduling, while using the estimated profile, it was only able to produce a program that ran about 20% faster. Nonetheless, in the case of `mpeg2_e`, the margin between the two cases is only a few percentage points. We note that the relative ranks of two paths only matter in superblock formation when they overlap, so the impact is also not as severe as the ranking inaccuracy shown in Figure 5 might suggest.

Another interesting feature to note in Figure 6 is that perfect path-profile based superblock formation actually produces a faster `epic_d` than when using maximal hyperblocks, and even the version produced using the estimated profile is competitive. Hence, predicate calculation overhead can be substantial even in real world programs. We believe that an examination of control heavy programs, such as some programs in the SpecInt benchmark suites (e.g., `perl`, `gzip`, etc.), would exhibit even greater performance gains when using superblocks. However, we were unable to run these using the current CASH compiler.

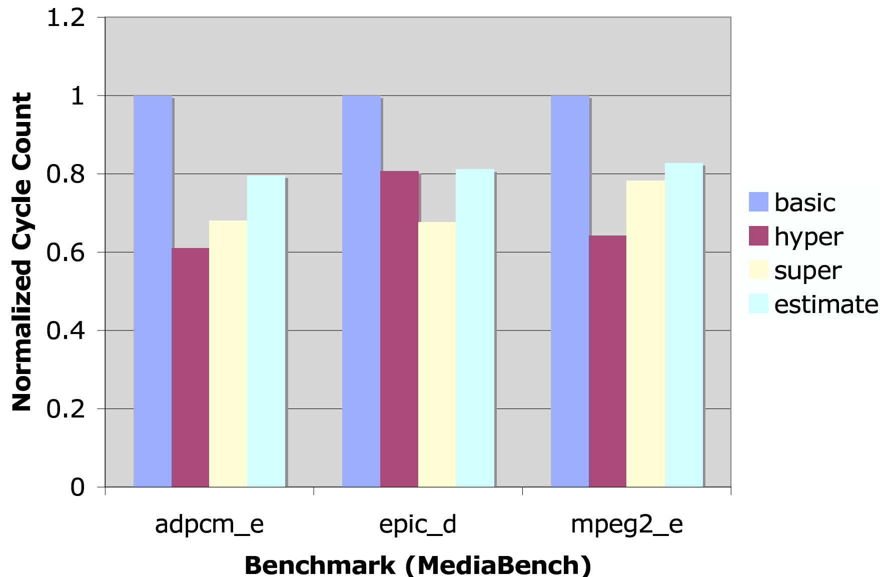

Figure 6: Performance of Superblock formation.

6 Conclusion

Like recent work [13], we showed that we can find 80-90% of the hot paths within a program using simple path estimation techniques on a small sample of short partial paths. However, we also found that despite this high accuracy in identifying hot paths, using an estimated path profile to perform *superblock* formation does not necessarily yield an equivalent boost in performance. We evaluated how estimated path profiles impact superblock formation in the CASH compiler and found that sometimes inaccuracies in the relative path ranks can reduce expected performance gains. In addition, we compared superblocks to hyperblocks and demonstrated that predicate calculation overhead can sometimes cause maximal hyperblocks to have inferior performance.

Thus, we believe that while there is promise in using modern architecture features to capture profiles with low overhead, there is additional research that can be done to make these profiles more accurate so that compiler optimizations can exploit them to their full potential.

Acknowledgements

We would like to thank Pedro Artigas, Tim Callahan, Tiberiu Chelcea, Mahim Mishra, and Dan Vogel for helping us with the CASH compiler. Our path sampling library also borrowed heavily from code by Tim Callahan and Cliff Young.

References

[1] The SUIF 1.x compiler system.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[4] T. Ball and J. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. *SIGARCH Comput. Archit. News*, 32(5):14–26, 2004.

[6] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, New York, NY, USA, 1989. ACM Press.

[7] H. Chen, W. Hsu, J. Lu, P. Yew, and D. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the International Symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003.

[8] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. In *IEEE Trans. Comput.*, pages 478–490, July 1981.

[9] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. In *The Journal of Supercomputing*, pages 229–248, 1993.

[10] R. Joshi, M. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[11] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsl.*, 23(1-2):45–54, 1992.

[12] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[13] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. Reddi, and D. Connors. Analysis of path profiling information generated with performance monitoring hardware. In *Proceedings of the 9th Workshop on Interaction between Compilers and Computer Architecture*, Feb. 2005.

[14] C. Young. The Harvard Atom-like Tool (Halt) manual.

[15] C. Young and M. D. Smith. Better global scheduling using path profiles. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 115–123, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.