An Empirical Study of Cross-Loop Reuse in the NAS Benchmarks

Keith Cooper
Ken Kennedy
Nathaniel McIntosh

CRPC-TR95519
March 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

revised September 1996

An Empirical Study of Cross-loop Reuse in the NAS benchmarks

Keith Cooper Ken Kennedy Nathaniel McIntosh
Department of Computer Science
Rice University
Houston, Texas, USA

Abstract

This paper describes an empirical study designed to quantify the level of *cross-loop* reuse occurring in a set of scientific Fortran programs, the NAS Benchmarks. Cross-loop reuse takes place when a set of data items or cache lines are accessed in a given loop nest and then accessed again within some subsequent portion of the program (usually another outer loop nest). In contrast to *intra-loop* reuse, which takes place during the execution of a single loop nest, cross-loop reuse is not always detectable by traditional compile-time reuse analysis techniques. In this study, the benchmark programs are instrumented and run through a cache simulator. The simulator gathers statistics on cross-loop reuse using a novel classification scheme that clearly identifies the different types of reuse. According to the simulation data, the level of cross-loop reuse varies widely from program to program, and depends greatly on the problem size and cache size. Some programs exhibit almost no cross-loop reuse, however other programs have significant levels of cross-loop reuse even for fairly small cache sizes. The data from this study suggest that cross-loop reuse information would be quite beneficial for compilers that attempt certain optimizations.

1 Introduction

One of the major trends in computer architecture over the last decade has been the widening gap between processor speed and memory latency. Main memory latencies for modern-day

machines are reaching 100 cycles and beyond, compared with the 1-5 cycle latencies of a decade ago. To deal with this growing disparity, system architects have increasingly turned to cache memories as a means to hide memory latencies and increase effective memory bandwidth. Because the role of the cache has become such an important one, the penalties for programs that don't effectively utilize cache are becoming more and more severe.

For cache memories to be effective with larger data sets, programs need to exhibit *locality of reference* – the tendency to access the same regions or locations repeatedly over time. Within the compiler community, researchers have developed a number of techniques to help programs exploit their locality of reference in order to improve cache utilization. For example, loop interchange [AC72] can be applied to help a program take advantage of inherent spatial locality. Other methods of this sort include loop tiling and unroll-and-jam [CCK90, WL91]. These techniques are driven by various forms of loop level analysis – they focus on identifying locations that are reused within a single loop nest ("intra-loop reuse").

Although intra-loop reuse is important, it is not the only type of reuse that occurs in practice. It is possible for data items to be brought into cache in one loop nest and then used in a subsequent loop nest, or possibly in a subsequently called procedure. We refer to this type of reuse as *cross-loop* reuse. There is very little research that deals with detecting this type of reuse; most people have chosen to focus on locating intra-loop reuse instead. There are a number of reasons for this. First, detecting intra-loop reuse is generally considered easier than detecting cross-loop reuse, since the compiler can focus on smaller regions of the program. Second, there are a variety of well-established loop transformations that can be used to exploit intra-loop reuse, whereas the common view has been that the program transformations for exploiting cross-loop reuse are fewer and more difficult to apply. Finally, there is the question of how frequently cross-loop reuse occurs in application programs. There is little empirical data available in this area, and compiler writers will no doubt be reluctant to develop sophisticated techniques for detection and exploitation of cross-loop reuse if there is no evidence that the work will pay off.

Our primary goal for this paper will be to describe a set of experiments that address the third concern above: quantifying the cross-loop reuse that exists in typical scientific programs. Our experiments are based on a simulation study of the NAS benchmarks [BBLS91a, BBLS91b]. In addition, we will explore possible applications of cross-loop reuse analysis.

An outline of the rest of this paper is as follows. Section 2 provides some motivation for our study by giving an example of a latency-hiding technique that might benefit from cross-loop reuse information. Section 4 describes the experimental framework we used to gather data on cross-loop reuse. In Section 5, we present and interpret the results of the experiments. Section 6 summarizes some of the previous research that relates to this work. Finally, in Section 7 we give our conclusions.

2 Motivating example: software prefetching

Compiler-directed software prefetching [MLG92] is a good example of a cache-management technique that could potentially benefit from cross-loop reuse information.

Software prefetching is a two-phase process. In the first phase, the compiler analyzes the program and seeks to determine the set of variable references that are likely to cause cache misses at run-time. In the second phase, the compiler transforms the program and annotates it with *prefetch* instructions. When executed, a prefetch instruction begins reading a particular data item from main memory into cache. The prefetch operation executes asynchronously – the processor continues to execute other instructions while the prefetch is in progress. If the prefetch instruction for a reference is scheduled sufficiently far in advance of the reference itself, then data required will be in cache when the actual reference takes place, and the latency from the cache miss will be avoided.

