Appears in International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept. 2001.

On the Stability of Temporal Data Reference Profiles

Trishul M. Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Abstract

Growing computer system complexity has made program optimization based solely on static analyses increasingly difficult. Consequently, many code optimizations incorporate information from program execution profiles. Most memory system optimizations go further and rely primarily on profiles. This reliance on profiles makes off-line optimization effectiveness dependent on profile stability across multiple program runs. While code profiles such as basic block, edge, and branch profiles, have been shown to satisfy this requirement, the stability of data reference profiles, especially temporal data reference profiles that are necessary for cache-level optimizations, has neither been studied nor established.

This paper shows that temporal data reference profiles expressed in terms of hot data streams, which are data reference sequences that frequently repeat, are quite stable; an encouraging result for memory optimization research. Most hot data streams belong to one of two categories—those that appear in multiple runs with their data elements referenced in the same order, and those with the same set of elements referenced in a different order—and this category membership is extremely stable. In addition, the fraction of hot data streams that belong to the first category is quite large.

1. Introduction

Computer systems continue to grow in complexity both at the architectural and micro-architectural level. At the architectural level, multiple levels of high speed cache memories result in variable data access times. At the micro-architectural level, speculation and out-of-order execution have a large impact on program performance. This increasing machine complexity makes it harder to statically anticipate and improve program performance.

To address this problem, many traditional code optimizations have evolved to incorporate profile information in the optimization analysis. For example, profiles are used to choose between several optimization plans based on the expected run-time benefit [1, 2, 13, 23], or to specify the optimization scope by directing procedure inlining [14] or scheduling [15, 20].

While code optimizations use profiles to complement static program analysis, most memory system optimizations that attempt to improve a program's data cache performance by changing the data layout, rely primarily on profiles [4, 6, 7, 9, 17, 26]. The success of many of these optimizations depends on access to accurate, fine-grain temporal data reference information. Since static code analysis cannot provide this level of detail, these optimizations rely on temporal data reference profiles. For example, both Calder et. al [4], and Chilimbi and Larus [6], use the temporal relationship graph (TRG) [12], which approximates a temporal data reference profile by maintaining weighted edges between data references that occur within a pre-selected reference distance.

This reliance on profiles makes the effectiveness of off-line optimizations, especially memory system optimizations, dependent on profile stability across multiple runs of the same program with different inputs. The stability of code profiles, such as procedure, basic block, edge, and branch profiles, has been demonstrated [11, 27]. On the other hand, the stability of data reference profiles, especially temporal data reference profiles which are necessary for cache-level optimizations, has neither been studied nor established. This is particularly unfortunate since data layout optimizations that target cache performance rely almost entirely on some form of temporal data reference profile.

The paper attempts to address this problem by investigating the stability of temporal data reference profiles. In recent research, Chilimbi describes a technique for efficiently computing hot data streams, which are sequences of data references that frequently repeat, from a program's data reference trace [8]. Since these hot data streams precisely capture the temporal profile information needed by cache-level optimizations, we use them to represent a program's temporal data reference profile. We "name" each of these hot data streams by their access signature, which is an ordered list of the load/store PCs that generate the hot data stream references. This naming permits comparison of hot data streams across different program runs, despite changes in the addresses of the data objects referenced.

Our experimental analysis of hot data stream profiles for several of the SPECint2000 benchmarks, boxsim, a state-of-the-art graphics application [5], and espresso, a heap-intensive boolean minimization program, indicate that these profiles are quite stable across different program runs with the relative contributions of different hot data streams to the overall profile varying by only small amounts. Most important (i.e., very "hot") hot data streams belong to one of two categories—those that occur in different program runs with the same order of data element references, and those that appear in different runs with the same set of data elements referenced in a different order—and this category membership is highly stable. In addition, for hot data stream sizes of up to twenty elements (sufficiently long for cache-level optimizations), the fraction of important hot data streams that occur in different runs with data elements referenced in the same order is quite large. These are encouraging results for off-line cache-level optimizations that depend on such temporal data reference profiles. Finally, for optimizations where the order of data stream references is important, combined profiles from multiple training runs outperform individual train profiles by a small amount. On the other hand, if the optimization does not rely on the order of data stream references, then any individual train profile performs as well as a combined train profile.

The rest of the paper is organized as follows. Section 2 surveys the different metrics used to compare profiles and motivates our choice of comparison function for temporal data reference profiles. Section 3 describes our representation of a program's temporal data reference profile and discusses how these profiles may be compared across different program runs despite changes in data object addresses. Section 4 presents experimental results that demonstrate the stability of temporal data reference profiles.

2. Comparing Profiles

The literature discusses several methods for comparing profiles [3, 10, 11, 18, 24, 27]. While some of the comparison functions described are abstract and can be used for comparing various types of profiles, they have been primarily applied to code profiles. Calder et al. compare branch profiles with a coverage function that they define as the percentage of branches from a program run that were also executed during a training run [3]. Fisher and Freudenberger combine branch direction predictability with branch density to compare branch profiles [11]. Wall uses a key matching and weight matching metric to compare the top n elements of two profiles [27]. Kistler and Franz treat an *n* item profile as a vector in *n*-dimensional space and define a similarity metric that computes a measure of the geometric angle between the two profile vectors [18]. They use this similarity metric in the context of a dynamic optimization system to decide when program behavior has changed sufficiently to warrant reoptimization. Savari and Young treat a frequency profile as a probability distribution and compute the relative entropy of the two probability distributions, which is a measure of the inefficiency of assuming that the distribution is the train profile when it actually is the test profile [24]. Feller uses an overlap percentage metric to compare value profiles [10]. The overlap for a profile event is the minimum of its percentage contribution to the test and train profile. The overlap percentage for two profiles is the sum of these overlaps for all profile events. For example using the data in Table 1, *Overlap*(p1, p2) = *min*(25, 74.6) + *min*(25, 24.9) + *min*(25, 0.5) + *min*(25, 0) = 50.4.

