*Appears in ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI), June 2001.*

# Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality

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

## ABSTRACT

With the growing processor-memory performance gap, understanding and optimizing a program's reference locality, and consequently, its cache performance, is becoming increasingly important. Unfortunately, current reference locality optimizations rely on heuristics and are fairly ad-hoc. In addition, while optimization technology for improving instruction cache performance is fairly mature (though heuristic-based), data cache optimizations are still at an early stage. We believe the primary reason for this imbalance is the lack of a suitable representation of a program's dynamic data reference behavior and a quantitative basis for understanding this behavior.

We address these issues by proposing a quantitative basis for understanding and optimizing reference locality, and by describing efficient data reference representations and an exploitable locality abstraction that support this framework. Our data reference representations (Whole Program Streams and Stream Flow Graphs) are compact—two to four orders of magnitude smaller than the program's data reference trace—and permit efficient analysis—on the order of seconds to a few minutes—even for complex applications. These representations can be used to efficiently compute our exploitable locality abstraction (hot data streams). We demonstrate that these representations and our hot data stream abstraction are useful for quantifying and exploiting data reference locality. We applied our framework to several SPECint 2000 benchmarks, a graphics program, and a commercial Microsoft database application. The results suggest significant opportunity for hot data stream-based locality optimizations.

## 1. INTRODUCTION

In the never-ending quest for greater performance, machine architectures continue to grow more complex. Achieving near-peak performance on these modern machines places an immense burden on software. Machine complexity makes statically anticipating and improving a program's performance increasingly difficult. A more pragmatic approach is to observe a program's dynamic execution behavior and use this data to optimize the program.

Program paths—consecutively executed basic block sequences—can offer insight into a program's dynamic control flow behavior and have been successfully used in compilers for program optimizations [1, 2, 10, 12]. However, though paths provide a basis for understanding a program's dynamic control flow, they supply an incomplete picture of a program's dynamic behavior as they do not capture a program's data accesses. With the growing processor-memory performance gap, understanding and optimizing a program's data accesses is becoming increasingly important. Recent research has attempted to address this problem by optimizing a program's data layout for caches [3, 4, 5, 6, 9, 15, 24]. Since a cache miss can be up to two orders of magnitude slower than a cache hit, these techniques promise significant performance improvements.

Unfortunately, while aggregate data access frequency information may be sufficient for page-level data layout [22], cache-level layout requires fine-grain temporal information about sequences of data references [3, 4]. This is because cache blocks are much smaller than pages resulting in zero-tolerance for incorrect data co-location. This temporal information can be obtained by instrumenting every program load and store, but analyzing the resulting gigabytes of trace data is impractical. In addition, the need for data reference sequence information makes it hard to use statistical sampling of loads and stores to reduce the amount of data generated.

We believe that the lack of suitable data reference representations and abstractions are the primary obstacle to analysis and optimization of a program's dynamic data reference behavior. Unlike the control flow graph and program paths, which are a convenient and compact abstraction of a program's control flow, no corresponding analogue exists for a program's data accesses. Compounding this problem is the fact that current reference locality optimizations, both for code and data, are fairly ad-hoc and rely primarily on heuristics.

This paper attempts to address these issues by proposing a quantitative basis for understanding and improving reference locality, and by describing efficient data reference representations and an exploitable locality abstraction that support this framework. We show that a hierarchical compression algorithm called SEQUITUR [20] can be used along with simple data abstractions to construct a compact representation of a program's dynamic data reference behavior. The resulting structures, called *Whole Program Streams* (WPS), are small and permit analyses without decompression. The paper shows that these WPSs can be used to efficiently compute our exploitable locality abstraction—*hot data streams*—which are frequently repeated sequences of consecutive data accesses, and are analogous to hot program paths. While WPSs capture a program's complete dynamic data reference behavior, this level of detail is often unnecessary for many optimizations. To address this, we show that hot data streams can be combined with the SEQUITUR compression technique to produce a series of representations with increasing compactness, but lower precision. These representations, which are efficient to analyze, can be up to four orders of magnitude smaller than the original data reference trace, yet still retain enough information about data reference

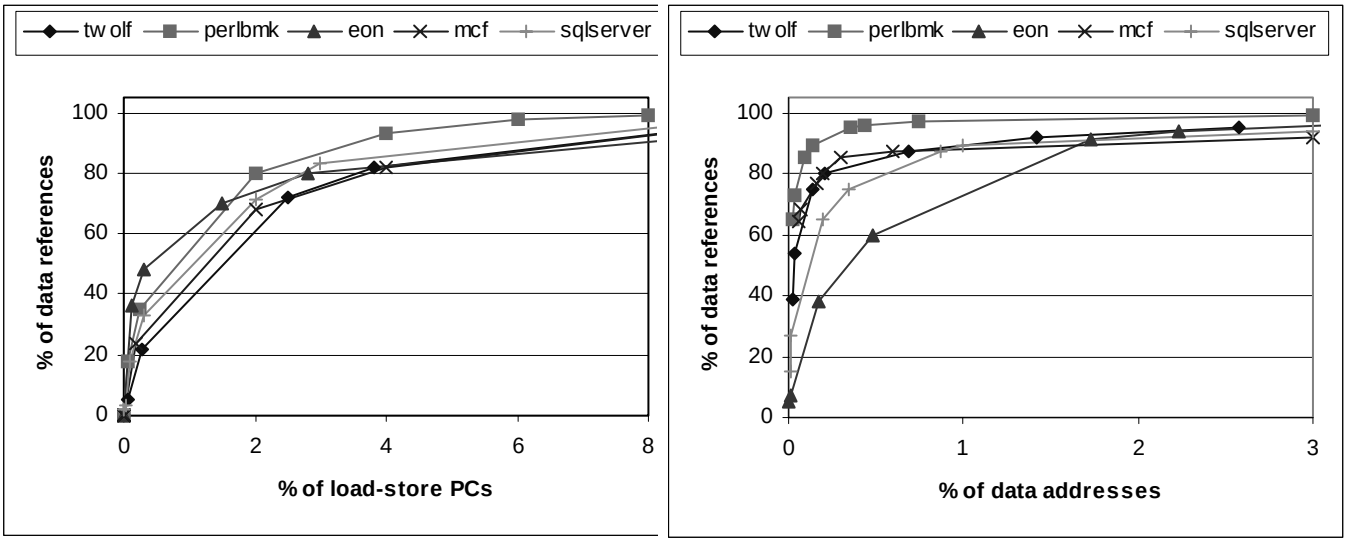


**Figure 1. Program data reference skew in terms of load-store PCs and data addresses.**

behavior for optimization purposes.

The paper evaluates these representations and the hot data stream abstraction by applying them to quantify and exploit data reference locality in several SPECint 2000 benchmarks, boxsim, a graphics application, and Microsoft SQL Server, a commercial database. It shows that the hot data stream abstraction can be used to compute inherent exploitable locality metrics that are indicative of a program's algorithmic spatial and temporal reference locality, and a realized locality metric that evaluates the effectiveness of the program's mapping of data objects to memory addresses at exploiting the inherent reference locality. Since these locality metrics can be efficiently computed at the granularity of individual hot data streams, they allow locality optimizations to focus solely on streams that show the greatest potential benefit. In addition to guiding locality optimizations, hot data streams, which contain data objects that are frequently accessed together, can help drive locality optimizations, such as clustering and prefetching. The paper demonstrates that such hot data stream-based locality optimizations have the potential to significantly improve cache performance.

The rest of the paper is organized as follows. Section 2 presents a quantitative basis for studying reference locality and defines an exploitable locality abstraction—hot data streams. It discusses several exploitable locality metrics that can be computed in terms of this hot data stream abstraction. Section 3 describes the construction of Whole Program Streams and other more compact, though less precise, representations of a program's dynamic data reference behavior. It also discusses efficiently computing the hot data stream abstraction from these representations. Section 4 outlines the role of our data reference representations, hot data streams abstraction, and locality metrics in exploiting reference locality. Section 5 presents experimental results for several SPECint 2000 benchmarks, boxsim, a graphics application, and Microsoft's SQL database. Finally, section 6 briefly discusses related work.

