Array Merging: A Technique for Improving Cache and TLB Behavior[*] - Extended Abstract -

Daniela Genius, INRIA Rocquencourt
Siddhartha Chatterjee, University of North Carolina at Chapel Hill
Alvin R. Lebeck, Duke University

June 5, 2001

1 Introduction

It is common knowledge that the memory bottleneck has become increasingly acute in recent years. For decades now, *memory hierarchies* have been inserted between the processor and main memory. Scientific computing applications access large portions of memory in regular patterns, and are particularly sensitive to memory hierarchy misses. A standard technique to overcome such misses is to change to loop structure. In addition, the data layout can be changed. Translation look-aside buffers (TLBs) have long been underestimated as a major source of performance degradation.

This paper suggests a simple *array merging* scheme to achieve better page utilization in the fully associative TLB, while also preventing cross interference in the cache hierarchy levels. The focus is on scientific FORTRAN codes without pointers or procedure calls. The remainder of this paper is organized as follows. Section 2 contains related work. Section 3 introduces basic terminology. Section 4 presents the method, while Section 5 provides a case study. Section 6 gives detailed experimental results for this example on four different memory hierarchies. The paper concludes by outlining future work. A longer version of the work presented here can be found in a technical report of UNC [9].

2 Related Work

The well-established compiler approach to improving the memory hierarchy behavior is to apply loop transformations, usually focusing on improvement for the the first level data cache [15]. As shown in [13, 8], some techniques like *iteration space tiling*, while improving largely on capacity misses, can lead to an increase in cache conflicts for certain tile sizes. A loop transformation affects all arrays in the loop, sometimes destroying good locality for one array while enhancing it for the other. Furthermore, conflicts are difficult to control at the loop transformation level. Most *data layout* approaches such as matrix transposition and stride reordering [7] use techniques that try to reorder data according to the access structure for better spatial reuse. However, these techniques neglect cache conflicts, and some might even increase them. [1] In the mid-1990s, several researchers have established *padding* [4, 16, 18] as a data layout technique for avoiding conflicts. Originating from memory bank conflict avoidance in the highly interleaved memory systems on vector computers without a cache [5], this technique advocates the insertion of gaps filled with unused data (the *pad*) into data structures in order to prevent interferences. Ghosh et al. used cache miss equations [11] to select both padding and tile size. Characteristic of padding is the possible inflation of data structures: pad sizes that are optimal wrt. cache conflicts can become quite large so that there is an imminent danger of exceeding the capacity of higher memory hierarchy levels.

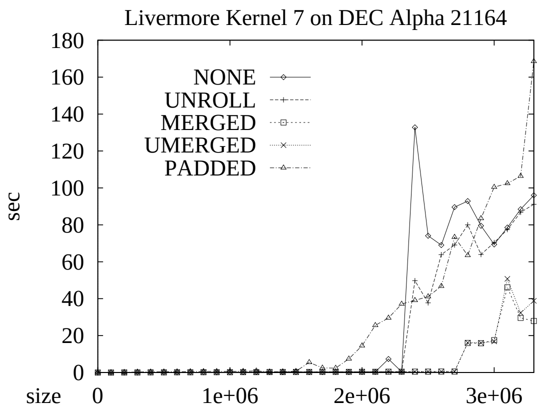
An alternative approach to prevent arrays from interfering with each other is to *merge* them by alternately taking elements of each one; obviously this eliminates any chance of them interfering, but it glues them together

[*] This work was performed when the first author was a postdoctoral researcher with the TUNE project

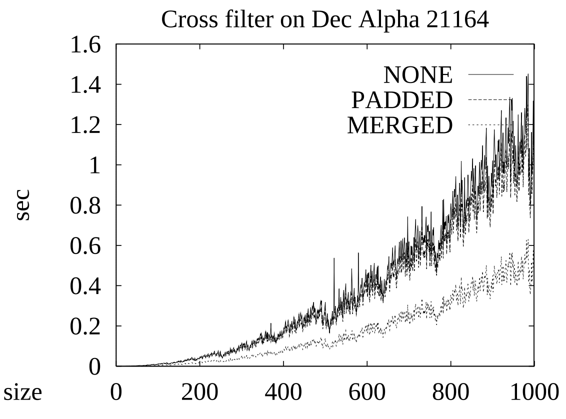
[1] As an example, take the matrix multiplication code $C_{i,j} = C_{i,j} + A_{i,k} * B_{k,j}$. Transposing A for column-wise access forces accesses to A and B into cadence, which for certain array sizes causes a conflict in every iteration.