Table 1: Example profiles

Profile	Event 1	Event 2	Event 3	Event 4
P1	250	250	250	250
P2	1500	500	10	0
P3	300	150	10	0
P4	0	0	400	450
P5	400	500	600	0

We use the example profiles shown in Table 1, which are taken from [24], to motivate our choice of comparison function for computing the stability of temporal data reference profiles. We applied Calder et al.'s static and dynamic coverage metric, Kistler and Franz's similarity metric, Savari and Young's relative entropy metric, and Feller's overlap percentage metric to the example profiles. The results are shown in Table 2. All comparison metrics agree on the best train-test profile combination. Not surprisingly, static and dynamic coverage, which were devised to compare branch profiles, do a poor job of distinguishing between the profiles. The other three comparison functions mostly seem to be in agreement. Relative entropy and overlap percentage always agree on the relative ordering of the different train-test profile combinations. Similarity and relative entropy are fairly expensive to compute. On the other hand, overlap percentage is simple to compute and intuitively appears to be the most satisfactory. Hence, we use overlap percentage for comparing temporal data reference profiles.

3. Representing Temporal Data Reference Profiles

This section describes our representation of a program's temporal data reference profile. It discusses comparing data reference profiles across multiple program runs despite changes in the addresses of the data objects referenced.

3.1 Hot Data Streams

In recent research, Chilimbi describes a compact representation of a program's data reference behavior that permits precise temporal data reference profiles to be computed efficiently [8]. The process is summarized in Figure 1. He uses an incremental, linear-time compression algorithm called SEQUITUR [21, 22] to generate a context-free grammar from an input data reference trace. This context-free grammar produces a single string, which is the data refer-

Table 2: Comparing profiles using different metrics.

Test profile	Train profile	Static Coverage	Dynamic Coverage	Similarity	Relative Entropy	Overlap
P1	P2	**0.75**	**0.75**	0.64	25.43	50
	P3	**0.75**	**0.75**	0.69	24.85	52
	P4	0.5	0.5	0.71	48.33	50
	P5	**0.75**	**0.75**	**0.86**	**24.12**	**75**
P2	P1	**1**	**1**	0.64	1.15	50
	P3	**1**	**1**	**0.99**	**0.04**	**90**
	P4	0.33	0.01	0	98.31	0
	P5	**1**	**1**	0.62	0.97	52
P3	P1	**1**	**1**	0.69	0.95	52
	P2	**1**	**1**	**0.99**	**0.05**	**90**
	P4	0.33	0.02	0.02	96.47	2
	P5	**1**	**1**	0.68	0.74	62
P4	P1	**1**	**1**	**0.71**	**1**	**50**
	P2	0.5	0.42	0	55.36	0
	P3	0.5	0.42	0.02	54.36	2
	P5	0.5	0.42	0.45	52.38	40
P5	P1	**1**	**1**	**0.86**	**0.43**	**75**
	P2	**1**	**1**	0.62	2.28	52
	P3	**1**	**1**	0.68	1.35	62
	P4	0.33	0.4	0.45	58.66	40

ence trace. The grammar can be represented as a DAG, as shown in Figure 1. The DAG representation, called Whole Program Streams (WPS), can be efficiently analyzed (i.e., in linear time) to discover hot data streams, which are frequently occurring data reference sequences. The hot data stream detection algorithm accepts three parameters—the minimum hot data stream length, the maximum hot data stream length, and the threshold (product of data stream length and number of stream repetitions in the reference trace) above which a data stream is deemed "hot". The minimum hot data stream length is set at 2, and the maximum hot data stream length is set at 100, since studies of several programs yielded extremely few hot data streams of longer length [8]. The "heat" threshold is set such that the resulting hot data streams account for 90% of all data references (see [8] for details), and thus are representative of the program's data reference behavior. Finally, these hot data streams can be combined with the WPS representation to construct a Stream Flow Graph (SFG) as shown in Figure 1. The nodes of the SFG are hot data streams and weighted directed edges, <*src*,*dest*>, indicate the number of times an access to hot data stream *src* is followed by an access to hot data

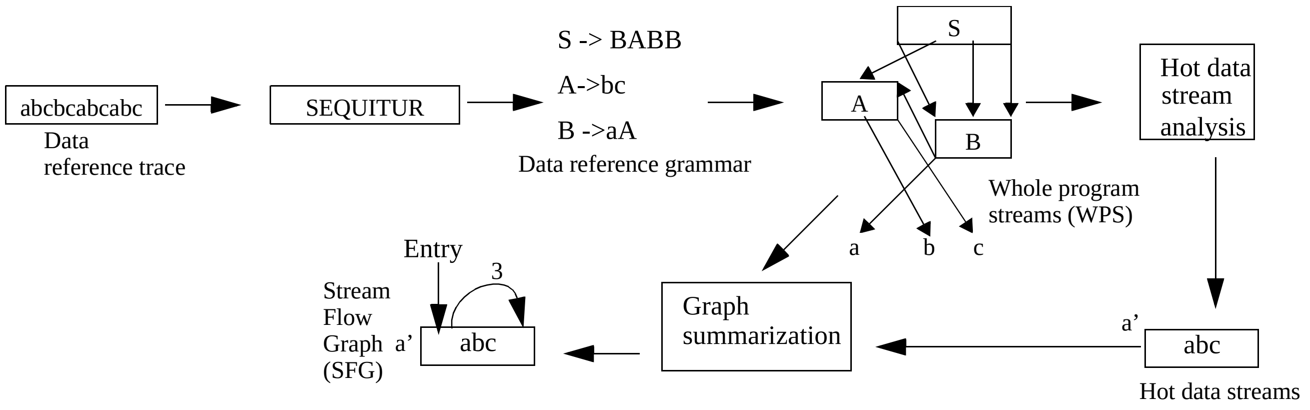

Figure 1. Temporal data reference representations and abstractions.

