Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks

KATHRYN S. McKINLEY
University of Massachusetts
and
OLIVIER TEMAM
Paris XI University

This article 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 at the nest level. 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 SPEC'95 and 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, although most reuse is within a loop nest, in line with popular assertions, most misses are internest capacity misses, and they correspond to potential reuse between nearby loop nests. In addition, we find that temporal and spatial reuse have balanced roles within a loop nest and that most reuse across nests and the entire program is temporal. These results are consistent with high hit rates (80% or more hits), but go against the commonly held assumption that spatial reuse dominates. 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.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Performance attributes*; *Measurement techniques*

General Terms: Measurement, Performance

Kathryn S. McKinley is supported by an NSF CAREER Award CCR-9624209. Olivier Temam is supported by DGXIII Esprit IV LTR Project MHAOTEU, No. 24942. This work is also supported by NSF Grant EIA-9726401, an NSF Infrastructure Grant CDA-9502639, DARPA Grant 5-21425, and Compaq. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

Authors' addresses: K. S. McKinley, Computer Science Department, University of Massachusetts, 140 Governor's Drive, Amherst, MA 01003-4610; email: mckinley@cs.umass.edu; O. Temam, LRI, Université de Paris XI, Batiment 490, 91405 Orsay Cedex, France; email: temam@lri.fr.

Additional Key Words and Phrases: Caches, locality, reuse, loop nest behavior, program behavior, scientific programs, spatial locality, temporal locality

1. INTRODUCTION

Because processor speed is increasingly outpacing memory speed, an enormous amount of research focuses on improving the cache behavior of numerical programs (for example, see Smith's bibliographies on hardware aspects of cache memories [Smith 1986; 1991] and compiler techniques that exploit cache memories [Ghosh et al. 1998; Lam et al. 1991; McKinley et al. 1996; Mowry et al. 1992; Wolf and Lam 1991]). 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 [Callahan et al. 1991; Chen and Baer 1995; Drach 1995; Klaiber and Levy 1991; Mowry et al. 1992]. Many researchers [Li and Pingali 1992; McKinley et al. 1996; Mowry et al. 1992; Wolf and Lam 1991] 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 sometimes exploit locality more selectively than previous caches [Hennessy and Patterson 1996; McKee and Wulf 1996]. For example, the HP-7200 [Hennessy and Patterson 1996] 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 [Hennessy and Patterson 1996; Hill and Smith 1989; Sugumar and Abraham 1993] which suggests that loop nest optimizations such as tiling [Gannon et al. 1988; Wolf and Lam 1991; Wolfe 1987] may be profitable. As we show, most of these misses actually correspond to locality across distinct nests, and thus tiling will not eliminate them.

In this article, 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

Table I. Assertions about Reuse Characteristics and Cache Behavior of Numerical Programs

Assertions		Related Issues, Assertions, and Questions	
1	Most reuse occurs within a nest rather than across nests [McKinley et al. 1996; Temam et al. 1993; Wolf and Lam 1991].	a	Targeting individual loop nests is sufficient.
		b	Is internest reuse difficult to exploit?
2	Spatial reuse is the dominant form of reuse [McKinley et al. 1996; Jouppi 1990; Kaplan and Winder 1973; Przybylski et al. 1988; Smith 1982; 1987].	a	Temporal reuse avoids fewer misses than spatial.
		b	Cache blocks effectively exploit spatial reuse [Kaplan and Winder 1973; Przybylski et al. 1988; Smith 1982; 1987].
		c	Most cache pollution is due to *spatial-only* blocks, i.e., blocks which are only reused spatially
3	Capacity misses occur more frequently than conflict misses, and both are significant sources of misses.	a	Do ping-pong conflicts occur frequently?
		b	Two-way set-associative caches remove the majority of conflict misses [Agarwal and Pudar 1993; Hill and Smith 1989].
4	Many memory references within numerical codes correspond to regular references.	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.

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 I, arise from extensive measurements of complete programs [Belady 1966; Gee et al. 1993; Hennessy and Patterson 1996; Hill 1988; Hill and Smith 1989; Przybylski et al. 1988; Smith 1982; Sugumar and Abraham 1993], slightly more narrow measurements that evaluate proposed hardware or software techniques [Agarwal and Pudar 1993; Chen and Baer 1995; Drach 1995; Jouppi 1990; Klaiber and Levy 1991; Mowry et al. 1992], and software models [Gannon et al. 1988; McKinley et al. 1996; Mowry et al. 1992; Wolf and Lam 1991]. Our results dispute Assertions 2 and 3, and confirm Assertions 1 and 4 for our benchmark suites.

Our quantification of locality characteristics suggests that the scope of some optimizations should be revisited; even though most reuse is within a nest (Assertion 1), most misses occur between nest executions. This result suggests compiler writers should next focus on optimizing multiple nests at once. In addition, we find that loop nest locality significantly differs from

whole program locality; loop nests attain more spatial reuse (Assertion 2) and dramatically fewer capacity misses than programs (Assertion 3). We show that bandwidth and the cache are frequently wasted due to loads of blocks in which only one word is referenced (Assertion 2). This result suggests that architects should focus more on adjustable block sizes and cache bypass instructions.

We also compare the characteristics of Perfect and SPEC'95, finding that Perfect programs tend to use the cache less well and have more varied behavior than SPEC'95 programs. We then compare the influence of the architecture and compiler pairing on our measurements using the DEC-Alpha and Sun-Sparc, and show similar results. In summary, this article 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 article is organized as follows. Section 2 describes our experimental framework. We then examine our results for SPEC'95 from four different perspectives:

—Is locality *intranest* or *internest* (Section 4)?

—What characterizes *intranest data locality* (Section 5)?

—What characterizes *internest* and *whole-program* data locality (Section 6)?

—What characterizes load/store instruction locality (Section 7)?

Within each of the sections, we examine the relevant assertions and 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. Section 8 compares SPEC'95 and Perfect, finding mostly similar results. We then compare results for Perfect on different architectures to examine the influence of the compiler architecture pairing on these results in Section 9. Finally, Section 9.3 examines the loops with the most misses from Perfect, relating source code structure to locality properties.

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 *intranest* locality in loop nests that are at most three deep, without calls, and with only one loop at each level. (Note that SPEC'95 and Perfect 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

```
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
```

Fig. 1. A few classic examples.

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. For 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 intranest measurements, and the loop is split into the three nests.

The program trace then becomes a set of **in-nest** statement executions separated by **out-nest** statements. We define **intranest** events as ones that occur within a single execution of a nest. For example, if the same cache block is referenced more than once during a single nest execution, we call all these references but the first one **intranest** locality. Thus, we exclude the first reference to a block from the intranest statistics. Any locality the first reference incurs is internest locality. **Internest** 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.

2.1.1 *Examples*. Consider the array A in *Matrix-Vector Multiply* and in *Stencil*. Assume the following: (1) no interference in the cache from the other arrays accessed in the L loop, (2) A is not originally in the cache, (3) R elements of A fit on a cache block, and (4) 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 intranest spatial reuse. If A is still in the cache when *Stencil* executes, these hits are internest and temporal. If on the second iteration of L, *Matrix-Vector*

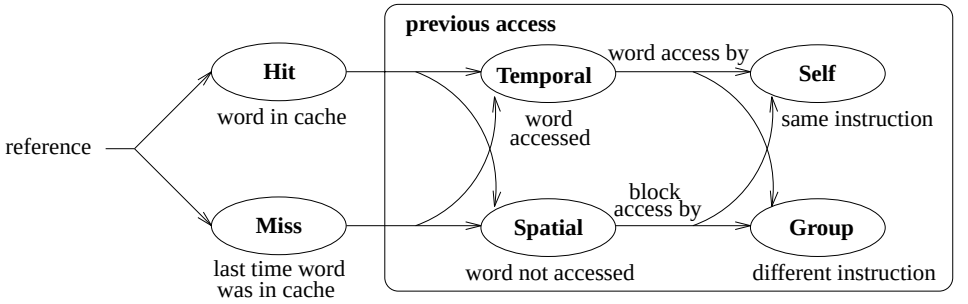

Fig. 2. Locality classifications.

Multiply still finds A in the cache this reuse is also internest temporal reuse for all N^2 references to A. The intranest locality of A on the second execution of *Matrix-Vector Multiply* is the same as it was on the first execution of the nest, intranest spatial locality for $\lfloor (R - 1)/R*N^2 \rfloor$ references. This example illustrates why even though most references occur in loops, intranest, internest, and program characteristics may differ significantly.

2.2 Classifying Locality and Reuse

The classical definitions of locality properties [Hennessy and Patterson 1996] found in programs are

—*Temporal Locality:* If an item is referenced, it will tend to be referenced again soon.

—*Spatial Locality:* If an item is referenced, an adjacent item will tend to be referenced soon.

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, and the previous accesses to the block determine if the reference experiences temporal or spatial locality. If it hits and a previous reference to the same word occurred while the block has resided in the cache, the reference achieves **temporal reuse**. Otherwise, if no previous reference to the same word has occurred since the block was loaded in cache, the reference achieves **spatial reuse**. If the reference misses, its locality depends on accesses to this block the last time it was in

cache. If a previous reference to the same word occurred the last time the block was in cache, the reference is a **temporal miss** (temporal reuse that is not achieved), otherwise it is a **spatial miss**. 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; (2) spatial reuse can occur only once for each word every time the block is loaded; and (3) 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 (32KB, 32-byte block, two-way), though we also examine a second cache configuration (8KB, 32-byte block, one-way) to better understand how varying cache parameters changes our locality classifications.

To discover why a reference misses in the cache, we use Hill's [Hill 1987; Hill and Smith 1989] 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.

—*Conflict Misses:* Additional misses due to mapping constraints in set-associative caches.

Like Hill, we measure conflict and capacity misses with respect to a fully associative cache using an LRU replacement policy; we did not use the more accurate but more time-consuming metric of Sugumar and Abraham [1993].

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.

2.2.1 *Examples*. To measure *intranest 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 intranest 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 5.1 for additional examples.)

2.3 Test Suite

For our test suite, we used the SPEC'95 [Reilly 1995] and Perfect Benchmarks [Cybenko et al. 1990]. For each benchmark, Tables II and IV present the percent of references and misses in nests and out of nests; the total

Table II. SPEC Benchmark Loop Behavior

	% Percent of Nest %				Instrumented Nests			
	References		Misses			at depth		
Code	In	Out	In	Out	Total	1	2	3
APSI	93	7	99	1	159	105	36	18
HYDRO2D	100	0	99	1	101	46	55	0
MGRID	99	1	99	1	21	12	3	6
SU2COR	89	11	83	17	68	55	12	1
SWIM	100	0	100	0	16	8	8	0
TOMCATV	99	1	100	0	7	5	2	0
TURB3D	89	11	99	1	35	18	10	7
Average	95.6	4.4	97.0	3.0				

Table III. SPEC Benchmark Characteristics

Code	Number of Data References Dec	Working Set Size (Eight-Byte Words)	Miss Rate Dec 32KB, Two-way 32-Byte Block
APSI	10,280,886,150	251,292	7.8
HYDRO2D	15,272,678,839	1,101,735	14.9
MGRID	28,420,271,421	955,533	4.8
SU2COR	8,465,773,694	3,041,375	17.3
SWIM	11,588,011,301	1,842,317	7.0
TOMCATV	10,384,736,543	1,833,547	13.3
TURB3D	29,655,434,439	3,245,779	4.2

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 first-level cache miss rate.

Our nest selection, on average, considers more nests than McKinley et al. [1996] consider for optimization. The numbers are not directly comparable because McKinley et al. do not include single loops and consider more complex nesting structures than we do, but their similarities show that our nest selection is comparable to those selected by loop optimizations. In seven of the eight 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 (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.

We used seven of the SPEC'95 Benchmark programs which range in number of noncomment lines from 107 to 4211, averaging 1480 lines. We ran the SPEC'95 experiments using `-O4` (without unrolling; `-O5` adds software pipelining) of the DEC Fortran compiler on an Alpha 21164 server; for the experiments in Section 9.2, we used a SuperSparc20 and

Table IV. Perfect Benchmark Loop Behavior

	% Percent of Nest %				Instrumented Nests			
	References		Misses			at depth		
Code	In	Out	In	Out	Total	1	2	3
ADM	88	12	90	10	159	105	36	18
ARC2D	99	1	99	1	141	76	59	6
BDNA	95	5	98	2	154	133	18	3
DYFESM	93	7	84	16	108	60	42	6
FLO52	98	2	99	1	87	39	39	9
OCEAN	99	1	99	1	76	35	41	0
QCD2	39	61	77	23	90	80	6	4
TRFD	96	4	89	11	29	13	9	7
Average	88.4	11.6	91.9	8.1				

Table V. Perfect Benchmark Characteristics (asterisk denotes partial trace).

				Miss Rates, 32-Byte Blocks		
Code	Number of Data References Dec = (D); Sun = (S)		Working Set Size (Four-Byte Words)	Sun, 8K 1-way	Dec, 8K 1-way	Dec, 32K 2-way
ADM	543,767,970	(D)	18,922	5.6	10.0	1.4
	918,546,427	(S)	29,753			
ARC2D	2,077,628,477	(D)	1,798,990	9.9	27.4	15.4
	4,000,000,000	(S) *	2,001,043			
BDNA	771,237,387	(D)	381,788	4.2	10.9	6.5
	2,331,001,541	(S)	429,217			
DYFESM	453,048,921	(D)	16,718	3.2	3.2	0.5
	452,663,362	(S)	28,409			
FLO52	662,514,235	(D)	207,638	6.8	7.7	5.4
	956,221,083	(S)	214,622			
OCEAN	2,826,270,564	(D)	150,818	6.2	8.3	6.9
	4,000,000,000	(S) *	156,688			
QCD2	298,577,303	(D)	459,838	1.1	1.7	0.7
	508,546,774	(S)	463,133			
TFRD	729,920,480	(D)	1,351,612	5.0	9.1	0.3
	1,429,808,439	(S)	1,354,205			

with the optimization level `-O2` (the highest level without unrolling). All of the codes were run to completion. We disabled loop unrolling because it obscures self- and group locality. The cache miss rates for SPEC in Table III are for a 32KB, 32-byte block size, two-way set-associative data cache (the latest Alpha 21264 cache).

To compare the effects of different compilers and platforms, we also used eight of the Perfect Benchmark programs which range in number of noncomment lines from 485 to 6105, averaging 3509 lines. (See Table IV.) We ran the Perfect experiments using the Sun's F77 compiler at optimization level `-O2` which includes all optimizations except optimizations on global variables and loop unrolling on a SuperSparc20 workstation. For Perfect, the cache miss rates in Table V range from 1.1 to 9.9% on an 8KB,

32-byte block size, direct-mapped data cache (an earlier version of the Alpha 21164 data cache [DEC 1994]). 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.

We use SPEC'95 benchmarks because they are well known and because their working set size is large enough to incur nontrivial miss rates (4.2 to 17.3%), even on a large cache, which makes them ideal for testing popular assertions. Although the Perfect Benchmarks have smaller miss rates even on a smaller cache than SPEC'95 and are thus showing their age, they tend to have more complexity in terms of number of loops and source lines. They are, thus, potentially more representative of actual scientific applications. We first examine the SPEC'95 Benchmarks with respect to our assertions, and then we compare trends between Perfect and SPEC'95. Finally, we discuss the impact of the compiler and architecture on our results.

2.4 Instrumentation and Analysis

For experiments on the Alpha server, we used the Digital tracing tool ATOM [Srivastava and Eustace 1994]. For experiments on the Sun SuperSparc20, we used the *Spy* tracing tool from the *Spa* package.[1] Both ATOM and *Spy* use object codes as inputs, so no special compilation flag is required. ATOM traces most system calls, and *Spy* traces all calls to system libraries but does not handle a number of traps. Six of the eight benchmarks Perfect Benchmarks were run till completion on the Sparc. The trace length was limited to four billion entries for two codes (OCEAN and ARC2D) because of intrinsic limitations in Spy. These eight Perfect Club codes were run to completion on the Alpha Server along with the SpecFP'95 codes.

To detect loops, we instrumented the programs with Sage++, a source-to-source Fortran compiler [Bodin et al. 1994]. Our Sage++ routine detects loop nests as described in Section 2.1 and inserts variables and subroutine calls to uniquely identify nests. ATOM and Spy catch 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

Many of the figures in this article present average statistics over the different benchmarks. We did not weigh the averages, in order to balance the impact of each benchmark. Figures 6–11 plot the fraction of reuse

[1]By G. Irlam, 1991.