Prefetching is not "free"; adding prefetch instructions to a program will increase both static and dynamic instruction counts for the program. One of the primary goals for the compiler is to reduce the overhead from prefetching as much as possible. As a result, the precision of the reuse analysis employed by the compiler is important: prefetches issued for data items that are already in cache will needlessly increase the number of instructions the program executes. If the reuse analysis techniques are confined to examining single loop nests in isolation, they will have to assume that some references are cache misses, when in fact (due to cross-loop reuse) they are cache hits.

Figure 1 shows an abstracted Fortran program. Consider the two loops in the subroutine "`foo`". If the compiler considers the "`do j`" loop in isolation, it will have to assume that the reference to `x(j)` will always result in a cache miss [1] and will have to be prefetched, since it has no other information to the contrary. If the cache is sufficiently large, however, then the elements from the section `x(1:n)` may already be in cache when the second loop executes, eliminating the need to prefetch `x(j)` within that loop.

In fact, if the compiler is able to examine reuse patterns across procedure boundaries, then it may be able to eliminate the prefetch of `x` in *both* the loops in `foo.f`, since `foo` is called repeatedly from the enclosing loop in `main`.

3 Methodology

One of the challenges we face is that it is difficult to measure cross-loop reuse using traditional metrics (i.e. overall miss rates); summary statistics tend to obscure the sources of reuse within a given program. In this section, we describe a new scheme for classifying cache access metrics. Our scheme requires the use of a simulator, and also depends on the

[1] We assume in this example that the cache line size is equal to the size of one array element.

main.f:

```
program main
   do t = 1, n
      call foo
      do k = 1, n
         ...
      enddo
   enddo
end
```

foo.f:

```
subroutine foo
   do i = 1, n
      ... x(i) ...
   enddo
   do j = n-1, 2, -1
      ... x(j) ...
   enddo
end
```

Figure 1: Example program

addition of some instrumentation to the program being examined.

3.1 Program partitioning

Compiler techniques already exist for detecting intra-loop reuse; we instead focus on the reuse that be can't be detected by these techniques. To prepare for this task, we divided our benchmark programs into logical units that could potentially be analyzed by conventional techniques, and then measured the reuse that took place between these units.

For the programs we considered, we performed this partitioning by examining each of the subroutines and identifying their outermost loops. Each outer loop was given a unique identifier, or *segment* number. Segments were identified without using interprocedural information. This meant that in some cases, a loop was marked as outermost even though it was contained in some other loop in a calling procedure. For example, consider the loops appearing in Figure 1. For the two functions in the example program, we would flag all of the loops except the "`do k`" loop in `main.f` as segments (outermost loops), when in fact only the "`do t`" loop is truly an outermost loop. This is consistent with our intent to consider units of the program that can be analyzed by conventional techniques, since existing reuse analysis techniques provide little or no support for loops containing subroutine calls.

3.2 Cache hit partitioning

We can imagine the flow of time within the execution of the program as being measured in terms of *intervals*. A new interval begins each time the program enters or leaves a segment.

Time is kept using an interval clock, which is simply a counter that is incremented whenever a segment boundary is crossed.

Figure 2 depicts a period of program execution. Within the diagram, the elongated boxes represent data items that are brought into cache at various points. Within a given box, the dark grey circle depicts the initial miss, and the triangles and squares represent cache hits, or reuse of data items.

