A Quantitative Analysis of Loop Nest Locality *

Kathryn S. McKinley
University of Massachusetts

Olivier Temam
Versailles University

Abstract

This paper analyzes and quantifies the locality characteristics of numerical loop nests in order to suggest future directions for architecture and software cache optimizations. Since most programs spend the majority of their time in nests, the vast majority of cache optimization techniques target loop nests. In contrast, the locality characteristics that drive these optimizations are usually collected across the entire application rather than the nest level. Indeed, researchers have studied numerical codes for so long that a number of commonly held assertions have emerged on their locality characteristics. In light of these assertions, we use the Perfect Benchmarks to take a new look at measuring locality on numerical codes based on references, loop nests, and program locality properties. Our results show that several popular assertions are at best overstatements. For example, we find that temporal and spatial reuse have balanced roles within a loop nest and most reuse across nests and the entire program is temporal. These results are consistent with high hit rates, but go against the commonly held assumption that spatial reuse dominates. Another result contrary to popular assumption is that misses within a nest are overwhelmingly conflict misses rather than capacity misses. Capacity misses are a significant source of misses for the entire program, but mostly correspond to potential reuse between different loop nests. Our locality measurements reveal important differences between loop nests and programs; refute some popular assertions; and provide new insights for the compiler writer and the architect.

1 Introduction

Because processor speed is increasingly outpacing memory speed, an enormous amount of research focuses on improving the cache behavior of numerical programs (see for example, Smith's bibliographies on hardware aspects of cache memories [Smi86, Smi91] and compiler techniques that exploit cache memories [CM95, CMT94, LRW91, MLG92, WL91]). Most of this work depends on loop nests to provide predictable and regular data accesses. Techniques to improve data cache performance typically target and model locality characteristics found in loop nests. For example, software and hardware prefetching exploit the spatial locality of regular accesses in loop nests [CB95, CKP91, Dra95, KL91, MLG92]. Wolf and Lam [WL91] model data locality by distinguishing four categories of locality which they use to drive loop optimizations: *spatial* – reuse of adjacent locations in a cache block; *temporal* – reuse of the same location; *self* – reuse from the same data reference; and *group* – reuse from distinct references. They do not however measure programs to determine how often these locality types occur. Even though many of these approaches yield significant improvements, there exists no broad quantitative study of loop nest locality, of which we are aware, driving this exploration.

New memory architectures can exploit locality more selectively than previous caches [HP95, MW96]. For example, the HP-7200 [HP95] uses a form of cache bypass for a reference stream with only spatial locality. These architectures require detailed knowledge about the locality properties of loop nests and programs. Of course, much of the research on memory optimizations analyzes relevant locality characteristics. These studies are typically conducted across the entire application, while many optimizations just target loop nests. For instance, several studies find numerous capacity misses in applications [HP95, HS89, SA93]. If these misses actually correspond to locality across distinct nests, loop nest optimizations are unlikely to eliminate them.

In this paper, we investigate the locality behavior of loop nests in order to suggest targets for future software and hardware research. We focus on data cache behavior although the data cache may only have a small impact on the total performance for programs with very low miss rates. We investigate in detail how well caches exploit locality characteristics. We quantify both achieved and potential locality as a function of the distance between load/store references to the same word or cache block. We measure and quantify locality within a nest, across nests, and for the entire program.

Because numerical codes have been the target of memory optimizations for so long, a number of popular assertions on their memory characteristics have emerged. To provide a framework for our study, we examined the literature on memory optimizations and extracted some of the most prevalent assertions. These assertions, listed in Table 1, arise from extensive measurements of complete programs [Bel66, GHPS93, Hil88, HS89, HP95, PHH88, Smi82, SA93], slightly more narrow measurements that evaluate proposed hardware or software techniques [AP93, CB95, Dra95, Jou90, KL91, MLG92], and software models [CMT94, GJG88, MLG92, WL91]. Our results dispute Assertions 1 and 2, and confirm Assertions 3 and 4 for our benchmark suite.

For instance, we find that loop nest locality significantly differs from whole program locality; loop nests attain more spatial reuses (Assertion 1) and dramatically fewer capacity misses than programs (Assertion 2). Our quantification of locality characteristics suggests that the scope of some optimizations should be revisited; even though most reuse is within a nest (Assertion 3), most misses occur between nest executions. This result suggests compiler writers should next focus on optimizing multiple nests at once. We also show that bandwidth and the cache are frequently wasted due to loads of blocks in which only one word is referenced (Assertion 1). This result suggests that architects should focus more on adjustable block sizes and cache bypass instructions. In summary, this paper explores and checks a number of popular assertions on data locality, and provides new quantitative insights on locality within and across numerical loop nests.

The remainder of this paper is organized as follows. Section 2 describes our experimental framework. We then examine our results from three different perspectives: *intra-nest* data locality (Section 4), *inter-nest* and *whole program* data locality (Section 5), and load/store instruction locality (Section 6). Within each of the sec-

*Kathryn S. McKinley was supported by NSF grant CCR-9525767. Authors' addresses: K. S. McKinley, Computer Science Department, LGRC, University of Massachusetts, Amherst MA 01003-4610; email: mckinley@cs.umass.edu; O. Temam, PRiSM Laboratory,Versailles University, 78000 Versailles, France; email: temam@prism.uvsq.fr.

To appear in ASPLOS-VII: Seventh International Conference on Architectural Support for Programming Languages and Operating Systems.

Table 1: Assertions about reuse characteristics and cache behavior of numerical programs

Assertions	Related Issues, Assertions, & Questions
1. Spatial reuse is the dominant form of reuse [CMT94, Jou90, KW73, PHH88, Smi82, Smi87].	a. Temporal reuse avoids fewer misses than spatial. b. Cache blocks effectively exploit spatial reuse [KW73, PHH88, Smi82, Smi87]. c. Most cache pollution is due to *spatial-only* blocks, i.e., blocks which are only reused spatially [Pou94].
2. Capacity misses occur more frequently than conflict misses, and both are significant sources of misses [HP95, HS89, SA93].	a. Do ping-pong conflicts occur frequently? b. 2-way set-associative caches remove the majority of conflict misses [AP93, HS89].
3. Most reuse occurs within a nest rather than across nests [CMT94, TGJ93, WL91].	a. Targeting individual loop nests is sufficient. b. Is inter-nest reuse difficult to exploit?
4. Many memory references within numerical codes correspond to regular references [BC91].	a. What is the fraction of misses due to scalar references? b. The most commonly used stride value is 1. c. Loop nest structures are mostly rectangular and triangular.

tions, we examine the relevant assertions and mention related work. When each assertion is encountered for the first time, we further justify it. At the end of each section, we summarize the results with respect to the relevant assertions.

2 Experimental Framework

In this section, we delineate loop nests and locality. We describe the programs we used, their basic characteristics, and how we instrumented them to obtain our results.

2.1 Loop Nests

We chose to measure *intra-nest* locality in loop nests that are at most 3 deep, without calls, and with only one loop at each level. (Note the Perfect Benchmarks do not make use of mathematical libraries.) The loops need not be perfectly nested, i.e., there can be statements between nesting levels, as long as the intervening statements do not affect control flow. We apply these restrictions to try to select the nests usually targeted by software and hardware optimizations. Section 2.3 shows that this selection of nests is comparable to published targets of nest optimizations and finds the nests responsible for the majority of references in the programs. As an example, consider the nests in Figure 1. Given these loops, we collect statistics on the three inner nests: *Matrix-Vector Multiply*, *Matrix-Matrix Multiply*, and *Stencil*. Because the outer L loop contains multiple loops at the same nesting level, it is not included in the intra-nest measurements and the loop is split into the 3 nests.