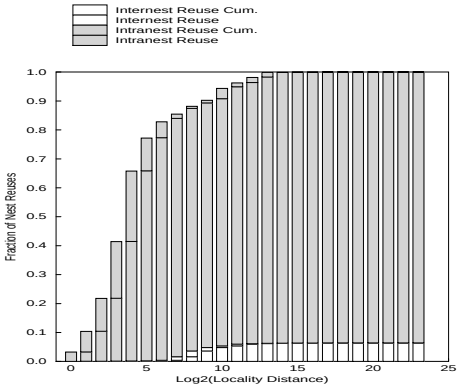

Fig. 3. Intranest versus internest reuse.

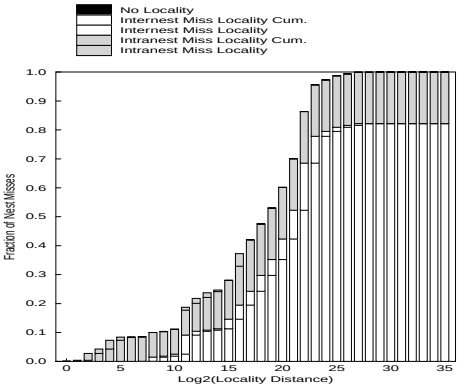

Fig. 4. Intranest versus internest misses.

(Figures 6, 7, and 8) and misses (Figures 9, 10, and 11) as a function of the locality distance in $\log_2$ between references to the same word or cache block for SPEC'95. (Many other figures use the same x-axis.) In Figures 6–11, 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 6, 7, and 8 reveals the differences between intranest, internest, and program reuse. Compare Figures 9, 10, and 11 similarly for misses. The differences in locality between reuse and misses are demonstrated by comparing pairwise between Figures 6 and 9 for intranest locality, Figures 7 and 10 for internest locality, and Figures 8 and 11 for program locality. Section 5 and Section 6 are organized around this later comparison, and Section 7 focuses on load/store instruction locality.

4. IS IT INTRANEST OR INTERNEST DATA LOCALITY?

The vast majority of hardware and software techniques to improve cache performance target individual nests. This strategy implies Assertion 1: *most reuse occurs within a nest rather than across nests*. Most researchers correctly state that the majority of execution time is spent in loops, which is subtly different from Assertion 1. Figures 3 and 4 partition misses by load/store distances in $\log_2$, and the y-axis indicates the fraction of the locality that is intranest or internest. The boxes represent the increment added for the corresponding distance. The results confirm Assertion 1, finding that 95% of hits are intranest, and 5% are internest. HYDRO2D,

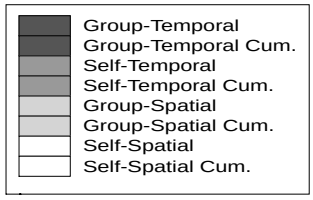

Fig. 5. Legend for Figures 6–11.

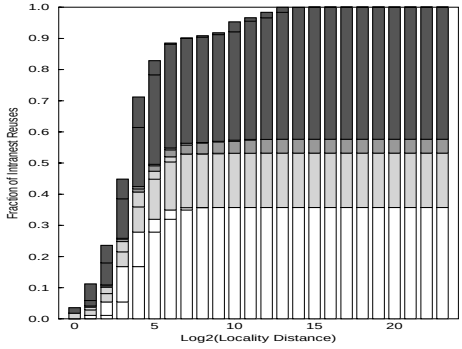

Fig. 6. Distribution of intranest reuse locality.

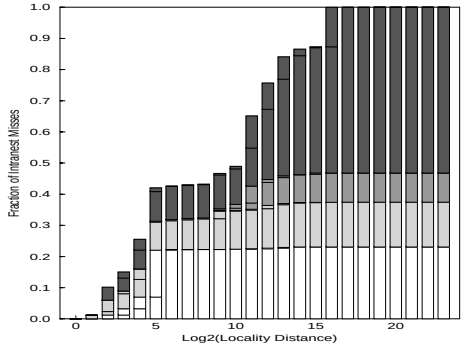

Fig. 9. Distribution of intranest miss locality.

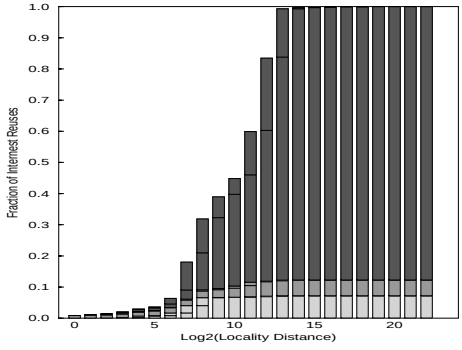

Fig. 7. Distribution of internest reuse locality.

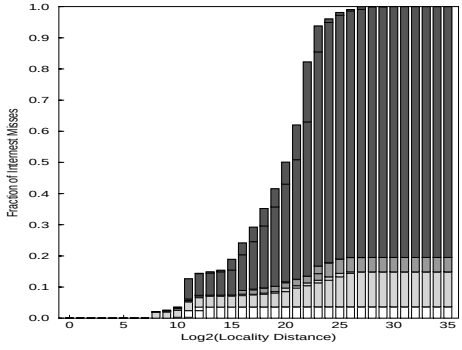

Fig. 10. Distribution of internest miss locality.

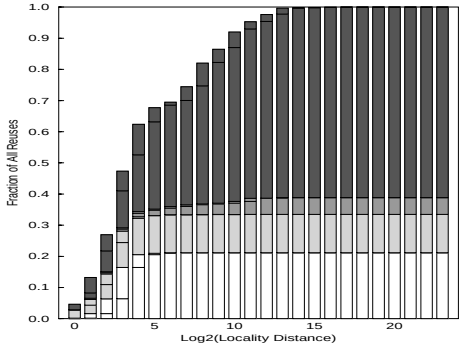

Fig. 8. Distribution of program reuse locality.

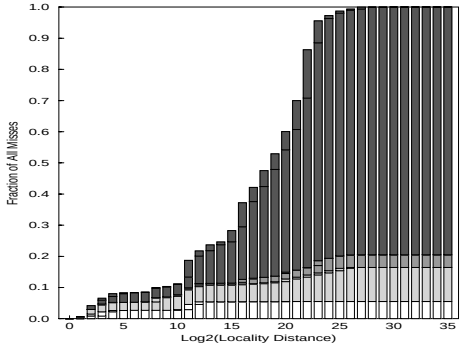

Fig. 11. Distribution of program miss locality.

MGRID, SU2COR, SWIM, and TOMCATV all achieve 100% intranest reuse. APSI achieves 18% internest reuse, and TURB3D achieves 23% internest reuse. McIntosh et al. confirm this result for small caches, and show, when caches are much larger and hit rates drop, that internest reuse is more prevalent [Cooper et al. 1995].

Although most reuse is intranest (Figure 3), more than 80% of misses are internest (Figure 4). We analyze these misses in more detail in Section 6.1.

5. INTRANEST LOCALITY

This section focuses on *intranest* references and explores their behavior in the context of Assertions 2 and 3.

5.1 Locality of Intranest Reuse

One of the most widespread assertion we address is Assertion 2: *Spatial reuse is the dominant form of reuse* [Jouppi 1990; Kaplan and Winder 1973; McKinley et al. 1996; Przybylski et al. 1988; Smith 1982; 1987]. Spatial locality enables hardware and software prefetching to achieve most of its improvements [Chen and Baer 1995; Drach 1995; Jouppi 1990; Klaiber and Levy 1991; Mowry et al. 1992]. Other software techniques have also attributed their effectiveness to improved spatial locality [McKinley et al. 1996].

Figure 6 shows, that, on average, 52% of intranest reuse is spatial. MGRID, SWIM, and TOMCATV achieve a majority of temporal reuse (85%, 80%, and 51%, respectively), but HYDRO2D, APSI, SU2COR, and TURB3D achieve a majority spatial reuse (52%, 55%, 58%, and 94%, respectively). Except for TURB3D, these results are in contrast to Assertions 2 and 2.a, since spatial reuse is not the single overwhelming factor in six of the seven programs.

Compiler algorithms to improve locality target group and self, spatial and temporal locality [McKinley et al. 1996; Temam et al. 1993; Wolf and Lam 1991], 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 [McKinley et al. 1996; Wolf and Lam 1991]. 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, while `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 90% of intranest temporal reuse is group-temporal reuse. In Figure 6, over half of group-temporal reuse occurs at short distances (16

references or less), like X in *Matrix-Matrix Multiply*. Except for a concentration of group-temporal reuse at short distances, the reuse distance for group-temporal reuse is distributed relatively evenly up to a distance of 2^{13} = 8K (slightly more than the cache size in words, 2^{12}). 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 we see in Figure 6.

5.2 Intranest Misses

First recall, that Figure 4 demonstrates that only 18% of misses are intranest. Now compare Figures 6 and 9 and notice a drop in the impact of spatial locality: most intranest misses are temporal (64%) instead of spatial (36%). The intramisses are, however, more bimodal than the hits with respect to spatial and temporal locality. For HYDRO2D, MGRID, and TOMCATV 100% of misses are temporal. For SWIM, 95% of misses are spatial. APSI, TURB3D, and SU2COR are more balanced, with 65%, 58%, and 42% spatial misses respectively. Intranest misses account for 25% or more misses in four programs: APSI (25%), TOMCATV (27%), MGRID (35%), and TURB3D (35%). We will thus concentrate on these misses in the remainder of this section.

5.2.1 *Intranest Capacity and Conflict Misses*. Previous research demonstrates Assertion 3: *capacity misses occur more frequently than conflict misses, and both are significant sources of misses* for whole programs [Hennessy and Patterson 1996; Hill and Smith 1989; Sugumar and Abraham 1993]. Figures 12(a)–(d) divide the intranest misses of APSI (25%), TOMCATV (27%), MGRID (35%), and TURB3D (35%) into self and group, conflict and capacity misses.[2] Figures 13 and 14 divide internest and program misses into self and group, capacity and conflict misses, again plotting the fraction of misses against their distance. All seven programs have only *group* conflict or capacity misses which occur when a different load/store instruction causes the miss.

In line with Assertion 3, MGRID's intranest misses are 100% capacity misses, and most are at a distance of 2^{16} references (remember the cache size is 32K = 2^{12} words, and many of these references have locality). Eighty-five percent of the misses in TOMCATV are capacity misses at a distance of 2^{14} or fewer references, and 15% are conflict misses at a distance of 2^{11} references. TOMCATV and MGRID also have large working set sizes: 1.8M and 0.9M eight-byte words respectively (see Table III). All of MGRID and TOMCATV's intranest misses are group-temporal. Several software techniques focus on selecting tile sizes that eliminate capacity misses and do not introduce self-interference misses [Coleman and McKin-

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

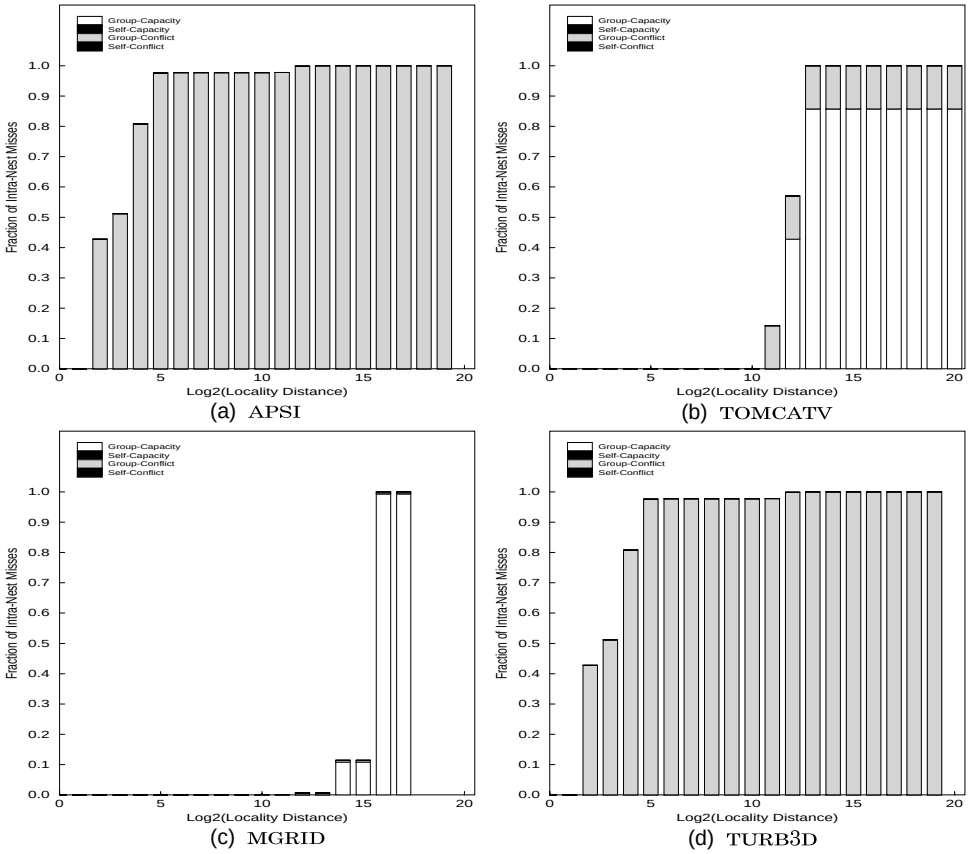

Fig. 12. Capacity/conflict intranest misses.

ley 1995; Lam et al. 1991]. These tiling studies focus on kernels and target self-temporal capacity misses.

On the other hand, all of APSI and TURB3D's misses are conflict misses, and most occur at a distance of 16 or fewer references. These results show a surprising number of conflict misses for APSI and TURB3D which contradicts Assertion 3. Most of these conflict misses are spatial in APSI (66%) and TURB3D (58%). 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.

5.2.2 *Conflict Misses*. A classic source of conflict misses 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 interfere during execution, causing conflict misses. More generally, if $n + 1$ arrays conflict for the same cache set in an n-way associative cache, then conflict misses occur. The probability that the first elements of two arrays fall in the same cache block is low, about 1/*number of cache blocks*,

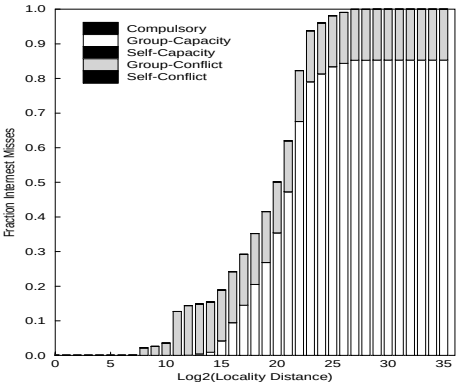

Fig. 13. Capacity/conflict internest misses.

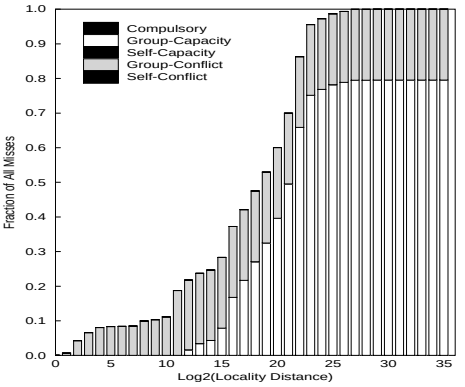

Fig. 14. Capacity/conflict program misses.

suggesting this pattern of interference does not occur frequently (Issue 3.a). However, interference misses due to power-of-two array sizes are not uncommon. TURB3D suffers from this problem. It has six 3D arrays of size $64 \times 64 \times 64$, and typical loop nests reference six to three of these arrays. It also uses several 1D arrays of size 129. Loop nests in TURB3D typically access several of these arrays at a time in one loop. APSI uses two nonpower-of-two array sizes, and a typical loop nest often accesses arrays of both sizes. Hardware mechanisms like cache bypass, victim caches [Jouppi 1990], or higher set associativity are likely to eliminate these misses (see Section 6.2.1).

5.2.3 *Pollution*. Figures 15(a) and (b) illustrate the locality of the lines that come in the cache and remove a line that is still needed for APSI and TURB3D. These graphs illustrate that the intranest references *pollute* the cache, by removing useful lines, and that the replacing (polluting) lines are not reused well. For APSI, 80% of the polluting lines are not reused before the line they replace is needed again. An additional 9% are only reused spatially. In TURB3D, 45% are not reused at all, and 19% are only reused spatially.

In all the other programs, lines that replace a line that will be used have excellent locality. (These results echo the high amount of intranest reuse shown in Figure 3.) Ninety-eight percent of the replacing lines in SWIM and 75% in MGRID are reused extensively both temporally and spatially. The remainder in MGRID are reused only spatially. In TOMCATV (84%), HYDRO2D (58%), and SU2COR (91%) the polluting lines are reused only spatially. The spatial hits of the polluting lines in TOMCATV, HYDRO2D, and SU2COR are echoed in correspondingly high numbers of spatial intranest hits. These results confirm that it may be possible to use a stream buffer [McKee and Wulf 1996] to achieve better cache utilization. A stream buffer bypasses cache to avoid pollution from blocks with only spatial locality.