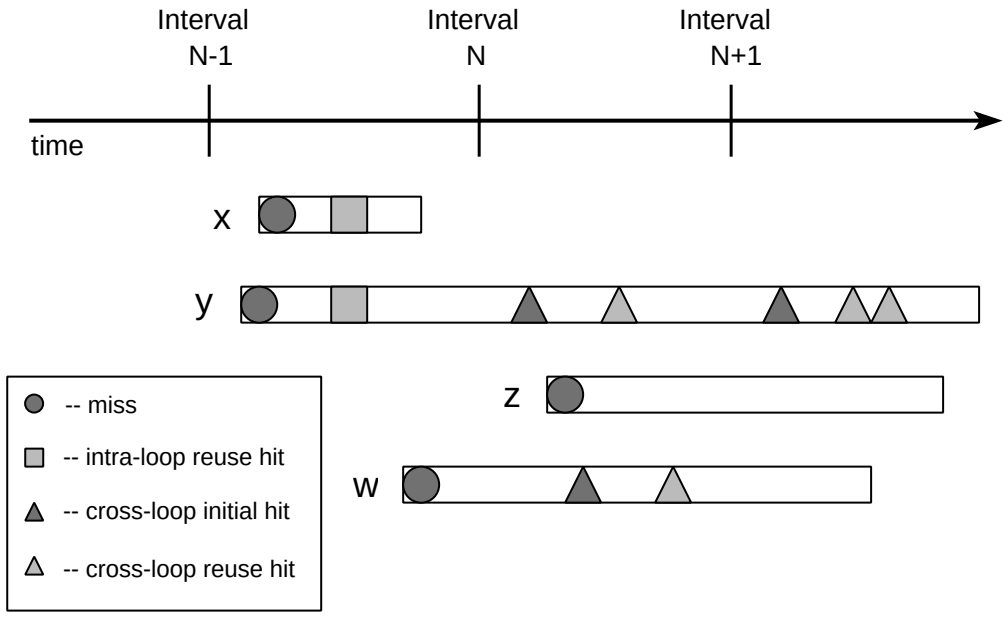

Figure 2: Simulator execution timeline

By keeping track of interval information, we can classify cache hits according to their relation to current and previous intervals. The first type of cache hit we call an "intra-loop reuse hit"; it is designated by a light grey square in the figure. This type of hit takes place when there is an access to a line that was brought into the cache during the same interval. We call the the second type of hit a "cross-loop initial hit". This corresponds to the *first* hit within the current interval on a line that was brought into the cache during a previous interval. This is represented in the figure by a dark grey triangle. The final type of cache hit is called a "cross-loop reuse hit". This type corresponds to a hit on a line that was brought into cache during a previous interval, but has in the intervening time been accessed within the current interval. Cross-loop reuse hits are shown as light triangles within the figure.

From the perspective of static or compiler-based reuse analysis, consider a hypothetical compiler that only looks at a given loop nest (segment) in isolation, but has near-perfect knowledge about the behavior of the loop nest. This compiler should be able to detect all of the intra-loop reuse hits (squares), and all of the cross-loop reuse hits (light triangles), but

it would be unable to detect any of the cross-loop initial hits, and would have to assume that they would result in cache misses.

In the case of software prefetching, if a compiler could detect the set of references that result in cross-loop initial hits, it would be able to reduce the overhead for prefetching, since prefetches for these references could be eliminated.

4 Experimental framework

4.1 Programs

For our experiments, we used the entire collection of programs from the NAS benchmark suite [BBLS91a, BBLS91b], plus a single program (OCEAN) from the PERFECT benchmark suite [CKPK90]. In the case of the NAS benchmarks, the "sample" versions of the programs were used in order to reduce simulation times and memory requirements.[2]

Program	Lines	Funcs	Data (KB)	Runtime (secs)
BT	3274	19	1261	85
LU	2198	17	1591	110
SP	2259	26	281	57
IS	199	6	794	5
CG	418	13	2892	31
EP	141	4	1048	573
FT	470	11	14683	100
MG	497	15	1058	5
ocean	1956	39	796	625

Figure 3: Important program characteristics

Figure 3 gives some of the vital statistics for these programs. "Lines" is the number of non-comment, non-blank source lines in the program. "Funcs" is the number of functions or subroutines. "Data" is the total size of the arrays declared in the program, in kilobytes. "Runtime" is the approximate number of seconds the program takes to execute on an unloaded Sparc 2 workstation with 32 megabytes of memory.

[2]The NAS package provides two versions of each benchmark program: a standard version and a reduced or "sample" version. The reduced versions operate on smaller data sets, making it more feasible to run the programs on inexpensive workstations, as opposed to high-end workstations or supercomputers.

4.2 Compilation

The benchmark programs were compiled using Sun Fortran V1.4, SC1.0.1, running under SunOS Release 4.1.3. The flags passed to the compiler were: "-dalign -O3" (the `-dalign` flag allows the compiler to use double-word loads and stores for double-precision floating point data).

As with all studies of this nature, much depends on the optimization techniques employed by the compiler used to compile the programs being simulated. Certain compiler transformations have the potential to change the way data items are reused. One such transformation is *loop fusion* [Wol89, AC72]. Fusing two adjacent loops may bring successive uses of a given location closer together, increasing the likelihood that the reused location will be found in cache, but also potentially converting cross-loop reuse into intra-loop reuse. The compiler that we used for our experiments did not apply loop fusion, thus we were unable to gauge its effects on cross-loop reuse. Previous studies, however, have shown that fusion is less widely applicable than most other locality-enhancing transformations. The available data indicates that fusion would not substantially alter our results [CMT94].

4.3 Simulation framework

Our cache simulator was built using the SPARC Performance Analysis Toolkit; it is layered on top of the tool `shade` [CK93]. `Shade` provides an extensible mechanism for writing execution-driven simulators; it operates by interpreting a SPARC executable and passing a trace of the instructions to a user-written trace analyzer. In our case, the trace analyzer simulates a particular cache configuration.