# 2. QUANTIFYING LOCALITY

Programs appear to exhibit a property termed *locality of reference* [16]. Informally, this locality of reference principle says that the most recently used data is likely to be accessed again in the near future. This phenomenon is widely observed and a rule of thumb, often called the 90/10 rule, states that a program spends 90% of its execution time in only 10% of the code. While cache miss rate provides an indirect measure of a program's reference locality, rarely do we see a program's reference locality directly quantified independent of the mapping of data objects to memory addresses, and much less this quantification used as a basis for optimization. This section discusses how data reference locality can be computed, shows that data references exhibit locality, describes a new concept that we call exploitable locality, and defines metrics for quantifying exploitable locality.

## 2.1 Data Reference Locality

A data reference sequence exhibits locality if the reference distribution is non-uniform in terms of either the data addresses/objects being accessed or the load/store instructions responsible for the accesses. A consequence of data reference locality is that some addresses/objects or load/store instructions are accessed much more frequently than others making these highly referenced entities attractive optimization targets. One possible quantifiable definition for data reference locality in the spirit of the 90/10 rule is the smallest percentage of accessed data addresses/program instructions that are responsible for 90% of the data references. Thus, good reference locality implies a large skew in the reference distribution. The 90/10 rule predicts a reference locality of 10% and a uniform reference distribution possesses a reference locality of 50%. Figure 1 indicates the fraction of program data references (90%) that are attributable to the most frequently referenced data addresses (1–2% of accessed data addresses), and to the hottest loads and stores (4–8% of load/store accesses), for several SPECint 2000 benchmarks, and Microsoft SQL Server, a commercial database.[1] The graphs indicate that programs possess significant data reference locality, even more than the 90/10 rule predicts. Interestingly, they also indicate that data addresses often exhibit greater access skew than a program's load/store PCs, making them

[1] To avoid biasing the data in favor of addresses, stack references were excluded. In addition, we modified the programs to prevent reuse of heap addresses by removing all system calls that free memory.

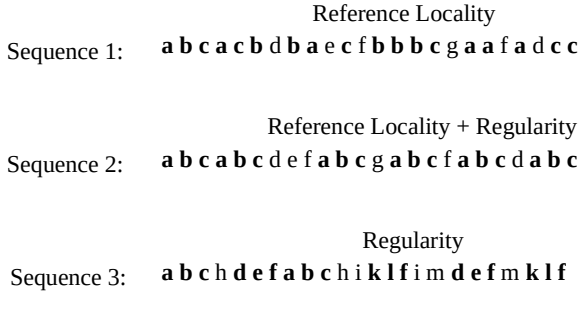


**Figure 2. Data reference sequence characteristics.**

good optimization targets. Thus, in the rest of this paper we focus on reference locality expressed in terms of data addresses/objects.

## 2.2 Data Reference Regularity

From an optimization standpoint, locality is only interesting if it can be exploited to improve program performance. Processor caches, which are the ubiquitous hardware solution to exploiting reference locality, form the first line of attack. We believe that the key property that makes data reference locality exploitable by software optimization (over and above caches) is *regularity.*

For a data reference subsequence to exhibit regularity, it must contain a minimum of two references and must appear at least twice (non-overlapping) within a data reference sequence. In the first sequence in Figure 2, the only regular subsequence is *bc*, and in the second sequence, it is *ab, bc*, and *abc.* We quantify regularity in terms of four associated concepts called *regularity magnitude*, *regularity frequency*, *spatial regularity*, and *temporal regularity.* To make the definitions concrete, we apply them to quantify the regularity of subsequence *abc* in the second sequence shown in Figure 2. Regularity magnitude is defined as the product of the number of references in the regular data subsequence (not necessarily to unique data objects) and the non-overlapping repetition frequency of that subsequence in the data reference sequence. The regularity magnitude of subsequence *abc* is 18. Regularity frequency is defined as the number of non-overlapping occurrences of the regular reference subsequence within the data reference sequence. The regularity frequency of *abc* is 6. Spatial regularity is defined as the number of references (not necessarily to unique data objects) in the regular subsequence. The spatial regularity of *abc* is 3. Temporal regularity is defined as the average number of references between successive non-overlapping occurrences of the data reference subsequence that exhibits regularity. The temporal regularity of *abc* is 1.2.

## 2.3 Exploitable Data Reference Locality

While hardware caches are capable of exploiting reference locality independent of regularity, software locality optimizations are heavily dependent on regularity. Figure 2 illustrates this with three data reference sequences. The first two data reference sequences have exactly the same reference locality or reference skew. However, the first data reference sequence is less predictable than the second sequence. The second data reference sequence's regularity makes it a much more attractive optimization target. For example, a prefetching optimization could exploit the regularity of subsequence *abc* to initiate prefetches for data objects *b*, and *c*, once object *a* has been observed. A clustering optimization could attempt to allocate data objects *a*, *b*, and *c*, together, in the same cache block or page.

If a data reference sequence exhibits regularity then do we need it to exhibit reference locality or reference skew as well? Or is regularity all we need? The third data reference sequence in Figure 2 attempts to address these questions. While it is certainly possible for software optimizations to take advantage of the sequence's regularity, this reference sequence does not appear as promising as the second sequence since any optimization must track and analyze a larger number of data objects than the second sequence to get the same performance benefit. In any case, since a program's data reference sequence exhibits locality (see Figure 1), such sequences are extremely unlikely to occur in practice. Thus, from an optimization perspective, data reference sequences that possess both reference locality and regularity are ideal optimization targets. Hence, we refer to the combination of reference locality and regularity as *exploitable locality,* and it is the focus of the rest of this paper.

### *2.3.1 Exploitable Data Reference Locality Abstraction*

Since exploitable locality forms the basis of our optimization framework, we define an exploitable locality abstraction called *hot data streams*. A data stream is a data reference subsequence that exhibits regularity. If the regularity magnitude of a data stream exceeds a predetermined threshold, then it is termed a *hot data stream.* In the spirit of the 90/10 rule, this heat threshold above which a data stream is labelled as "hot" is set such that the collection of hot data streams together cover 90% of all program data references (see Section 3 for details). Note that with this heat threshold, the 90/10 rule ensures that only a small fraction of accessed data addresses/objects participate in hot data streams. We confirm this in Section 5 and additionally demonstrate that this heat threshold generates a small number of hot data streams, making them attractive optimization targets. Hot data streams are by definition regular, and since they are extracted from program data reference traces, which are known to exhibit reference locality (see Figure 1), they qualify as exploitable locality units. These hot data streams, which are analogous to hot program paths, can be efficiently extracted from large program data reference traces, as we demonstrate in Section 3. In addition, the collection of hot data streams taken together are representative of a program's data reference behavior since they cover 90% of all program data references.

## 2.4 Quantifying Exploitable Locality

This section presents two sets of exploitable locality metrics that are useful from an optimization perspective. These metrics are defined in terms of hot data streams, our exploitable locality abstraction. The first set of metrics quantify the *inherent* exploitable data reference locality of an algorithm. This information may be useful to a programmer and could suggest algorithmic modifications. The second kind of exploitable locality metric quantifies the impact of a particular mapping of data objects to memory addresses, and the interaction of this mapping with hardware caches. This *realized* locality metric quantifies how effectively hardware caches exploit a program's inherent reference locality and provides a measure of the remaining opportunity for software-based locality optimizations. Together, these locality metrics define limits on the performance improvements due to data reference locality optimizations. In addition, these locality metrics, which can be efficiently computed from hot data streams, can help guide data locality optimizations as described in Section 4.