The program trace then becomes a set of **in-nest** statement executions separated by **out-nest** statements. We define **intra-nest** events as ones that occur within a single execution of a nest. For example, if the same block is referenced more than once during a single nest execution, we call all these references but the first one **intra-nest** locality. We thus exclude the first reference to a block from the intra-nest statistics. Any locality the first reference incurs is inter-nest locality. **Inter-nest** events are between different executions of nests, including different executions of the same nest. **Program** locality, of course, includes all the references in the program.

Figure 1: A Few Classic Examples

```
DO L = 1, N
                    Matrix-Vector Multiply
    DO I = 1, N
      DO J = 1, N
        C(I) = C(I) + A(J,I) * D(J)
      ENDDO
    ENDDO
                    Matrix-Matrix Multiply
    DO I = 1, N
      DO K = 1, N
        DO J = 1, N
          X(J,I) = X(J,I) + Y(J,K) * Z(K,I)
        ENDDO
      ENDDO
    ENDDO
                          Stencil
    DO I = 1, N
      DO J = 1, N
        A(J,I) = A(J,I+1) + B(J,I) + B(J+1,I)
                 + E(I,J) + E(I+1,J)
      ENDDO
    ENDDO
ENDDO
```

Examples. Consider the array A in *Matrix-Vector Multiply* and in *Stencil*. Assume no interference in the cache from the other arrays accessed in the L loop, A is not originally in the cache, that R elements of A fit on a cache block, and $N > R$. On the first iteration of the L loop, *Matrix-Vector Multiply* misses once every R iterations of the J loop, and then hits on the intervening R-1 accesses, yielding intra-nest spatial reuse. If A is still in the cache when *Stencil* executes, these hits are inter-nest and temporal. If on the second iteration of L, *Matrix-Vector Multiply* still finds A in the cache this reuse is also inter-nest temporal reuse for all N^2 references to A. The intra-nest locality of A on the second execution of *Matrix-Vector Multiply* is the same same as it was on the first execution of the nest, intra-nest spatial locality for $\lfloor (R-1)/R * N \rfloor$ references. This example illustrates why even though most references occur in loops, intra-nest, inter-nest, and program characteristics may differ significantly.

2.2 Classifying Locality and Reuse

The classical definitions of locality properties found in programs are: *temporal locality – if an item is referenced, it will tend to be referenced again soon*; and *spatial locality – if an item is referenced, an adjacent item will tend to be referenced soon* [HP95]. Given a reference, cache designers exploit temporal locality by placing the referenced word into the cache, and spatial locality by using a cache block size greater than one word that places adjacent words in the cache at the same time. References with *locality* thus have the *potential* for reuse in the cache. Reuse is simply a *hit* in the cache that achieves its locality. References with locality can also *miss* in the cache. Unfortunately, we cannot directly use the definitions of temporal and spatial locality to measure which of the properties the cache best exploits.

Figure 2 illustrates our new classification system. This classification enables us to measure locality properties with respect to a given cache organization in terms of individual references in programs. Given a reference, it either hits or misses in the cache. If it hits, the word is in the cache and a previous reference either accessed the same word (**temporal reuse**) or another word in the same cache block (**spatial reuse**). If a reference misses and the cache previously contained the word, the cache is not exploiting locality and there is potential for improvement. A **temporal miss**

Figure 2: Locality Classifications

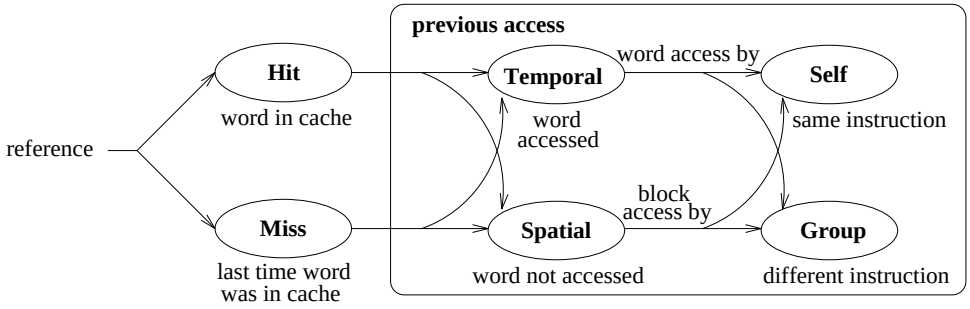


Table 2: Test Suite Characteristics

Code	% References In-Nest	% References Out-Nest	Percent of Misses In-Nest	Percent of Misses Out-Nest	Number of Data References f=full, p=partial	Program Working Set Size in 4 byte words	Instrumented Nests Total	Instrumented Nests at depth 1	2	3	Cache Miss Rate 8K, 32B block direct-mapped
ADM	88	12	90	10	918,546,427 (f)	29,753	159	105	36	18	5.6
QCD2	39	61	77	23	508,546,774 (f)	463,133	90	80	6	4	1.1
BDNA	95	5	98	2	2,331,001,541 (f)	429,217	154	133	18	3	4.2
OCEAN	99	1	99	1	4,000,000,000 (p)	156,688	76	35	41	0	6.2
DYFESM	93	7	84	16	452,663,362 (f)	28,409	108	60	42	6	3.2
ARC2D	99	1	99	1	4,000,000,000 (p)	2,001,043	141	76	59	6	9.9
FLO52	98	2	99	1	956,221,083 (f)	214,622	87	39	39	9	6.8
TRFD	96	4	89	11	1,429,808,439 (f)	1,354,205	29	13	9	7	5.0
average	88.4	11.6	91.9	8.1							

occurs if the last time the cache contained the word, the word was referenced. A **spatial miss** occurs if the last time the cache contained the word, the word was not referenced. For either a hit or a miss, if the previous reference to the word or cache block came from the same instruction, we call it **self-locality**. If it came from a different instruction, we call it **group-locality**.

Notice that (1) given a one word cache block, references with temporal reuse would still be classified as temporal; and (2) spatial reuse is a function of the cache block size. It is not obvious how other changes to associativity, cache block size, and cache size affect this classification. In this study, we focus on a single cache organization and do not examine how varying cache parameters changes our locality classifications.

To discover why a reference misses in the cache, we use Hill's miss classification which is orthogonal to the above: **compulsory misses** – misses that occur on the first reference to a block; **capacity misses** – additional misses resulting from the limited capacity of a cache; and **conflict misses** – additional misses due to mapping constraints in set-associative caches [Hil87, HS89]. Like Hill, we measure conflict and capacity misses with respect to a fully-associative cache using an LRU replacement policy.

We measure the **locality distance** in terms of the number of memory references (load/store references in the trace) between two references to the same word or cache block. We chose this metric because it is more context-independent than the number of cycles.

Examples. To measure *intra-nest spatial reuse*, we consider only the load/store references in a single execution of the nest. If the nest references a word which has not yet been referenced, but the nest has previously accessed the cache block on which the word resides, the nest exhibits intra-nest spatial reuse. If the reference was from the same instruction, we further classify the locality as self-spatial reuse. Consider the reference to `Y` in Matrix-Matrix Multiply in Figure 1. It has self-spatial locality on the inner loop. The first reference to `X` has group-spatial locality with the second reference to `X`. (See Section 4.1 for additional examples.)

2.3 Test Suite

For our test suite, we used 8 of the Perfect Benchmark programs which range in number of non-comment lines from 485 to 6105, averaging 3509 lines [CKPK90]. Although the Perfect Benchmarks suffer from the same flaw as SPEC92 [Uni89], namely small data set sizes, they are real applications, and thus are more likely to exhibit classic programming patterns. For each benchmark, Table 2 presents the percent of references and misses in nests and out of nests; the total number of references; the working set size of the program; the number of instrumented loop nests partitioned into nests of depth 1, 2, and 3; and the program miss rate for an 8-Kbyte, 32-Byte block size, direct-mapped data cache (the DEC 21164 [Dig94] first-level data cache implements these parameters). These miss rates vary from 1.1 to 9.9%. We selected the smallest cache parameters available in current processors to keep the ratio of data set size to cache size as close as possible to that of real programs run on future-generation processors. All of our figures use this basic configuration. We also discuss a few results for a 2-way set-associative version of the same cache (the Intel P6[Res95] first-level data cache implements these parameters).