permanently. Although the technique is acknowledged as a standard technique for hand optimization in the computer architecture literature [12], it has not yet found its way into compilers. By profiling, Lebeck and Wood reveal the opportunities for merging on a significant fraction of the SPEC92 floating point benchmarks in [14]. The authors combine an ad hoc merging with other optimizations for half of the benchmarks with good run time improvements. There have also been suggestions for merging into cache blocks. A data flow analysis based approach appears in the master's thesis of Rawat [17], but handles direct-mapped caches only and treats arrays as a whole. Thus, cache miss rates are significantly overestimated. Calder et al. [6] use a variant of merging for heap allocation. In [10] merging is applied to simple pointered data structures such as linked lists and hashtables, with a beneficial effect on data cache behavior.

During previous experimentation, we met with extremely encouraging results for merging with some smaller codes. Livermore kernel 7 (Figure 1.a) is a fragment of a hydrology kernel, consisting of a single loop sequentially



(a)



(b)

Figure 1: Run times for Livermore kernel 7 and the cross filter on Alpha 21164

accessing four arrays[2]. If only inter-array padding can be applied, small to medium array sizes merging and padding yield approximately the same run times. Yet, an interesting effect can be observed for large arrays: as the TLB is fully associative and the accesses traverse the array sequentially, every new page has already been evicted — we call this TLB *thrashing*. The sharp increase in run time usually attributed to TLB thrashing is postponed for *MERGED*, while for the padded code, the overall size of the portion of memory required is inflated, causing worse page utilization and an earlier onset of thrashing. For the cross filter, padding makes no difference compared to the unoptimized layout — the two curves are approximately the same and can hardly be discerned in the graph. On the other hand, merging improves run time by up to a factor of two (Figure 1.b).

The remainder of the paper will show that while not inflating the array size, merging is cheap at compile time, and suitable for inter-loop optimization.

3 Basic Notions

Data cache hierarchies in Harvard architectures have separate first level data and instruction caches. In a n-level memory hierarchy [12], we have the parameters A, B, C of **A**ssociativity, **B**lock size and **C**apacity on each level. The first level data cache is denoted as the L1 cache with parameters A_1, B_1, C_1, and so forth. In scientific codes, common data types on 32 bit architectures are `float, int` of size 4 bytes, and `double, long` of size 8 bytes[3], determining B_0. Block sizes B in the cache vary between 16 and 128 bytes.

Associativity A gives the number of choices given when a block is mapped to a level of the memory hierarchy For *direct mappings* ($A = 1$), the address m completely determines the cache location $m \bmod \frac{C}{B}$. *Full*

[2]As this code profits from unrolling, also the unrolled variants (UNROLL, UMERGE) are shown.

[3]On 64 bit architectures, data types can be larger

associativity ($A = \frac{C}{B}$) on the other hand is expensive to implement, and associativities $A > 8$ generally give only marginal improvement inperformance [12]. The set is selected by $m \bmod \frac{C}{B*A}$. Within a set, replacement algorithms like LRU and FIFO apply.

Let us assume the *inclusion property* holds, i.e. a block present in level i is also present in level 1 to $i-1$. Whenever a block requested by the program is not present in level i, a *cache miss* occurs. In this case, the block containing the requested location is loaded from level $i+1$. For higher levels, the time required to retrieve the block, the *memory latency*, grows by orders of magnitude.

Data memory addresses are translated with the help of the (data) *Translation Look-aside Buffer* (TLB), which works like a small cache with very large block size. B_{TLB} is the size of a portion of memory that is addressed through one TLB entry (here 8 or 16 KB). All accesses go to the TLB. As a consequence, while there are fewer misses on higher cache levels due to hits in the lower levels, this does not extend to the TLB. TLB misses are known to occur rarely (lower than 1%), but at a high penalty (20 cycles)[12].