5.2.4 *Miss Characteristics of Nests*. We also measured how the individual nests contribute to the overall misses. In each graph of Figure 16, the

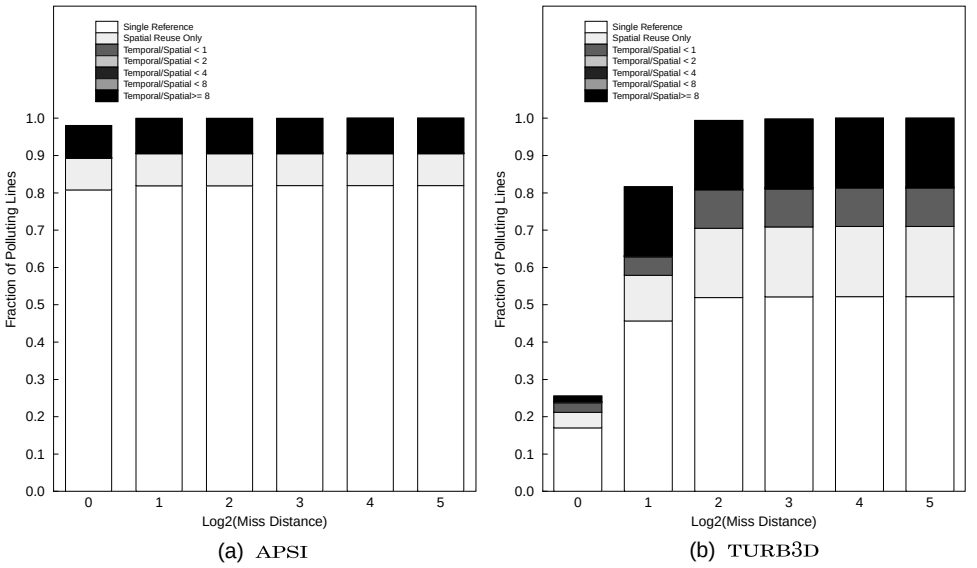

Fig. 15. Polluting line locality.

x-axis numbers all the nests in the program by decreasing number of misses. Each graph contains two curves. The solid line is the cumulative number of misses over all nests, and the dotted line is the miss ratio of each nest. These graphs will show us if there is a correlation between the overall miss contribution of each nest and nest miss ratio.

These results differ widely based on the code. In APSI and TURB3D, many of the nests that contribute most to the overall number of misses are also nests with high miss ratios (between 20% and almost 100%). In contrast, the dominant nests in TOMCATV, SU2COR, and HYDRO2D only exhibit at most a 25% miss ratio. The division of misses among nests also changes significantly from code to code. In SWIM, MGRID, and TOMCATV three nests contribute to between 80% and 99% of the overall misses, while most other codes exhibit a more regular distribution. Except for HYDRO2D, the distribution of misses is skewed; 10 nests usually encompass at least 80% of the misses.

5.2.5 *Summary of Intranest Locality Results*. The major findings of this Section damage Assertions 1–3. Our results confirm that most (95%) reuse is intranest (Assertion 1), but finds that 82% of misses are instead internest. To further improve performance will, thus, require hardware and software techniques that target multiple loop nests. Instead of a preponderance of spatial reuse (Assertion 2), we find a balanced role between intranest spatial (48%) and temporal (52%) reuse. Even for these large data sets, most hits (46%) are intranest, group temporal, and many (37%) are intranest, self-spatial. The hits tended toward short reuse distances (85% of hits are within 32 references) for the two-way set-associative 32KB

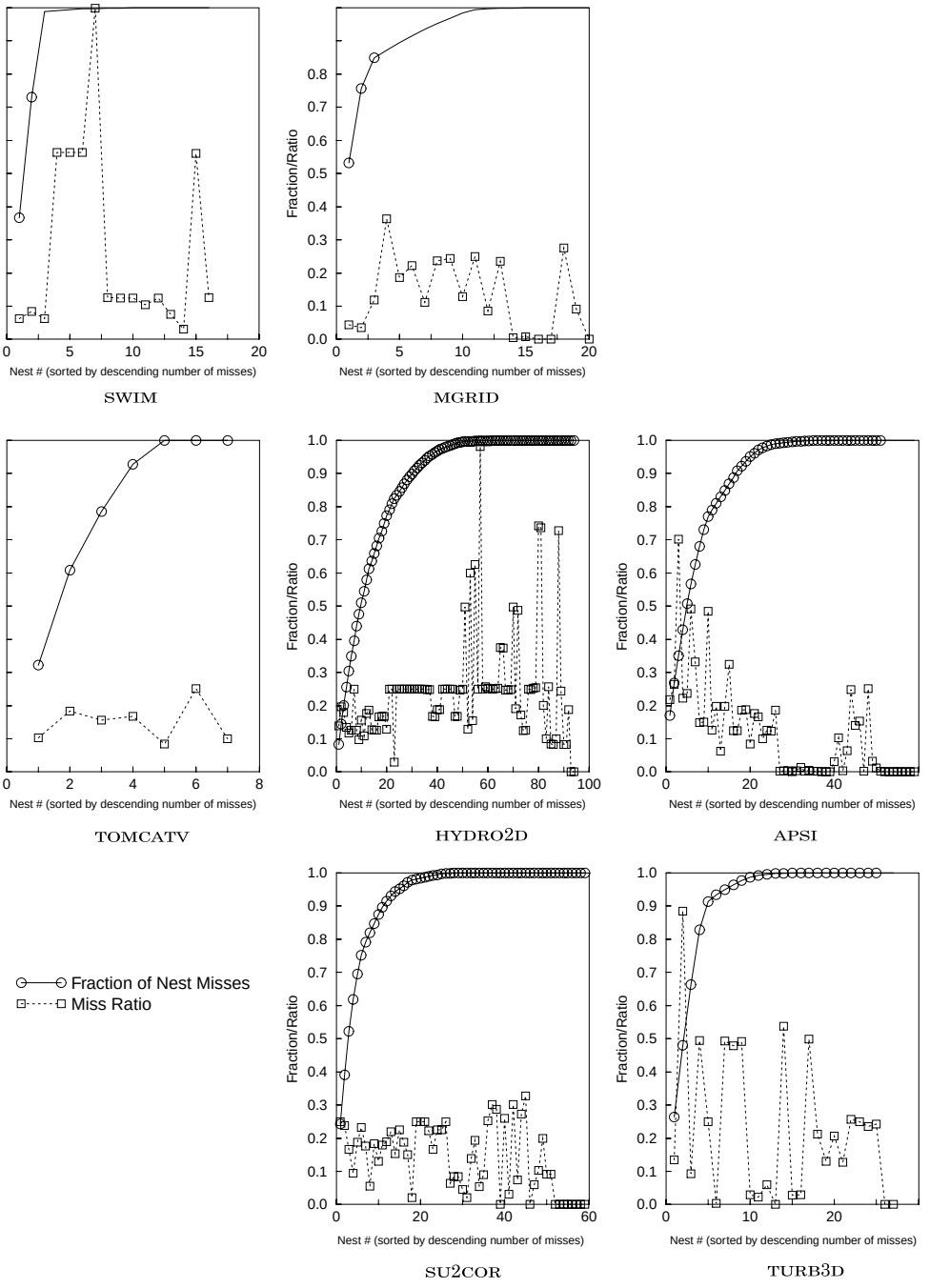

Fig. 16. Miss characteristics of nests.

cache. Only four programs contained significant numbers (25% to 35%) of intranest misses: APSI, TOMCATV, MGRID, and TURB3D. MGRID and TOMCATV contain mostly capacity misses, following Assertion 3. However,

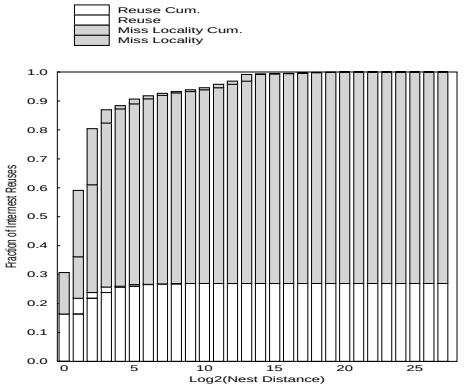

Fig. 17. Nest distance of internest locality.

the intranest misses in APSI and TURB3D are conflict misses with both spatial and temporal locality that is poorly exploited by the cache. In these programs, cache blocks that are only referenced once or a few times also significantly pollute cache. To use the cache more effectively will require new hardware, software, and combined solutions.

6. INTERNEST AND PROGRAM DATA LOCALITY

In this section, we investigate internest locality, locality that occurs between loop nests and across outer nests. We then compare intranest, internest, and program behavior.

6.1 Internest Reuse and Misses

As mentioned in Section 4, only 5% of reuse is internest; 95% is intranest (see Figure 3). Only two programs have any internest reuse at all: APSI and TURB3D have 18% and 23% internest reuse, respectively. For both programs, these hits occur between a distance of 128 and 4048 references. Not surprisingly, all of these hits are temporal, and over 90% are group temporal.

Although most hits are intranest, Figure 4 shows that 82% of misses are internest. Figure 13 illustrates that 85% of these misses are capacity misses. APSI and TURB3D are responsible for all of the conflict misses (58%) and (45%) respectively. These misses are spatial and temporal, occurring from distances of 2^8 to 2^{26} references. For the remaining five programs, 100% of misses are capacity misses. The majority of these misses are group temporal. SU2COR and MGRID have around 15% self-temporal misses. These temporal capacity misses occur from distances of 2^{15} to 2^{25} references.

To decrease these misses, optimizations cannot simply focus on nest optimizations but need to consider more than one nest, in contrast to Assertion 1.a. An internest miss is a miss on a block that was previously referenced in another nest. The natural question posed by Assertion 1.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 17 illustrates the fraction of internest misses as a function of *nest distance*, the number of nest executions since the last reference to the data. Only 25% of internest locality is exploited, but fortunately, more than 75% internest misses correspond to short nest distances of at most four adjacent nests, around 50% of internest misses are between adjacent nests (*nest distance* = 1). (Since these are nest executions, some of these are the same nest.) This result implies fusion and other cross-nest optimizations have the potential to reduce misses. Perhaps an internest tiling optimization would be appropriate, although this loop transformation technology does not currently exist. Cooper et al. [1996] propose an internest reuse analysis to exploit locality, but its effectiveness with conventional cache designs was limited. Prefetching may also work in this case, but most prefetching algorithms are focused on a single nest and do not consider internest effects. We examine prefetching further in Section 7.1.4.

6.2 Intranest versus Internest versus Program Locality

If we compare Figures 6 and 7, and Figures 9 and 10, we see that spatial locality accounts for very little locality (reuse or misses). Distinct nests are unlikely to achieve 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 93% and self-temporal for 4% of internest reuse. Only 7% of internest reuse is group-spatial. Only APSI and TURB3D have a significant portion of internest reuse, and all of their reuse is temporal. Since a nest of depth 3 can be surrounded by a loop, reuse from one execution to the next can result in self-reuse, but none of these programs exhibit this behavior.

Figure 8 illustrates average program reuse without regard to the location of a reference. The reader might expect that Figure 8 would be a combination of Figures 6 and 7, weighted by their relative importance from Figure 3, but it is not. As discussed in Section 2.1, intranest reuse does not include hits due to references from outside the loop; internest reuse captures just the reuse due to shared accesses from different nests; and program reuse includes every reference in the program. These measurements are, thus, different views of 96% of the references, and program references include the additional 4% of references (see Table II).

Comparing Figures 6 and 8 shows a drop in the importance of spatial locality from 52% to 33%, in contrast with Assertion 3. The high number of internest spatial references is amenable to loop nest techniques that hide latency, like prefetching and stream buffers. Since these techniques only target nests, they are not as effective across the whole program. The programs with the highest miss rates—SU2COR (17.3%), HYDRO2D (14.9%), and TOMCATV (13.3%)—have the highest portion of spatial locality, at 61%, 52%, and 46%. The other four programs have lower miss

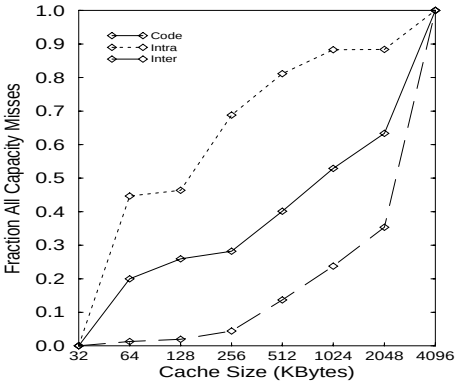

Fig. 18. Cache size versus capacity misses.

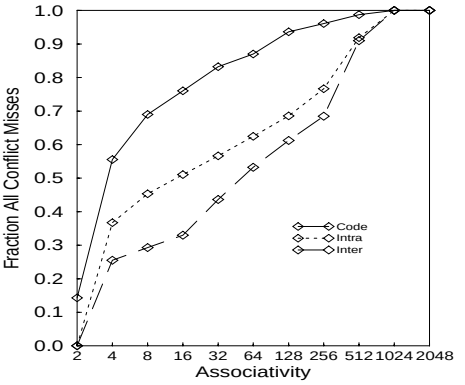

Fig. 19. Associativity versus conflict misses.

rates (less than 8%) and very high (80%) temporal reuse.[3] This result implies that achieving high hit rates requires a high degree of temporal locality across the program. Indeed if we look at Figures 10 and 11, 85% of misses are temporal. To further decrease miss rates, architects and compilers will, thus, need to focus on temporal misses across loop nests. Simply increasing cache sizes may not be sufficient because the bigger the cache sizes the bigger the problems users will likely want to solve.

6.2.1 *Capacity and Conflict Misses*. Figures 13 and 14 demonstrate that the fractions of capacity misses are significantly higher for program misses and internest misses than for intranest misses. These results are in line with Assertion 3. Internest capacity misses range from 45% to 100%, averaging 84%. Internest misses are mostly capacity misses because the locality distances are large and because there is a high probability that the number of words reloaded between two references to the same location 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 32KB two-way set-associative cache with a 32-byte block (Figure 18), and the minimum associativity required to remove each conflict miss that occurs in the same cache for intranest, internest, and whole-program misses (Figure 19).

Figure 18 shows the effects of increasing the cache size on intranest, internest, and program capacity misses. Our results show that doubling the cache to 64KB removes 20% of program capacity misses, but further gains are more difficult to achieve. A 64KB cache eliminates all the intranest misses in MGRID and TOMCATV. (These two programs are the only programs with significant numbers of intranest capacity misses, but this fact is not represented in Figure 18 due to averaging on a per-program

[3]See Table III for miss rates.

basis.) A cache size of 512KB just begins to remove internest misses, and a 2048KB cache is required to eliminate just 35% of internest misses, although it eliminates 60% of program misses. To eliminate all capacity misses requires a cache size of 4096KB. This trend clearly reveals the motives of the SPEC'95 benchmark designers to exceed the capacity of modern first-level caches. Even a relatively large cache size of 256KB leaves over 95% of the internest capacity misses for software techniques to attack.

Figure 19 plots the fraction of conflict misses removed by higher degrees of set associativity. A four-way set-associative cache removes 36% of intranest conflict misses. It takes more than 128-way set associativity to eliminate all conflict misses, in contrast with Assertion 3.b. This result is due to the limited number of conflict misses in the first place. The two codes with significant numbers of intranest and internest conflict misses—APSI and TURB3D—need less associativity. APSI needs four-way set associativity to eliminate over 90% of its intranest conflict misses, but 64-way set associativity to eliminate over 90% of its internest conflict misses. Eight-way set associativity eliminates over 90% of TURB3D's intranest and internest conflict misses. These results demonstrate that conflict misses are still problematic and are in contrast with Assertion 3.b.

6.3 Summary of Internest and Program Locality Results

In this section, we confirmed Assertion 1; most reuse is intranest. However, we find that most misses are internest capacity misses. The lack of intranest 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. Large caches do not eliminate many of these misses. Internest misses often occur between adjacent nests, so it may be possible to remove some of them with software techniques. In a few programs, high set associativity is sometimes required to eliminate conflict misses (Assertion 3.b). We show that the locality patterns between intranest, internest, and program locality are usually very different.

7. LOAD/STORE LOCALITY

Hardware and software optimizations target not only nests, but the load/store instructions within nests. For instance, prefetching tables [Chen and Baer 1995] 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 [Callahan et al. 1990; Carr and Kennedy 1994], collapse several array references into a single reference. Even with these caveats, load/store statistics tend to reflect the behavior of array references.

