

# Increasing Perfect Nests in Scientific Programs

Tarek S. Abdelrahman and Robert Sawaya  
Department of Electrical and Computer Engineering  
The University of Toronto  
Toronto, Ontario, Canada M5S 3G4  
tsa@eecg.toronto.edu

## Abstract

Loop optimizations for data locality often require perfect loop nests. In this paper, we report on the extent to which perfect nests are present in 23 applications from 4 standard benchmark suites. Further, we evaluate the effectiveness of 3 transformations for converting imperfect nests into perfect ones: code sinking loop distribution and loop fusion. We find that on average, perfect nests contribute to only 39% of the execution time of the benchmarks. Code sinking causes the largest increase in the relative contribution to execution time of perfect nests (to 56%) followed by loop distribution (to 45%). Loop fusion results in no significant improvement. We also evaluate the overhead of applying the transformations. Based on our results, we suggest a strategy for using the transformations in a compiler.

**Keywords** — Parallelizing compilers, data locality, loop transformations, benchmark characterization.

## 1 Introduction

Scalable shared-memory multiprocessors (SSMMs) have become increasingly viable as platforms for high-performance computing by efficiently supporting coherent shared memory in hardware for large numbers of processors [4]. Examples include the Convex SPP1000 [6], the Stanford FLASH [10], and the University of Toronto NUMAchine [18]. Although the memory in SSMMs is logically shared by all processors, it is physically distributed to provide scalability. As a result, memory accesses are non-uniform; access latency for remote memory is considerably higher than for local memory. SSMMs heavily rely on data caching to reduce the effective memory access latency. Consequently, enhancing locality, in addition to exploiting parallelism is becoming a crucial responsibility of parallelizing compilers.

Many loop transformations are used by parallelizing compilers to improve data locality (see [2] for a survey). Examples include loop permutation [15], tiling [12], access normalization [13] and unimodular transformations [20]. However, the majority of these transformations target *perfect* loop nests; i.e. nests that consist of loops with no intervening code between their headers and no intervening code between their tails. Consequently, the applicability and effectiveness of such optimizations are limited by the relative contribution to execution time of perfect nests in applications.

However, there are virtually no studies that indicate how common perfect loop nests are in programs. Moreover, although researchers generally suggest certain loop transformations, such as code sinking and loop distribution, to increase perfect nests, there exists no study of the effectiveness of these transformations in real code. In this paper, we study 23 applications from 4 benchmark suites to determine the extent to which perfect nests are common and the extent to which they contribute to execution time. Moreover, we evaluate three transformations that can be used to convert imperfect nests into perfect ones. These transformations are: code sinking, loop distribution and loop fusion.

We extend a research compiler to instrument loop nests in the benchmarks and obtain execution time profiles of all nests. We also implement the 3 transformations in the compiler and apply them to the benchmarks. The profile information along with transformed code are then used to determine how common perfect nests are and to determine the effectiveness of each of the 3 transformations. We find that on the average, perfect nests contribute to only 39% of execution time. Code sinking causes the largest increase in the relative contribution to execution time of perfect nests (to 56%) followed by loop distribution (to 45%). Loop fusion is found to result in no significant improvement.

## 2 Loops and Loop Nests

A DO-loop (referred to simply as loop) is defined as a structured program construct consisting of a loop header statement, a sequence of inner statements called the loop body, and a loop tail statement that marks the end of the loop:

```
do  $i = i_b, i_f, s$            $\Leftarrow$  loop header  
   $\langle statements \rangle (i)$        $\Leftarrow$  loop body  
end do                       $\Leftarrow$  loop tail
```

where  $i$  is the loop index variable, and  $i_b, i_f, s$  are integer-valued expressions, called lower bound, upper bound and step, respectively. On entry to the loop, the index variable takes the value of  $i_b$ . With each iteration of the loop, the index variable is incremented by  $s$ . The loop is exited when the index variable is outside the  $i_b$  to  $i_f$  range. A *loop nest*  $N = (L_1, L_2, \dots, L_n)$ , is a sequence of loops  $L_1, L_2, \dots, L_n$ , where  $n \geq 1$ , and where every loop  $L_i$ , ( $1 \leq i < n$ ), directly encloses loop  $L_{i+1}$ . Loop  $L_n$  may or may not enclose another loop. Loop  $L_1$  is called the *outermost* loop of  $N$ , while loop  $L_n$  is called

```

do i = ib, if, 1
  do j = jb, max(jb,jf), 1
    if (j == jb) then
      stmt1
    endif
    if (jb ≤ jf) then
      stmt2
    endif
  do j = jb, jf, 1
    if (j == max(jb,jf)) then
      stmt3
    endif
  enddo
enddo
enddo