### *2.4.1 Inherent Exploitable Locality Metrics*

Spatial and temporal locality are well-known concepts used to express locality [13]. However, these are unsuitable as inherent exploitable locality metrics for three reasons. First, the standard definitions are qualitative and do not indicate how these metrics

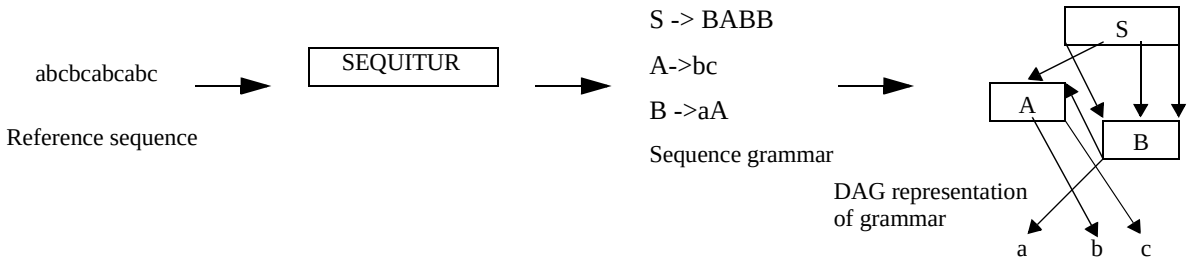


**Figure 3. Sequence compression using SEQUITUR.**

may be computed. Next, they do not incorporate regularity, which is important for quantifying exploitable locality. Finally, spatial locality in particular is defined in terms of cache blocks, making it ill-suited as a measure of inherent locality.

To address this, we define exploitable algorithmic spatial and temporal data reference locality metrics in terms of our hot data stream abstraction. The inherent exploitable spatial locality of a hot data stream is its spatial regularity. A data reference sequence's inherent exploitable spatial locality is the weighted average spatial regularity across all of the sequence's hot data streams, where a hot data stream's weight is its regularity magnitude. Long hot data streams indicate good inherent exploitable spatial reference locality. Similarly, the inherent exploitable temporal locality of a hot data stream is its temporal regularity. A data reference sequence's inherent exploitable temporal locality is the weighted average temporal regularity across all of its hot data streams. A hot data stream that repeats in close succession has good inherent temporal reference locality. As noted earlier, these metrics are characteristic of the algorithm and are independent of the mapping of data objects to memory addresses.

### *2.4.2 Realized Exploitable Locality Metric*

Caches exploit a program's spatial locality by transferring data in cache blocks that encompass multiple words (typically 32 to 128 bytes). Caches exploit temporal locality by storing the most recently accessed data. Our realized exploitable locality metric—cache block packing efficiency—attempts to quantify the success of caches at exploiting a program's inherent locality. This cache block packing efficiency metric is determined by the mapping of data objects to memory addresses. Ignoring cache capacity and associativity constraints, an ideal mapping would enable hardware to fully exploit a program's inherent locality. More typically, a sub-optimal mapping suggests software optimization opportunities. Finite cache capacity and limited associativity provide additional optimization opportunities.

A hot data stream's cache block packing efficiency is the ratio of the minimum number of cache blocks required by its data members if they could be re-mapped to different memory addresses, to the actual number of cache blocks required, given the current mapping. A ratio close to one indicates that the existing data layout is good at exploiting the hot data stream's inherent spatial locality. In addition, if the weighted average cache block packing efficiency over all program hot data streams is close to one, it indicates that the data layout assists the cache in exploiting the temporal locality of hot data streams since it keeps to a minimum the number of cache blocks referenced between successive accesses to the same hot data stream (ignoring cache associativity constraints).

## 3. DATA REFERENCE REPRESENTATIONS FOR COMPUTING LOCALITY

Compression algorithms discover and exploit sequence regularity to produce a compressed sequence. Thus, they suggest themselves as obvious candidates for efficiently discovering our exploitable locality abstraction—hot data streams—from a data reference trace. They also offer the additional benefit of producing a reduced data reference trace representation that is more efficient to analyze by virtue of its compactness. This section explores these issues.

Nevill-Manning and Witten's SEQUITUR compression algorithm infers hierarchical structure from a sequence of symbols [20]. This linear time, on-line algorithm constructs a context-free grammar for its input. It does this by recursively replacing repetitions in the sequence with a grammatical rule that generates the repeated string. Figure 3 illustrates this for the string *abcbcabcabc*. The resulting grammar requires fewer symbols than the input string and constitutes a hierarchical representation of the string that explicitly captures the repeated sequences, bc and abc. In addition, the grammar can be compactly represented as a DAG, as shown. Larus used this DAG representation of the SEQUITUR grammar, which he called *Whole Program Paths* (WPP), to represent a program's complete dynamic control flow behavior [17]. He showed that this WPP representation can be efficiently analyzed to extract hot program paths, which are acyclic path sequences that repeat frequently. For example, if the sequence in Figure 3 represents a program's acyclic path sequence, then Larus showed how to efficiently compute the hot program path, *abc*, from the WPP representation of this sequence.

This section extends the Whole Program Paths idea to provide a series of representations of a program's dynamic data reference behavior. Each representation in the series is more compact than its predecessor, but comprises a less precise representation of the data reference trace. We describe simple data abstractions that enable the construction of these representations, which we call *Whole Program Streams*$_i$ (*WPS*$_i$), from raw data address traces. The resulting representations are small and permit analyses without decompression (see Section 5.2). These representations can be used to efficiently discover *hot data streams*, our exploitable locality abstraction. While WPSs capture a program's complete dynamic data reference behavior, this level of detail is often unnecessary for many optimizations. To address this, we discuss how WPSs can be combined with hot data streams to construct *Stream Flow Graphs* (SFGs), which are yet more compact summarized representations that are similar to control flow graphs in that reference sequence information is no longer retained.

## 3.1 Whole Program Streams (WPS)

While acyclic paths are a convenient and compact abstraction of a program's dynamic control flow, no corresponding analogue exists for a program's data accesses. Thus, direct application of Larus' WPP technique to a program's data reference trace faces several challenges. First, using SEQUITUR to identify repetitions in a sequence of raw data addresses may obfuscate several interesting patterns that may only be apparent at the granularity of data

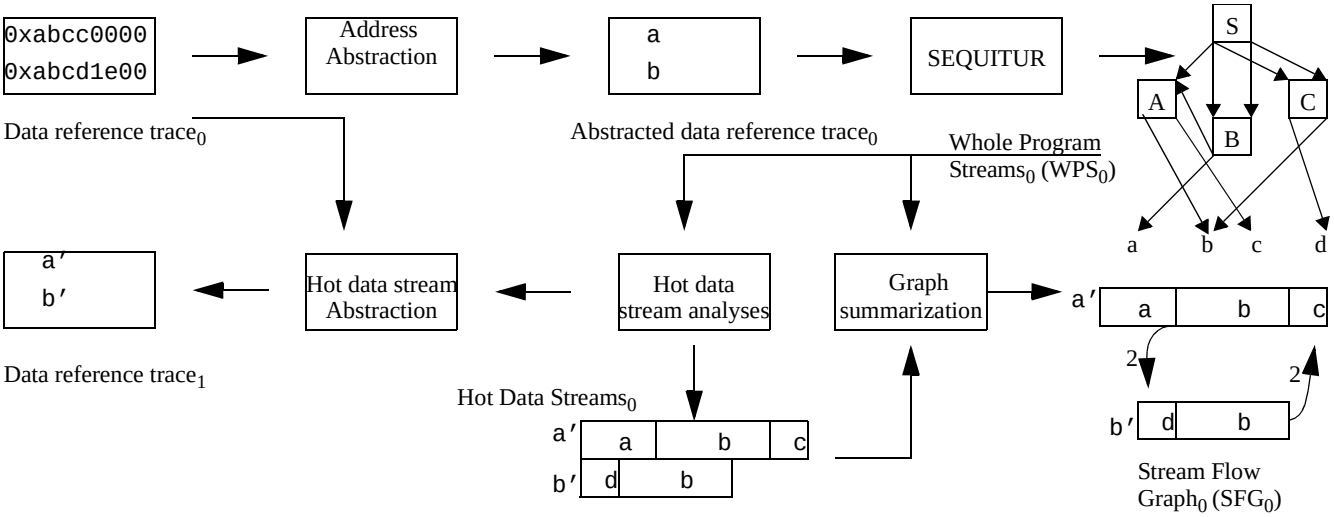


**Figure 4. Constructing data reference representations.**

objects. Second, the size of the resulting $WPS_0$ grammar may be too large to be efficiently analyzed. Finally, the processing time required to construct a $WPS_0$ from a raw data address trace may be too large for the technique to be practical. Abstracting data addresses may alleviate these problems by increasing the regularity in the data reference stream and consequently reducing the size of the $WPS_0$ grammar.