7.1 Stream Locality

7.1.1 *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 three consecutive references (two strides) is identical (the hardware characterization proposed by Chen and Baer [1995]). 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 18 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 18).

—*Nonstream References:* Two consecutive executions of the load/store instruction have distinct strides.

Stream references clearly dominate (100% of references in SWIM, 97% in MGRID, 91% in TOMCATV, and 90% in HYDRO2D), confirming Assertion 4. With respect to Assertion 4.a, the programs have few scalar references, 11% in APSI, 10% in SU2COR, 9% in TOMCATV, 6% in HYDRO2D, 2% in TURB3D, and 1% in MGRID. These scalar references may be due to register spilling, even though we compiled using F77's scalar optimizations.

7.1.2 *Stream Misses*. Figure 21 divides the fraction of total nest misses (intranest and internest 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 [1993] findings that misses are concentrated in particular load/store instructions. In addition, Figure 21 illustrates that the concentration is much higher for stream misses than scalar and nonstream misses. Stream misses occur within a stream, and on average 7% of all stream misses are in a single instruction. For instance, 24% of TURB3D's and 10% of MGRID's stream misses occur within a single instruction. For stream and nonstream misses, the address referenced changes on each new execution of the load/store instruction, so it is not surprising that the fraction of misses and references is correlated.

7.1.3 *Types of Streams and Stream Strides*. Figure 20 splits streams into four categories:

(1) *Fixed-Length Streams:* The stride change is always the same (this behavior is typical of rectangular loop nests).

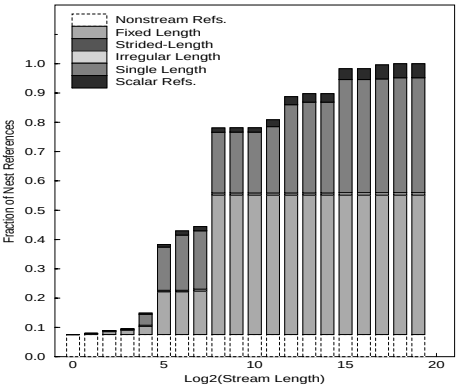

Fig. 20. Stream types, lengths, and frequencies.

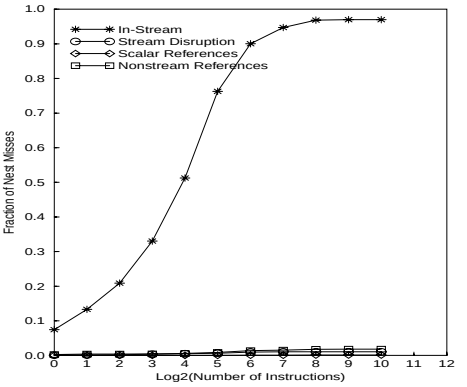

Fig. 21. Stream, scalar, and nonstream misses.

(2) *Strided-Length Streams:* The stride change itself is regular (this behavior is typical of triangular loop nests).

(3) *Irregular Streams:* The stride change is irregular.

(4) *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 from the loop bound, 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 18: 38% and 48% of stream references respectively. For these numerical codes, this result confirms the first part of Assertion 4.c: *loop nest structures are mostly rectangular and triangular.* All SPEC'95 benchmarks we used contain just a few strided-length streams generated by triangular loop nests, and contain only 1% irregular stream lengths.

Table VI presents the stride distributions for each program and confirms Assertion 4.b: *the most commonly used stride value is 1.* The table includes the five most frequently used strides and its fraction of total stream references. Stride-1 references largely dominate. These program are dominated by stride values of 0, 1, and 2. Most programs use only one stride over 60% of the time. Stride values of 0 represent scalars and array references that only vary according to outer-loop indices and are thus invariant in an inner loop. These loop-invariant array references dominate SU2COR. 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. Two bits are sufficient, since there are usually less than four strides.

Table VI. Stride Values in Streams

Program	Stride Value (Percentage of All Stream References)				
APSI	2 (60%)	1 (16%)	–2 (13%)	0 (11%)	
HYDRO2D	1 (99%)	0 (1%)			
MGRID	1 (95%)		2 (5%)		
SU2COR	1 (29%)	0 (17%)	2 (15%)	–1 to –510 (19%)	3 to 510 (20%)
SWIM	1 (100%)				
TOMCATV	1 (100%)				
TURB3D	1 (64%)	4 (19%)	66 (17%)		

7.1.4 *Stream Lengths and Prefetching*. The x-axis in Figure 18 plots the number of references between stride changes. Two programs have large numbers of references between stride changes: HYDRO2D has 40% at 256 references, but 60% at 2^{15} references, and SU2COR has most stride changes at 2^{12} references. However, most programs have fewer references between stride changes. For MGRID, APSI, and TURB3D these stream lengths often consist of 32 or fewer 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 submatrices of larger matrices. When a nest scans 2D submatrices, 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, and the stream ends, hence the short stream lengths. Short stream lengths can disrupt hardware prefetching. For instance in prefetch tables [Chen and Baer 1995], prefetching stops if the stride changes, and the instruction must make three new stream references to stabilize. Tagged prefetching [Jouppi 1990], which upon access to one cache block simply fetches the next, 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 32 references to adjacent words correspond to eight 32-byte cache blocks, tagged prefetching may fail every eight prefetch requests. This result suggests prefetching should handle stride changes [Mowry et al. 1992] as well as stream references [Chen and Baer 1995].

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 [Drach 1995]. In Figures 20(a) and (b), the fraction of intranest 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 for APSI and TURB3D. Since prefetching only targets intranest misses, these are the only programs with significant numbers of intranest misses that are amenable to prefetching. None of the intranest capacity misses in MGRID

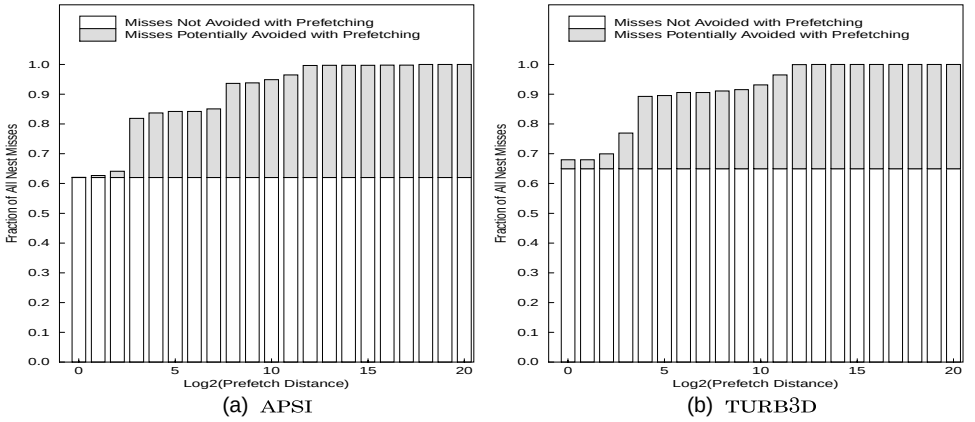

Fig. 22. Potential effectiveness of prefetching.

or TOMCATV are avoided with prefetching. Observe that virtual tagged prefetching breeds 65% useless prefetch requests for APSI (the data are already in the cache) and 62% for TURB3D. Many of these misses are conflict misses (see Section 5). Figure 20 also quantifies prefetch distances in number of references and shows that the useful prefetch requests are used immediately, which is more amenable to software rather than hardware techniques.

7.2 Summary of Load/Store 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 these codes do not contain triangular or irregularly strided loops. Most loops are rectangular (Assertion 4.c). We also found most of these programs contain a majority of stride-1 references (Assertion 4.b), but that six of seven programs also contained other consistent stride values. Finally, we briefly evaluated the impact of these observations on data prefetching.

8. PERFECT

In this section, we compare similarities and differences on Alpha traces between SPEC'95 on a 32KB, 32-byte block, two-way set-associative cache and the Perfect Benchmarks on an 8KB, 32-byte block, direct-mapped cache. We choose a smaller cache and less associative cache for Perfect because it brings the average miss rates between the two benchmarks closer together: the miss rate of SPEC'95 is 9.9%, and Perfect is 9.8%. However, the large miss rate of ARC2D in Perfect skews the average a bit.

In previous work, we present the average graphs and discuss in detail the Perfect Benchmarks on SuperSparc20 traces for the same cache organization [McKinley and Temam 1996]. Since the Sparc and Alpha results are

similar, as we will discuss in Section 9, we do not present the average graphs for Perfect on the Alpha in this section. The appendix contains these graphs and those for SPEC'95 for reference and comparison. Since, on average, both benchmarks yield similar results, the remainder of this section briefly overviews the similarities and mainly focuses on the differences.

8.1 Where are the Hits and Misses?

Most reuse is intranest in both SPEC'95 and Perfect, 95% and 90% respectively, and most misses are internest, 82% and 70% respectively (see Figures 31, 34, 32, and 35 in the appendix). The temporal/spatial breakdown of intranest reuse and internest misses breaks down as follows. Fifty-two percent of intranest reuse is temporal in SPEC'95, and 70% in Perfect probably due to the larger data set size of SPEC'95. For internest misses, locality is mostly temporal: 86% temporal in SPEC'95, and 73% temporal in Perfect. In the appendix, see Figures 52–65 for spatial and temporal locality breakdowns of misses and reuse. The higher hit rates and smaller working set sizes in Perfect explain its 20% higher proportion of intranest reuse. We expect internest misses to be mostly temporal, and both benchmarks bear out this result. Nest distances for hits and misses in Perfect are also short (see Section 6.1 for SPEC'95, and Figure 36 and 39 in the appendix).

One interesting difference in Perfect is that four programs—FLO52, ADM, ARC2D, and BDNA—contain more than 15% intranest misses, 15%, 34%, 54%, and 90% respectively. In SPEC'95, no program has more than 35% intranest misses, but four programs fall in the 25% to 35% range: APSI, TOMCATV, TURB3D, and MGRID have 25%, 27%, 35%, and 35% respectively.

8.2 Capacity and Conflict Misses in Perfect

Most misses in both benchmarks are capacity misses: 68% for SPEC'95 and 53% for Perfect (combining the results from Figures 35 and 38 with 40 and 43 in the appendix). To eliminate these misses, software techniques must span loop nests. Only a few misses are intranest capacity: 5% in SPEC'95 and 7% in Perfect. Very few misses could, thus, benefit from tiling, a single nest transformation.

Clearly, as we increase cache size with respect to working set size, conflict misses will decrease in importance. Perfect has more conflict misses than SPEC'95: on average, 13% of SPEC'95's and 20% of Perfect's intranest misses are conflicts, and 12% of SPEC'95's and 20% of Perfect's internest misses are conflicts. Thus SPEC'95 has 25%, and Perfect has 40% inter- and intranest conflict misses. These differences are due to the smaller working set size of Perfect, and less associativity in the smaller 8KB direct-mapped cache that we used for Perfect as compared to the 32KB, two-way cache for SPEC'95.

Table VII. Stride Values in Streams

Program	Stride Value (Percentage of All Stream References)						
ADM	0 (50.28%)	1 (46.66%)	–1	(3.03%)	–63	(0.03%)	
ARC2D	1 (55.23%)	289 (43.79%)	0	(0.54%)	8	(0.02%)	6 (0.02%)
BDNA	0 (38.12%)	1 (20.06%)	3	(19.01%)	4	(0.91%)	2 (0.49%)
DYFESM	1 (47.14%)	0 (46.41%)	10	(1.15%)	2	(1.04%)	23 (0.79%)
FLO52	1 (46.92%)	0 (44.43%)	97	(7.89%)	17	(0.15%)	57 (0.14%)
OCEAN	1 (43.16%)	129 (33.46%)	0	(18.06%)	–1	(1.81%)	–129 (1.75%)
QCD2	0 (45.61%)	1 (45.26%)	4	(3.26%)	5	(3.00%)	–9 (1.40%)
TRFD	1 (58.37%)	820 (19.03%)	630	(11.20%)	465	(6.08%)	325 (2.96%)

We found that conflicts between two arrays occurs repeatedly in Perfect without the power-of-two problem found in SPEC'95, but for short periods of time. The recurrence of the same sequence of conflict misses results in very irregular miss distributions across cache blocks: a high of 6% of misses in DYFESM are concentrated in a single cache block.

8.3 Spatial and Temporal Locality in Perfect

One of the biggest differences between Perfect and SPEC'95 is the nature of their intranest reuse: only 30% of Perfect's intranest reuse is spatial, whereas 52% of SPEC'95's is spatial (Figures 53 and 56). These results are echoed also in our pollution results: when a cache line replaces a line in SPEC'95 that line tends to be reused spatially: on average, 42% of lines are reused only spatially, and an additional 38% are reused both spatially and temporally (Figures 67 and 70). Only 20% of replacing lines in SPEC'95 never get reused, but in Perfect 50% of replacing lines are never used. In Perfect, these lines typically contain spatial reuse, but it occurs across an outer loop and was thus too far apart to exploit. If we examine the individual programs, we see that the pollution in APSI and TURB3D from SPEC'95 actually follows the same trends as in Perfect, but in all the other SPEC'95 programs the polluting lines have reuse. The higher spatial locality in SPEC'95 also means that prefetching will be more effective: prefetching has the potential to eliminate 80% of misses in SPEC'95, but fewer, 60%, in Perfect (Figures 68 and 71).

Cache blocks in Perfect are, thus, not efficiently exploited and many loaded words are not used, wasting cache bandwidth. Other works support these results [Burger et al. 1996; Tyson et al. 1995; Wood et al. 1991]. If programs are more like Perfect than SPEC'95, future architectural and software improvements to cache performance should be able to load the cache selectively and/or use adjustable cache block sizes.

8.4 Streams

Perfect contains fewer streams and more strides than SPEC'95 (Figures 73–75). In fact, Perfect contains 5% scalar misses, which probably come from register spilling. Since Perfect codes are bigger and more complex, this result is not too surprising. Almost 50% of Perfect's references are

nonstream references, whereas less than 20% of SPEC'95's references are nonstream. This result echos SPEC'95's greater degree of spatial locality. Perfect has a richer set of stride values for its stride references: In SPEC'95, all programs use strides of 0, 1, or 2; in Perfect (see Table VII) seven programs use at least one stride value greater than 1, which again is probably due to the greater complexity and larger code size, as compared with SPEC'95.

8.5 Summary of Perfect versus SPEC'95 Benchmarks Results

We find Perfect and SPEC'95 have many similarities. For example, the internest misses have similar portions of spatial and temporal locality, and both achieve a vast majority of intranest hits and internest misses. Capacity and conflict misses occur in similar proportions in both benchmarks, although Perfect has more conflict misses due to its smaller working set size, smaller cache, and less set associativity. We find significantly more spatial intranest hits in SPEC'95, which impacts both pollution and the potential of prefetching. There are fewer streams in Perfect, and a richer set of stride values in the streams it does contain. In summary, Perfect seems to break the assertions more than SPEC'95, and we believe this result is because SPEC'95 is less complex than Perfect and is more highly optimized for current caches and architectures.

9. THE INFLUENCE OF THE ARCHITECTURE AND COMPILER

In the previous sections, we extensively analyzed the behavior of SPEC'95 codes on a 32KB, 32-byte block, two-way cache and then compared them with the Perfect Benchmarks on an 8KB, 32-byte block, direct-mapped cache, using traces collected on a Dec Alpha. In the following three sections, we address (1) how the cache architecture influences our locality statistics, (2) how the architecture/compiler platform influences these statistics, and (3) how the behavior of the individual nests with the most misses changes when the architecture and compiler platform varies.

To answer these questions, we did not perform extensive experiments, but analyze the results of two limited sets of experiments. To study the influence of cache parameters, we ran the Perfect Club suite on a Dec Alpha with an 8KB, 32-byte block, direct-mapped cache, and we compare the results with a 32KB, 32-byte block, two-way set-associative cache. To study the influence of the architecture/compiler pair, we compare the Perfect Benchmarks using an 8KB, one-way cache on a Dec Alpha with the same experiment on a Sun Sparc. We then vary these same parameters and discuss their effects on the loop nests that miss the most in the Perfect Benchmarks.