Our nest selection, on average, considers more nests than Carr et al. [CMT94] consider for optimization. The numbers are not directly comparable because Carr et al. do not include single loops and consider more complex nesting structures than we do. In 7 of the 8 programs, 88% or more of the references occur within the nests we instrumented. In QCD2, only 39% references are in nests. The reasons for this low number are essentially (1) references not considered because of the limitations of the instrumentation, (2) loops programmed with `GOTOs` and counters, and (3) loops with call statements where the called subroutine only contains the loop body and is thus not instrumented. We include QCD2 because 77% of misses are within the nests we instrument. Although we collected statistics for MDG, we exclude them from our statistics because less than 50% of references and misses are in nests.

2.4 Instrumentation and Analysis

We modified the *Spy* tracing tool from the *Spa* package by G. Irlam [Irl91] to trace each benchmark on a Sparc-20 workstation. *Spy* uses object codes as inputs so no special compilation flag is required. *Spy* traces all calls to system libraries but doesn't handle a number of traps. Six of the eight benchmarks were run till completion. The trace length was limited to 4 billion entries for two codes (OCEAN and ARC2D). We compiled the benchmarks with Sun's F77 compiler using `-O2` which includes all optimizations except optimizations on global variables and loop unrolling. We disabled loop unrolling because it obscures self and group locality.

We instrumented the programs with Sage++, a source-to-source Fortran compiler [BBG+94]. Our Sage++ routine detects loop nests as described in Section 2.1 and inserts variables and subroutine calls to uniquely identify nests. Spy catches the instrumented subroutine calls. Since the subroutine calls are outside of the nests and contain very few assembly instructions, the trace perturbation is negligible. We also used Sage++ to identify arrays and scalars based on their declarations. We collected locality statistics about each word, block, and reference (loads and stores). Since we record several pieces of information about each word and reference, the working set size of the analyzer is several times that of the traced code.

3 Explanation of Figures

All the figures in this paper present average statistics over the different benchmarks. We did not weigh the averages in order to balance the impact of each benchmark. Figures 3 thru 8 plot the fraction of reuse (Figures 3, 4, and 5) and misses (Figures 6, 7, and 8) as a function of the locality distance in $\log_2$ between references to the same word or cache block. (Figures 9 thru 14 use the same x-axis.) Each bar partitions the references into four locality groups: self-spatial (white), group-spatial (light gray), self-temporal (dark gray), and group-temporal (black). These categories reveal which type of locality is responsible for hits and misses and if these references come from the same (self) or a different (group) instruction. For a given distance 2^i, a box encloses the fraction of references which incurred the locality between the distances 2^{i-1} and 2^i. The unboxed portion represents the fraction of references incurring locality between 1 and 2^{i-1}, the cumulative locality. Since all categories of locality may increase at each distance, we plot both the increment and the accumulated portions to make comparisons between distances easier.

Comparing Figures 3, 4, and 5 reveals the differences between intra-nest, inter-nest, and program reuse. Compare Figures 6, 7, and 8 similarly for misses. The differences in locality between reuses and misses are demonstrated by comparing pairwise between Figures 3 and 6 for intra-nest locality; Figures 4 and 7 for inter-nest locality; and Figures 5 and 8 for program locality. Section 4 and Section 5 are organized around this later comparison and Section 6 focuses on load/store instruction locality.

4 Intra-Nest Data Locality

In this section, we focus on *intra-nest* references and explore their behavior in the context of Assertions 1 and 2 from Table 1.

Probably the most widespread assertion we address is Assertion 1: *Spatial reuse is the dominant form of reuse* [CMT94, Jou90, KW73, PHH88, Smi82, Smi87]. For example, all modern caches use block sizes greater than one. Spatial locality enables hardware and software prefetching to achieve most of its improvements [CB95, Dra95, Jou90, KL91, MLG92]. Software techniques have also attributed their improvements to improved spatial locality [CMT94].

4.1 Intra-Nest Reuse

Figure 3 shows that on average, only 40% of intra-nest reuse is spatial. QCD2, TRFD, and DYFESM achieve a majority of spatial reuse (51%, 56%, and 61%, respectively), but FLO52, ARC2D, ADM, OCEAN, and BDNA achieve a majority of temporal reuse (61%, 61%, 66%, 68%, and 87%, respectively). These results also hold for 2-way set-associativity and are in contrast to Assertion 1 and 1.a since spatial reuse is never the single overwhelming factor.

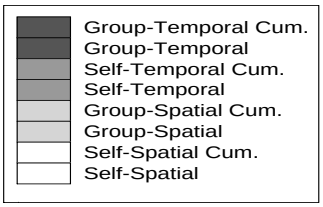
Compiler algorithms to improve locality target group and self, spatial and temporal locality [CMT94, TGJ93, WL91]. All of which have a role in these results. Two classic examples in Figure 1 that demonstrate a mixture of self-temporal, group-temporal, self-spatial, and group-spatial reuse are *Matrix-Vector Multiply* and *Matrix-Matrix Multiply*. Both are written in the best order for exploiting short-term data locality, assuming Fortran's column-major order [CMT94, WL91]. In *Matrix-Vector Multiply* on the inner `J` loop, the cache should exploit group-temporal locality for `C`, self-spatial for `A`, and self-spatial for `D`. For the entire nest, `C` is group-spatial and temporal, `A` is self-spatial, and `D` is self-spatial and temporal. In *Matrix-Matrix Multiply*, the cache should exploit self-temporal and self-spatial locality for all arrays. In *Stencil*, `A` and `B` exhibit self-spatial and group-temporal locality, and `E` exhibits group-spatial and group-temporal locality. While compiler optimizations often use classic examples like *Matrix-Vector Multiply* that suggest self-temporal reuse is significant, our results show that 86% of intra-nest temporal reuse is group-temporal reuse. In Figure 3, half of group-temporal reuse occurs at short distances, like `X` in *Matrix-Matrix Multiply*. Except for a concentration of group-temporal reuses at short distances, the reuse distance for group-temporal reuse is distributed relatively evenly. The group-temporal reuse distance for references like `A(J,I)` and `A(J,I+1)` in *Stencil* is the number of load/store references in the inner loop. Longer distances may be achieved for a complete execution of inner loops. These three factors contribute to the relatively even distribution of group-temporal reuse distances.

4.2 Intra-Nest Misses

4.2.1 Ping-Pong

Comparing Figure 3 and 6 reveals one impact of spatial locality: spatial misses are the dominant source of misses. QCD2 and TRFD are completely dominated by spatial misses. The other benchmarks, BDNA, ADM, OCEAN, DYFESM, ARC2D and FLO52 have more similar proportions of all locality types for both their reuses and misses, but all have more spatial misses than reuses. The misses are spread out over distances ranging from 1 to 2^{28}. Without loop nest optimizations, spatial locality probably cannot be exploited for reuse distances greater than 1024 (2^{10}). Loop nest optimizations have demonstrated the ability to turn intra-nest spatial locality with long distances into spatial reuse with short distances in ARC2D. Most misses in the original program are at a distance of 2^{10} or greater. Loop interchange and fusion improve ARC2D's performance by a factor of 2.15 on an IBM/RS6000 [CMT94]. However, a significant portion of intra-nest self-spatial misses occur for very short reuse distances. A given load/store instruction is thus not able to exploit all the spatial reuse within a cache block before a miss occurs due to another load/store instruction that uses the same cache block for other data. This phenomenon is called *ping-pong*. In OCEAN, TRFD, DYFESM, and QCD2, 23%, 25%, 33%, and 95%, respectively, of all intra-nest misses correspond to spatial locality with a distance of only 8 references or less.