An obvious candidate for abstracting data addresses is to use the name or type of the object that generates the address. While this approach works for static objects, heap addresses present an additional challenge. Heap space is reused as heap data is dynamically allocated and freed. Consequently, the same heap address can correspond to different program objects during execution. These may need to be distinguished for several reasons. First, heap addresses are unlikely to be reused in the same manner across multiple program runs with different inputs. In addition, source-level optimizations need to distinguish between data objects that share the same heap address. A possible candidate for abstracting heap addresses is to use heap object names based on allocation site calling context. Allocation site calling context (depth 3) has been shown to be a useful abstraction for studying the behavior of heap objects [22]. An alternative approach that permits greater discrimination between heap objects is to associate a global counter with allocation sites and increment the counter after each allocation. The combination of the allocation site with this global counter (birth identifier) can then be used to abstract heap addresses. Unlike program paths, all these data abstractions are lossy—it is not possible to regenerate the address trace once these abstractions have been applied. However, depending on the nature of subsequent analyses on the $WPS_0$ structure, such an abstracted program data access history may suffice. Figure 4 illustrates the process of constructing a WPS representation from a data reference trace.

The next step is to analyze the WPS representation to discover hot data streams. As defined earlier, a hot data stream is a data reference subsequence that exhibits regularity and whose regularity magnitude exceeds a predetermined "heat" threshold, H. More informally, a hot data stream is a frequently repeated sequence of consecutive data references. A minimal hot data stream is the minimal prefix of a data stream with a heat of H or more. To ensure that the collection of hot data streams are representative of a program's data reference behavior, the heat threshold H is set appropriately such that the hot data streams references taken together account for 90% of all data references.

The algorithm used for detecting hot data streams in WPSs is the same algorithm Larus used to compute hot subpaths in WPPs (see [17] for further details). The algorithm performs a postorder traversal of the DAG, visiting each node once. At each interior node, it examines each data stream formed by concatenating the substreams produced by two or more of the node's descendents. The algorithm examines only concatenated strings, as the hot data streams produced solely by a descendent node are detected in a recursive call. The algorithm finds minimal hot data streams. It runs in time O(EL), where E is the number of edges in the WPS, and L is the maximum length of a data stream. The space requirements are O(PL), where P is the number of partially visited nodes in the DAG (which is typically far lower than the number of nodes in the DAG). Our experiments indicate that the algorithm is efficient in practice, requiring at most a minute to analyze the $WPS_0$ representation, even for complex programs, such as MS SQL server.

## 3.2 Trace Reduction

Since SEQUITUR is an on-line algorithm with tight time and space bounds, the resulting grammars are not minimal. In particular, the grammars contain many redundant productions. While this does not affect the hot data streams computed from the WPS representation, it does impact the WPS size and the analysis time required to the compute the hot data streams. To address this, Larus proposed and used SEQUITUR(1), which looks ahead a single symbol before introducing a new rule to eliminate a duplicate digram, in his implementation of WPPs, but the grammars produced are not significantly smaller [17].

We take an orthogonal approach that uses the hot data streams as an abstraction mechanism. As shown in Figure 4, hot data streams can be used as an abstraction mechanism to produce a reduced reference trace, $trace_{i+1}$, which is composed solely of hot data streams with all cold references (which can be considered noise) excluded, from a $trace_i$. This is done by traversing the $WPS_i$ to regenerate the reference trace with hot data streams encoded as single symbols and cold references eliminated. This reduced trace, $trace_{i+1}$, can again be processed by SEQUITUR to produce a $WPS_{i+1}$ representation that can be used to detect *hot data*

*streams*$_{i+1}$. The encoding of hot data streams as single symbols and elimination of cold addresses (i.e., noise) enables SEQUITUR to discover larger repetition patterns and accounts for the much more compact grammar. This process can be repeated as many times as required. Each iteration produces a more compact WPS representation and fewer, hotter hot data streams, but includes less of a program's original data reference sequence since hot data streams are selected to cover 90% of references. Thus $WPS_0$ includes all original data references, hot data $streams_0$ include 90% of these references, $WPS_1$ includes 90% of original data references, and hot data $streams_1$ includes 81% of original data references. The reduced representations are useful since the $WPS_0$ representation captures a program's complete dynamic data reference behavior, a level of detail that is often unnecessary for many optimizations. In addition, the reduced representations still retain information about frequently observed data reference behavior.

## 3.3 Graph Representation

While WPSs are compact, they retain data reference sequence information that can complicate analysis that does not require this information. For example, many analyses operate efficiently on a program's control flow graph, which does not retain precise basic block sequence information. To address this, we can combine the WPS with hot data streams to construct a *Stream Flow Graph (SFG)*, which is similar to a control flow graph with hot data streams replacing basic blocks as graph nodes, as shown in Figure 4. In the SFG, each hot data stream is represented by a single node and weighted directed edges, <*src*, *dest*>, indicate the number of times an access to hot data stream *src* is immediately followed by an access to hot data stream *dest*. The SFG representation has the advantage that many control flow graph analyses can be directly adapted to operate on it. For example, dominators in the SFG may suggest program load/store points to initiate prefetching. In addition, the SFG captures temporal relationships that are potentially more precise than Gloy et al.'s Temporal Relationship Graph (TRG) since they are not determined by an arbitrarily selected temporal reference window size [11].

## 3.4 Discussion

The previous discussion suggests using the WPS, and SFG representations, and the hot data stream abstractions within a data locality optimization framework, much as control flow graphs and program paths are used for code optimizations. However, a major difference between these representations is that the control flow graph is a static representation and hence represents all possible program executions, whereas WPSs, and SFGs are abstract representations of one observed execution. Offsetting this is the fact that most code optimizations have to be conservative since an incorrect transformation will affect program correctness, whereas many data locality optimizations can afford to be aggressive as incorrect application affects only a program's performance and not its correctness. In addition, our experiments indicate that hot data streams, when expressed in terms of the program loads and stores that generate the references, are relatively stable across program executions with different inputs [7].

# 4. EXPLOITING LOCALITY

Our exploitable locality abstraction, hot data streams, and our associated exploitable locality metrics can be used to improve data reference locality in at least four ways, which are described further in this section. First, they can help identify programs likely to benefit from data locality optimizations by computing exploitable locality metrics for them. Since these exploitable locality metrics are computed at the granularity of individual hot data streams, locality optimizations can focus solely on the streams that show the largest potential benefit. Second, they can help select the most suitable locality optimization for a particular hot data stream. Next, hot data streams and our data reference representations can help drive locality optimizations. Finally, they can be used to compute the potential benefits of data locality optimizations in the limit.

## 4.1 Tools for Improving Data Locality

We built a tool called DRiLL (Data Reference Locality Locator) to help programmers improve a program's data reference locality. DRiLL enumerates all of a program's hot data streams. Clicking on a hot data stream displays its *regularity magnitude (heat)*, *spatial regularity* (our inherent exploitable spatial locality metric), *temporal regularity* (our inherent exploitable temporal locality metric), and its *cache block packing efficiency* (our realized exploitable locality metric). In addition, DRiLL displays the program source responsible for the reference to the stream's first data member in a code browser window. The hot data stream can be traversed in data member order to see the code and data structures responsible for the stream references.

We have used DRiLL to improve the data reference locality of a few programs, including boxsim [8], by hand. We focused on hot data streams with high heat and poor cache block packing efficiencies. Streams with poor cache block packing efficiencies indicate data objects that should be in the same cache block but are currently in different blocks. We attempted to co-locate these data objects in the same cache block by modifying structure definitions to reorder fields, splitting structures, and merging split portions of different structures. Preliminary results appear promising as our transformations improved execution time by 8–15%. Measurements of the optimized programs confirmed that these improvements were due to improved cache block packing efficiencies.