The simulator deals with *data* cache behavior only; it does not simulate an instruction cache. Simulated cache characteristics were as follows. The cache line size was 32 bytes. We used a 4-way set-associative configuration, with an LRU replacement policy within each set. The cache was write-back, with an allocate-on-write-miss policy. See Section 5.2.1 for a discussion of the effect of line size on our results.

An important concern for our study is that the degree to which reuse can be exploited for a particular program/machine combination depends on the size of the machine's cache. We address this problem by providing simulation data for a wide range of cache sizes (see Section 5.1 for details).

4.4 Program instrumentation

To instrument our programs, we first identified the outer loops within each subroutine and assigned them unique *segment* numbers (as described in Section 3). Calls to runtime

routines were added to the program at the beginning and end of each outer loop; these calls allowed the simulator to detect segment transitions and to gather cache statistics for each segment (in addition to statistics on the entire program).

We then added calls to runtime routines that demarcated the regions of the program's address space containing array data (as opposed to scalar data). As will be seen in Section 5, it was helpful to distinguish between array and non-array accesses, since in our study the two types of accesses tended to have very different cache behavior.

4.5 Simulator details

In addition to performing the expected functions of a cache simulator, our simulator maintains some additional state in order to gather cross-loop reuse data.

First, the simulator keeps a "current interval" clock, incremented each time the program passes a segment boundary at runtime. Second the simulator keeps track of the regions within the program's address space that correspond to array data; this allows it to keep separate hit/miss statistics for array cache accesses and non-array cache accesses. Third, the simulator stores two additional values for each cache line: "`interval-in`", an integer that records the value of the interval counter when the line was first brought into cache (this is only set on cache misses), and "`interval-use`", an integer that records the value of the interval counter when the line was last accessed (this is set on every access to the line). These tags allow the simulator to divide cache hits into the three categories described in Section 3.2. Intra-loop reuse hits take place when the `interval-in` value of the accessed line is equal to the current interval number. A cross-loop initial hit is characterized by a hit on a line whose `interval-use` tag is less than the current interval number. Cross-loop reuse hits take place when there is a hit on a line whose `interval-in` is less than the current interval number, but whose interval-use tag is equal to the current interval number.

5 Results

5.1 General cache statistics

Tables 4, 5, and 6 provide general cache statistics for the benchmark programs. Each program was run using eight different cache sizes, ranging from 8 kilobytes to 1 megabyte. Each step from left to right in the tables represents a doubling of the cache size.

The overall cache miss rates for each program are shown in Table 4. Table 5 shows the miss rates for scalar and array data, respectively. Finally, Table 6 shows the percentage of all misses that are to scalar data.

The average miss rates for the benchmarks are quite comparable to those reported for

	Cache size							
Program	8K	16K	32K	64K	128K	256K	512K	1M
BT	3.78	2.96	2.86	2.83	2.73	2.64	2.20	0.27
LU	1.96	1.92	1.90	1.80	1.63	1.53	1.28	0.23
SP	4.90	4.22	4.16	3.73	1.31	0.07	0.00	0.00
IS	1.46	1.15	1.07	1.01	0.98	0.88	0.16	0.09
CG	21.48	13.48	9.25	8.83	8.81	8.77	8.71	3.20
EP	0.62	0.61	0.61	0.61	0.61	0.61	0.61	0.01
FT	7.05	6.56	6.55	6.54	6.54	6.54	6.54	6.54
MG	2.09	1.97	1.92	1.29	1.21	1.12	0.92	0.08
ocean	6.28	5.60	5.29	2.33	0.88	0.34	0.08	0.00
average	5.51	4.27	3.73	3.22	2.74	2.50	2.28	1.16

Figure 4: Overall miss rates (in percentages)