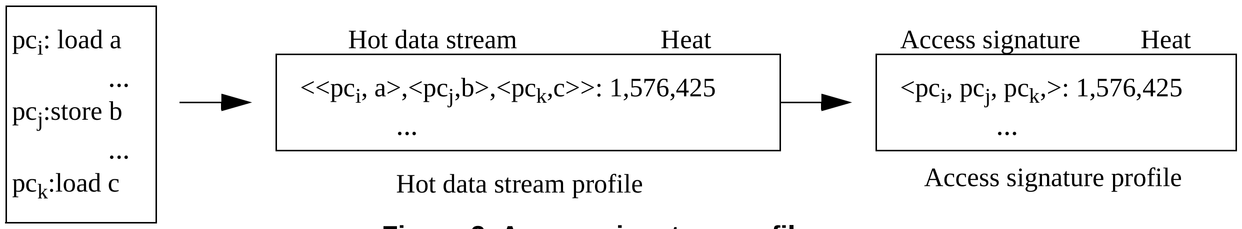

Figure 2. Access signature profile.

stream *dest*. The hot data streams and the SFG capture temporal relationships that are potentially more precise than the TRG since they are not determined by an arbitrarily selected temporal reference window size. In addition, the hot data streams represent a compact summary of a program's temporal data reference relationships since these account for 90% of all data references. For these reasons, we select a program's hot data stream profile, which consists of a list of hot data streams along with their respective "heat" values, to represent its temporal data reference profile. Each element of a hot data stream is a <PC, data object> pair, where the PC corresponds to the instruction that references the data object. Global data objects are represented by their address, whereas heap objects are represented by a <*start_address*, *allocation_time*> pair, where *start_address* corresponds to the beginning of the chunk of heap memory allocated for the object, and *allocation_time* is the value of a global counter that is incremented after each heap allocation (Stack references are filtered out prior to constructing the context-free grammar from the trace since these typically have good reference locality and are rarely the focus of data locality optimizations).

3.2 Naming Hot Data Streams

The hot data stream profile described in the previous section cannot be compared across multiple program runs since data object addresses change from one run to another. One possible solution to this problem and the approach taken in this paper is to compare the load/store instructions that generate the hot data stream references instead. In other words, for comparison purposes, we represent a hot data stream by its access signature, which is an ordered list of instruction PCs that reference the hot data stream objects (see Figure 2). While this naming scheme may introduce aliases, where the same access signature corresponds to multiple hot data streams, any off-line data locality optimization cannot expect access to more precise hot data stream information.

4. Experimental Evaluation

This section investigates the stability of temporal data reference profiles using several benchmarks that were run with different input data sets. Further experiments explore the differences between the best and worst training profiles and consider the effect of combining different training profiles to synthesize a single representative train profile. Finally, we present preliminary data that examines the effectiveness of train profiles in driving a hot data stream-based prefetching optimization.

4.1 Experimental Methodology

The programs used in this study include several of the SPECint2000 benchmarks, boxsim, a state-of-the-art graphics application [5], and espresso, a heap-intensive boolean minimization program. The benchmarks (and the standard libraries) were instrumented with Microsoft's Vulcan tool to produce a data address trace along with information about heap allocations. Vulcan is an executable instrumentation system similar to ATOM and EEL [19, 25]. The benchmarks were run with a minimum of three different input data sets as described in Table 3. 253.perlbmk's train inputs were scaled to generate a smaller data reference trace. Our choice of benchmarks was influenced by the availability of multiple input data sets that generated data reference traces on the order of a few gigabytes. Stack references were filtered out and do not appear in the trace. The heap allocation information was processed to build a map of heap objects. A heap object is a <*start_address*, *allocation_time*> pair as described earlier, where *allocation_time* is a global counter that is incremented after each allocation. We used this naming scheme to achieve maximum discrimination between heap objects. Heap addresses in the trace were replaced by their corresponding heap object name. The abstracted data reference trace was fed to SEQUITUR, which produced a context-free grammar. The DAG representation of this grammar, called Whole Program Streams (WPS), was analyzed to identify hot data streams (see Figure 1). For each input data set, eleven hot data stream profiles were collected. These hot data stream profiles differed in the maximum hot data stream size permitted (minimum size was set to 2 in all cases), which was varied from 5, 10, 20,..., 100. The hot data stream profiles were computed such that in each case the hot data stream references represented 90% of all program data references. Each of these hot data streams, which were represented by their load/store PC access signature, were written to a file along with their computed "heat" (product of number of stream data references and stream repetition frequency). These hot data stream access signature profiles were used to investigate the stability of the program's temporal data reference profiles.

4.2 Comparing Temporal Data Reference Profiles

Figure 3 illustrates using the overlap percentage metric to

Table 3: Benchmarks and input data sets.