The classic case of ping-pong interference is when the beginning of two arrays fall into the same cache block. References to these arrays in the same nest with identical linear subscript expressions will then ping-pong during execution. The probability that the first elements of two arrays fall in the same cache block is low, about $1/(\textit{number of cache blocks})$, suggesting ping-pong does not occur frequently (Issue 2.a). We found however that ping-pong between two arrays occurs repeatedly, but for short periods of time. The recurrence of the same ping-pong sequences may result in very irregular miss distributions across cache blocks: a high of 6% of misses in DYFESM are concentrated in a single cache block.



Legend for Figures 3 thru 8

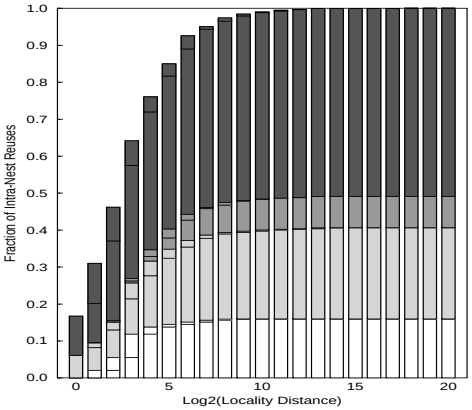


Figure 3: Distribution of Intra-Nest Reuse Locality

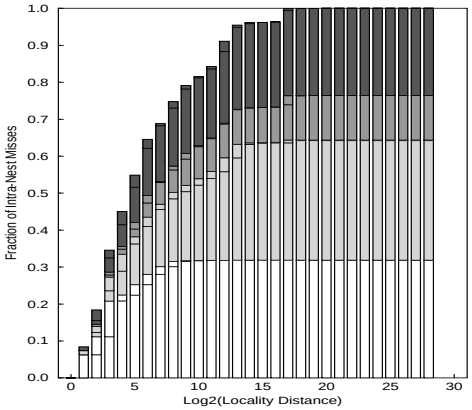


Figure 6: Distribution of Intra-Nest Miss Locality

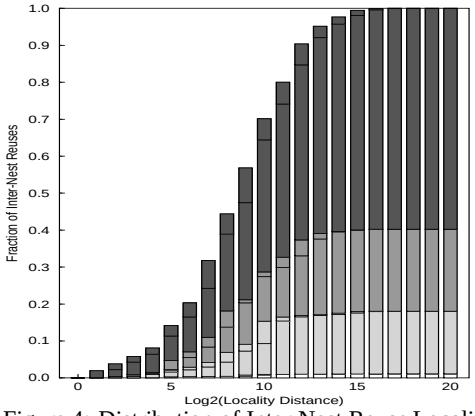


Figure 4: Distribution of Inter-Nest Reuse Locality

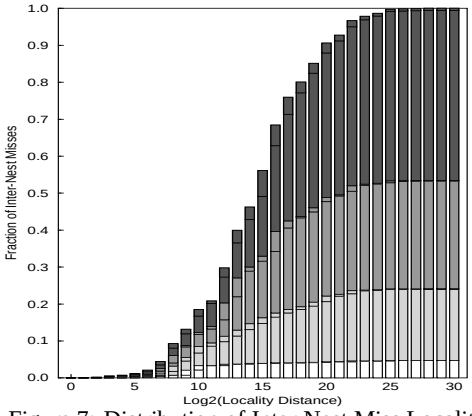


Figure 7: Distribution of Inter-Nest Miss Locality

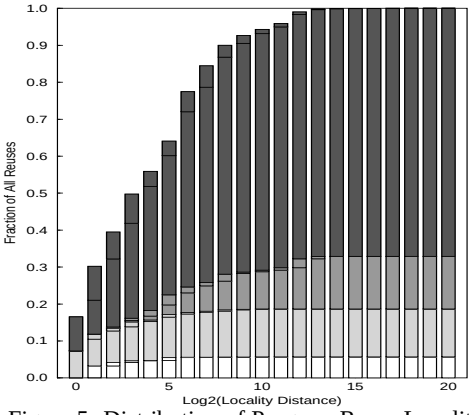


Figure 5: Distribution of Program Reuse Locality

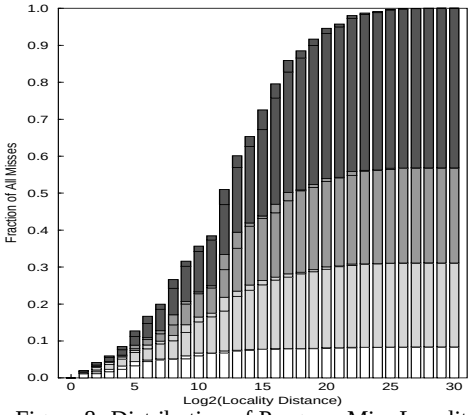


Figure 8: Distribution of Program Miss Locality

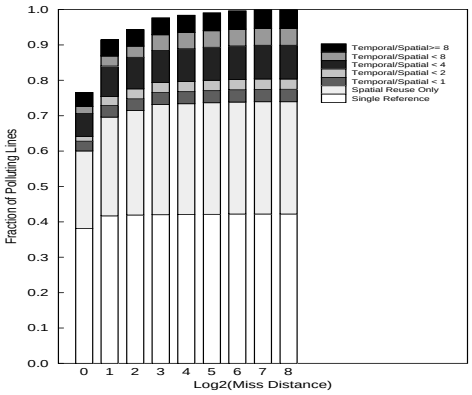


Figure 9: Locality of Polluting Lines

Looking at the program source reveals that array subscripts are often identical, but that the array sizes differ. During the execution of a loop, their relative position changes. Consider the references `A(J,I)` and `B(J,I)` declared as `A(100,100)` and `B(150,150)`. On each iteration of `I`, the relative cache position of `A` and `B` is $(a + 100*\mathtt{I})\ mod\ \mathtt{C}_S - (b + 150*\mathtt{I})\ mod\ \mathtt{C}_S$ where a and b are the array base addresses and $\mathtt{C}_S$ is the cache size in words. While `A` and `B` may not interfere initially, eventually one or more iterations of the `I` loop map them to the same cache block, thus inducing ping-pong misses. The 2-way set-associative 8-Kbyte cache eliminates most of this ping-pong effect and the resulting spatial misses at short reuse distances.

4.2.2 Pollution

References with spatial locality across outer loops also cause spatial misses, for example, `E(I,J)` where `I` is the outer loop and `J` the inner loop in *Stencil*. The spatial locality occurs after a complete execution of the inner loop. Since the spatial locality is far apart, it is difficult to exploit. As we discussed in Section 4.2, loop interchange can solve this problem for some nests [CMT94].

We found such references to be responsible for a high share of global pollution, in line with Assertion 1.c: *Most cache pollution is due to spatial-only blocks*. This assertion arises from the assumption that references that only exhibit spatial locality like `A(J,I)` in *Matrix-Vector Multiply* are the culprits in pollution. The *Assist-Cache* of the HP-7200 [Pou94] and stream buffers [MW96] use cache bypass to avoid pollution from blocks with only spatial locality. In Figure 9, we measure polluting blocks within nests by evaluating the ratio of temporal to spatial reuses for each block upon its eviction from cache. This ratio defines the 7 classes indicated in Figure 9. Between the time a block is flushed and it is reused again, we record the number of blocks in each class that *polluted* the locality, i.e., that were loaded in the same cache block between the two references. Figure 9 accumulates these statistics over all cache lines and all reuses. For example, the temporal/spatial classes demonstrate that less than 30% of cache blocks that pollute the cache are reused both spatially and temporally (a temporal/spatial ratio greater than 0).