	Cache size							
Program	8K	16K	32K	64K	128K	256K	512K	1M
BT	$\frac{7.00}{0.42}$	$\frac{5.73}{0.08}$	$\frac{5.57}{0.03}$	$\frac{5.52}{0.02}$	$\frac{5.33}{0.01}$	$\frac{5.16}{0.01}$	$\frac{4.30}{0.01}$	$\frac{0.52}{0.00}$
LU	$\frac{7.63}{0.02}$	$\frac{7.51}{0.01}$	$\frac{7.45}{0.01}$	$\frac{7.06}{0.01}$	$\frac{6.36}{0.01}$	$\frac{5.99}{0.01}$	$\frac{5.00}{0.01}$	$\frac{0.90}{0.00}$
SP	$\frac{11.81}{0.25}$	$\frac{10.38}{0.08}$	$\frac{10.28}{0.05}$	$\frac{9.21}{0.05}$	$\frac{3.21}{0.03}$	$\frac{0.16}{0.01}$	$\frac{0.01}{0.00}$	$\frac{0.01}{0.00}$
IS	$\frac{7.60}{0.01}$	$\frac{6.00}{0.01}$	$\frac{5.57}{0.01}$	$\frac{5.27}{0.01}$	$\frac{5.10}{0.01}$	$\frac{4.58}{0.01}$	$\frac{0.80}{0.01}$	$\frac{0.43}{0.01}$
CG	$\frac{21.91}{3.29}$	$\frac{13.75}{1.99}$	$\frac{9.45}{0.87}$	$\frac{9.03}{0.36}$	$\frac{9.01}{0.33}$	$\frac{8.97}{0.33}$	$\frac{8.91}{0.33}$	$\frac{3.27}{0.22}$
EP	$\frac{24.99}{0.02}$	$\frac{24.96}{0.00}$	$\frac{24.95}{0.00}$	$\frac{24.95}{0.00}$	$\frac{24.95}{0.00}$	$\frac{24.95}{0.00}$	$\frac{24.95}{0.00}$	$\frac{0.36}{0.00}$
FT	$\frac{10.36}{0.55}$	$\frac{9.87}{0.07}$	$\frac{9.87}{0.04}$	$\frac{9.87}{0.02}$	$\frac{9.87}{0.01}$	$\frac{9.87}{0.01}$	$\frac{9.87}{0.00}$	$\frac{9.87}{0.00}$
MG	$\frac{5.67}{0.04}$	$\frac{5.39}{0.01}$	$\frac{5.25}{0.01}$	$\frac{3.53}{0.01}$	$\frac{3.32}{0.01}$	$\frac{3.06}{0.01}$	$\frac{2.51}{0.01}$	$\frac{0.22}{0.01}$
ocean	$\frac{8.42}{0.23}$	$\frac{7.53}{0.14}$	$\frac{7.12}{0.14}$	$\frac{3.11}{0.12}$	$\frac{1.17}{0.04}$	$\frac{0.45}{0.01}$	$\frac{0.10}{0.00}$	$\frac{0.00}{0.00}$

Figure 5: Miss rates in percentages for array [top] and scalar [bottom] data

	Cache size							
Program	8K	16K	32K	64K	128K	256K	512K	1M
BT	5.39	1.39	0.49	0.37	0.23	0.22	0.25	0.71
LU	0.81	0.40	0.37	0.37	0.37	0.31	0.36	1.41
SP	3.11	1.13	0.73	0.75	1.22	6.06	17.93	17.92
IS	0.63	0.71	0.73	0.75	0.77	0.86	4.47	6.02
CG	0.36	0.34	0.22	0.09	0.09	0.09	0.09	0.16
EP	2.65	0.26	0.18	0.18	0.18	0.18	0.18	10.92
FT	2.62	0.36	0.19	0.11	0.06	0.04	0.02	0.02
MG	1.28	0.46	0.42	0.43	0.41	0.42	0.46	4.66
ocean	0.94	0.68	0.67	1.34	1.25	0.82	1.04	11.91

Figure 6: Percentage of all misses that are to scalars

the floating point programs in the SPEC benchmarks, when run using the same cache size and configuration [GHPS91]. The miss rates for the NAS programs appear to be somewhat lower for the smaller cache sizes (8K – 32K), but the disparity is less evident for cache sizes above 32K.

Miss rates and miss percentages for scalars are very low[3], as can be seen in Tables 5 and 6. This combination of low miss percentage and low miss rate suggests that there is very little payoff to be gained by the application of cache optimizations to scalar accesses in these programs.

For the remainder of this section, we concentrate on the cache statistics for array data only.

5.2 Cross-loop reuse statistics

Figures 7, 8, and 9 provide a graphical breakdown of the cache accesses involving array data. As with the overall statistics above, each program is run using a series of cache sizes, ranging from 8 Kbytes to 1 Mbyte. The four components of each bar represent cache misses, intra-loop reuse hits, cross-loop initial hits, and cross-loop reuse hits as a percentage of all cache accesses during the program's execution.

Not surprisingly, there is a wide range of behavior with regard to the level of cross-loop reuse present. For programs such as `BT`, nearly all of the cache accesses result in misses

[3]Note that for a few of the programs, the percentage of misses that are to array data tapers off with the larger cache sizes (example: for `SP` at the 1 Mbyte cache size, only 83% of all misses are to array data). This is due to the fact that at this cache size, the entire data set for the program (281K) fits easily into cache.