Benchmark	Description	Inputs
175.vpr	Performs placement and routing in FPGAs (SPEC 2000)	Input 1: Place and route from SPEC test input Input 2: Place and route from modified SPEC test input Input 3: Place and route from modified SPEC test input
186.crafty	Chess playing program (SPEC 2000)	Input 1: game 1 from SPEC test input Input 2: game 2 from SPEC test input Input 3: game 3 from SPEC test input
252.eon	Probabilistic ray tracer (SPEC 2000)	Input 1: chair.cook from SPEC test input Input 2: chair.kajiya from SPEC test input Input 3: chair.rushmeier from SPEC test input Input 4: chair.cook from SPEC train input
253.perlbmk	Interpreter for the Perl scripting language (SPEC 2000)	Input 1: modified perfect from SPEC train input Input 2: modified scrabbl from SPEC train input Input 3: modified diffmail from SPEC train input
boxedsim	Event-driven simulation of spheres bouncing in a box [5]	Input 1: 2 balls simulated for 100 msec Input 2: 4 balls simulated for 300 msec Input 3: 7 balls simulated for 500 msec
espresso	Performs boolean function minimization	Input 1: x2dn Input 2: opa Input 3: ti

compare hot data stream profiles through their access signatures. As discussed, eleven different temporal reference profiles (corresponding to increasing limits on the maximum size of hot data streams) were gathered for each benchmark input data set. The overlap percentage for each data point reported in the graph shown in Figure 3 represents the average of all possible train-test permutations for the different input data sets, excluding using the same data set as the test and train profile. The term *exact overlap* is used to indicate that access signatures from two different profiles were considered to represent the same profiled event only if they were identical in terms of both the load/store PCs and the order of these PCs. This metric is most relevant to optimizations such as prefetching, for which the order of data stream references is important. With this comparison metric, boxsim's profiles are the most stable and 252.eon the least. Not surprisingly, the graph shows that as the maximum hot data stream size is increased, the exact overlap between different profiles decreases as longer streams are less likely to have access signatures that match exactly. For certain programs, such as 175.vpr and 252.eon, the dropoff is fairly pronounced, whereas for others it is more gradual. Most of the dropoff occurs as the maximum stream length is increased from 5 to 40, with little additional dropoff occurring as the stream length is increased from 40 to 100. Closer examination of the profile data revealed that this is because longer hot data streams do not contribute substantially to the overall profile "heat".

We wanted to estimate the impact of access signature load/store PC order on matching hot data streams across different program runs. To do this, we ignored access signature load/store PC order and treated access signatures as sets for matching purposes. We called this *set overlap* and the results are shown in Figure 4. This metric is most relevant to data layout optimizations such as clustering, that do not rely on the order of data stream element references. The overlap percentage metric is affected if high frequency events do not appear in both the test and train profiles or if the relative weight of events is significantly different in the test and train profile. Figure 4 confirms that the relative contribution of different hot data streams to the overall profile is extremely stable (especially for hot data streams of length 2 to 20) across the different profiles. In addition, comparison with Figure 3 indicates that most important hot data streams match (for e.g., see 252.eon) if the access signature load/store PC order is ignored. Further, the decrease in overlap as longer hot data streams are included in the temporal profiles (maximum stream size is increased from 5 to 40) is much less significant when using *set overlap* as the comparison metric. With this comparison metric, 186.crafty and boxsim

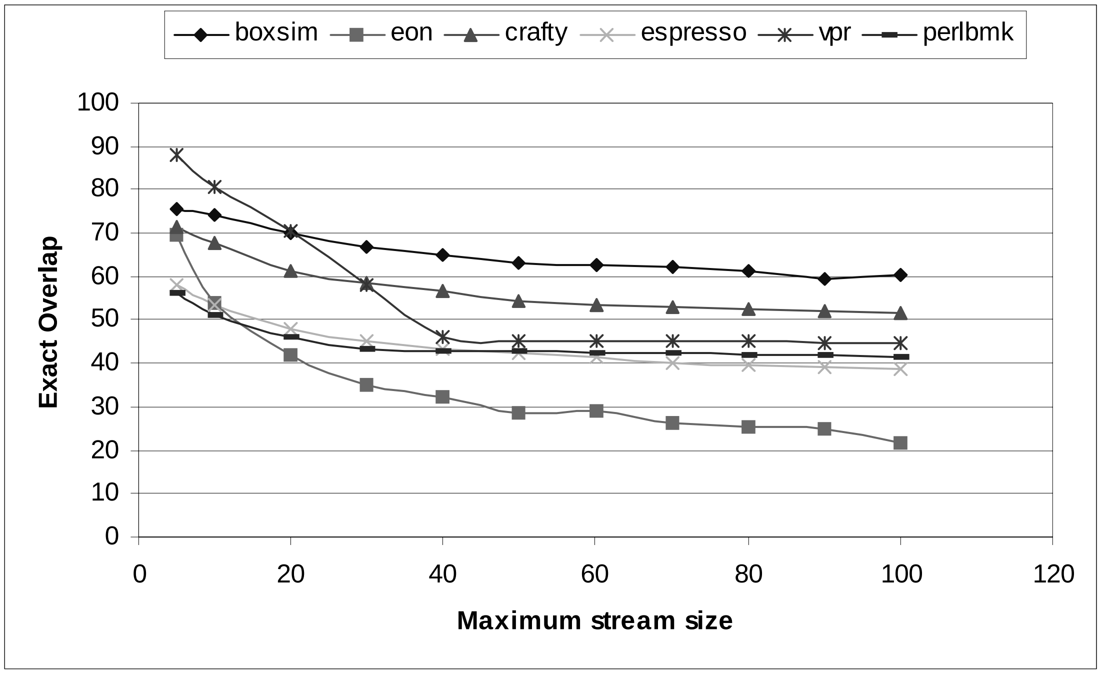

Figure 3. Exact overlap comparison of hot data stream profiles.

have the most stable profiles and espresso the least. In addition, the difference between the most and least stable profiles is much less pronounced than the corresponding difference in Figure 3.