## 4.2 Implementing Data Locality Optimizations

A program's data reference locality can be improved by changing the order in which it accesses data (i.e., its inherent locality), or by changing the mapping of data objects to memory addresses (i.e., its realized locality). While changing the order in which a program accesses data has been used to improve the locality of dense matrix codes that access arrays, it is not a practical optimization technique for pointer-based data structures [5]. However, prefetching, which changes the order in which the memory system sees data requests, can be used to improve data locality by tolerating memory access latency [14, 19]. Techniques for changing a program's mapping of data objects to memory addresses include clustering, coloring, and compression [5]. Of these, compression typically requires source code modification and is not considered further. Coloring reduces cache conflict misses caused by multiple blocks mapping to the same cache location. Higher associativity caches—4 way, 8 way and higher associativity caches are becoming increasingly common—reduce the benefit of this optimization. Thus, we focus on clustering as the primary data layout optimization for improving locality. We briefly outline how our exploitable locality metrics, hot data stream abstraction, and data reference representations can be used to identify data locality optimization targets, select the most effective optimization combination for a given target, and drive the selected optimization. The discussion is preliminary as this is a topic of current research.

### *4.2.1 Identifying Data Locality Optimizations Targets*

The locality metrics described in Section 2 can be used to identify

data locality optimization targets. The best optimization targets are long hot data streams that are not repeated in close succession, and that have poor cache block packing efficiency. Short hot data streams are indicative of poor inherent exploitable spatial locality and limit the benefit of any data locality optimization. Hot data streams that are repeated in close succession are likely to be cache resident on subsequent accesses and consequently unlikely to benefit from data locality optimizations. Poor cache block packing efficiency signifies that a hot data stream occupies a larger number of cache blocks than are strictly necessary.

### *4.2.2 Selecting Data Locality Optimizations*

Clustering and prefetching have different strengths and weaknesses. In addition, we distinguish between two types of prefetching—intra-stream prefetching, which fetches the data members of the stream being currently accessed, and inter-stream prefetching which fetches the data members of a stream that is different from the one being currently accessed. Clustering alone is less effective for hot data streams with poor exploitable temporal locality when the improvement in cache block packing efficiency is insufficient to make the stream's data members cache resident. In addition, in the absence of continuous reorganization, clustering cannot address competing layout constraints caused by data objects belonging to multiple hot data streams. Prefetching, on the other hand, can address both these shortcomings, but requires intelligent scheduling to be effective. In addition, it can increase a program's memory bandwidth requirements. Given these constraints, clustering should be used for hot data streams with poor cache block packing efficiency to enforce the dominant data layout. Inter-stream prefetching should be used for hot data streams with poor exploitable temporal locality. Finally, intra-stream prefetching should be used for those streams with good exploitable spatial locality that have poor cache block packing efficiency even after clustering due to competing layout requirements.

### *4.2.3 Driving Data Locality Optimizations*

In addition to guiding data locality optimizations, such as prefetching, and clustering, our data reference representations and hot data streams can be used to drive these optimizations. For prefetching optimizations, hot data streams supply an ordered list of data addresses to be fetched. For inter-stream prefetching, the SFG can be analyzed to determine candidate pairs. In addition, dominators in the SFG suggest program load/store points to initiate prefetching. Clustering optimizations use an object affinity graph to determine objects that need to be co-located [4]. The SFG can be used as a more precise replacement for the object affinity graph.

We quantified the potential benefit of hot data stream-based locality optimizations for several programs. The detailed results are reported in Section 5.4. They indicate that locality optimizations based on hot data streams appear promising, and can produce cache miss rate reductions of up to 92%. In addition, preliminary results for an initial implementation of a hot data stream-based prefetching optimization indicate cache miss rate improvements of 15–43% for three benchmarks when different data reference profiles were used as train and test profiles [7].

## 5. EXPERIMENTAL EVALUATION

This section presents results from applying our techniques to several programs and demonstrates that they produce compact representations of data reference behavior that support efficient analysis. We show that our hot data stream abstraction is useful for quantifying data reference locality and can guide data locality optimizations by using it to compute locality metrics for our benchmark programs. Finally, we compute the potential benefit of data locality optimizations guided by our representations and abstractions.

### 5.1 Experimental Methodology

The programs used in this study include several of the SPECint2000 benchmarks, *boxsim*, a graphics application that simulates spheres bouncing in a box [8], and Microsoft SQL server 7.0, a commercial database. 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 [23]. Stack references were not instrumented to avoid biasing the data reference locality results. In addition, they typically exhibit good locality and are seldom data locality optimization targets. For experimentation purposes, the traces were written to a file, rather than processed on-line. Each data reference occupied 9 bytes in the trace (one byte encodes the reference type and the program counter and data address each occupy four bytes). The heap allocation information was processed to build a map of heap objects. A heap object is a <Start address, global counter> pair, where the global counter 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 trace was fed to SEQUITUR, which produced a context-free grammar. The DAG representation of this grammar (i.e., the $WPS_0$) was analyzed to identify hot data $streams_0$. These hot data $streams_0$ were used in conjunction with the $WPS_0$ to construct the $SFG_0$ representation. The process was repeated as described in Section 3 to construct the $WPS_1$ representation. Finally, the $WPS_1$ was analyzed to identify hot data $streams_1$ and these were used to construct the Stream Flow $Graph_1$ ($SFG_1$). Measurements were performed on a dual processor 550 Mhz Pentium III Xeon PC with 512 MB of memory running Windows 2000 Server. The SPEC benchmarks were run with their smallest input data set (test) with the exception of eon which was run with the larger train input set. *boxsim* was used to simulate 100 bouncing spheres. The SQL server measurements were performed while running the TPC-C benchmark. This is an on-line transaction processing benchmark that consists of a mix of five concurrent transactions of different types and complexity. The

**Table 1: Benchmark characteristics**

| Benchmark | Refs. millions | Heap millions | Global millions | Addresses heap+ global | Refs./ Address |
|---|---|---|---|---|---|
| 176.gcc | 464.7 | 125.0 | 146.1 | 22,647 | 11,972 |
| 197.parser | 1,255.7 | 516.4 | 530.5 | 9,977 | 104,929 |
| 252.eon | 1,784.0 | 165.4 | 152.5 | 18,744 | 16,961 |
| 253.perlbmk | 112.1 | 36.3 | 27.4 | 26,715 | 2,385 |
| 255.vortex | 3,384.4 | 778.4 | 637.3 | 200,810 | 7,050 |
| 300.twolf | 91.6 | 39.3 | 27.6 | 17,770 | 3,764 |
| boxsim | 183.4 | 60.1 | 43.6 | 75,677 | 1,371 |
| SQL server | 279.2 | 139.4 | 39.8 | 1,606,890 | 112 |

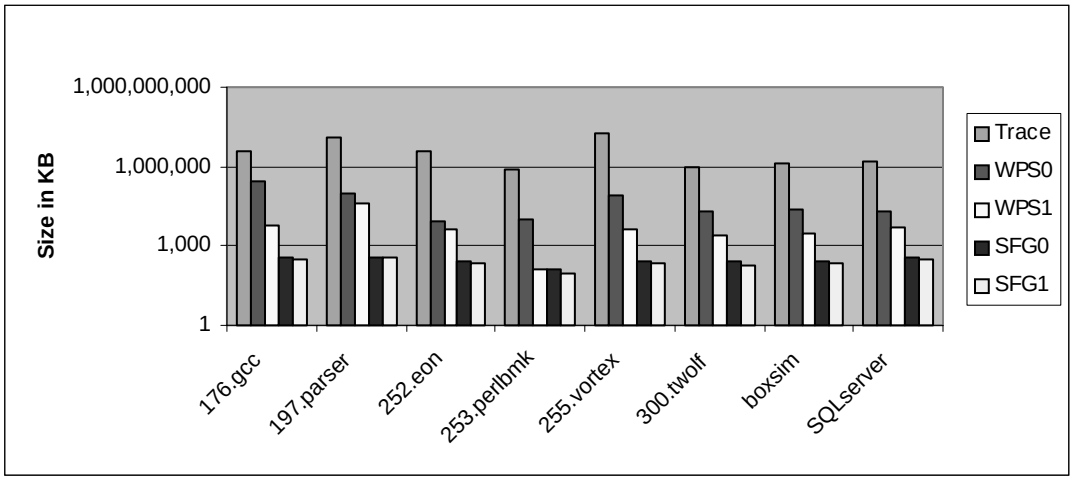


**Figure 5. Relative sizes of the different data reference representations.**

benchmark runs for a fixed length of time, in this case a short (non-standard run) of 60 seconds. Unlike the SPEC benchmarks, which are single threaded, SQL executes many threads. The current system distinguishes data references between threads and constructs a separate WPS for each one. Table 1 reports some overall characteristics of the benchmark's data references. The last column indicates the average number of references to each global and heap data address.