In line with Assertion 1.c, very few polluting blocks exhibit numerous temporal reuses. But the very high fraction, about 40%, of polluting blocks that have no spatial reuse (only a single word is used) is surprising. (This result is consistent with the intra-nest temporal locality demonstrated in Section 4.1.) When pollution is measured over all the references in the program, then the ratio of polluting blocks with no spatial reuse is smaller, but still remains high at 20% (and for 40% of blocks, all words are not used). These statistics contradict Assertion 1.b. An important corollary to this observation is that cache blocks are not efficiently exploited and many loaded words are not used, wasting cache bandwidth. Other works support these results [BKG96, TFMP95, WHK91]. Future architectural and software improvements to cache performance may therefore need to selectively load the cache and/or use adjustable cache block sizes.

4.2.3 Capacity and Conflict Misses

Previous research demonstrates Assertion 2: *Capacity misses occur more frequently than conflict misses, and both are significant sources of misses* for whole programs [HP95, HS89, SA93]. Conflict misses play the lesser role in these results, typically making up about 30-40% of misses. Figure 12 divides program misses into self and group,[1] capacity and conflict misses, again plotting the fraction of misses against their distance. **Self-conflict** misses occur when both the reuse and the pollution results from different memory references from the same load/store instruction interfering in the cache, i.e., mapping to the same block. Such conflicts can occur more frequently when loop boundaries are different from array boundaries. **Group-conflict** misses occur when a different load/store instruction causes pollution. The graph partitions capacity misses similarly. Figure 10 provides the same statistics for intra-nest misses, and Figure 11 illustrates inter-nest misses.

Perhaps the most dramatic difference between intra-nest statistics and previous work on whole programs is that on average, more than 80% of intra-nest misses are *conflict* misses (100% are conflict misses for ADM, QCD2, DYFESM and OCEAN). Only in ARC2D did capacity misses exceed conflict misses (77% vs 23%). The conflict locality distances are short, suggesting they could be avoided. On average, conflict misses reduce to 68% of intra-nest misses with 2-way set-associativity, but are still a significantly larger fraction than the average of 30 to 40% conflict misses on whole programs for direct-mapped caches in Figure 12 and other studies [SA93, HP95].

Considering that these programs have relatively high hit rates (see Table 2), this result should not have been surprising. The working set of most executions of a loop nest should fit and already be in the cache, implying inter-nest misses should be conflict misses. We confirmed this result by measuring nest working set sizes. The vast majority of intra-nest misses are in nests with working set sizes of only 32 to 512 words. In QCD2 and TRFD, several nests had large working set sizes, about 256 Kbytes, but these misses barely contribute to the total number of intra-nest misses.

Several software techniques focus on selecting tile sizes that eliminate capacity misses and do not introduce self-interference misses [CM95, LRW91]. These tiling studies focus on kernels, and have yet to demonstrate effectiveness on complete applications. The lack of capacity misses in Figure 10 indicates that the data set sizes for these programs are too small to make tiling effective. Wolf supports this conclusion as he was unable to achieve improvements from tiling SPEC92 [Wol92]. Larger data set sizes would probably result in increased capacity misses that may be eliminated by tiling. Even if tiling increases self-conflict misses, it is also clear from Figure 10 that eliminating group-conflict misses is also important. Group-conflicts are however much more difficult to analyze and reduce [TFJ94]. Hardware mechanisms like victim caches [Jou90] or higher set associativity may be more likely to eliminate these misses (see Section 5.1.1).

4.2.4 Miss Characteristics of Individual Nests

We also measured how the individual nests contribute to the overall misses. Misses are not concentrated in a few nests, but rather distributed over 10 or more nests in all benchmarks, with 90% of misses requiring 40 nests in FLO52. We also found that the miss rates of nests that induced the most raw misses could vary from a high of 30% in ADM to a low of 10% in TRFD. To determine if optimizations can have the same effect across different executions of the same nest, we measured nest miss variations. Nest misses proved to be very stable. 90% of nests have nearly the same number

[1] We use group-conflict instead of the more frequently used term *cross-conflict* for consistency with the rest of the paper.

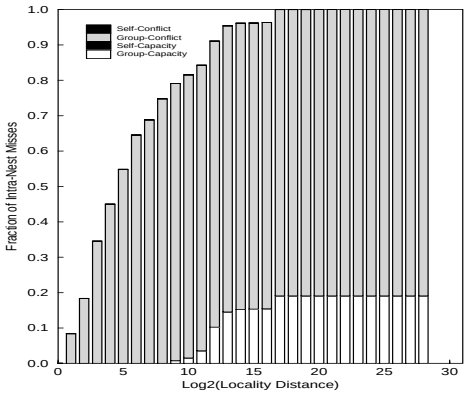


Figure 10: Capacity/Conflict Intra-Nest Misses

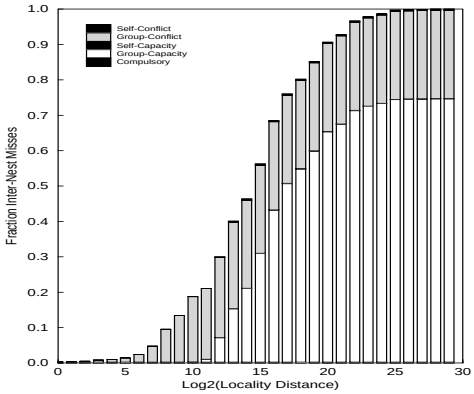


Figure 11: Compulsory/Capacity/Conflict Inter-Nest Misses

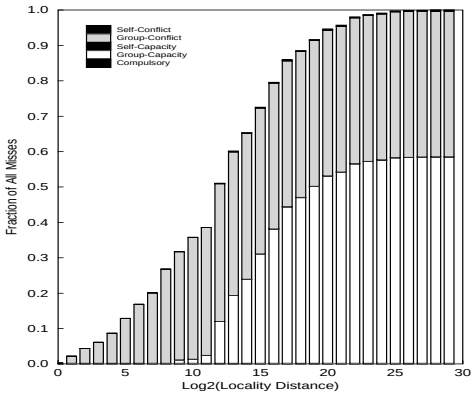


Figure 12: Compulsory/Capacity/Conflict Program Misses

of misses on every execution, and the other 10% only have small variations, possibly because loop bounds are changing.

Another side-effect of the small working set sizes in nests is that each nest only uses a small part of the cache (see Section 4.2.3). The cache is thus often underutilized within a nest, and many blocks reside in cache even though they are dead and will not be used again. These blocks often have long *agonies*, i.e., they stay in the cache a long time after they die, and on average 60% of dead blocks are replaced by blocks from a different nest. Long cache block agonies have also been observed by Wood et al. [WHK91] and Burger et al. [BGK95].

4.2.5 Summary of Results

The major findings of this section refute Assertions 1 and 2. Instead of a preponderance of spatial reuse, we find a more balanced role between spatial and temporal reuse (Assertion 1.a), particularly group-temporal reuse. The majority of intra-nest misses are spatial, and tended towards short reuse distances for the direct-mapped cache. The high number of blocks with few spatial or temporal reuses reveals that cache blocks are poorly exploited (Assertion 1.b). Cache blocks that are only referenced once or a few times also significantly pollute the cache and are responsible for many misses (Assertion 1.c). We find a large fraction of intra-nest misses are group-conflict misses, in contrast to previous work on complete applications. This result is consistent with higher hit rates, since programs that fit in the cache will have relatively few capacity misses, and capacity misses will thus increase in importance. Ping-pong occurs relatively frequently (Assertion 2.a) and accounts for some conflict misses. Spatial misses with large strides are also a source of conflict misses. To use the cache more effectively will require new hardware, software, and combined solutions.