From Figures 3 and 4 it appears that a large fraction of the most important hot data streams have their data elements referenced in the same order across different program runs (especially for streams of length 2 to 20), while the rest have their elements referenced in a different order. Hence, it appears reasonable to investigate a comparison metric where access signatures are first matched exactly and set matching is used for the remaining unmatched access signatures. We call this *hybrid overlap* and the results are shown in Figure 5. The impact of hot data stream size on profile overlap is smaller than in the case of exact and set overlap. With this comparison metric, 175.vpr has the most stable profile, and espresso the least, just as with the set overlap metric. However, boxsim's profile appears relatively less stable than it did with the set overlap metric.

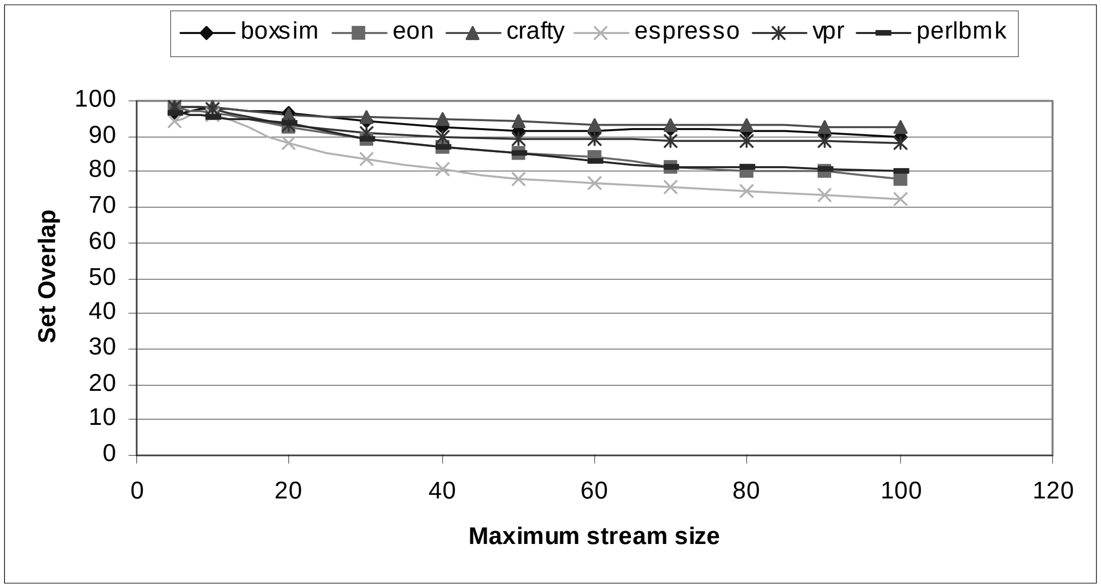

Figure 4. Set overlap comparison of hot data stream profiles.

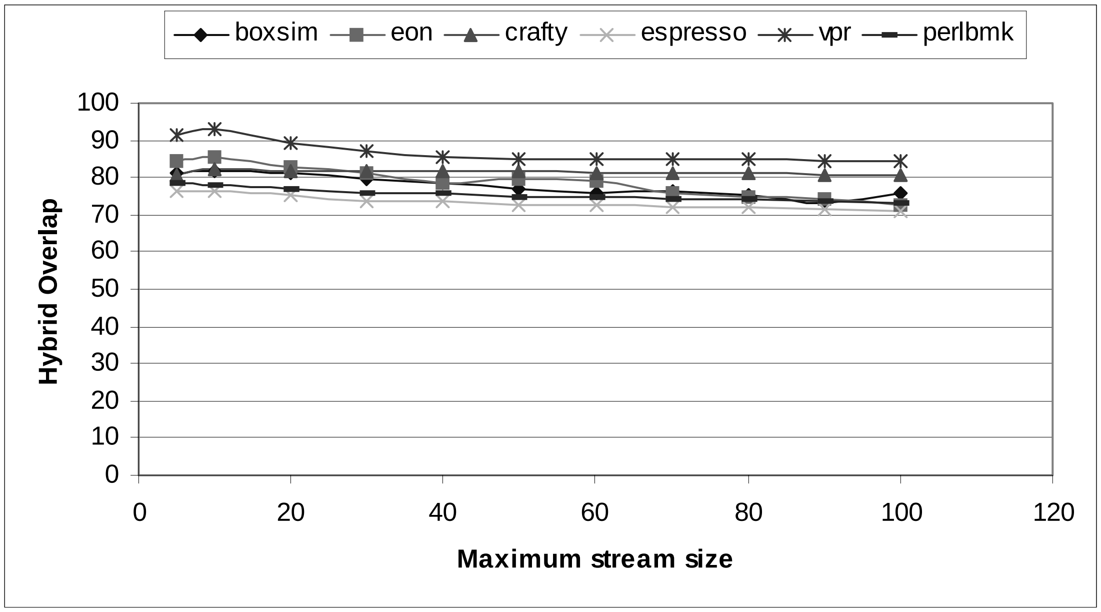

Figure 5. Hybrid overlap comparison of hot data stream profiles.