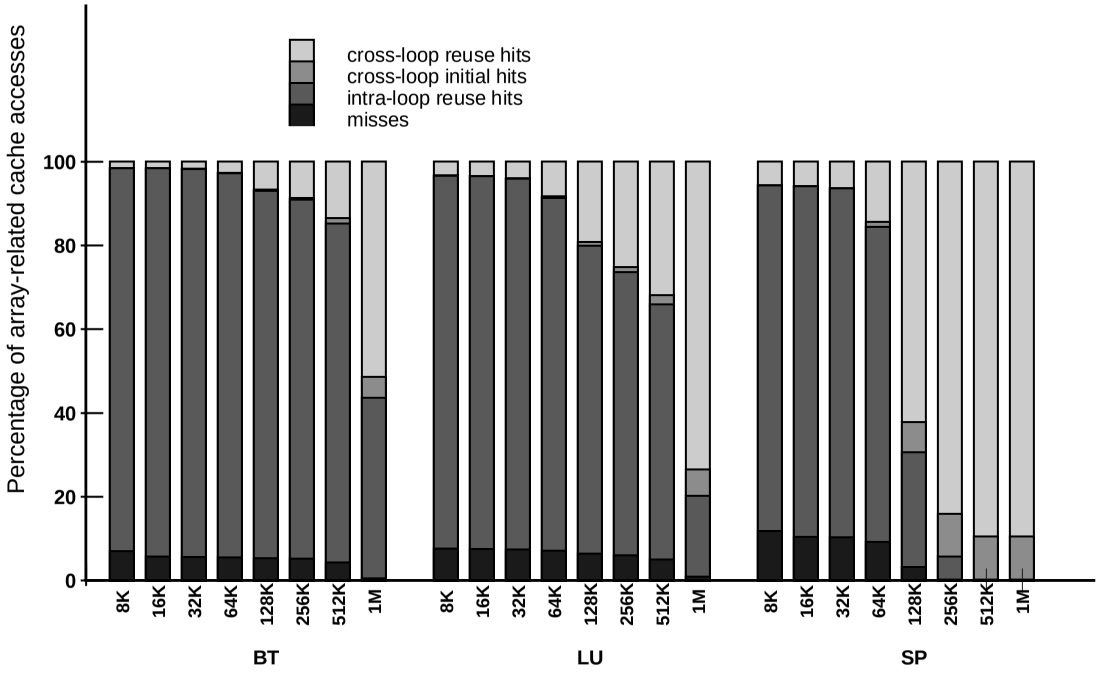

Figure 7: Cross-loop reuse data, part 1

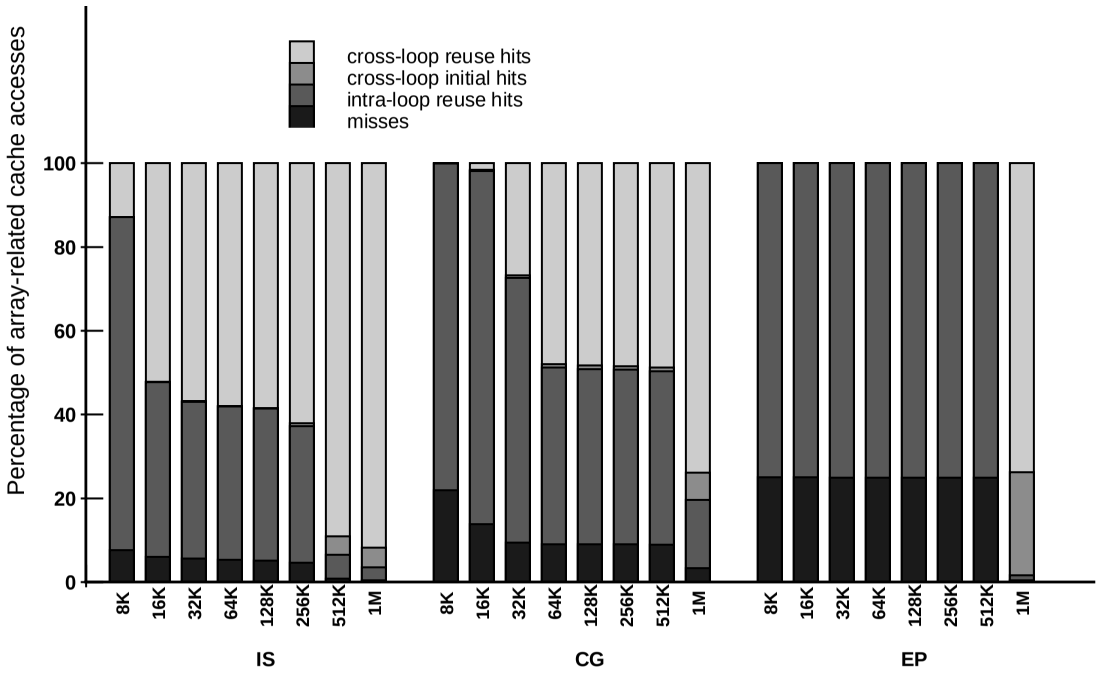

Figure 8: Cross-loop reuse data, part 2

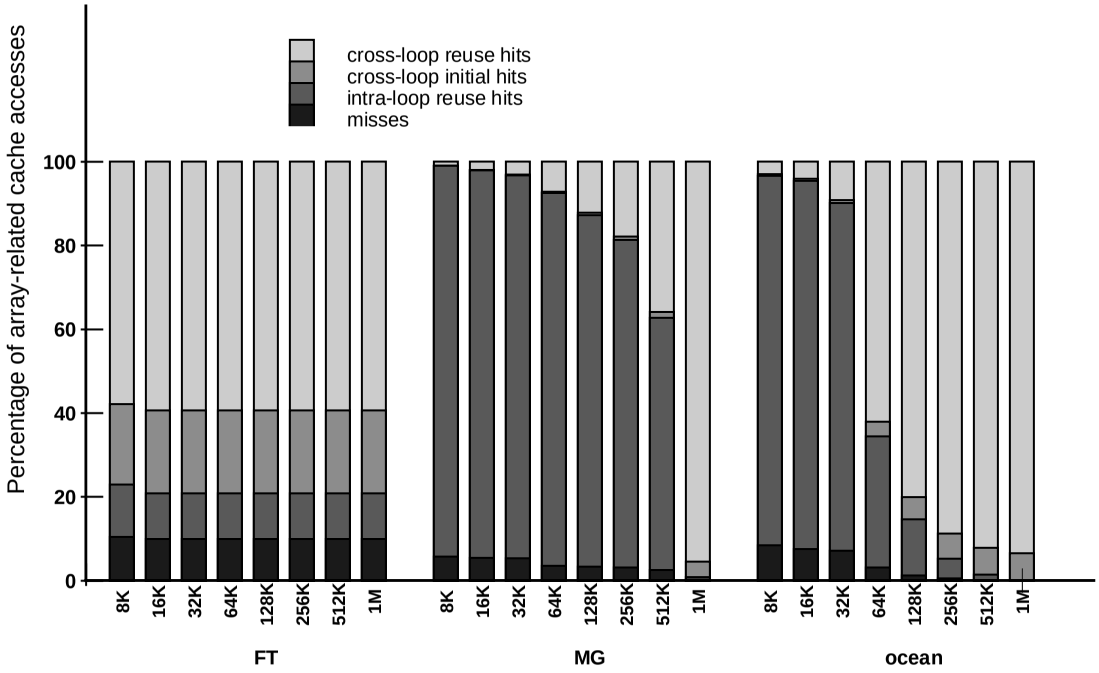

Figure 9: Cross-loop reuse data, part 3

or intra-loop reuse hits. This pattern continues until the cache size exceeds the size of the program's data set. For other programs, cross-loop initial hits and cross-loop reuse hits account for a significant percentage of the total cache accesses at smaller cache sizes. For example, the program `ocean` has a fair amount of cross-loop reuse at the 64K cache level: more than 50% of the cache hits are to data items that were brought into cache by previous loops. Note that the overall miss rate at this cache size is still significant (i.e. the program's data set does not completely fit into cache). The program `FT` represents an extreme case in terms of cross-loop reuse: it shows high degrees of cross-loop reuse even at the smallest cache sizes.

5.2.1 Effects of line size

Although the data is not presented here, we performed the same simulations using cache line sizes of 8 bytes and 128 bytes. We found that cache line size had little apparent effect on the cross-loop reuse results. In general, changing the line size produced no change in the ratio of intra-loop accesses (misses plus intra-loop reuse hits) to cross-loop accesses (cross-loop initial hits plus cross-loop reuse hits). Since the programs had good spatial locality, however, changing the line size did change the proportion of misses vs. intra-loop reuse hits and the corresponding proportion of cross-loop initial hits vs. cross-loop reuse hits. From this we can conclude that most cross-loop reuse is primarily temporal as opposed to

spatial in nature.

5.3 Implications for compiler developers

If a compiler doesn't attempt any transformations that might benefit from cross-loop reuse information, then there is little incentive to try to detect it. As discussed in Section 2, however, software prefetching is one technique that could benefit. In contrast with other techniques, cross-loop reuse information is used for decreasing compiler-introduced overhead, as opposed to reordering the computation to reduce the number of cache misses.