## 5.2 Evaluating the Representations and Abstractions

Figure 5 reports the sizes of the different data reference representations. The $WPS_0$ and $WPS_1$ size is the size of the ASCII grammar produced by SEQUITUR (the binary representation can be two times smaller), and the $SFG_0$, $SFG_1$ size is the size of the respective Stream Flow Graphs. The $WPS_0$s are one to two orders of magnitude smaller than the data reference traces, on average. The $WPS_1$ and $SFG_0$, $SFG_1$ offer an additional order of magnitude size reduction, reducing a gigabyte trace to a representation that occupies a few megabytes or even a few hundred kilobytes. These small representations permit in-memory trace processing. Most promisingly, in all cases the $WPS_0$s were small enough to permit efficient processing (avoiding the need to use the $WPS_1$ or SFG representations, which are lossy representations of the data reference trace), with the analysis time for hot data stream identification ranging from a few seconds to a minute. Hence, all following results are computed on the $WPS_0$ representation unless noted otherwise. In addition, the compression is a measure of the regularity in a program's data reference stream and the high compression ratios suggest a large amount of inherent exploitable reference locality.

The hot data stream analysis requires three parameters—the minimum and maximum length of a hot data stream, and the threshold above which a data stream is marked as "hot". We set the minimum hot data stream length at 2 and the maximum length at 100 since it is unclear whether there is significant opportunity for exploiting longer streams and our data (not shown) indicated that only a few streams are longer than this value. Because the goal is to use hot data streams for locality optimizations, we would like a large majority of the data references in the trace (at least 90% of references) to belong to hot data streams, for the optimization to have a significant impact on overall program locality. However, if this requires setting the threshold so low that it produces an extremely large number of hot data streams, the optimization opportunity becomes less attractive.

To investigate this issue, we defined our threshold for identifying hot data streams as follows—it is the smallest value at which over 90% of data references participate in hot data streams. If a program's references were uniformly distributed over its address space, each address would be accessed (total references)/ (total addresses) times. We use multiples of this 'unit uniform access' for each program to report the threshold. This normalization process permits comparison across the programs, independent of the number of data references in the trace. Greater the threshold, higher the program's data reference regularity. Since the threshold value represents a plausible empirical measure of exploitable reference locality, we refer to it as 'exploitable locality threshold' or just 'locality threshold'.

Table 2 reports the 'locality threshold' for the various programs as well as hot data stream information at this 'locality threshold'. The locality thresholds range from 1 for 176.gcc, to 126 for 252.eon. The table indicates that SQL server has a significantly larger num-

**Table 2: Hot data stream information.**

| Benchmark | Locality threshold ('unit uniform access' multiple) | Number of hot data streams | # of distinct addresses in hot data streams | % of total program data addresses |
|---|---|---|---|---|
| 176.gcc | 1 | 7,461 | 3,912 | 17.27 |
| 197.parser | 69 | 105 | 14 | 0.14 |
| 252.eon | 126 | 60 | 80 | 0.43 |
| 253.perlbmk | 58 | 228 | 181 | 0.68 |
| 255.vortex | 75 | 475 | 250 | 0.12 |
| 300.twolf | 5 | 1,260 | 419 | 2.36 |
| boxsim | 4 | 3,896 | 1,701 | 2.25 |
| SQLserver | 8 | 77,319 | 32,616 | 2.03 |

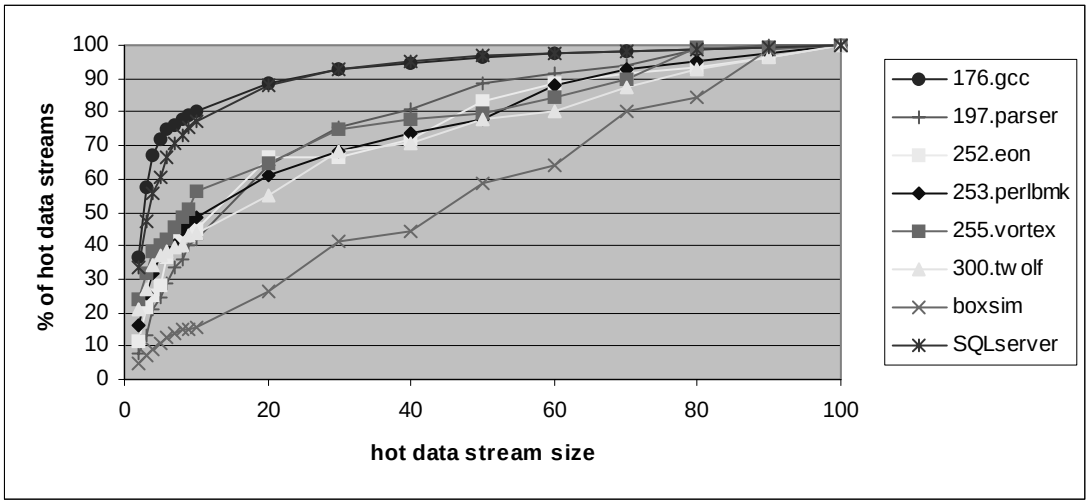


**Figure 6. Cumulative distribution of hot data stream sizes.**

ber of hot data streams than any of the other benchmarks, probably reflecting its more complex data reference behavior. In addition, SQL server has a higher number of distinct data addresses that participate in hot data streams. However, when viewed as a percentage of the total number of data addresses accessed during program execution, the number is comparable to the other benchmarks, and at 2.03% is quite small. The sole exception is 176.gcc, where data addresses that participate in hot data streams comprise 17.3% of all data addresses. These results suggests that commercial applications such as SQL server, despite their richer data reference behavior, possess regularity in their data reference stream much like the SPEC benchmarks. In addition, the results indicate that setting the "heat" threshold so that 90% of program data references participate in hot data streams, does not produce an excessive number of hot data streams, and the resulting hot data streams include only a small fraction of all program data addresses. This is encouraging for data locality optimizations based on the hot data stream abstraction. Finally, examination of the program source associated with these hot data streams indicate that while some streams occur in loops, many span procedure and data structure boundaries, exposing potentially new optimization opportunities.

## 5.3 Using Metrics to Quantify Locality

Two factors contribute to the regularity magnitude or 'heat' of a data stream. One is its spatial regularity, which is the number of references it contains, and the other is its regularity frequency, which is the number of non-overlapping occurrences of the stream. Another hot data stream characteristic is its temporal regularity, which is the average reference distance between consecutive stream occurrences. As described in Section 2.4, these characteristics can be used to compute exploitable locality metrics. We performed experiments to compute these metrics and the data is presented in Figures 6, 7 and summarized in Table 3. Figure 6 illustrates the cumulative distribution of hot data stream sizes (i.e., spatial regularity), which is our inherent exploitable spatial locality metric. The programs divide into three classes, with 176.gcc and SQLserver in one class, boxsim in a category by itself, and the rest of the benchmarks in a third class. 176.gcc and SQLserver have relatively the worst inherent exploitable spatial locality as 90% of their hot data streams are less than 20 references long. On the other hand, boxsim has the best inherent exploitable spatial locality with a fairly uniform distribution of hot data stream sizes from 2 to 100.