4.3 Combining Temporal Data Reference Profiles

The next set of experiments examine the effect of combining multiple train profiles and evaluating the resulting synthesized profile. Fisher and Freudenberger tried three different strategies—unscaled, scaled, and polling—for combining branch profiles [11]. Scaled normalizes the execution profiles to give each data set equal total weight. Unscaled simply adds the execution counts of the same event in different profiles, giving more weight to longer running profiles. Polling, which gives one vote on branch direction to each profile, is not applicable to temporal data reference profiles. They reported that polling performed poorly and that scaled and unscaled were on average indistinguishably close. We used both the scaled and unscaled strategies to combine our temporal data reference profiles (we used exact matching on access signatures to generate the combined profile), but report results only for scaled since it was consistently better that unscaled, albeit by a very small amount (1–3% on average)

For each of the three different comparison strategies—exact, set, and hybrid—we report results for temporal data reference profiles where the hot data streams were limited to a maximum size of twenty data elements. We compute six data points for each benchmark. The first three data points measure the average, best, and worst overlap between individual train profiles and a test profile, excluding using the same profile as train and test. The next three data points measure the average, best, and worst overlap between scaled train profiles and individual test profiles. Concretely, if a benchmark has four profiles: p1, p2, p3, p4. Then scaled

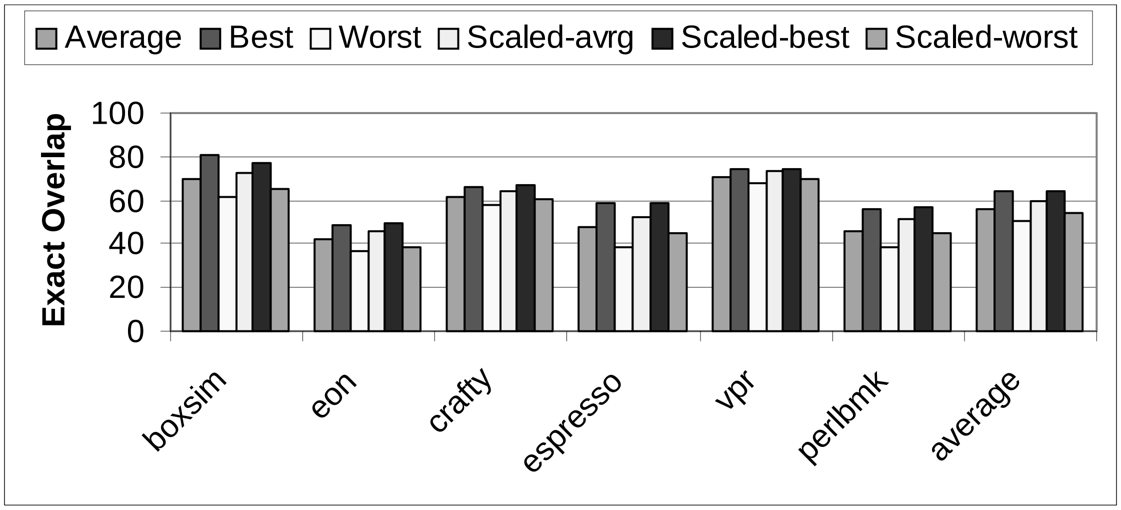

Figure 6. Exact overlap profile comparison (max. hot data stream size = 20).

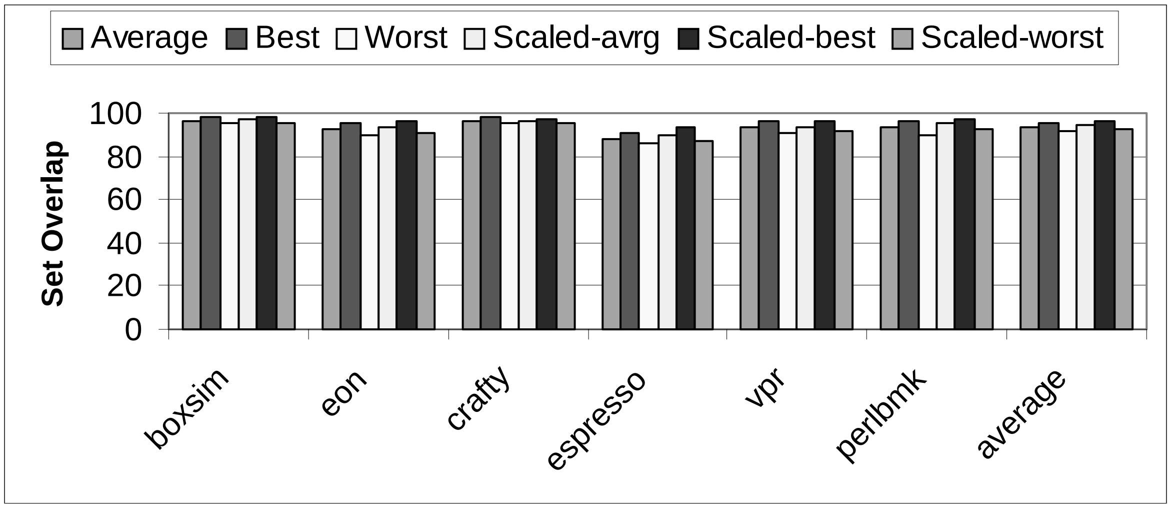

Figure 7. Set overlap profile comparison (max. hot data stream size = 20).

profiles p1p2p3, p1p2p4, p1p3p4, p2p3p4 were computed and used as train profiles for p4, p3, p2, and p1 respectively.

The results for exact overlap comparison are shown in Figure 6. Scaled profiles are on average slightly better than individual profiles by around 4%. In addition, the difference between the best and worst individual train profiles while not always insignificant (18–20% for espresso, boxsim, 253.perlbmk), is much smaller than the difference reported for branch profiles by Fisher and Freudenberger [11]. Using scaled train profiles slightly reduces this variation. The average profile overlap is around 60% when using scaled train profiles. This suggests that a large fraction of important hot data streams repeat across runs with the same data element reference order. Closer examination of the profile data indicates that these hot data streams that exactly match are highly stable, with roughly 95% of these streams identical across all the different train profiles.