```

(a)

```

do i = ib, if, 1
  do j = jb, jf, 1
    if (j == jb) then
      stmt1
    endif
    if (j == jf) then
      stmt3
    endif
  enddo
enddo

```

(b)

```

do i = ib, if, 1
  do j = jb, jf, 1
    if (j == jb) then
      stmt1
    endif
    stmt2
    if (j == jf) then
      stmt3
    endif
  enddo
enddo

```

(c)

Figure 1: An example of code sinking.

the *innermost* loop of  $N$ . The *dimensionality* of the nest is the number of loops in the nest, i.e.,  $n$ . A loop nest is said to be *perfect* if there is no intervening code between the headers of the loops, there is no intervening code between the tails of the loops, and the innermost loop does not enclose any loops.

### 3 The Transformations

#### 3.1 Code Sinking

Code sinking [21] moves a block of statements inside the body of a loop, as shown in Figure 1. The nest in Figure 1(a) is imperfect because of statements *stmt1* and *stmt3*. Code sinking pushes these two statements inside the inner loop, as shown in Figure 1(b). Guards around these statements are added to ensure that they are executed the correct number of times. Since it may not be known whether the inner loop is a zero-trip loop, its bounds must be changed to have a trip count of at least 1. Moreover, the original statements of the inner loops must be guarded to execute only if the original inner loop executes at least once. However, if it is known that  $j_b \leq j_f$  then the bounds of the inner loop need not be changed and the guard around the inner loop statements is not required, as shown in Figure 1(c).

Code sinking is only legal if there are no dependences between the blocks of statement that move inside a loop and the header of this loop. Hence, in Figure 1(a), code sinking is legal if no dependences exist between *stmt1* and the header of the inner loop.

Computational overhead due to code sinking stems from the execution of the guards. Although the sunk code will be executed only once for the trip count of

```

L1 do i = 1, 10, 1
S1   a(i) = b(i) + c(i)
L2   do j = 1, 10, 1
S2     d(i,j) = b(j) + c(j)
E2   enddo
E1 enddo

```

(a)  $N_1$ 

```

L'1 do i = 1, 10, 1
S'1   a(i) = b(i) + c(i)
E'1 enddo
L''1 do i = 1, 10, 1
L''2   do j = 1, 10, 1
S''2     d(i,j) = b(j) + c(j)
E''2   enddo
E''1 enddo

```

(b)  $N'_1$  and  $N''_1$ 

Figure 2: An example of loop distribution.

the inner loop, the *ifstmt* used to guard the sunk code will be executed trip count times, which increases execution time. Moreover, the *ifstmt* conditional may cause pipeline hazards or cache misses.

#### 3.2 Loop Distribution

Loop distribution [21] splits a loop nest into two or more nest. It can be used to transform an imperfect nest into a perfect one, as shown in Figure 2. The original imperfect nest  $N_1 = (L_1, L_2)$ , is split into two distinct nests,  $N'_1 = (L'_1)$  and  $N''_1 = (L''_1, L''_2)$ . The statement  $S_2$ , common to both loops  $L_1$  and  $L_2$  in  $N_1$ , becomes the body  $S''_2$  of the perfect nest  $(L''_1, L''_2)$ . The statement  $S_1$ , enclosed by the outer loop  $L_1$  only, constitutes the body  $S'_1$  of the loop  $L'_1$ .

Loop distribution is legal if and only if there exists no dependence cycles between the blocks of statements to be placed in different loops [11]. Dependences carried by outer loops that enclose the loop to be distributed do not affect the legality of distribution because they will always be satisfied by the outer loops [21].

We classify the reasons loop distribution may fail to obtain perfect nests into the following categories:

- **unsafe**: the loop is entered or exited by a *goto* statement, or contains I/O or subroutine calls. The latter case requires inter-procedural analysis which is not available for this research.
- **arraydep**: there exists a dependence cycle that prevents distribution, which is made up of only array dependences and scalar flow dependences.
- **scalardep**: all dependence cycles that prevent distribution are because of scalar dependences.