9.1 Varying the Cache Architecture

In this section, the 8KB, 32-byte block, direct-mapped cache is called cache-1, and the 32KB, 32-byte block, two-way cache is called cache-2. Both experiments are done on the Dec Alpha. When moving from cache-1 to

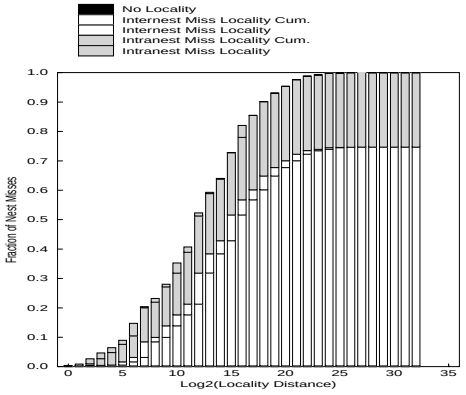

Fig. 23. Cache-1 intranest versus internest misses.

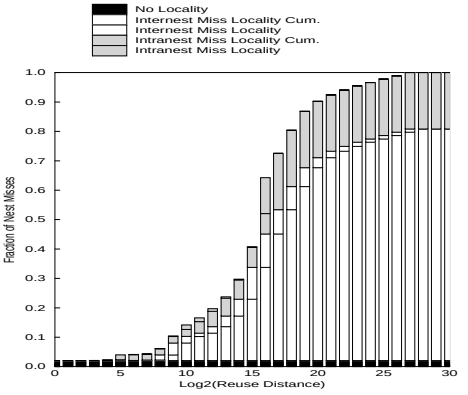

Fig. 24. Cache-2 intranest versus internest misses.

cache-2, the raw miss ratios show an expected decrease of at least 30% and of 50% and more in most cases (see Table V). Since both size and associativity increase, both conflict and capacity misses decrease.

9.1.1 *Intranest versus Internest Locality*. Comparing Figures 23 and 24 shows that the average fraction of intranest misses decreases by only 10% when moving from cache-1 to cache-2. Looking at individual program statistics reveals more contrasts. Many codes like DYFESM, OCEAN, TRFD, and QCD2 have almost no intranest misses on cache-1, and, thus, the increases in size and associativity in cache-2 do not decrease their intranest misses. On the other hand, ADM, BDNA, and ARC2D exhibit large numbers of intranest misses using cache-1. For ADM and ARC2D, the percentage of intranest misses decreases by more than 25%. Moving from cache-1 to cache-2, thus, drastically reduces intranest misses in these codes. On average (Figures 23 and 24), the codes with no intranest misses dampen this effect. The ability of cache-2 to reduce intranest misses is particularly effective for misses with a short locality distance. For instance, in BDNA with cache-1, 50% of intranest misses are concentrated at distances of 64 or fewer references, while with cache-2 there are almost no misses below a locality distance of 1024. As mentioned in Section 5, intranest misses are mostly conflict misses, while intermisses are mostly capacity misses. The higher associativity of cache-2 is responsible for decreasing most of the intranest conflict misses, but a larger cache size also contributes to their reduction. Going from cache-1 to cache-2 reduces the miss rate less in codes with few intranest misses (e.g., from 8.3 to 6.9 for OCEAN), than in codes that start with large numbers of intranest misses. In other words, intranest misses tend to have a large effect on overall miss rate. For all these codes, either capacity misses dominate already with cache-1, or they become clearly dominant with cache-2. This behavior suggests that increasing cache size to 32KB does help reduce intranest conflict misses, but it does not have a strong impact on intranest capacity misses. On the other hand, TRFD and DYFESM have few intranest misses

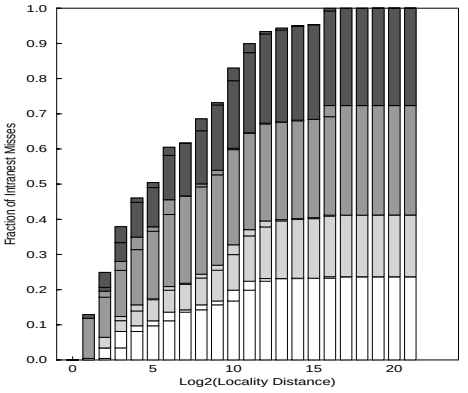

Fig. 25. Cache-1 intranest miss locality.

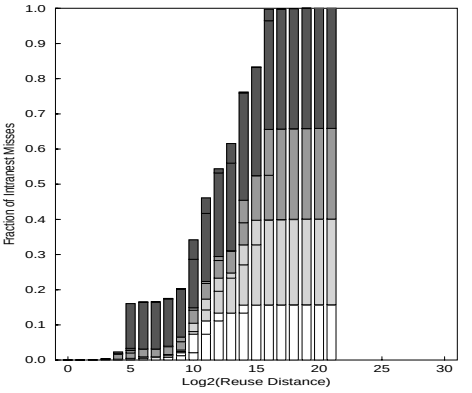

Fig. 26. Cache-2 intranest miss locality.

and exhibit a significant decrease of overall miss ratio when using cache-2 instead of cache-1. We found that a 32KB fully associative cache would remove almost all conflict misses for these codes, hence the significant decrease.

9.1.2 *Locality Characteristics*. The following analysis of locality characteristics for reuse and misses corroborates the above remarks. The breakdown of reuse into group/self, and spatial/temporal for cache-1 and cache-2 is almost identical, which is not surprising considering the large hit rates (we do not include the figures for this result). Figures 25 and 26 demonstrate essentially the following two differences between the miss locality characteristics for cache-1 and cache-2. (These figures use the legend from Figure 5.)

(1) The fraction of self-spatial misses is halved (from 25% to 12%) and is correlated to the reduction of intranest conflict misses at short locality distances. Self-spatial misses at short locality distances in a direct-mapped cache correspond to conflicts between two or more load/store instructions simultaneously competing for the same cache line. A load/store instruction, thus, spatially reuses only few words per line, and misses occur in bursts at short time intervals. As expected, two-way associativity significantly reduces these misses.

(2) Consequently, while most misses are concentrated in short locality distances for cache-1 (log shape), the distribution for cache-2 has a shape that is similar to the internest miss distribution (hyperbolic shape).

As a side-effect of the reduction of intranest conflict misses, prefetching potentially performs better in cache-2 than in cache-1. The intranest self-spatial conflict misses induce unpredictable misses at short locality distances that are difficult to predict and avoid with prefetching. When these misses are removed, prefetching has the potential to reduce a significantly larger portion of remaining nest misses, from 60% with cache-1 to 80% with cache-2.

In summary, the main impact of changing from cache-1 for cache-2 is to decrease intranest misses, essentially by removing intranest self-spatial conflict misses at short locality distances.

9.2 Processor/Compiler Influences

There are many parameters in a processor/compiler environment that influence the statistics gathered in this study. Because of the large number of these parameters, our explanations must be speculative, and we therefore restrict our comments to major trends and specific examples.

We first compare the raw statistics of the eight Perfect benchmarks from Table V. First, notice that the number of references with the Dec is almost always half of those on the Sun except for DYFESM. Because the Sparc processor is 32 bits, double-precision loads actually require two references to cache, while Dec is 64-bit and therefore can load eight bytes in a single cache fetch; in DYFESM, double precision is not used. Because the number of references varies so much, we compare the raw number of misses rather than the miss ratios. The number of misses is fairly similar and decreases between 1% to 10% from Sun to Dec, with the notable exception of ARC2D where the number of misses increases by 43%.

A comparison of the main statistics (i.e., the division of misses and reuse into intra- and internest, the division of misses and reuse into spatial/temporal and self/group, and the division of misses into conflict, capacity, and compulsory misses) show a remarkable similarity between almost all of the *average* graphs. The only exceptions are the locality characteristics of both intranest reuse and misses. The Sparc statistics show about 10% more intranest temporal reuse and 20% more intranest self-temporal reuse. One explanation could be more extensive use of scalar variables (possibly due to register spilling) by the Sun compiler.

A Sun SuperSparc20 has a 16KB, 32-byte block, four-way set-associative data cache that is less sensitive to conflicts than the Dec 21164 8KB, direct-mapped cache, so we can expect Sun compiler optimizations may use the cache to store scalar variables. This explanation can also justify the even larger difference between the two intranest miss locality characteristics. On the Sparc, there are about 65% spatial misses, and only 40% on the Alpha. Group-spatial misses fall from 35% down to 15% of overall intranest misses. It is possible that scalar references within loop nests sporadically flush useful lines within loop nest iterations.

Although the average graphs are fairly similar, there are significant variations in individual programs. Consider, the intranest misses versus internest misses of the individual codes of the Sparc and Alpha in Figures 27 and 28, respectively. QCD2, OCEAN, FLO52, and TRFD have fairly similar proportions of intra- versus internest misses. But ADM and DYFESM have more intranest misses on the Sparc than on the Alpha Dec, 10% and 50% more respectively, while BDNA and ARC2D have more intranest misses on the Alpha than on the Sparc, 25% and 30% more respectively. While the overall decrease in misses from the Sparc to the

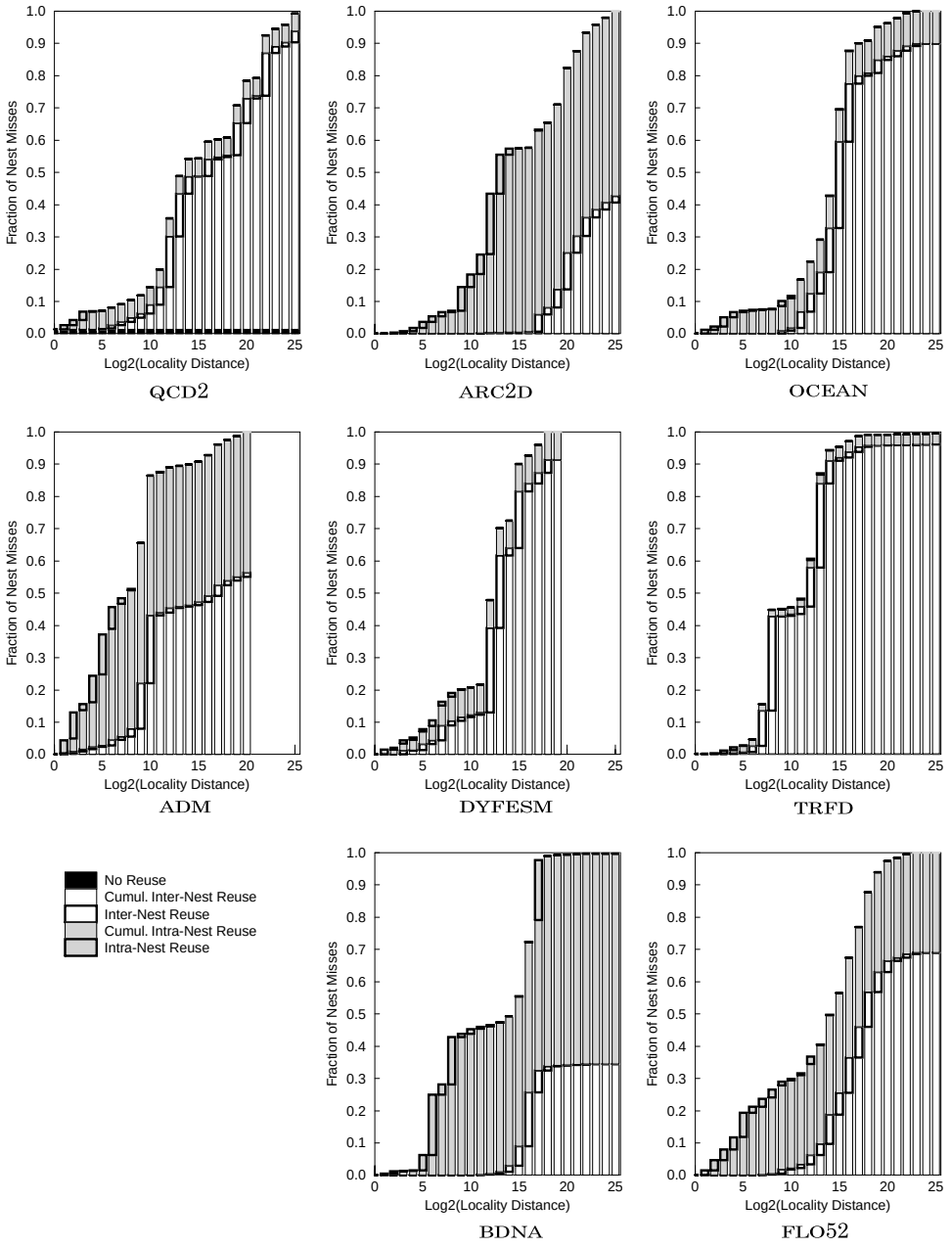

Fig. 27. Sun intranest versus internest misses.

Alpha is consistent except for ARC2D, these individual variations are far less consistent. The same observation applies to the temporal/spatial and group/self locality characteristics. There are many differences in the individual nests that do not show a systematic evolution between architectures. For instance, QCD2 and OCEAN have 45% and 35% intranest self-spatial

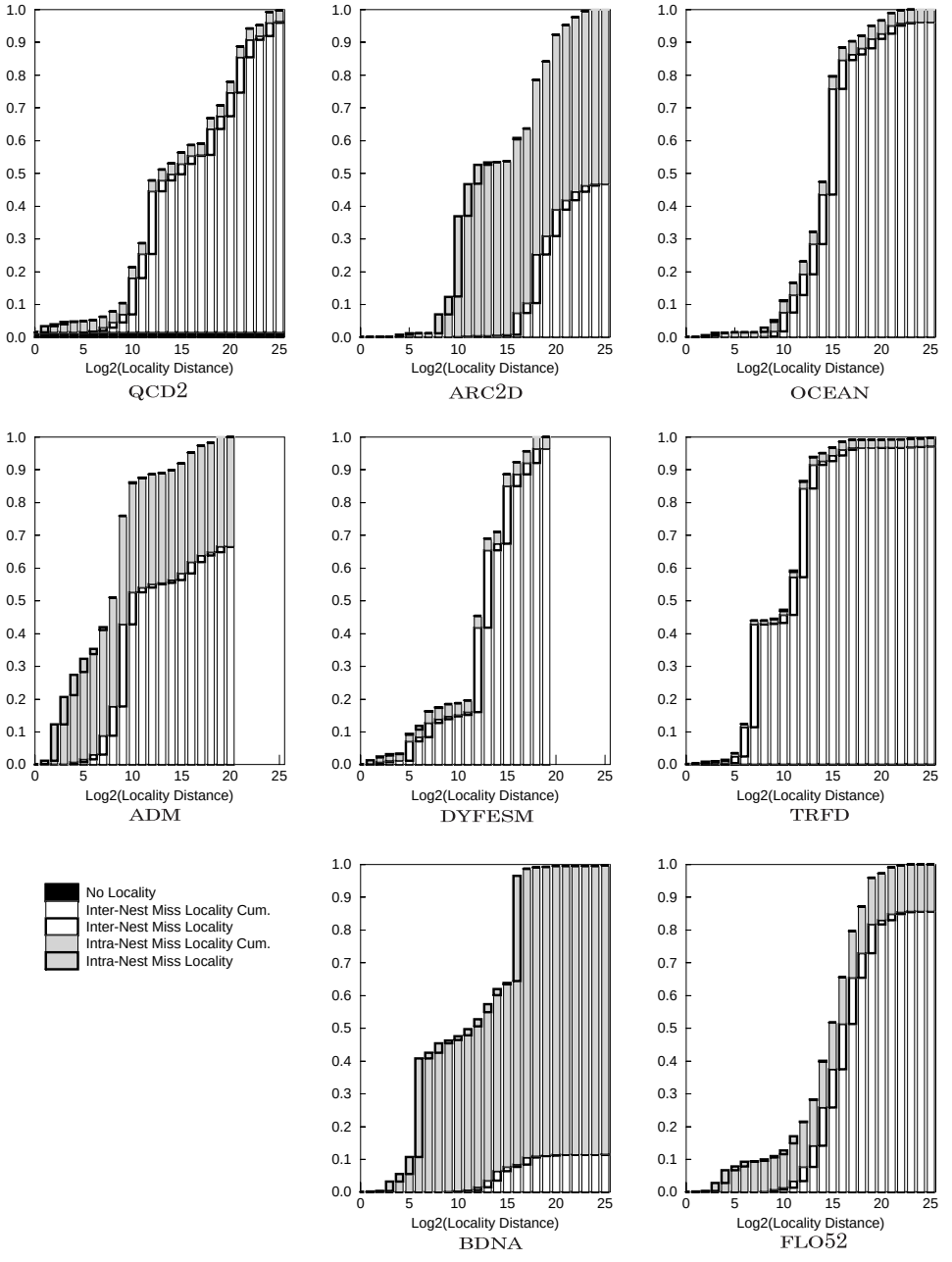

Fig. 28. Dec intranest versus internest misses.

reuse respectively on the Sparc and 35% and 10% respectively on the Alpha, while ARC2D has 20% on the Sparc and 35% on the Alpha. There are many other examples of this type of dichotomy. It is difficult to extract

the locality characteristics of a code just by analyzing the source code, since the compilation process greatly affects locality characteristics.