	Cache size							
Program	8K	16K	32K	64K	128K	256K	512K	1M
BT	0.2	0.3	0.4	1.3	4.7	7.8	23.2	90.7
LU	0.1	0.3	1.1	5.1	12.3	17.4	30.9	87.5
SP	0.3	0.4	0.7	11.0	69.0	98.5	99.9	99.9
IS	1.1	2.1	2.5	2.8	3.2	12.2	84.6	91.6
CG	0.2	1.9	6.3	8.2	8.4	8.7	9.2	66.7
EP	0.0	0.0	0.0	0.0	0.0	0.0	0.0	98.6
FT	65.0	66.7	66.7	66.7	66.7	66.7	66.7	66.7
MG	1.1	2.0	3.3	8.8	14.2	21.0	35.3	94.4
ocean	4.0	5.5	9.7	52.7	82.0	93.0	98.4	100.0
average	7.99	8.80	10.07	17.42	28.95	36.14	49.80	88.46

Figure 10: Percentage of prefetches likely to be useless

The table in Figure 10 illustrates the potential for reducing useless prefetches based on cross-loop reuse information. Recall that in the absence of cross-loop reuse information, a compiler must insert prefetches for any reference that looks as though it will be a cache miss, including those which result in cross-loop initial hits. In this table, we plot the number of cross-loop initial hits as a percentage of all the cache accesses that would be prefetched by conventional prefetching compiler (i.e. cache misses plus cross-loop initial hits). A value of N for a given program run indicates that $N\%$ of the prefetches inserted by a loop-level compiler would in fact be useless (since the references would be cache hits). This should be considered "best case" data, since a real compiler would not always be able to gather perfect cross-loop reuse information.

Figure 10 shows that for cache sizes below 32K, the values tend to be very small for most programs, indicating that a single-loop strategy is quite effective. For cache sizes above 32K, anywhere from 10 to 50 percent of the prefetches generated by a single-loop strategy will be useless. For certain programs, the use of cross-loop reuse information has the potential to reduce useless prefetches by a factor of 2, which is fairly significant.

6 Related work

There is an extensive body of previous research on memory reference behavior and cache performance of various benchmark applications [GHPS91, CP90, SZ88, Smi82]. These studies tend to focus on examining the effects of various architectural features (line size, associativity, etc) on cache performance, however, and do not try to classify or categorize the various sources of reuse.

A number of researchers have developed compiler techniques useful for improving cache behavior [CMT94, AL93, WL91, CCK90]. Most of these techniques apply to individual loop nests, however, and are not designed to exploit cross-loop reuse. Two exceptions are *loop fusion* and *affinity regions*. McKinley et al have proposed using loop fusion to improving locality and cache behavior [KM93]. Affinity regions are a technique that allows a compiler to give locality-improving hints to the loop scheduler for a parallel program running on a shared-memory multiprocessor. Compile-time identification of affinity regions was proposed by Appelbe et al [AL93]. No comprehensive studies have been done on the applicability of affinity regions in practice, however.

7 Conclusions

In this paper, we have presented a new strategy for classifying cache accesses that enables a simulator to discriminate between reuse taking place within a single loop nest and reuse taking place between loop nests. By running instrumented versions of the NAS benchmarks through our simulation framework, we have been able to gather data on the level of cross-loop reuse present in these programs.

Our experiments have show that cross-loop reuse does exist in practice, although the level of cross-loop reuse exhibited by a program is dependent on the data set size, the cache size, and the program itself. Two out of the nine programs that we studied exhibit high levels of cross-loop reuse (more than 50% of all cache hits were to data brought in by previous loop nests) for cache sizes that were well below the size of the program's data sets used.

Finally, we have explored the implications of these results for compiler developers. Cross-loop reuse information promises to be very useful for compilers that employ software prefetching; preliminary results indicate that for some programs, useless prefetches can be reduced by a factor of two if precise cross-loop reuse information is employed.

References

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin,

editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[AL93] B. Appelbe and B. Lakshmanan. Optimizing parallel programs using affinity regions. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II–246–II–249, St. Charles, IL, August 1993.

[BBLS91a] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[BBLS91b] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.

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

[CK93] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12; UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington, 1993.

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

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

[CP90] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[GHPS91] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC benchmark suite. Technical Report TR 1049, University of Wisconsin, September 1991.

[KM93] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), pages 62–73, Boston, MA, October 1992.

[Smi82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 13(3):473–530, September 1982.

[SZ88] K. So and V. Zecca. Cache performance of vector processors. In *Proceedings of the 15th International Symposium on Computer Architecture*, 1988.

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

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.