If a loop is found to be unsafe then the reason for distribution failure is denoted as **unsafe**, and the other reasons are not checked. Analogously, if **scalardep** is denoted as the reason for failure then the loop is safe, there are no dependence cycles made of only array dependences, but there exists at least one scalar dependence cycle.

```

L1 do i = ib, if, 1
L2   do j = jb, jf, 1
S2     st1(j)
E2   enddo
L3   do k = kb, kf, 1
S2     st2(k)
E3   enddo
E1 enddo

```

(a) N<sub>1</sub>

```

L'1 do i = ib, if, 1
L'23 do h = min(jb, kb), max(jf, kf), 1
C'1   if (jb ≤ h ≤ jf)
S'1     st1(h)
E'1   endif
C'2   if (kb ≤ h ≤ kf)
S'2     st2(h)
E'2   endif
E'2 enddo
E'1 enddo

```

(b) N'<sub>1</sub>

Figure 3: Example of loop fusion.

### 3.3 Loop Fusion

Loop fusion [21] can be used to transform an imperfect nest into a perfect one [5], as illustrated in Figure 3. The nest N<sub>1</sub> has two inner loops L<sub>2</sub> and L<sub>3</sub> at the same nesting level. The perfect nest N'<sub>1</sub> is obtained after fusing these two inner loops. The headers of the loops L<sub>2</sub> and L<sub>3</sub> need not be compatible. The guards C'<sub>1</sub> and C'<sub>2</sub> ensure that the bodies of L<sub>2</sub> and L<sub>3</sub> respectively, are executed the correct number of times. Also, the loop indices are unified and the array subexpressions are modified accordingly. The details of the loop fusion algorithm we use in our compiler are described in [17].

Fusing two adjacent loops L<sub>1</sub> and L<sub>2</sub> with bodies B<sub>1</sub> and B<sub>2</sub> respectively, to yield L<sub>12</sub> with a body made of B<sub>1</sub> followed by B<sub>2</sub>, is legal if two conditions are satisfied. First, in L<sub>12</sub> there should be no dependences, unless carried by an outer loop, whose source is B<sub>2</sub> and sink is B<sub>1</sub> [21]. Second, the header of each loop to be fused should not depend on the body of the other loop.

We classify reasons that limit loop fusion in obtaining perfect nests into the following categories:

- **unsafe:** one of the loops to be fused is entered or exited by a *goto* statement, or has I/O or subroutine calls in its body.
- **structure:** the structure of one of the nests disallows fusion. For instance, there may remain an *ifstmt* that encloses one of the loops to be fused.
- **intercode:** some code between the loops to be fused cannot be moved to make the loops adjacent.
- **backdep:** a dependence in the fused loop indicates that fusion is illegal.
- **headerdep:** there is a dependence between the body of one of the loops to be fused and the header of the other.

If one of the candidate loops is found to be unsafe then the reason for fusion failure is denoted as **unsafe**, and the other reasons are not checked. Analogously, if **headerdep** is denoted as the reason for failure, then all candidate loops are safe; their structure allows fusion; there is no intercode that cannot be moved between the loops to be fused; there are no backward dependences in the resulting fused loop; but there exists a dependence between the header of the first loop and the body of the second, or vice-versa.

## 4 Experimental Evaluation

### 4.1 Experimental Setup

We extended the Polaris compiler [9] to generate an instrumented source code and a set of profile information for each benchmark application. The source is instrumented to measure execution time of loop nests and subroutine calls in the program. The profile contains information such as the number of loops and the nesting structures in the application. The instrumented code is then compiled using a native F77 compiler, and the executable is used to generate timing information. This timing information along with the profile information are stored in a database, which allows subsequent queries that derive the various metrics we present in this paper.

### 4.2 Benchmark Applications

In total, 23 benchmark applications are considered. Table 1 lists the applications and their respective benchmark suites. The applications are chosen from 4 suites: Perfect Club [8], NAS [3], SPEC92 [7] and NCSA [16]. The table also shows the abbreviations used in the remainder of the paper to denote the applications; the label AV represents the average for all 23 applications.

### 4.3 Timing of Loop Nests

The *total time* of a loop is the time it takes to execute the loop. Hence, the total time of a perfect nest is the total time of its outer loop. The *net time* of a loop is equal to its total time minus the total time of all the loops and subroutines directly enclosed by it. Hence, the net time of a perfect nest is equal to the net time of its innermost loop.