9.3 Analysis of Individual Nests

In previous sections, we evaluated the impact of the cache architecture and processor/compiler platform on locality. In this section, we examine in more detail the relationship between locality and source code structure. For that purpose, we isolate the three nests from each Perfect Benchmark with the most misses from the results on the Sparc with cache-1 (8KB, 32-byte block, direct-mapped). Table VIII shows the total number of misses and the total miss rate of these nests, ranked by number of misses.

For most codes, we found the nests that miss the most on the Alpha and Sparc are usually the same, and they are often ranked the same. However, the two different cache architectures often change the nest rank, and sometimes the top three nests with the most misses also differ. Even though most Sparc traces have double the number of references due to two memory loads for each fetch of a double-precision number, the number of misses is often similar.

Below, we briefly analyze the nests with the most misses in each code. For some of the nests discussed below, the explanation for their high number of misses is sometimes straightforward: they are simply one of the biggest nest in a heavily called routine. Other nests present more interesting behavior that often is the result of interference.

9.3.1 *ADM*. Nest 153 in Figure 29 is over 20 lines of code with plenty of floating-point computations. Its total number of misses is at least 1.5 to 2.5 times higher than all other nests in ADM, and thus its impact on total code performance is significant. Its miss ratio is rather high (26% in Dec-8k) which is due to conflict misses between the multiple array references. Notice the boundary code is within the loop, which introduces unnecessary branches in the inner loop.

9.3.2 *QCD2*. The source code of Nest 7 is shown below. It has more than three times the number of misses than the next highest missing nest (almost eight times for Dec-32k). It is included in a small routine that is heavily referenced through out the code. Its miss ratio is rather low, so there is no conflict between the two arrays. The only strategies for optimizing such loops are interprocedural analysis or inlining.

```
      do 1 i = 1,n
    1 outmat(i) = inmat(i)
```

9.3.3 *BDNA*. Nest 137 in Figure 30 in BDNA is one of the largest nests. It largely dominates the code in terms of misses except for Nest 237 with the Sun-8k. However, this latter nest has a significantly higher miss ratio, and it disappears from the top-three list in the Dec configurations. This result suggests that changes in the base address of the arrays remove the conflict misses. The only surprise for this nest is its low miss rate, considering the very high number of array references; note the arrays are singly dimensioned, and the accesses are stride-1.

Table VIII. Top Three Missing Nests in the Perfect Benchmarks on Sun-8k

Code	Arch/Cache	Nest 1	Misses	Miss Rate	Nest 2	Misses	Miss Rate	Nest 3	Misses	Miss Rate
ADM	Sun/8k	153	8548125	0.13	12	5577120	0.20	16	3352320	0.12
ADM	Dec/8k	153	8118226	0.26	16	3470400	0.27	12	3239280	0.23
ADM	Dec/32k	12	2078642	0.15	153	1304918	0.04	69	362022	0.02
QCD2	Sun/8k	7	2522079	0.07	8	728779	0.02	90	321405	0.01
QCD2	Dec/8k	7	2517463	0.07	8	665078	0.02	90	292549	0.01
QCD2	Dec/32k	7	1702075	0.04	88	122184	0.04	90	79068	0.00
BDNA	Sun/8k	137	49123493	0.03	237	29259933	0.13	133	10382977	0.02
BDNA	Dec/8k	137	44403656	0.07	133	6524433	0.07	134	2422868	0.33
BDNA	Dec/32k	137	18422230	0.03	133	2223877	0.02	134	1691669	0.23
OCEAN	Sun/8k	73	70255661	0.05	62	45861789	0.13	63	32382390	0.11
OCEAN	Dec/8k	73	68795630	0.08	62	40404109	0.12	63	32238427	0.11
OCEAN	Dec/32k	73	50888948	0.06	62	39542650	0.11	63	30637108	0.10
DYFESM	Sun/8k	64	4703557	0.01	11	2661243	0.09	70	759731	0.05
DYFESM	Dec/8k	64	4513137	0.01	11	2264845	0.08	70	1177128	0.06
DYFESM	Dec/32k	11	978548	0.03	64	600100	0.00	97	167130	0.08
ARC2D	Sun/8k	42	46165329	0.19	45	43756713	0.15	79	30329901	0.17
ARC2D	Dec/8k	42	71204900	0.67	45	66390600	0.42	79	46723200	0.58
ARC2D	Dec/32k	45	31447800	0.19	48	22179600	0.21	42	21300200	0.20
FLO52	Sun/8k	28	8421252	0.07	25	7384713	0.06	22	5764679	0.07
FLO52	Dec/8k	28	5986890	0.12	25	4780648	0.09	22	4489392	0.11
FLO52	Dec/32k	25	3447399	0.07	28	3428811	0.07	22	3337138	0.08
TRFD	Sun/8k	16	37288100	0.05	22	9595896	0.04	18	8744435	0.02
TRFD	Dec/8k	16	36216625	0.11	22	8840572	0.07	18	7861260	0.04
TRFD	Dec/32k	3	421955	0.24	20	419780	0.14	16	197308	0.00

9.3.4 *OCEAN*. Nest 73 (shown below) is short and is more comparable to the other nests in OCEAN. Even though the nest miss ratio is low (less than 10%), it has a high impact on the total number of misses because it is heavily referenced (334604 times). Notice that even though the arrays are only one dimensional, the access patterns are not stride-1.

```
do 106 jj = jl,npts,i2kp
  do 106 mm = 1,mtrn
       js = (jj - 1) * nskip + (mm - 1) * mskip + 1
       h = data(js) - data(js + i2ks)
       data(js) = data(js) + data(js + i2ks)
       data(js + i2ks) = h * exj
106 continue
```

9.3.5 *DYFESM*. Nest 64 clearly dominates for 8KB caches. It is the inner nest of a matrix-matrix multiplication (with control flow statements between other loops, hence the single-level loop) and is the classical target of voluminous amounts of research. Despite the fact that matrix multiply is widely targeted for optimization, this instance is the only program in which the main nest is a straight matrix-matrix or matrix-vector multiplication.

```
  do 200 i = 1,l
    c(i,k) = c(i,k) + a(i,j) * temp
200 continue
```

```
      do 10 k = 1,nz
         if (i .eq. 1 .or. i .eq. nx) then
             tx = 0.5 * pott(i,j,k)
           wux = 0.5 * ux(i,j,k)
           wvx = 0.5 * vy(i,j,k)
         else
           tx = 0.25 * (pott(i - 1,j,k) + pott(i + 1,j,k))
           wux = 0.25 * (ux(i - 1,j,k) + ux(i + 1,j,k))
           wvx = 0.25 * (vy(i - 1,j,k) + vy(i + 1,j,k))
         endif
         if (j .eq. 1 .or. j .eq. ny) then
            ty = 0.5 * pott(i,j,k)
           wuy = 0.5 * ux(i,j,k)
           wvy = 0.5 * vy(i,j,k)
         else
           ty = 0.25 * (pott(i,j - 1,k) + pott(i,j + 1,k))
           wuy = 0.25 * (ux(i,j - 1,k) + ux(i,j + 1,k))
           wvy = 0.25 * (vy(i,j - 1,k) + vy(i,j + 1,k))
         endif
         xw = (ux(i,j,k) + wux + wuy) / 2. + um(k)
         yw = (vy(i,j,k) + wvx + wvy) / 2. + vm(k)
         u(k) = sqrt (xw * xw + yw * yw)
         t(k) = (pott(i,j,k) + tx + ty) / 2. + tm(k)
10    continue
```

Fig. 29. The nest with the most misses in ADM.

9.3.6 *ARC2D*. ARC2D is characterized by its high miss rate as is its main nest, Nest 42 in Figure 31, with a 67% miss ratio on the Dec-8k configuration. These misses are conflict misses, and thus two-way associativity has a profound effect, decreasing the miss rate to 20%.

9.3.7 *FLO52*. Nest 28 in Figure 30 from the EFLUX routine incurs the most or second most number of misses for the different configurations. It is a very standard case: its miss ratio is relatively low, and the routine corresponds to a large fraction of total references, misses, and execution time. Its references and subscripts are representative of many other floating-point codes: stride-1 accesses and group-temporal reuse.

9.3.8 *TRFD*. There are 10 nests in the same routine (OLDA). The first nest in Figure 33, Nest 16 and the second, Nest 22, have almost the same shape and execute the same number of times, but Nest 16 has significantly more misses. Looking at the Dec-32k statistics, it seems the low miss rate of the Nest 16 (0.05 on Sun-8k and 0.11 on Dec-8k) is misleading, since it sharply drops to 0.0006 on Dec-32k, suggesting there were many conflicts or capacity misses in this nest.

9.4 Summary of Architecture and Compiler Influences

In Section 9.1, we increase both the size and associativity of the cache and find the expected decrease in miss rates, and corresponding decreases in the intranest misses. However, the increase in cache size from 8K to 32K does not eliminate many intranest capacity misses. The eliminated misses are conflict misses at short distances. These effects combine to increase the effectiveness of prefetching on cache-2. Section 9.2 demonstrates that

```
      do 350 i = 1,nsp
         ins = (i - 1) * isit
         do 500 ia = 1,natoms
            k = ia + nop
            xd = x0(i) - x0(k)
            yd = y0(i) - y0(k)
            zd = z0(i) - z0(k)
            xd = xd - 2.d0 * alengt * dble (int (xd * alengm))
            yd = yd - 2.d0 * alengt * dble (int (yd * alengm))
            zd = zd - 2.d0 * dble (int (zd))
            dx = xd + sx(ins + 1)
            dy = yd + sy(ins + 1)
            dz = zd + sz(ins + 1)
            dd = dx * dx + dy * dy + dz * dz
            ddi = 1.d0 / dd
            dsqi = sqrt (ddi)
            d6i = ddi * ddi * ddi
            d12i = d6i * d6i
            t1 = aa(1,ia) * dsqi
            t2 = bb(1,ia) * d6i
            t3 = cc(1,ia) * d12i
            ewat = t1 + t2 + t3
            ggg = t1 + 6.d0 * t2 + 12.d0 * t3
            ggg = ggg * ddi
            fox(i) = fox(i) + dx * ggg
            foy(i) = foy(i) + dy * ggg
            foz(i) = foz(i) + dz * ggg
            fax(ia) = fax(ia) + dx * ggg
            fay(ia) = fay(ia) + dy * ggg
            faz(ia) = faz(ia) + dz * ggg
            dx = xd + sx(ins + 2)
            dy = yd + sy(ins + 2)
            dz = zd + sz(ins + 2)
            dd = dx * dx + dy * dy + dz * dz
            ddi = 1.d0 / dd
            dsqi = sqrt (ddi)
            d6i = ddi * ddi * ddi
            d12i = d6i * d6i
            t1 = aa(2,ia) * dsqi
            t2 = bb(2,ia) * d6i
            t3 = cc(2,ia) * d12i
            ewat = ewat + t1 + t2 + t3
            ggg = t1 + 6.d0 * t2 + 12.d0 * t3
            ggg = ggg * ddi
            f1x(i) = f1x(i) + dx * ggg
            f1y(i) = f1y(i) + dy * ggg
            f1z(i) = f1z(i) + dz * ggg
            fax(ia) = fax(ia) + dx * ggg
            fay(ia) = fay(ia) + dy * ggg
            faz(ia) = faz(ia) + dz * ggg
            dx = xd + sx(ins + 3)
            dy = yd + sy(ins + 3)
            dz = zd + sz(ins + 3)
            dd = dx * dx + dy * dy + dz * dz
            ddi = 1.d0 / dd
            dsqi = sqrt (ddi)
            d6i = ddi * ddi * ddi
            d12i = d6i * d6i
            t1 = aa(3,ia) * dsqi
            t2 = bb(3,ia) * d6i
            t3 = cc(3,ia) * d12i
            ewat = ewat + t1 + t2 + t3
            ggg = t1 + 6.d0 * t2 + 12.d0 * t3
            ggg = ggg * ddi
            f2x(i) = f2x(i) + dx * ggg
            f2y(i) = f2y(i) + dy * ggg
            f2z(i) = f2z(i) + dz * ggg
            fax(ia) = fax(ia) + dx * ggg
            fay(ia) = fay(ia) + dy * ggg
            faz(ia) = faz(ia) + dz * ggg
350         ewa = ewa + ewat
500      continue
```

Fig. 30. The nest with the most misses in BDNA.

```
      do 212 j = jlow,jup
         jp1 = jplus(j)
         jm1 = jminu(j)
         jp2 = jplus(jp1)
         jm2 = jminu(jm1)
         do 210 k = klow,kup
            c2m = coef2(jm1,k) * dtd
            c4m = coef4(jm1,k) * dtd
            c2 = coef2(j,k) * dtd
            c4 = coef4(j,k) * dtd
            work(j,k,1) = xyj(jm2,k) * c4m
            work(j,k,2) =  (-(c2m + 3. * c4m + c4)) * xyj(jm1,k)
            work(j,k,3) = xyj(j,k) * (c2m + 3. * c4m + c2 + 3. * c4)
            work(j,k,4) =  (-(c2 + 3. * c4 + c4m)) * xyj(jp1,k)
            work(j,k,5) = xyj(jp2,k) * c4
212         continue
210      continue
```

Fig. 31. The nest with the most misses in ARC2D.

```
      do 878 j = 2,jl
         do 879 i = 2,il
            xx = x(i,j,1) - x(i - 1,j,1)
            yx = x(i,j,2) - x(i - 1,j,2)
            pa = p(i,j + 1) + p(i,j)
            qsp = (xx * w(i,j + 1,3) - yx * w(i,j + 1,2)) / w(i,j + 1,1)
            qsm = (xx * w(i,j,3) - yx * w(i,j,2)) / w(i,j,1)
            fs(i,j,1) = qsp * w(i,j + 1,1) + qsm * w(i,j,1)
            fs(i,j,2) = qsp * w(i,j + 1,2) + qsm * w(i,j,2) - yx * pa
            fs(i,j,3) = qsp * w(i,j + 1,3) + qsm * w(i,j,3) + xx * pa
            fs(i,j,4) = qsp * (w(i,j + 1,4) + p(i,j + 1)) + qsm *
     (w(i,j+1,4) + p(i,j))
30          continue
879          continue
878      continue
```

Fig. 32. The nest with the most misses in FLO52.

```
C     Nest 16
      do 20 mi = 1,morb
         xrsiq(mi,mq) = xrsiq(mi,mq) + val * v(mp,mi)
         xrsiq(mi,mp) = xrsiq(mi,mp) + val * v(mq,mi)
20    continue

C     Nest 22
      do 220 mk = mi,morb
         xijks(mk,ms) = xijks(mk,ms) + val * v(mr,mk)
         xijks(mk,mr) = xijks(mk,mr) + val * v(ms,mk)
220   continue
```

Fig. 33. The nest with the most misses in TRFD.

platform effects are on average small, but can vary significantly between programs and that the trends between platforms are inconsistent across benchmarks. Finally, Section 9.3 confirms the significance of intranest

conflict misses, and the inability of modest increases in cache size to remove intranest and internest capacity misses.

10. CONCLUSIONS

In this article, 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 many misses are actually internest 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 intranest reuse, but temporal locality dominates internest, and program reuse in our test suite. We find conflict misses the most significant source of intranest misses and two-way set associativity unable to resolve many of them. Our results are fairly consistent across our test suites, a few cache organizations, and architectures.

Based on these results, 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 interloop reuse; and compilers and architects still need to address the interference problem. In summary, there is still plenty of room to improve the efficiency and effectiveness of data caches.

APPENDIX

COMPARING SPEC'95 AND PERFECT

For the purposes of comparison, the figures in this appendix present the average graphs for SPEC'95 and Perfect Benchmarks from Alpha traces for all the experiments described in Sections 4–8. Section 8 references these graphs, explaining and summarizing their differences and similarities. The SPEC numbers use a 32KB two-way set-associative cache. The Perfect numbers use an 8KB direct-mapped cache. Both use a 32-byte block size (cache line).

ACKNOWLEDGMENTS

We would like to thank Amer Diwan, Christine Eisenbeis, Mark Hill, Toni Juan Hormigo, Nathaniel McIntosh, Sharad Singhai, Darko Stefanovic, and Chau-Wen Tseng for their suggestions after reading various versions of this article.

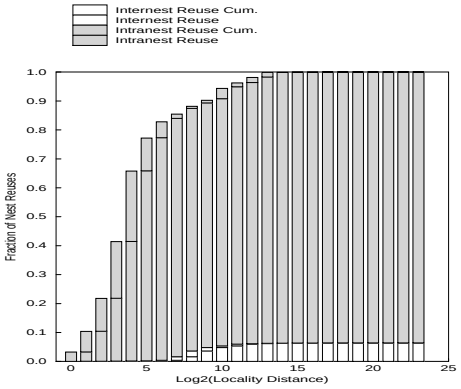

Fig. 34. SPEC: Intranest versus internest reuse.