Figure 7 examines the effect of using set overlap to compare profiles. The graph indicates consistently high overlaps (over 90% in all cases), irrespective of whether individual train profiles or scaled train profiles are used. In addition, the differences between the best and worst profiles are negligible. This suggests that for optimizations where the order of elements in a data stream is not important, practically any train profile can be used to guide the optimization. Comparing this graph with Figure 6 suggests that almost all the important hot data streams match across different program runs if the data element reference order within a stream is ignored.

Finally, Figure 8 shows a similarly plotted graph with hybrid overlap as the comparison metric. Again, there is little difference on average between using individual train profiles or scaled train profiles. In addition, the differences between the best and worst train profiles are around 10% on average, which is a smaller variation than in the case of the exact overlap metric but a larger variation than observed

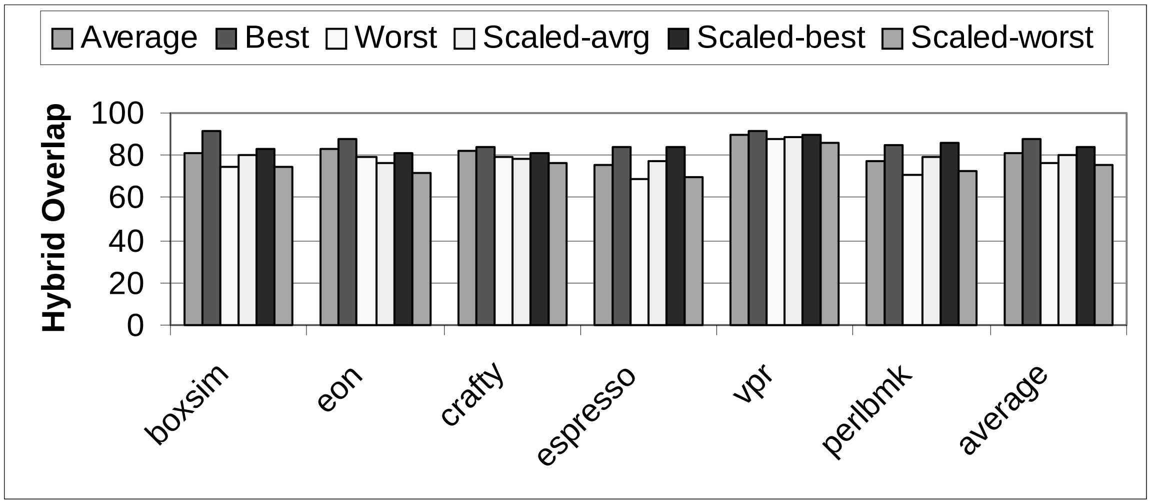

Figure 8. Hybrid overlap profile comparison (max. hot data stream size = 20).

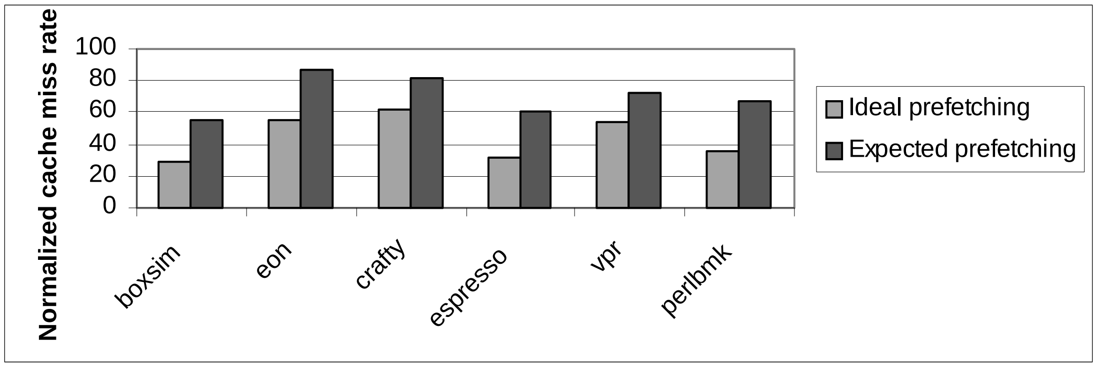

Figure 9. Impact on prefetching optimization.

with the set overlap metric.

4.4 Impact on Optimization

The previous experiments compared and analyzed hot data stream profiles from different runs of a program. While the results appear encouraging, especially for hot data streams that are smaller than twenty elements, the true test is to explore the effectiveness of using a train profile to drive an optimization.

We attempt to address this question by computing the potential impact on cache miss rate of a prefetching scheme based on hot data streams (we ignore misses that occur when the data is prefetched, since these do not affect access latency if the prefetch is scheduled sufficiently in advance). While the details of this prefetching scheme is beyond the scope of this paper, we briefly sketch how it works. It uses prefetch arrays [16], and populates these arrays with pointers to hot data stream elements at runtime. All the load/store PCs that are associated with any hot data stream are instrumented, and load/store PCs that correspond to the start of any hot data stream have a prefetch array associated with them. When the program executes any load/store instruction that corresponds to the start of any hot data stream, the associated set of prefetch arrays (possibly more than one since multiple hot data streams may share the same start PC) are examined and if none of them matches the current reference, a fresh array is allocated (or an existing array is reused). Subsequent references up to any load/store PC that ends a hot data stream are recorded in this array. If on the other hand a unique match is found, the remaining array elements are prefetched. If multiple prefetch arrays match, subsequent references are used to distinguish between the arrays until a unique match is found, or a predetermined limit is exceeded. Saturating counters are used as a confidence mechanism to avoid prefetching cold addresses that alias to the same set of load/store PCs.