To time a given loop L, calls to timing routines are inserted before the header and after the tail of the loop. The execution time of L is equal to the difference between the two timing calls. However, the loop L may be executed multiple times<sup>1</sup>. Thus, the execution time of L must be accumulated. Although the timing calls are intrusive, we believe that their effect is small.

#### 4.3.1 Timing of Nests Made Perfect by Code Sinking

Given a nest (L<sub>outer</sub>, L<sub>inner</sub>) that can be made tightly nested using code sinking then the time of the resulting perfect nest (L'<sub>outer</sub>, L'<sub>inner</sub>) is computed as follows. The total time of L'<sub>outer</sub> is considered equal to the total time

<sup>1</sup>L may be inside a subroutine that is called multiple times, or may be enclosed by an outer loop.

Benchmark Suite	Application	Abbreviation
NAS	appbt	BT
	embar	EP
	fftpde	FT
	buk	IS
	applu	LU
	mgrid	MG
	appsp	SP
PerfectClub	qcd	LG
	mdg	LW
	track	MT
	bdna	NA
	ocean	OC
	dyfesm	SD
	arc2d	SR
	flo52q	TF
	trfd	TI
Spec92	hydro2d	HY
	ora	OR
	su2cor	SU
	swm256	SW
	tomcatv	TO
	wave5	WV
NCSA	cmhog	HG

Table 1: Application benchmarks and their respective suites.

of  $L_{outer}$ . That is, the overhead of the guards used in code sinking is ignored. This is done to keep execution time of perfect nests from being artificially inflated. The total time of  $L'_{inner}$  is given by

$$\text{Total Time}(L'_{inner}) = \text{Total Time}(L_{inner}) + \text{Net Time}(L_{outer}),$$

which approximates the addition of the execution time of the sunk code to the total time of  $L_{inner}$ . This is only an approximation because the net time of  $L_{outer}$  includes the execution time for the header and tail of  $L_{outer}$ . We believe that this is a reasonable approximation because usually the body of a loop contributes to most of the execution time of the loop.

#### 4.3.2 Timing of Nests Made Perfect by Loop Distribution

Given a loop  $L_{outer}$  enclosing a loop  $L_{inner}$ , and given that  $L_{outer}$  can be distributed to obtain a single loop  $L_{single}$  and a perfect nest  $(L'_{outer}, L'_{inner})$ , then the times of the resulting loops are computed as follows. The total time of  $L'_{inner}$  is equal to the total time of  $L_{inner}$ . The total time of  $L'_{outer}$  is approximated as the total time of  $L_{inner}$ . The header and tail times of  $L'_{outer}$  are not included; they are expected to be small compared to the total time of  $L_{inner}$ . The total time of  $L_{single}$  is given by:

$$\text{Total Time}(L_{single}) = \text{Total Time}(L_{outer}) - \text{Total Time}(L_{inner}).$$

#### 4.3.3 Timing of Nests Made Perfect by Loop Fusion

Given a loop  $L_{outer}$  enclosing loops  $L_{in1}$  and  $L_{in2}$ , and given that these inner loops can be fused to obtain a loop  $L_{inner}$  which will be part of a perfect nest  $(L_{outer}, L_{inner})$ , then the times of the resulting loops are computed as follows. The total time of  $L_{outer}$  is unchanged. The total time of  $L_{inner}$  is approximated as the addition of the total times of  $L_{in1}$  and  $L_{in2}$ . This is an approximation because it ignores the saving in time due to combining the headers of  $L_{in1}$  and  $L_{in2}$ .

#### 4.4 Characterization of Execution Time

In order to gain better understanding of the potential of each transformation to increase the contribution of perfect nests to execution time, we characterize the execution time of the benchmarks according to the following categories:

- **UnEnclosed:** time for code segments not enclosed by any loop.
- **Single:** net time of single loops.
- **Perfect:** net time of perfect nests.
- **Code-motion:** net time of nests that may be made perfect using code sinking. Such nests can also be made perfect by loop distribution, but not by loop fusion.
- **Fusion:** net time of nests that require loop fusion to become perfect. Such nests can also be made perfect by loop distribution, but not by code sinking.