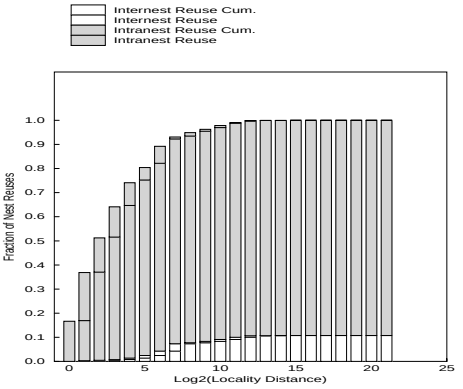

Fig. 37. Perfect: Intranest versus internest reuse.

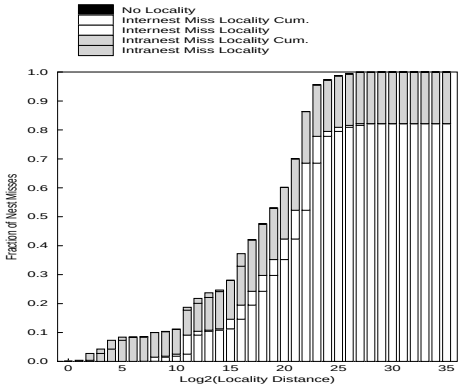

Fig. 35. SPEC: Intranest versus internest misses.

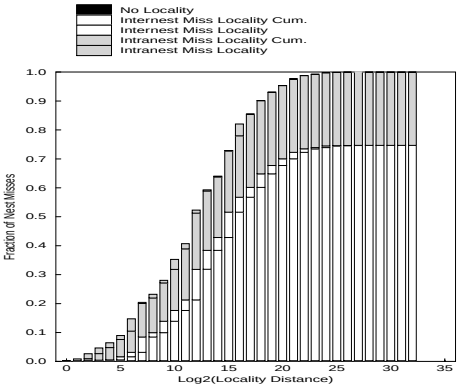

Fig. 38. Perfect: Intranest versus internest misses.

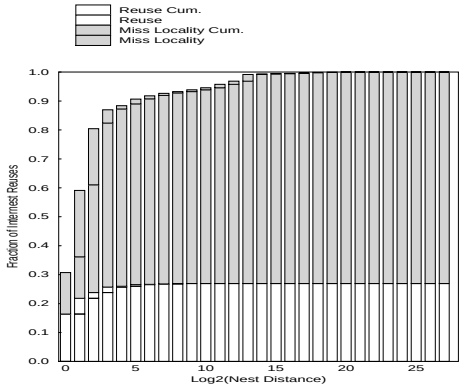

Fig. 36. SPEC: Nest distance of internest locality.

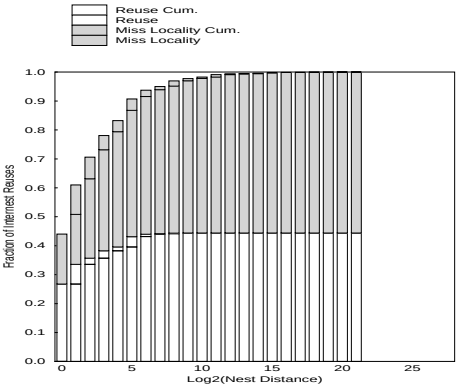

Fig. 39. Perfect: Nest distance of internest locality.

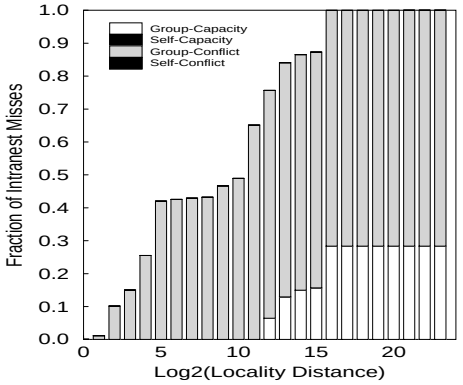

Fig. 40. SPEC: Capacity/conflict intranest misses.

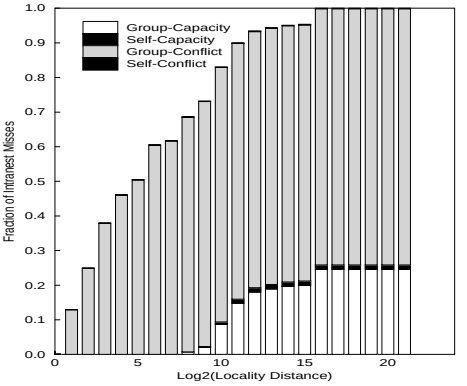

Fig. 43. Perfect: Capacity/conflict intranest misses.

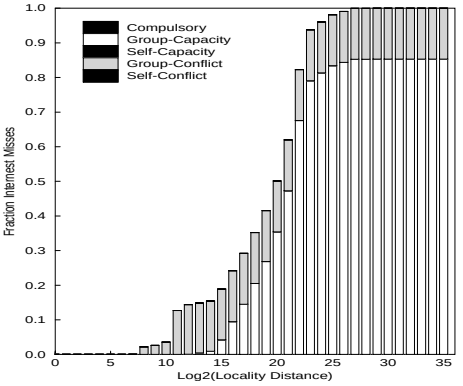

Fig. 41. SPEC: Capacity/conflict internest misses.

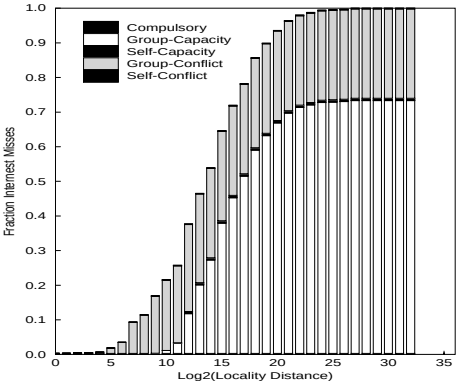

Fig. 44. Perfect: Capacity/conflict internest misses.

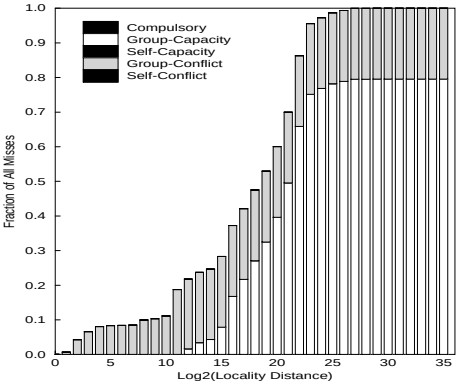

Fig. 42. SPEC: Capacity/conflict program misses.

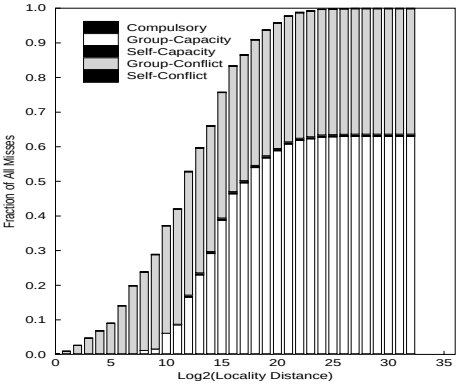

Fig. 45. Perfect: Capacity/conflict program misses.

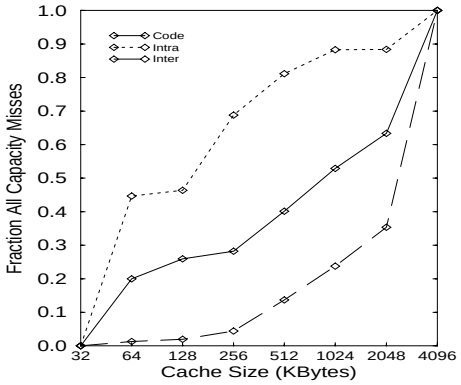

Fig. 46. SPEC: Cache size versus capacity misses.

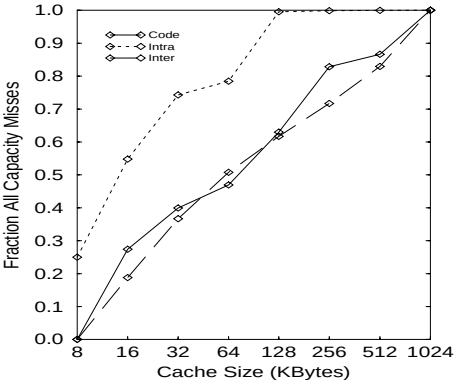

Fig. 49. Perfect: Cache size versus capacity misses.

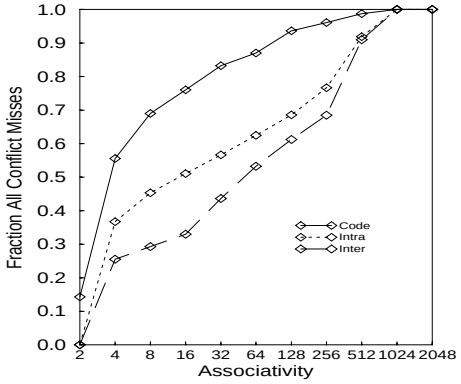

Fig. 47. SPEC: Associativity versus conflict misses.

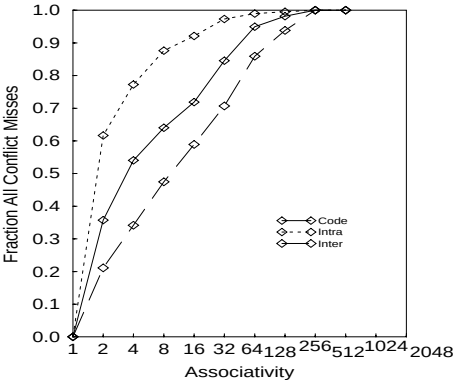

Fig. 50. Perfect: Associativity versus conflict misses.

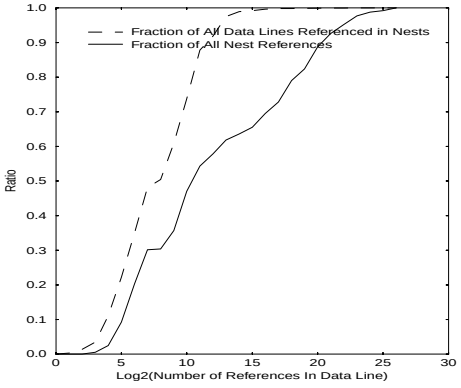

Fig. 48. SPEC: Uses per data line.

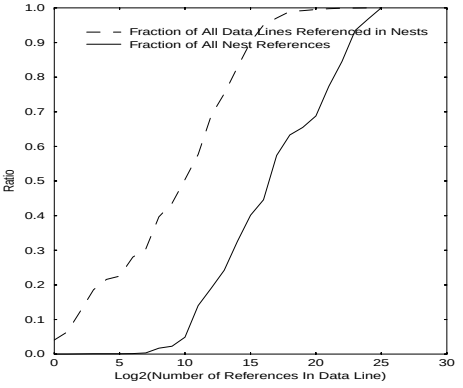

Fig. 51. Perfect: Uses per data line.

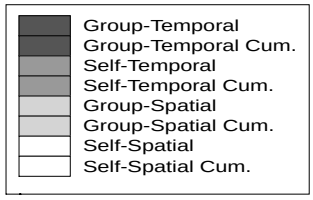

Fig. 52. Legend for Figures 53–58.

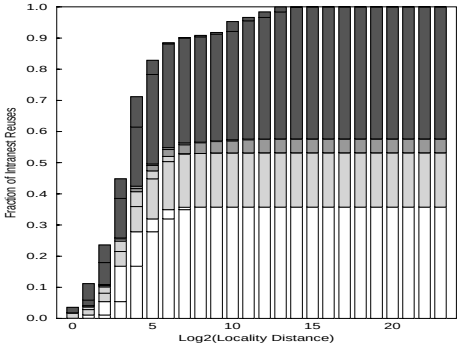

Fig. 53. SPEC: Intranest reuse locality.

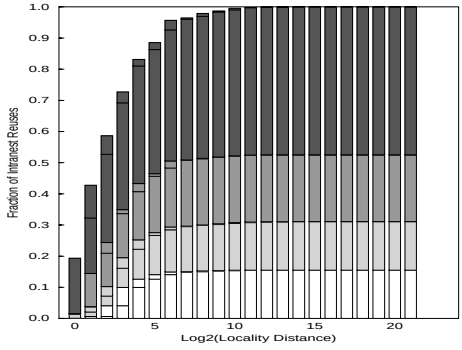

Fig. 56. Perfect: Intranest reuse locality.

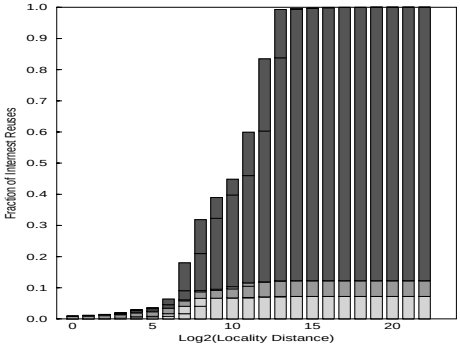

Fig. 54. SPEC: Internest reuse locality.

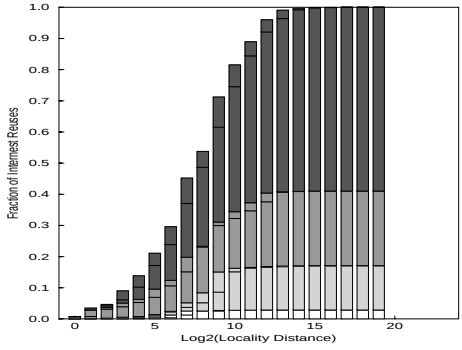

Fig. 57. Perfect: Internest reuse locality.

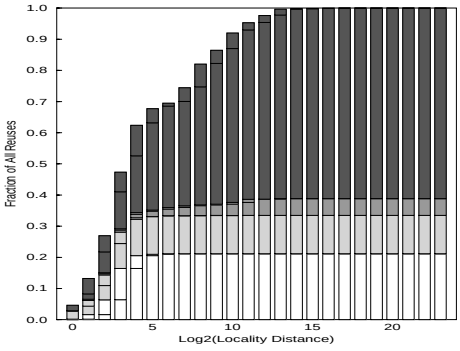

Fig. 55. SPEC: Program reuse locality.

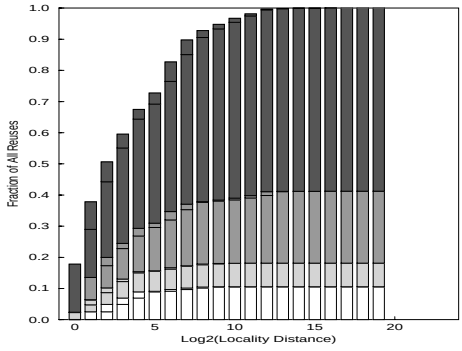

Fig. 58. Perfect: Program reuse locality.

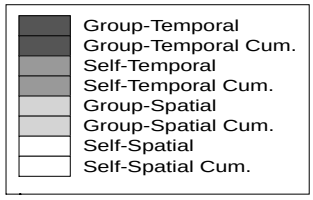

Fig. 59. Legend for Figures 60–65.

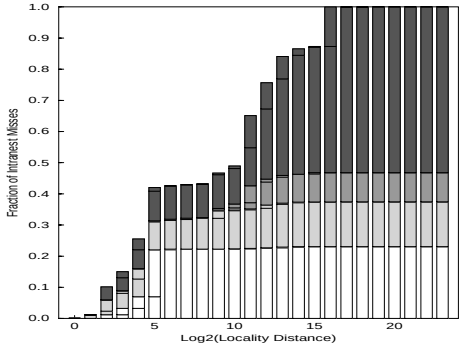

Fig. 60. SPEC: Intranest miss locality.

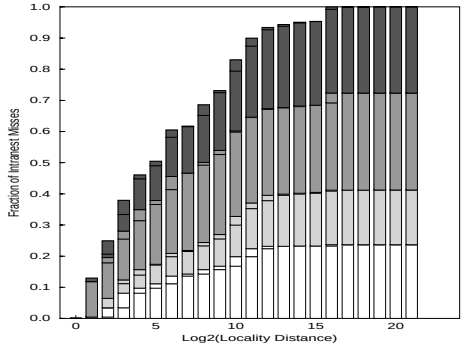

Fig. 63. Perfect: Intranest miss locality.