We simulated the effect of this prefetching scheme on cache miss rate and Figure 9 shows preliminary results for an 8K fully-associative cache with 64 byte blocks (we scaled the cache size since we did not use the SPEC benchmark's ref inputs). All miss rates are normalized with respect to the base cache miss rate. The Ideal prefetching bar was computed using the test input to generate its train profile. The Expected prefetching bar was computed using a combined scaled profile that did not include the test input profile. Since the optimization depends on the hot data stream element reference order to populate and use the prefetch arrays, only those hot data streams that exactly matched across all the individual train profiles used to synthesize the combined profile, had their load/store PCs instrumented. Most of the difference in results is attributable to this decision, since the Ideal prefetching scenario was not limited in this manner and could prefetch all hot data streams. Nevertheless, using the combined train profile still produced significant cache miss rate improvements.

5. Conclusions

This paper investigates the stability of temporal data reference profiles. We use hot data streams, which are sequences of data references that frequently repeat, to represent a program's temporal data reference profile. We "name" each of these hot data streams by their access signature, which is an ordered list of load/store PCs that generate the hot data stream references. This permits comparison of hot data streams across different program runs, despite changes in the addresses of the data objects referenced.

Experimental analysis of hot data stream profiles for several benchmarks indicate that these profiles are fairly stable across different program runs with the relative contributions of different hot data streams to the overall profile varying by only small amounts. Most important hot data streams belong to one of two categories—those that appear in different program runs with the same order of data element references, and those that appear in different runs with the same set of data elements referenced in a different order—and this category membership is highly predictable. In addition, for hot data stream sizes of up to twenty elements (sufficiently long for cache-level optimizations), the fraction of important hot data streams that appear in different runs with the same order of data element references is fairly large. These are encouraging results for off-line cache-level optimizations that depend on such temporal data reference profiles. Finally, for optimizations where the order of data stream references is important, combined train profiles should outperform individual train profiles by a small amount. On the other hand, for optimizations that do not rely on data stream reference order, any individual train profile should perform as well as a combined train profile.

6. REFERENCES

[1] R. Bodik, R. Gupta, and M. L. Soffa. "Complete Removal of Redundant Expressions." In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[2] R. Bodik, R. Gupta, and M. L. Soffa. "Load-Reuse Analysis: Design and Evaluation." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[3] B. Calder, D. Grunwald, and A. Srivastava. "The predictability of branches in libraries." In *Digital WRL Technical Report 95/6*, Oct. 1995.

[4] B. Calder, C. Krintz, S. John, and T. Austin. "Cache-conscious data placement." In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139-149, Oct. 1998.

[5] S. Chenney. "Controllable and scalable simulation for animation." *PhD. thesis, University of California at Berkeley*, 2000.

[6] T. M. Chilimbi, and J. R. Larus. "Using generational garbage collection to implement cache-conscious data placement." In *Proceedings of the 1998 International Symposium on Memory Management*, Oct. 1998.

[7] T. M. Chilimbi, B. Davidson, and J. R. Larus. "Cache-conscious structure definition." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[8] T. M. Chilimbi. "Efficient representations and abstractions for quantifying and exploiting data reference locality." To appear in *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, June 2001.

[9] C. Ding and K Kennedy. "Improving cache performance in dynamic applications through data and computation reorganization at run time." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 229-241, May 1999.

[10] P. T. Feller. "Value profiling for instructions and memory locations." *M.S. thesis CS98-581, University of California at San Diego*, April 1998.

[11] J. A. Fisher and S. M. Freudenberger. "Predicting conditional branch directions from previous runs of a program." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 85-95, Oct. 1992.

[12] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder "Procedure placement using temporal ordering information." In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[13] R. Gupta, D. A. Berson, and J. Z. Fang. "Path profile guided partial dead code elimination using predication." In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 1997.

[14] A. M. Holler "Optimization for a superscalar out-of-order machine." In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[15] W. Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation." In *Journal of Supercomputing, Kluwer Academic Publishers*, pages 229-248, 1993.

[16] M. Karlsson, F. Dahlgren, and P. Stenstrom. "A prefetching technique for irregular accesses to linked data structures." In *Symposium on High-Performance Computer Architecture, Jan.* 2000.

[17] T. Kistler and M. Franz. "Automated record layout for dynamic data structures." In *Department of Information and Computer Science, University of California at Irvine, Technical Report 98-22*, May 1998.

[18] T. Kistler and M. Franz. "Computing the similarity of profiling data." In *Workshop on Profile and Feedback-Directed Optimization*, Oct. 1998.

[19] J. R. Larus and E. Schnarr. "EEL: Machine-Independent Executable Editing." In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 291-300, 1995.

[20] S. A. Mahlke, D. C. Lin, W. Y. Chen, and R. E. Hank "Effective compiler support for predicated execution using the hyperblock." In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, 1992.

[21] C. G. Nevill-Manning and I. H. Witten. "Compression and explanation using hierarchical grammars." In *The Computer Journal*, vol. 40, pages 103-116, 1997.

[22] C. G. Nevill-Manning and I. H. Witten. "Linear-time, incremental hierarchy inference for compression." In *Proceedings of the Data Compression Conference (DCC'97)*, pages 3-11, 1997.

[23] G. Ramalingam. "Data flow frequency analysis." In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, May 1996.

[24] S. Savari and C. Young. "Comparing and combining profiles." In *Workshop on Profile and Feedback-Directed Optimization*, Nov. 1999.

[25] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools." In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196-205, May 1994.

[26] D. Truong, F. Bodin, and A. Seznec. "Improving cache behavior of dynamically allocated data structures." In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 1998.

[27] D. Wall. "Predicting program behavior using real or estimated profiles." In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 59-70, June 1991.