Figure 4 shows the execution time characterizations of the 23 benchmarks. The average contribution of perfect nests over all applications is only 39%. In the applications SR, HY and SW, perfect nests contribute to almost 100% of the execution time. Therefore, none of the transformations considered in this work are needed for these applications. In MG and TF, perfect nests contribute to a high percentage of execution time. Thus the effects of the transformations is expected to be minimal. The execution time labeled as “UnEnclosed” and “Single” is not targetable by any of the transformations. Therefore the relative contribution to execution time of perfect nests in IS cannot be increased. Code sinking targets the percentage of execution time labeled as “Code Motion”. Therefore, it has the potential to significantly improve the contribution to execution time of perfect nests in FT, LU, SP, OC, TI, OR and TI. Loop distribution targets the percentage of execution time classified under the categories “Code Motion” and “Fusion”. Finally, loop fusion only handles nests whose time is classified as “Fusion”. Thus, loop fusion can potentially increase the relative contribution to execution time of perfect nests in BT, EP, SP, LW, MT, NA, WV and HG. From the above, it can be concluded that while in some applications perfect loop nests are common, many applications can benefit from an increase in the contribution to execution time of perfect nests. Further, each of the 3 transformations has the potential to increase this contribution.

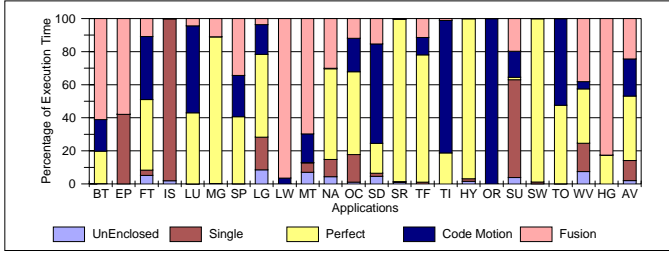


Figure 4: Execution time Profile of the applications.

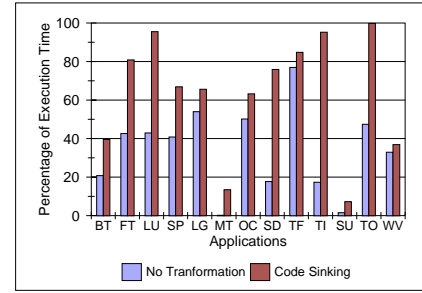
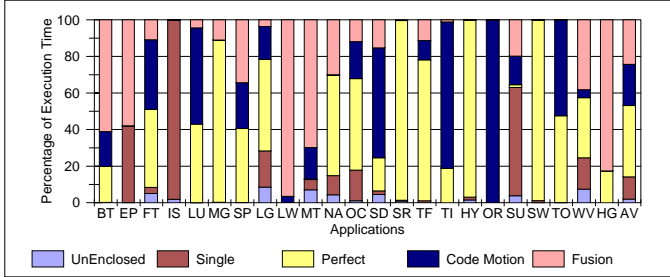


Figure 6: Effects of code sinking on perfect nests.



(a) Original

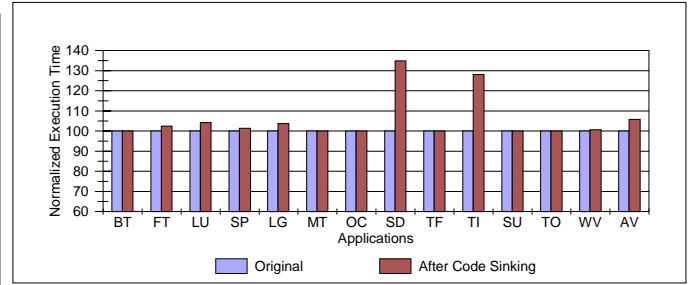
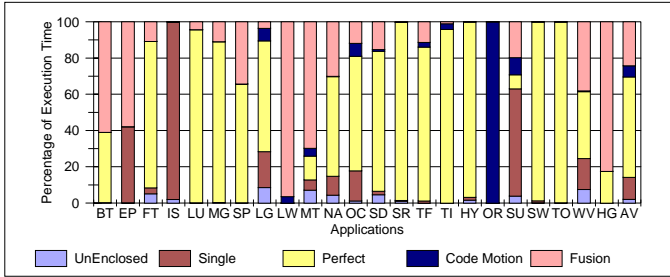


Figure 7: Overhead of code sinking.



(b) After Code Sinking

Figure 5: Execution time profile before and after code sinking.