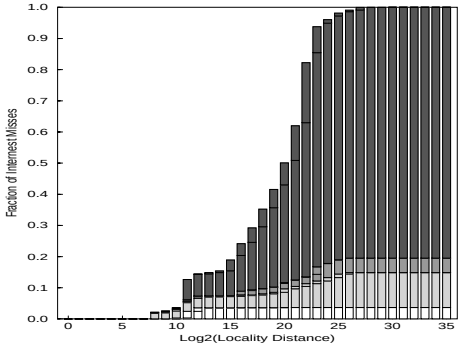

Fig. 61. SPEC: Internest miss locality.

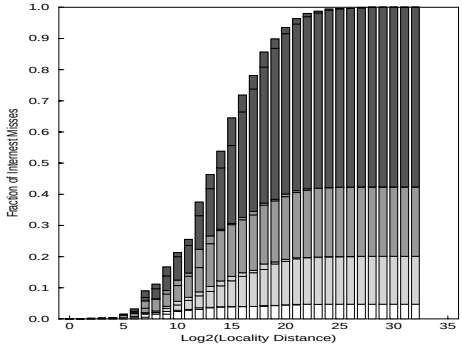

Fig. 64. Perfect: Internest miss locality.

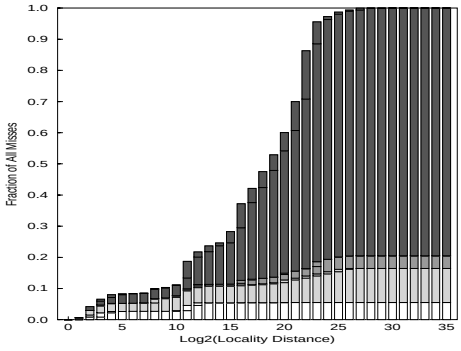

Fig. 62. SPEC: Program miss locality.

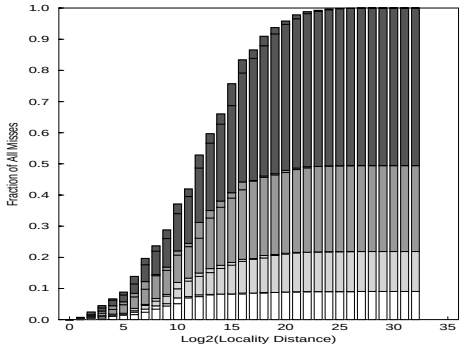

Fig. 65. Perfect: Program miss locality.

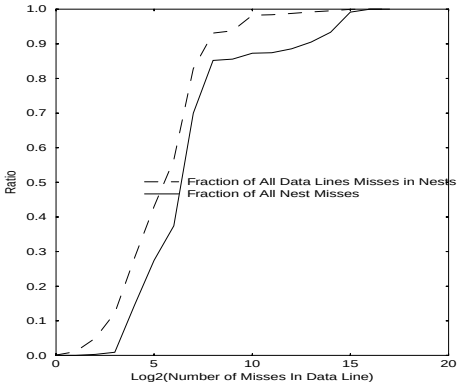

Fig. 66. SPEC: Line miss characteristics.

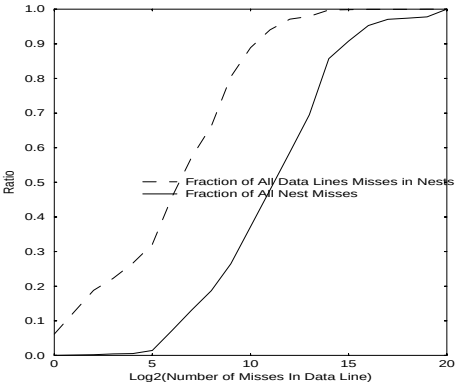

Fig. 69. Perfect: Line miss characteristics.

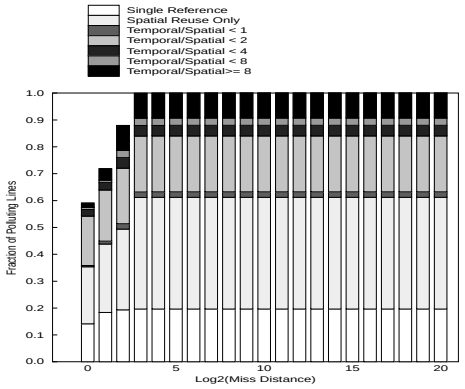

Fig. 67. SPEC: Polluting line locality.

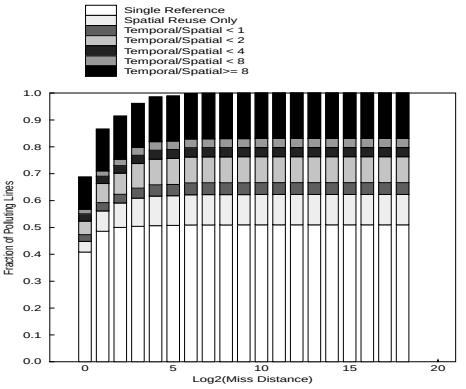

Fig. 70. Perfect: Polluting line locality.

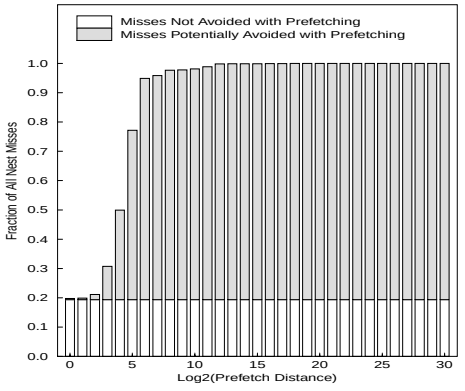

Fig. 68. SPEC: Potential of prefetching.

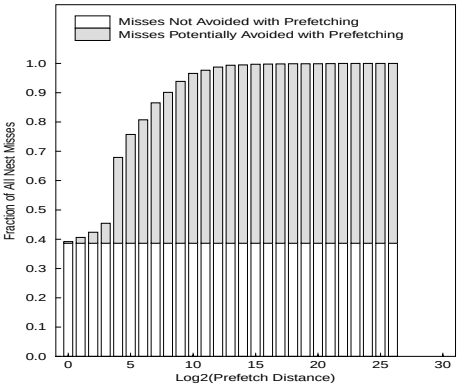

Fig. 71. Perfect: Potential of prefetching.

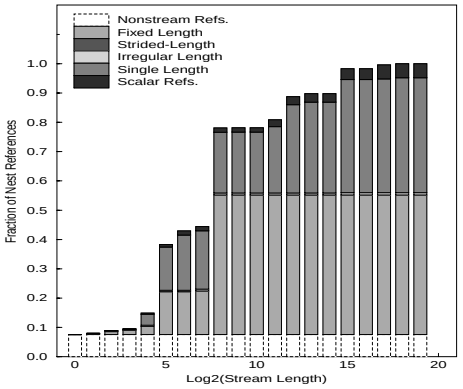

Fig. 72. SPEC: Stream types, lengths, and frequencies.

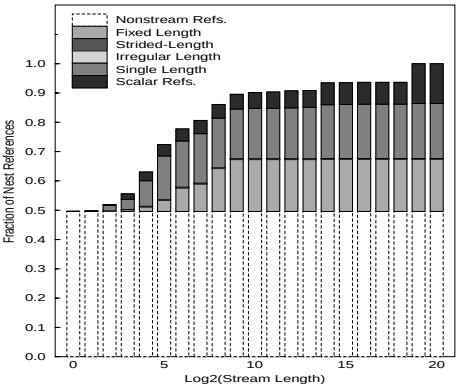

Fig. 74. Perfect: Stream types, lengths, and frequencies.

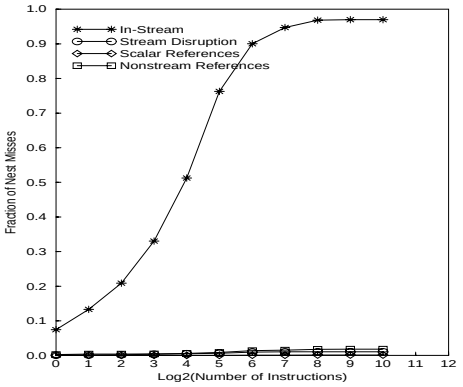

Fig. 73. SPEC: Stream, scalar, and out-of-stream misses.

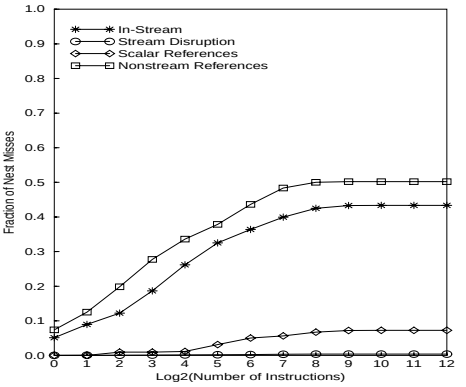

Fig. 75. Perfect: Stream, scalar, and out-of-stream misses.

REFERENCES

ABRAHAM, S. G., SUGUMAR, R. A., WINDHEISER, D., RAU, B. R., AND GUPTA, R. 1993. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture* (MICRO 26, Austin, TX, Dec. 1–3), A. Wolfe and W. Mangione-Smith, Eds. IEEE Computer Society Press, Los Alamitos, CA, 139–152.

AGARWAL, A. AND PUDAR, S. D. 1993. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *SIGARCH Comput. Arch. News 21*, 2 (May), 179–190.

BAER, J.-L. AND CHEN, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 Conference on Supercomputing* (Albuquerque, NM, Nov. 18–22), J. L. Martin, Ed. ACM Press, New York, NY, 176–186.

BELADY, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J. 5*, 2, 79–101.

BODIN, F., BECKMAN, P., GANNON, D., GOTWALS, J., NARAYANA, S., SRINIVAS, S., AND WINNICKA, B. 1994. Sage++: An object-oriented toolkit and class library for building Fortran and C++ structuring tools. In *Proceedings of the 2nd Annual Object-Oriented Numerics Conference* (OON-SKI '94, Sun River, OR, Apr.).

BURGER, D., GOODMAN, J. R., AND KÄGI, A. 1996. Memory bandwidth limitations of future microprocessors. *SIGARCH Comput. Arch. News 24*, 2, 78–89.

CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. *SIGPLAN Not. 25*, 6 (June), 53–65.

CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. 1991. Software prefetching. *SIGARCH Comput. Arch. News 19*, 2 (Apr. 1991), 40–52.

CARR, S. AND KENNEDY, K. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov.), 1768–1810.

CHEN, T. F. AND BEAR, J. L. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput. 44*, 5 (May), 609–623.

COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. *SIGPLAN Not. 30*, 6 (June 1995), 279–290.

COOPER, K., KENNEDY, K., AND MCINTOSH, N. 1995. An emprical study of cross-loop reuse in the NAS benchmarks. Tech. Rep. CRPC-TR95519-S. Center for Research on Parallel Computation, Rice University, Houston, TX.

COOPER, K., KENNEDY, K., AND MCINTOSH, N. 1996. Cross-loop reuse analysis and its application to cache optimizations. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing* (Santa Clara, CA).

CYBENKO, G., KIPP, L., POINTER, L., AND KUCK, D. 1990. Supercomputer performance evaluation and the Perfect Benchmarks. *SIGARCH Comput. Arch. News 18*, 3, 254–266.

DEC. 1994. Alpha 21164 microprocessor, hardware reference manual. Digital Equipment Corp., Maynard, MA.

DRACH, N. 1995. Hardware implementation issues of data prefetching. In *Proceedings of the 9th ACM International Conference on Supercomputing* (ICS '95, Barcelona, Spain, July 3–7, 1995), M. Valero, Ed. ACM Press, New York, NY, 245–254.

GANNON, D., JALBY, W., AND GALLIVAN, K. 1988. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput. 5*, 5 (Oct. 1988), 587–616.

GEE, J. D., HILL, M. D., AND PNEVMATIKATOS, D. N. 1993. Cache performance of the SPEC92 benchmark suite. *IEEE Micro 13*, 4 (Aug.), 17–27.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1998. Precise miss analysis for program transformations with caches of arbitrary associativity. *SIGPLAN Not. 33*, 11, 228–239.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HILL, M. D. 1987. Aspects of cache memory and instruction buffer performance. Ph.D. Dissertation. Computer Science Department, University of California at Berkeley, Berkeley, CA.

HILL, M. D. 1988. A case for direct-mapped caches. *IEEE Computer 21*, 12 (Dec. 1988), 25–40.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1612–1631.

JOUPPI, N. P. 1998. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *Computer Architecture* (ISCA '98), G. S. Sohi, Ed. ACM Press, New York, NY, 388–397.

KAPLAN, K. R. AND WINDER, R. O. 1973. Cache based computer systems. *IEEE Computer 6*, 3, 30–36.

KLAIBER, A. C. AND LEVY, H. M. 1991. An architecture for software-controlled data prefetching. *SIGARCH Comput. Arch. News 19*, 3 (May 1991), 43–53.

LAM, M., ROTHBERG, E., AND WOLF, M. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-IV, Santa Clara, CA, Apr. 8–11), D. A. Patterson, Ed. ACM Press, New York, NY, 63–74.

LI, W. AND PINGALI, K. 1992. Access normalization: Loop restructuring for NUMA compilers. *SIGPLAN Not. 27*, 9 (Sept. 1992), 285–295.

MCKEE, S. A. AND WULF, W. A. 1996. A memory controller for improved performance of streamed computations on symmetric multiprocessors. In *Proceedings of 25th International Conference on Parallel Processing* (Aug.).

McKinley, K. S. and Temam, O. 1996. A quantitative analysis of loop nest locality. *ACM SIGOPS Oper. Syst. Rev. 30*, 5, 94–104.

McKinley, K. S., Carr, S., and Tseng, C.-W. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4 (July), 424–453.

Mowry, T. C., Lam, M. S., and Gupta, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-V, Boston, MA, Oct. 12–15), S. Eggers, Ed. ACM Press, New York, NY, 62–73.

Przybylski, S., Horowitz, M., and Hennessy, J. 1988. Performance tradeoffs in cache design. In *The 15th Annual International Symposium on Computer Architecture* (ISCA '88, Honolulu, HI, May 30–June 2), H. J. Siegel, Ed. IEEE Computer Society Press, Los Alamitos, CA, 290–298.

Reilly, J. 1995. SPEC describes SPEC'95 product and benchmarks. *SPEC Newslett.* (Sept.). Available via http://www.spec.org/osg/news/articles/news9509/cpu95descr.html.

Smith, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept.), 473–530.

Smith, A. J. 1986. Bibliography and readings on CPU cache memories and related topics. *SIGARCH Comput. Arch. News 14*, 1 (Jan. 1986), 22–42.

Smith, A. J. 1987. Line (block) size choice for CPU cache memories. *IEEE Trans. Comput. C-36*, 9 (Sept. 1987), 1063–1076.

Smith, A. J. 1991. Second bibliography on cache memories. *SIGARCH Comput. Arch. News 19*, 4 (June 1991), 154–182.

Srivastava, A. and Eustace, A. 1994. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language, Design and Implementation* (PLDI '94, Orlando, FL, June 20–24, 1994), V. Sarkar, B. Ryder, and M. L. Soffa, Eds. ACM Press, New York, NY, 196–205.

Sugumar, R. A. and Abraham, S. G. 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. *SIGMETRICS Perform. Eval. Rev. 21*, 1 (June 1993), 24–35.

Temam, O., Granston, E. D., and Jalby, W. 1993. 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 the Conference on Supercomputing* (Supercomputing '93, Portland, OR, Nov. 15–19), B. Borchers and D. Crawford, Eds. IEEE Computer Society Press, Los Alamitos, CA, 410–419.

Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A. R. 1995. A modified approach to data cache management. In *Proceedings of the 28th annual international symposium on Microarchitecture* (Ann Arbor, MI, Nov. 29 - Dec. 1, 1995), T. Mudge and K. Ebcioğlu, Eds. IEEE Computer Society Press, Los Alamitos, CA, 93–103.

Wolf, M. E. and Lam, M. S. 1991. A data locality optimization algorithm. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (SIGPLAN '91, Toronto, Ontario, Canada, June 26–28), D. S. Wise, Ed. ACM Press, New York, NY, 30–44.

Wolfe, M. 1987. Iteration space titling for memory hiearchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing* (Dec.) SIAM, Philadelphia, PA.

Wood, D. A., Hill, M. D., and Kessler, R. E. 1991. A model for estimating trace-sample miss ratios. *SIGMETRICS Perform. Eval. Rev. 19*, 1 (May 1991), 79–89.

Received: April 1998; accepted: September 1999