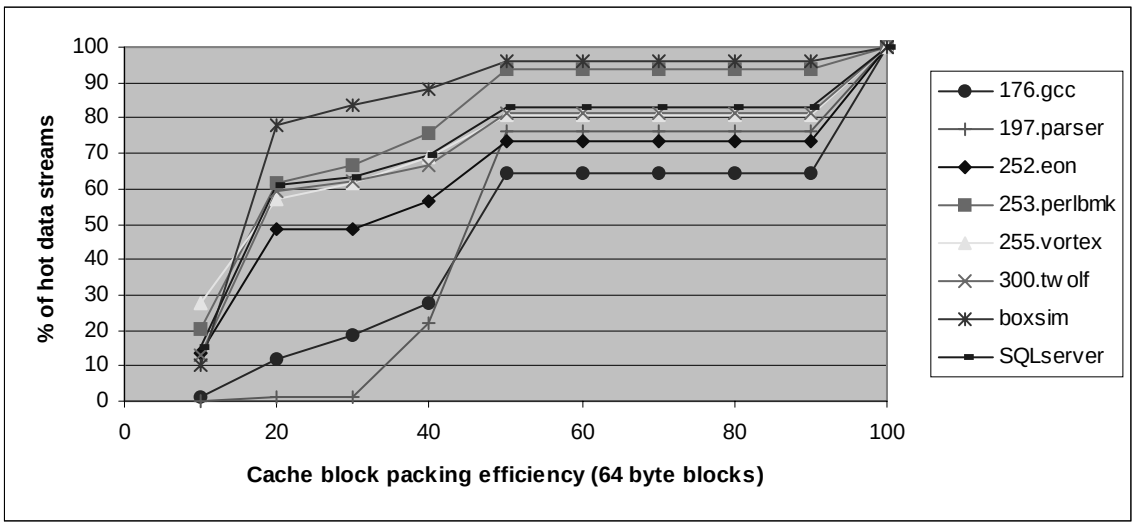


**Figure 7. Cumulative distribution of hot data stream cache block packing efficiencies.**

The other benchmarks lie in between these extremes with a slight bias towards short hot data streams. Table 3 shows weighted average hot data stream sizes, where the "heat" (i.e., regularity magnitude) of a data stream is used as its weight, so hotter data streams have a greater influence on the reported average value. As expected, 176.gcc and SQLserver have the smallest average hot data stream size. While boxsim has the longest streams, its weighted average is not significantly greater than the other benchmarks, indicating that its "hottest" streams are small. In addition, 255.vortex exhibits a similar behavior wherein its weighted average hot data stream size is smaller than Figure 4 indicates. From a cache optimization standpoint, the data is encouraging as stream sizes are sufficiently long to guide cache-conscious layouts or direct prefetching.

Due to space constraints, we omit the graph for cumulative distribution of average hot data stream repetition intervals (i.e., temporal regularity), which is our inherent exploitable temporal locality metric. Instead, we present the weighted average hot data stream repetition intervals, expressed in terms of references, in Table 3. The weighted average numbers reported are similar to the omitted graph with the exception of boxsim, which appears to have its "hottest" data streams repeat in closer succession than expected. Once again, the programs divide into three categories with SQL server, 176.gcc, and 300.twolf in one class, boxsim and 253.perlbmk in a second class, and 197.parser, 252.eon, and 255.vortex in the third class. The first class of benchmarks, which includes SQLserver, have relatively poor inherent exploitable temporal locality. 253.perlbmk and boxsim have reasonable exploitable temporal locality. Finally, 197.parser, 252.eon, and 255.vortex possess hot data streams that exhibit good exploitable temporal locality. Interestingly, these results correlate well with number of hot data streams present in these benchmarks, with the exception of boxsim (see Table 2). Benchmarks with a large number of hot data streams tend to have the same stream repeat less often.

The inherent exploitable locality metrics computed from hot data streams indicate that 176.gcc and SQLserver have relatively poor inherent exploitable spatial and temporal locality, boxsim has excellent inherent spatial locality but only fair temporal locality, 300.twolf has reasonably good inherent spatial locality but poor temporal locality, and the rest of the benchmarks have relatively good inherent exploitable temporal locality and reasonable inherent exploitable spatial locality.

**Table 3: Summary of inherent and realized locality metrics.**

| Benchmark | Wt. avg hot data stream size | Wt. avg. repetition interval | Wt. avg cache block packing efficiency |
|---|---|---|---|
| 176.gcc | 10.3 | 4,575.4 | 51.7 |
| 197.parser | 24.0 | 86.9 | 64.8 |
| 252.eon | 18.4 | 47.9 | 66.4 |
| 253.perlbmk | 23.1 | 334.8 | 31.0 |
| 255.vortex | 11.5 | 92.8 | 36.1 |
| 300.twolf | 23.9 | 847.7 | 39.8 |
| boxsim | 25.8 | 228.2 | 49.2 |
| SQLserver | 10.9 | 2,544.1 | 41.4 |

The next set of experiments use hot data streams to compute our realized exploitable locality metric for the benchmarks as outlined in Section 2.4.2. Figure 7 illustrates the cumulative distribution of the hot data stream's cache block packing efficiency for 64 byte cache blocks. The shape of the curves is remarkably similar for all benchmarks and indicates that their hot data streams fall into two distinct categories—a small fraction (5--35% of hot data streams) that have ideal cache block packing efficiency, and the remaining streams that have suboptimal cache block packing efficiency. This metric permits focusing optimization efforts on those hot data streams with the most potential for improvement. 176.gcc and 197.parser have the best packing efficiencies and 253.perlbmk and boxsim, the worst. From the weighted average data in Table 3 it appears that the boxsim's and 252.eon's "hottest" data streams have better than expected packing efficiencies. However, on average, the benchmark's hot data streams occupy 2 to 3 times the number of caches blocks that an ideal layout would require. This suggests that the current mapping of data objects to memory addresses does not do a very good job of exploiting the program's inherent reference locality and promises significant benefits from data locality optimizations.

Combining the inherent and realized exploitable locality metric data for the benchmarks it appears that boxsim and 300.twolf, which have good inherent spatial locality, fair/poor inherent temporal locality, and poor cache block packing efficiencies, would benefit most from data locality optimizations, while 197.parser and 252.eon, which have good inherent temporal locality, would benefit the least.

## 5.4 Evaluating the Potential of Stream-Based Data Locality Optimizations

We now evaluate the hot data stream abstraction for data locality optimizations along two dimensions. First, are the locality metrics computed using hot data streams useful for indicating the potential of different optimizations? If so, they could be used to select the data locality optimization likely to produce the greatest benefit. Second, how much improvement can we expect from locality optimizations based on the hot data stream abstraction?

Before attempting to answer these questions, we address a more basic issue. How much do we give up by focusing our optimization efforts on hot data streams exclusively? To answer this, we measured cache miss rates for a variety of cache configurations. Figure 8 shows the proportion of caches misses to references that participate in hot data streams as the cache miss rate increases. The graph indicates that if cache performance is a bottleneck (miss rate > 5%) then around 80% of misses are attributable to hot data stream references, with the exception of 197.parser for which this number is only 30%. This is not surprising as it suggests that when hot data streams fit in the cache, most misses are to cold addresses and the cache miss rate is low. For programs with a cache performance problem, their higher cache miss rates indicate a large number of misses to hot data stream references, making these streams suitable optimization targets.