## 4.5 Evaluation of Code Sinking

Figure 5(a) and Figure 5(b) show the characterization of execution time before and after code sinking, respectively. The figures indicate that code sinking is effective in transforming most nests labeled as “Code Motion” into perfect nests; in Figure 5(b), there remain little execution time labeled as “Code Motion”, except in OR where there is an unsafe loop that contribute to most of the execution time. The applications that have a low percentage of execution time labeled as “Code Motion” are not affected by code sinking. Code sinking increases the contribution of perfect nests from 39% to 56%, averaged over all applications. More specifically, it increases the relative contribution to execution time of perfect nests in 13 benchmarks. Figure 6 shows the increase in the relative contribution to execution time of perfect nests for these applications.

To assess the computational overhead due to code sinking, the execution times of the applications after code sinking, are compared with the execution times of the original applications. Figure 7, shows the overhead of code sinking for the applications affected by this

transformation. Code sinking caused no overhead in 8 of the 13 applications in which it increased the relative contribution to execution time of perfect nests. For 3 other applications, the overhead is less than 5%. Thus, for most applications code sinking introduces acceptable overhead. However, code sinking is prohibitive for SD and TI because of the high overhead of 35% and 28%, respectively. This large overhead is due to sinking code in loops that have small bodies (one or two statements), where guards have an execution time that is comparable to that of the body of the original loop.

## 4.6 Evaluation of Loop Distribution

Figure 8 shows the characterization of execution time after loop distribution (see Figure 5(a) for the characterization of the applications before loop distribution). The figure indicates that in 9 of 12 applications, loop distribution only benefits the execution time labeled “Code Motion”. In these applications, execution time labeled “Fusion” is not affected due to dependences. Only in MG, loop distribution succeeds in transforming all the imperfect nests, whose execution time is characterized as “Fusion”, to perfect nests. In SD and TF, loop distribution benefits loop nests whose times are characterized as either “Code Motion” or “Fusion”. Hence, most of the loops nests that can be made perfect by loop distribution have their execution time labeled as “Code Motion”, and can also be made perfect by code sinking. Loop nests that require loop fusion to become perfect, can rarely be made perfect by loop distribution.

Loop distribution increased the contribution of perfect nests from 39% to 45%, averaged over all applications, which is a modest increase. More specifically, it increased the relative contribution to execution time of perfect nests in 12 benchmark applications. Figure 9 shows the increase in the relative contribution to execution time of perfect nests for these applications. Loop

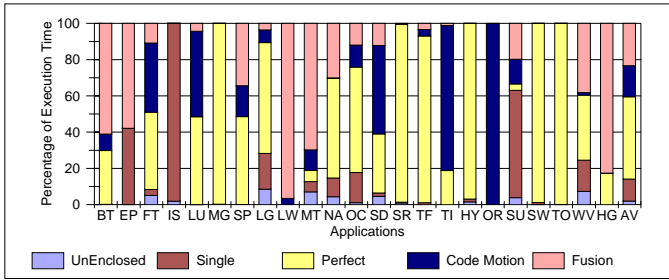


Figure 8: Execution time profile after loop distribution.

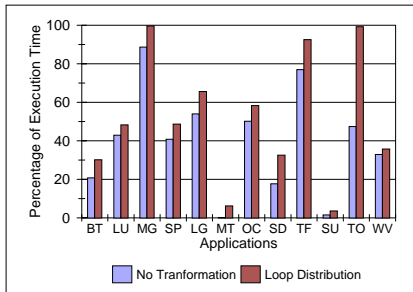


Figure 9: Contribution to execution time of perfect nests before and after loop distribution.

distribution succeeds in increasing the relative contribution to execution time of perfect nests to more than 95%, for 3 applications, namely MG, TF and TO. For 16 other applications, loop distribution is not very effective in transforming imperfect nests into perfect ones. Array dependences are often the main reason for disallowing distribution. However, for some applications (FT, LU, OC, SD and TI), scalar dependences also prevent loop distribution from obtaining perfect nests. Unsafe loops disallow distribution in only few applications.

The benchmark applications are executed after applying loop distribution to assess the overhead introduced. It is found that loop distribution added negligible overhead to the execution time in all of the applications.

#### 4.7 Evaluation of Loop Fusion

Loop fusion increases the relative contribution to execution time of perfect nests in only 2 applications. The relative contribution to execution time of perfect nests increases from 88% to 99% in MG. In SD, the relative contribution to execution time of perfect nests increases from 18% to 23%. Loop fusion caused no overhead in MG, and a 3% overhead in SD.