5 Inter-Nest and Program Data Locality

In this section, we investigate inter-nest locality, locality that occurs across outer nests and between loop nests. We explore Assertion 3 and compare intra-nest, inter-nest, and program behavior.

.

5.1 Inter-Nest and Program, Reuse and Misses

The vast majority of hardware and software techniques to improve cache performance target individual nests. This strategy implies Assertion 3: *Most reuse occurs within a nest rather than across nests*. Most researchers correctly state that majority of execution time is spent in loops which is subtly different from Assertion 3. Figure 13 and 14 partition misses by load/store distances and indicate if the locality is intra-nest or inter-nest. The boxes represent the increment added for the corresponding distance. It confirms Assertion 3, finding that 93% of hits are intra-nest, and 7% are inter-nest. ARC2D, OCEAN, BDNA, and FLO52 all achieve 100% intra-nest reuse. McIntosh et al. confirm this result for small caches, and show when caches are much larger and hit rates drop, inter-nest reuse is more prevalent [CKM95].

Although most reuse is intra-nest (Figure 13), more than 70% of misses are inter-nest (Figure 14). To decrease these misses, optimizations cannot simply focus on nest optimizations but need to consider more than one nest, in contrast to Assertion 3.a. An inter-nest miss is a miss on a block that was previously referenced in another nest. The natural question posed by Assertion 3.b is how far apart in the program are these two nests? The further apart they are, the harder it will be to optimize them in concert. Figure 15 illustrates the fraction of inter-nest misses as a function of *nest distance*, the number of nest executions since the last reference to the data. Only 57% of inter-nest locality is exploited, but fortunately, more than 70% inter-nest misses correspond to short locality distances of at most 4 adjacent nests, and 50% of inter-nest misses are between adjacent nests (*nest distance* = 1). (Since these are nest executions, they could be the same nest.) This result implies fusion and other cross nest optimizations have the potential to reduce misses. McIntosh et al. [CKM96] propose an inter-nest reuse analysis to exploit locality, but have not yet implemented it.

Figures 4, 5, 7, and 8 continue the trend away from spatial locality that we saw in Section 4.1 for intra-nest reuse. Spatial locality accounts for only 14% of inter-nest reuse. Nests are unlikely to achieve to achieve much spatial reuse since that would require one nest to access part of a cache block and subsequent nests to access other elements in the block. Group-temporal locality accounts for 58% and self-temporal for the other 28% of inter-nest reuse. Since a nest of depth 3 can be surrounded by a loop, reuse from one execution to the next can result in self reuse. For example in TRFD, 60% of reuses are self-temporal reuses because one large loop surrounds collections of 1 and 2-deep loop nests in the subroutine OLDA and elsewhere. Such constructs may be easy targets for inter-nest optimizations to reduce inter-nest misses.

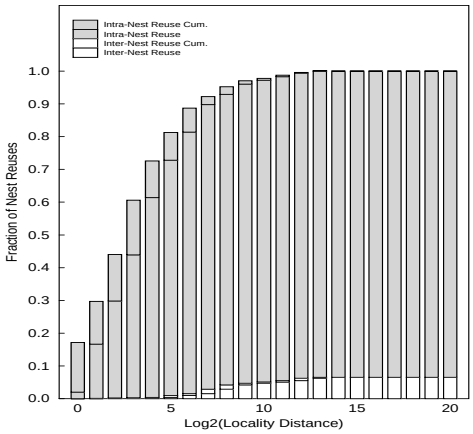


Figure 13: Intra-Nest versus Inter-Nest Reuses

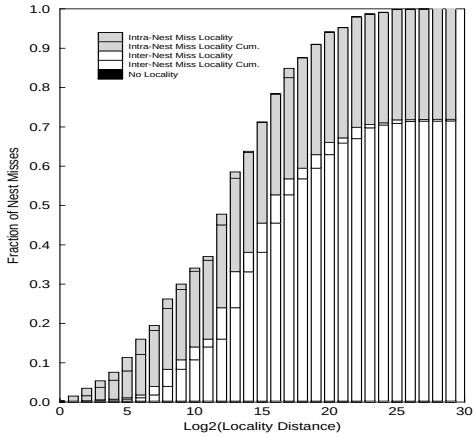


Figure 14: Intra-Nest versus Inter-Nest Misses

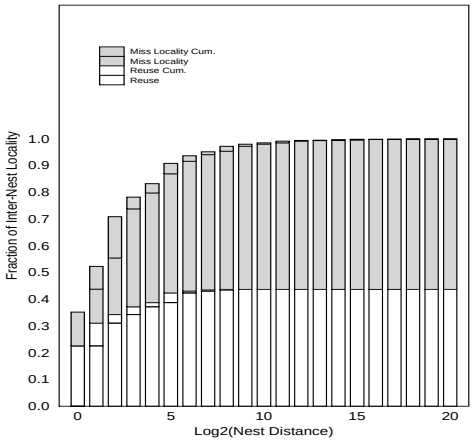


Figure 15: Nest Distance of Inter-Nest Locality

Similar to inter-nest reuse, program reuse is 86% temporal (Figure 5), with highs of 96% temporal reuse for QCD2 and 94% for ADM. The inter-nest and program reuse results are consistent with relatively high hit rates. The reader might expect that Figure 5 would be a combination of Figures 3 and 4, weighted by their relative importance from Figure 13, but it is not. As discussed in Section 2.1, intra-nest reuse does not include hits due to references from outside the loop; inter-nest reuse captures just the reuse due to shared accesses from different nest; and program reuse includes every reference in the program. These measurements are thus different views of the same references.

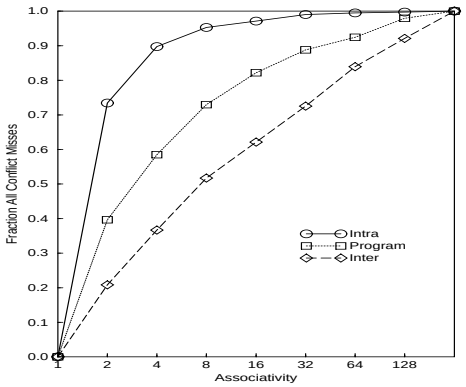


Figure 16: Set-Associativity that Eliminates Conflict Misses

5.1.1 Capacity and Conflict Misses

Figures 11 and 12 demonstrate that the fractions of capacity misses are higher for program misses and inter-nest misses than for intra-nest misses. Inter-nest capacity misses range from 20% to 100%, averaging 74%. Inter-nest misses are mostly capacity misses because the locality distances are large and there is a high probability that the number of words reloaded between two reuses is greater than the cache size. In other words, the working set size of a single nest tends to be small enough to fit in cache while the working set of multiple nests often exceeds cache size. To further understand the nature of capacity/conflict misses, we determined the minimum cache size necessary to remove each capacity miss that occurs in an 8-Kbyte direct-mapped cache with a 32-byte block, and the minimum associativity required to remove each conflict miss that occurs in the same cache for intra-nest, inter-nest, and whole program misses (Figure 16).

Our results show that doubling the cache to 16 Kbytes removes 40% of program capacity misses. Further gains are more difficult to achieve; a 128-Kbyte cache is required to achieve an additional 40% gain. As expected, a reasonably sized cache (32 Kbytes) eliminates most intra-nest capacity misses. However, even a large cache size of 128 Kbytes leaves 20% of inter-nest capacity misses for software techniques to attack.

Figure 16 plots the fraction of conflict misses removed by different set-associativities of the base cache. A 2-way set-associative cache removes 73% of intra-nest conflict misses and 90% with a 4-way cache, thus supporting Assertion 2.b. On the other hand, a 16-way set-associative cache eliminates about 65% of inter-nest conflict misses. In contrast to Assertion 2.b, 2-way set-associativity removes only 39% of program conflict misses; 16-way set-associativity is necessary to remove about 80% of conflict misses.