Figure 9 attempts to answer the previous two questions about the ability of hot data streams-based metrics to select data locality optimizations and the benefits of optimizations that target hot data streams. We first compute the potential impact on cache miss rate of an ideal prefetching scheme based on hot data streams that is able to schedule all prefetches sufficiently in advance of their use such that the data is cache-resident when referenced (we ignore misses that occur when the data is prefetched, since these do not

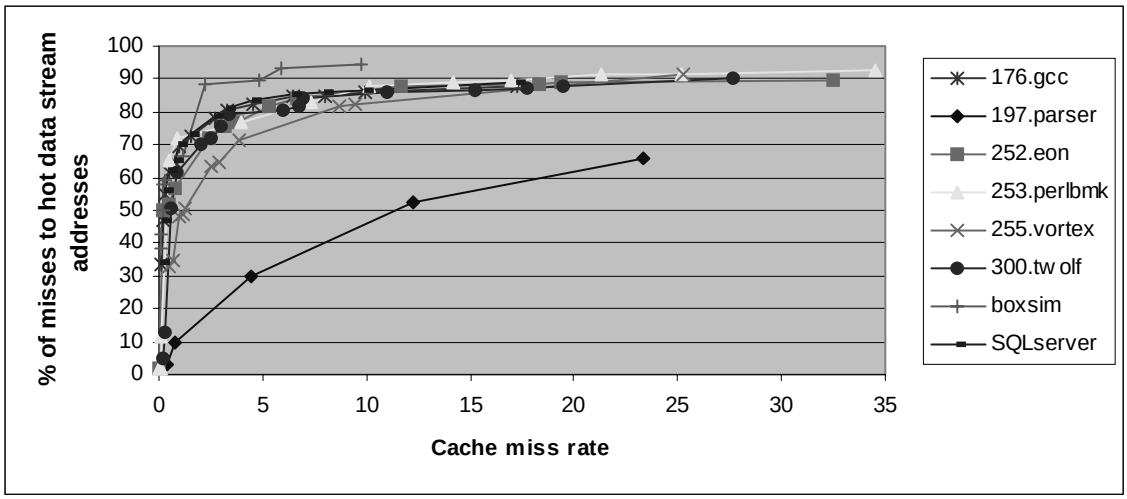


**Figure 8. Fraction of cache misses caused by hot data streams.**

affect access latency if the prefetch is scheduled sufficiently in advance). Second, we compute the effect of using hot data streams to cluster objects as mentioned in Section 4. Finally, we compute the effect of combining the two optimizations. The results are computed for a 8K fully-associative cache with 64 byte blocks (we scaled the cache size since we used the SPEC benchmark's test inputs) and are normalized to the base cache miss rate. As expected from the locality metric data, boxsim and 300.twolf benefit the most from locality optimizations and 197.parser (also see Figure 8), 252.eon, and 253.vortex, benefit the least. In addition, clustering appears to be less effective for 176.gcc, and SQLserver, possible due to their complex data reference behavior that suggests multiple competing layouts. However, with the exception of 197.parser (see Figure 8), locality optimizations based on hot data streams appear promising, producing cache miss rate reductions of 64–92%. Admittedly, these are ideal miss rate improvements. In a practical implementation the prefetch scheduling would not be perfect. In addition, program constraints may prevent a faithful implementation of the clustering scheme suggested by hot data streams. Nevertheless, the large cache miss rate improvements indicate that practical implementations of these hot data stream-based locality optimizations merit further consideration.

# 6. RELATED WORK

Larus used a hierarchical compression algorithm (SEQUITUR) to construct Whole Program Paths (WPP), which are a compact, yet analyzable representation of a program's dynamic control flow [17]. This paper demonstrates that a similar scheme can serve as an effective representation of a program's dynamic data reference behavior. Together, they provide a complete picture of a program's dynamic execution behavior.

Compressing program traces has been the subject of much research. For example, Plezkun described a two-pass trace compression scheme, which infers basic block's successors and tracks linear address patterns.This is used to produce a variable-length encoding that compresses address traces to a fraction of a bit per reference [21]. Larus's Abstract Execution scheme guides the re-execution of the address generating slice of a program with a small amount of run-time data [18]. While these techniques produce considerable compression, the encoded files are not as analyzable as WPSs, and they require significant post-processing to regenerate the address trace. In any case, the focus of this research is not on compression but on efficient representations and abstractions for analysis and optimization of dynamic data reference behavior.

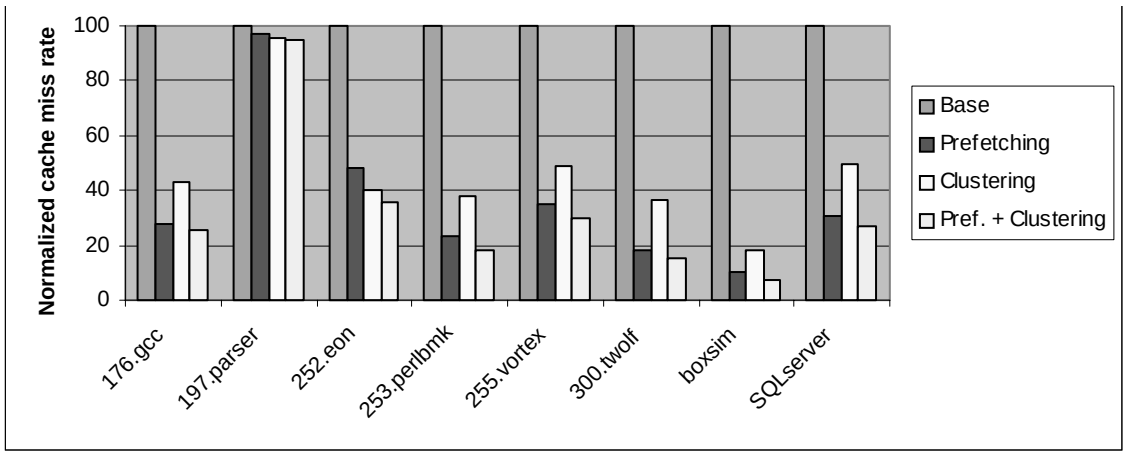


**Figure 9. Potential of locality optimizations based on the hot data stream abstraction.**

## 7. CONCLUSIONS

With the growing processor-memory performance gap, understanding and optimizing a program's data reference locality, and consequently, its cache performance, is becoming increasingly important. The paper address this by proposing a quantitative basis for understanding and improving reference locality. It describes data reference representations—Whole Program Streams, Stream Flow Graph—and an exploitable locality abstraction—hot data streams—that support this framework. The paper demonstrates that these data reference representations are compact and can be used to efficiently compute the hot data stream abstraction. In addition, it shows that hot data streams are useful for quantifying as well as exploiting data reference locality. The results reported in this paper suggest significant opportunity for hot data stream-based locality optimizations.

## 8. ACKNOWLEDGMENTS

Jim Larus generously provided his implementation of the SEQUITUR algorithm as well as code for detecting hot subpaths in Whole Program Paths. Tom Ball, Ras Bodik, Jim Larus, Scott McFarling, Ben Zorn and the anonymous referees provided useful comments on earlier drafts of this paper.

## 9. REFERENCES

[1] G. Ammons and J. R. Larus. "Improving data-flow analyses with path profiles." In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 72-84, 1998.

[2] R. Bodik, R. Gupta, and M. L. Soffa. "Redefining data flow information using infeasible paths." In *Proceedings of the ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, May 1997.

[3] 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.

[4] 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.

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

[6] 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.

[7] T. M. Chilimbi. "On the stability of temporal data reference profiles." In *Microsoft Research, Technical Report MSR-TR-2001-43*, Apr. 2001.

[8] S. Chenney. "Controllable and scalable simulation for animation." *Ph.D. Thesis, University of California at Berkeley*, 2000.

[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] J. A. Fisher. "Trace Scheduling: A technique for global microcode compaction." In *IEEE Transactions on Computers, vol. C-30*, pages 478-490, 1981.

[11] 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.

[12] 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.

[13] J. L. Hennessy and D. A. Patterson. "Computer Architecture: A quantitative approach, Second Edition." *Morgan Kaufmann Publishers, San Mateo, CA*, 1995.

[14] 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.

[15] 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.

[16] D. E. Knuth. "An empirical study of FORTRAN programs." In *Software—Practice and Experience, vol 1*, pages 105-133, 1971.

[17] J. R. Larus. "Whole program paths." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 259-269, May 1999.

[18] J. R. Larus. "Abstract Execution: A technique for efficiently tracing programs." In *Software—Practice and Experience, vol 20*, pages 1241-1258, 1990.

[19] C-K. Luk and T. Mowry. "Compiler-based prefetching for recursive data structures." In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Oct. 1996.

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

[21]A. R. Plezkun. "Techniques for compressing program address traces." In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 32-40, 1994.

[22] M. L. Seidl, and B. G. Zorn "Segregating heap objects by reference behavior and lifetime." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 12-23, Oct. 1998.

[23] 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.

[24] 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.