Loop fusion has the potential to increase perfect nests contribution in the applications that have a significant portion of its execution time labeled as “Fusion” in Figure 4. The main reason for the ineffectiveness of loop fusion is the presence of backward dependences in the fused loop which makes fusion illegal. The presence of INTERCODE (i.e. code between loop nests) that cannot be moved is a factor in OC only.

## 5 Related Work

There is a large body of work on locality enhancement techniques that assume perfect loop nests. Examples

include loop permutation [2], tiling [2, 12, 19], access normalization [13] and unimodular transformations [20].

The use of code sinking, loop distribution and loop fusion to obtain perfect nests has been proposed in the literature by many researchers in the past [1, 2, 5, 14, 20]. However, none conducted an extensive evaluation of the extent to which perfect nests are present in programs, nor of the effectiveness of these transformation in improving the structure of loop nests.

## 6 Conclusions

Many locality optimizations target perfect nests. However, there exists no studies of the extent to which perfect nests exist in programs, nor of the effectiveness of transformations that convert imperfect nests into perfect ones. In this paper, we experimentally evaluated the extent to which perfect loop nests are present in programs. We also evaluated the extent to which three transformations can increase perfect loop nests: code sinking, loop distribution and loop fusion. We profile loop nests in a program and use the profile results along with transformed code to determine the contribution of perfect nests after each of the above transformations is performed. We found that, on average, contribute to only 39% of the execution time of 23 benchmarks application. Therefore, increasing the presence of perfect nests across the applications may make locality enhancing optimizations more applicable, and hence, more beneficial. The average relative contribution to execution time of perfect nests is increased to 56% by code sinking, to 45% by loop distribution. Loop fusion was found to be relatively ineffective.

The sources of overhead introduced by each of the transformations were identified and measured. Loop distribution has negligible overhead. Code sinking caused small overhead (less than 5%) in the majority of applications, but when the size of the loop into which code is small, the overhead is high. Hence, given an imperfect nest that needs to be made perfect, our results suggest that loop distribution should be first attempted because it causes the smallest overhead. If loop distribution fails, then code sinking should be attempted next. If code sinking fails then loop distribution could be applied.

## References

- [1] S. Amarasinghe et al. An overview of a compiler for scalable parallel machines. In *Languages and Compilers for Parallel Computing*, pages 253–272, 1993.
- [2] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
- [3] D. Bailey, E. Barszcz, J. Barton, D. Browning, and R. Carter. The NAS parallel benchmarks. *Int’l J. Supercomputer Applications*, 5(3):63–73, 1991.
- [4] G. Bell. Ultracomputers: A teraflop before its time. *Comm. of the ACM*, 35(8):26–47, August 1992.

- [5] S. Carr, K. McKinley, and C-W. Tseng. Compiler optimizations for improving data locality. In *Proc. of ASPLOS*, pages 252–262, 1994.
- [6] Convex Computer Corporation. *Convex Exemplar system overview*. Richardson, TX, 1994.
- [7] K. Dixit. New CPU benchmarks from SPEC. *Digest of Papers, Spring COMPCON 1992*, pages 305–310, 1992.
- [8] M. Berry et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, CSRD, University of Illinois, 1989.
- [9] W. Blume et al. Polaris: Improving the effectiveness of parallelizing compilers. In *Languages and Compilers for Parallel Computing*, pages 141–154, 1995.
- [10] M. Heinrich et al. The Stanford FLASH multiprocessor. In *Proc. 21th Intl. Symp. on Computer Architecture*, pages 302–313, Chicago, IL., April 1994.
- [11] D. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.
- [12] M. Lam, E. Rotherberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. of ASPLOS*, pages 63–74, 1991.
- [13] W. Li and K. Pingali. Access normalization: loop restructuring for NUMA compilers. *ACM Trans. on Computer Systems*, 11(4):353–375, November 1993.
- [14] K. McKinley, S. Carr, and C-W. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, July 1996.
- [15] K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. of ASPLOS*, pages 94–104, 1996.
- [16] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. <http://zeus.ncsa.uiuc.edu:8080/archives/>.
- [17] R. Sawaya. A study of loop nest structures and locality in scientific programs. Master’s thesis, University of Toronto, Toronto, Ontario, Canada, 1998.
- [18] Z. Vranesic et al. The NUMAchine multiprocessor. Tech. Rep. CSRI-324, Computer Systems Research Institute, University of Toronto, Canada, April 1995.
- [19] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 30–44, 1991.
- [20] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
- [21] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 390 Bridge Parkway Redwood City, CA 94065, 1996.