5.2 Summary of Results

In this section, we confirmed Assertion 3; most reuse is intra-nest. We find that most misses are however inter-nest, suggesting new optimizations should target locality across nests (Assertion 3.a), and that high associativities are sometimes required to eliminate a majority of conflict misses (Assertion 2.b). Inter-nest misses often occur between adjacent nests so it may be possible to remove some of them (Assertion 3.b). The lack of intra-nest capacity misses is consistent with the high hit rates of these programs. The individual nests do not run out of cache, but multiple executions of the same nest or multiple nests do result in capacity misses. We show that the locality patterns between intra-nest, inter-nest, and program locality are usually very different.

6 Load/Store Locality

Hardware and software optimizations target not only nests, but the load/store instructions within nests. For instance, prefetching tables [CB95] analyze the stride of each load/store instruction within nests. Compilers of course translate array references into load/store instructions. Optimizing compilers may expand array references into several load/store instructions. Other advanced optimizations, like scalar replacement combined with unroll-and-jam [CCK90, CK94], collapse several array references into a single load/store. Even with these caveats, load/store statistics tend to reflect the behavior of array references.

6.1 Stream Locality

6.1.1 Stream Reuse

Probably the most classic characteristic of dense numerical codes is embodied in Assertion 4: *Many memory references within numerical codes correspond to regular references*, since load/store instructions for array references typically reference data with a constant stride. For this reason, numerical codes are often called *regular codes*. A load/store instruction induces a *regular reference* if the distance between 3 consecutive references (2 strides) is identical (the hardware characterization proposed by Chen and Baer [CB95]). A set of regular references using a constant stride is called *a stream*. The stream ends with the nest execution or when the stride changes. Figure 17 quantifies the fraction of references that belong to streams. There are three types of references: *scalar references* – the load/store instruction references the exact same address throughout program execution; *stream references* – the load/store instruction accesses addresses that differ by a constant stride (for the moment, ignore the different stream types in Figure 17); and *non-stream references* – two consecutive executions of the load/store instruction have distinct strides. Stream references clearly dominate (95% of references in TRFD and 90% in DYFESM), confirming Assertion 4. In some programs, stream references are slightly less pervasive, 50% of references in ADM and 65% for FLO52. With respect to Assertion 4.a, the programs also contain a surprising number of scalar references (24% on average), with 34% for ADM and OCEAN, even though we compiled using F77's scalar optimizations. While this result may be due to register spilling, it may also simply be dependent on the pairing of the compiler and instruction set.

6.1.2 Stream Misses

Figure 18 divides the fraction of total nest misses (intra-nest and inter-nest misses) into stream, scalar, and other references. Misses are accumulated over all nest instructions which are sorted by decreasing number of total misses. These results confirm Abraham et al.'s [ASW+93] findings that misses are concentrated in particular load/store instructions. In addition, Figure 18 illustrates that the concentration is even higher for stream misses, misses occurring within streams. For instance, 27% of DYFESM's stream misses occur within a single instruction. For stream and non-stream misses, the address referenced changes on each new execution of the load/store instruction so it seems natural that the fraction of misses and references is correlated. With respect to Assertion 4.a, we did not expect to see such a high fraction of scalar misses, since scalar references correspond to reuse of the same word. For ADM and DYFESM, scalar references account for respectively 13% and 21% of all nest misses. The same concentration of stream misses in specific instructions is also found in scalar misses.

6.1.3 Types of Streams and Stream Strides

Figure 17 splits streams into 4 categories: *fixed-length streams* – the stride change is always the same (this behavior is typical of rectangular loop nests); *strided-length streams* – the stride change itself is regular (this behavior is typical of triangular loop nests); *irregular streams* – the stride change is irregular; and *single-length streams* – the stride never changes during the stream and the stream ends with the nest. Fixed-length and single-length streams typically attain spatial reuse on the inner loop. For example, the reference to `Y` in *Matrix-Matrix Multiply* in Figure 1 is a single-length stream if the leading dimension of `Y` is equal to the loop bound `N`, because the stride would never change even between two executions of the `J` loop. If the declaration differs, the stream is a fixed-length stream because stride will change every time the `J` loop ends. Single-length and fixed-length streams are dominant in Figure 17. For these numerical codes, this result confirms the first part of Assertion 4.c: *Loop nest structures are mostly rectangular and triangular.* Triangular loops are very infrequent: 2% of all nests in these codes, except for 10% in DYFESM. Irregular stream lengths are also extremely infrequent. Single-length streams represent 43% of all references. These references correspond to loop invariant references in nests where loop bounds match array bounds.

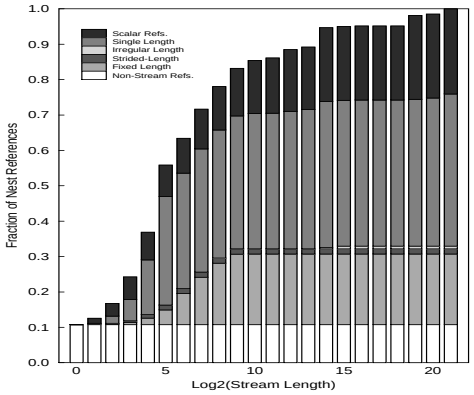


Figure 17: Stream Types, Lengths, and Frequencies

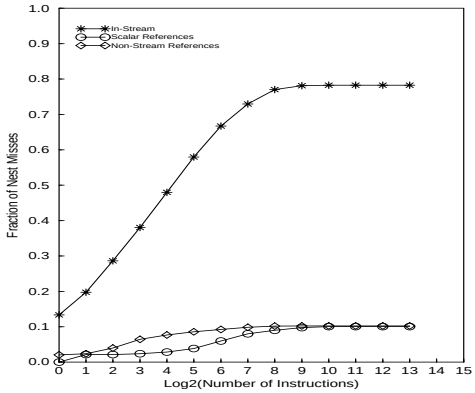


Figure 18: Stream, Scalar, and out of Stream Misses

Table 3 presents the stride distributions for each program and confirms Assertion 4.b: *The most commonly used stride value is* 1. The table includes the 5 most frequently used strides and its fraction of total stream references. Array element sizes in BDNA, TRFD, and ARC2D are double precision, and are complex in OCEAN, which results in the stride-1 value of 2 in Table 3. All other codes use single-precision, 1 word data. In QCD2, DYFESM, FLO52, ARC2D, and TRFD, stride-1 references account for 50% or more of all stream references (ranging from 29% for BDNA to 90% for FLO52). Aside from stride-1 streams, no other general rule on stride values across programs emerges. Stride values are however consistent *within* a program. Each program uses very few distinct strides. At most 5 different strides account for more than 95% stream references. For instance, 42% of stream references in QCD2 have a stride of 9. 18% of stream references in TRFD have a stride of 1640, and in ARC2D,

Table 3: Stride Values in Streams

Code	Stride Value (% of all stream references)											
ADM	0	(66.68%)	1	(33.32%)								
QCD2	1	(49.70%)	9	(41.96%)	0	(8.33%)						
BDNA	0	(55.54%)	2	(29.31%)	6	(12.35%)						
OCEAN	258	(50.90%)	2	(38.62%)	0	(7.67%)						
DYFESM	1	(72.75%)	45	(9.74%)	20	(7.18%)						
ARC2D	2	(68.01%)	578	(21.30%)	0	(10.41%)						
FLO52	1	(89.72%)	194	(8.99%)								
TRFD	2	(54.89%)	1640	(18.06%)	1260	(10.63%)	0	(6.12%)	930	(5.77%)		

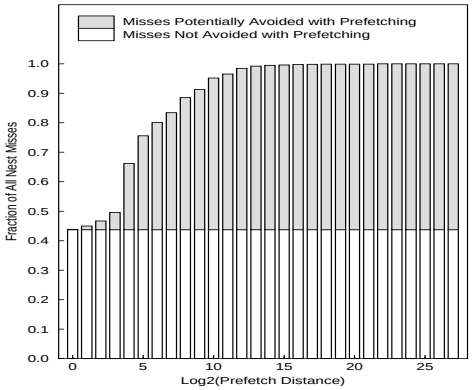


Figure 19: Potential Effectiveness of Prefetching

21% of stream references have a stride of 578. ADM has 67% null-stride references, though only 33% scalar references.[2] This result is due to array references that only vary according to outer-loop indices and are thus invariant in an inner loop. The small number of strides suggests load/stores could be tagged with a few bits to indicate which among the few specified strides should be used for prefetching. 2 bits are sufficient since there are usually less than 4 strides.

6.1.4 Stream Lengths and Prefetching

The x-axis in Figure 17 plots the number of references between stride changes. These stream lengths are quite short, often consisting of 16 to 32 references. Since many array declarations are much larger than these lengths, small data sets are not solely responsible for short stream lengths. Many linear algebra codes use sub-matrices of larger matrices. When a nest scans 2-D sub-matrices, the address of the last element of a row is not consecutive with the address of the first element of the next row. Therefore, the stride changes, the stream ends, and hence the short stream lengths. Short stream lengths can disrupt hardware prefetching. For instance in prefetch tables [CB95], prefetching stops if the stride changes and the instruction must make 3 new stream references to stabilize. Tagged prefetching fails to improve performance when a stride is larger than a cache block. (Strides that occur at the end of a stream are usually larger than a cache block.) Since 16 to 32 references to adjacent words correspond to 2 to 4, 32-byte cache blocks, tagged prefetching [Jou90] may fail every 2 to 4 prefetch requests. This result suggests prefetching should handle stride changes [MLG92] as well as stream references [CB95].

To determine the influence of short length streams and the potential of data tagged prefetching, we implemented a form of *virtual tagged prefetching*. Under the same conditions as tagged prefetching, on a miss or a hit of a prefetched block, the next block is marked as *virtually prefetched* but is not actually brought in cache. Thus, the potential of prefetching can be measured since virtual prefetching ignores cache side-effects such as flushing prefetched blocks too early, the pollution of useful data, coherence issues, and limitations due to the prefetch buffer size [Dra95]. In Figure 19, the fraction of intra-nest misses that can be potentially removed with prefetching is plotted as a function of the prefetch distance in number of references between the prefetch and the use. We observe that virtual tagged prefetching breeds an average of 43% useless prefetch requests (the data is already in the cache) with a high of 82% for ADM and a low of 27% for OCEAN. On average, tagged prefetching can potentially remove more than 57% intra-nest misses, as shown in Figure 19. Many of these misses are conflict misses (see Section 4). Figure 19 also quantifies prefetch distances in number of references and shows that half of useful prefetch requests are used many references later, which would require significant buffering.

[2] A stream can have a null-stride and not correspond to a scalar if the address referenced changes from one execution of the nest to another.

6.2 Summary of Results

In this section, we confirmed Assertion 4, but also found there were many short streams that require special care to exploit, and that the impact of scalars may not be negligible (Assertion 4.a). A more detailed study of stream length characteristics showed that triangular or irregularly-strided loops are very infrequent. Most loops are rectangular (Assertion 4.c). We also found many strides other than than stride-1 (Assertion 4.b), and that each program was characterized by a few specific stride values. Finally, we briefly evaluated the impact of these observations on data prefetching.

7 Conclusions and Future Work

In this paper, we demonstrate that nests differ from programs in some important locality characteristics, and that numerical codes are not yet as well understood as one might expect. We develop a new framework to quantify and measure program locality. We confirm and provide new insights to some popular assertions. For example, we confirm that most reuse occurs in nests, but most misses are actually inter-nest and the nests are nearby. We also confirm many references are in stream references of stride 1, and that other stride values vary across programs, but are consistent within a program. In addition, we present results that suggest several popular assertions are at best overstatements. For instance, we find spatial and temporal locality have balanced roles for intra-nest reuse, but temporal locality dominates inter-nest, and program reuse in our test suite. We find conflict misses the most significant source of intra-nest misses and 2-way set-associativity unable to resolve many of them. Our results are fairly consistent across our test suite, but we still intend to measure additional programs, including SPEC95, with larger data sets and a variety of cache organizations.

We pointed out several possible ways for improving cache performance. For instance, hardware and software could cooperate to use the cache more selectively with data bypass and prefetch; compilers should address inter-loop reuse; and compilers and architects still need to address the interference problem. In summary, there is still room to improve the efficiency and effectiveness of data caches.

Acknowledgments

We would like to thank Ron Cytron, Amer Diwan, Christine Eisenbeis, Mark Hill, Toni Juan Hormigo, Nathaniel McIntosh, Sharad Singhai, Darko Stefanovic, Chau-Wen Tseng, and the anonymous referees for their careful reading and suggestions.

References

[AP93] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[ASW+93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 28th International Symposium on Microarchitecture*, Austin, TX, December 1993.

[BBG+94] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Second Object-Oriented Numerics Conference*, 1994.

[BC91] J-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966.

[BGK95] D. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-puropose microprocessors. Technical Report 1216, Dept. of Computer Science, University of Wisconsin at Madison, January 1995.

[BKG96] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[CB95] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[CK94] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[CKM95] K. Cooper, K. Kennedy, and N. McIntosh. An empirical study of cross-loop reuse in the NAS benchmarks. Technical Report CRPC-TR95519-S, Center for Research on Parallel Computation, Rice University, October 1995.

[CKM96] K. Cooper, K. Kennedy, and N. McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1996.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[CKPK90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[CM95] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[CMT94] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[Dig94] Digital Equipment Corporation, Maynard, Massachussets. *Alpha 21164 Microprocessor, Hardware Reference Manual*, 1994.

[Dra95] N. Drach. Hardware implementation issues of data prefetching. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, pages 323–334, Barcelona, Spain, July 1995.

[GHPS93] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[Hil87] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Computer Science Dept., University of California, Berkeley, 1987. Available as Technical Report UCB/CSD 87/381.

[Hil88] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[HP95] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[HS89] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[Irl91] G. Irlam. *SPA package*, 1991.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a samll fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[KL91] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991.

[KW73] K. R. Kaplan and R. O. Winder. Cache based computer systems. *IEEE Computer*, 6(3):30–36, March 1973.

[LRW91] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[MW96] S. A. McKee and W. A. Wulf. A memory controller for improved performance of streamed computations on symmetric multiprocessors. In *Proceedings of the 1996 International Conference on Parallel Processing*, Honolulu, HI, April 1996.

[PHH88] S. Przybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 290–298, Honolulu, HI, June 1988.

[Pou94] D. Pountain. A different kind of RISC. *BYTE*, August 1994.

[Res95] MicroDesign Resources. Intel boosts Pentium Pro to 200 Mhz. *Microprocessor Report*, 9(17), November 1995.

[SA93] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, May 1993.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smi86] A. J. Smith. Bibliography and readings on cpu cache memories and related topics. *Computer Architecture News*, 14(1):22–42, January 1986.

[Smi87] A. J. Smith. Line (block) size choice for cpu caches. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[Smi91] A. J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, 1994.

[TFMP95] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun. A new approach to cache management. In *Proceedings of the 28th International Symposium on Microarchitecture*, Ann Arbor, MI, November 1995.

[TGJ93] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[Uni89] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[WHK91] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 79–89, San Diego, CA, May 1991.

[WL91] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol92] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.