A *data access* is a dynamic instance of a reference. A single reference can either read from or write to memory. A FORTRAN assignment can cause several references (e.g. `A(i) = B(i+1)` causes two references: a load to the address of array variable `B(i+1)` and a store to `A(i)`. The classical model [22] considers loop nests where *access vectors* are affine mappings of the loop index vector $\vec{i} = (i_1, \ldots, i_s)^T$. Let $\vec{d}$ be a vector of constants, and let n be the array dimensionality. The access matrix J relates n array dimensions and s loop indexes:

$$\begin{bmatrix} j_{1,1} & \cdots & j_{1,s} \\ \vdots & & \vdots \\ j_{n,1} & \cdots & j_{n,s} \end{bmatrix} \begin{bmatrix} i_1 \\ \vdots \\ i_s \end{bmatrix} + \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix}.$$

References are called *uniformly generated* if their accesses differ only in $\vec{d}$. Arrays containing only such references are *uniformly referenced.*

Usually, one distinguishes between *temporal reuse* and *spatial reuse.* Temporal reuse refers to the same data item, while spatial reuse refers to any item in the same cache line. If there are r references in a program, temporal reuse can ideally improve by a factor of r for the same element. Spatial reuse can only improve up to $\lfloor \frac{B}{B_0} \rfloor$, when all elements of the line are reused. Let us define the *reuse distance* as the number of loop iterations between two temporal reuses. If there is no reuse for an element, the reuse distance is 0.

Even with locality, in caches where $A < \frac{C}{B}$, *conflicts* can occur, often destroying reuse by evicting a cache block still in use. *Self interference* denotes conflicts between blocks that stem from the same array, and *cross interference* denotes conflicts between cache blocks from different arrays. By *thrashing* we denote the situation in which a block is evicted on every iteration; typically, this occurs with cross interference in uniformly referenced arrays.

3.1 Stencil codes

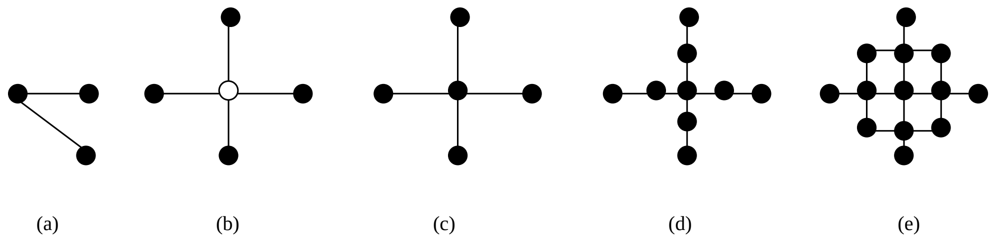


Figure 2: Stencils (a) 3-point (b) 4-point (c) 5-point (d) 9-point (e) 13-point

Stencil codes exhibit uniformly generated references, opening opportunities for reuse. Typical examples are filter algorithms, most Livermore kernels, and a large part of every collection of SPEC floating point benchmarks. In Figure 2, (a) occurs e.g. in multi-grid algorithms, (b) is typical for image filters. Example (c) occurs e.g. in Successive Overrelaxation, where the array position itself is overwritten; (d) and (e) are typical templates in various image filters and many of the Livermore kernels. Sellappa and Chatterjee shows in [19] that impressive results can be obtained with merging for multigrid algorithms. Uniformity wrt. the outer loop complicates things. In Figure 2, the reuse distance is N for (a)-(c), N resp. $2N$ for (d) and (e). Obviously, the notion of uniformity needs some finer-grained subdivision.

3.2 Data Layout

The *data layout* for an n-dimensional array a can follow any (injective) function $L : \mathbb{N}^n \rightarrow \mathbb{N}$. Compilers usually allocate contiguous memory space from an arbitrary starting address, with some alignment on a cache-line or Virtual Memory page boundary. FORTRAN compilers uses column major layout, i.e. for two-dimensional arrays array element $A_{k,l}$ is mapped to virtual memory address $o + k + N * l$, where o is the starting address offset. The basic idea of a compiler-directed data placement is to place data at suitable memory locations at compile time such that at run time, it is loaded into the desired cache blocks.

4 Cache-conscious Data Merging

4.1 Uniformity and Togetherness

Using classical compiler techniques, we analyze for *uniformly* generated references. In contrast to the classical notion, we consider also references to *different* array to be uniform if they satisfy the condition from section 3. Moreover, two array accesses are considered. Let us assume that simple data layout transformation have already taken place. Arrays are laid out in memory so that the dimension n of the array that is accessed in the innermost loop by index i_n is laid out contiguously. In particular, arrays that are accessed rowwise are laid out rowwise. Then, two accesses are

1. *innermost uniform*, denoted by u_{k_n}, where i_n is the index of the innermost loop.
2. *uniform wrt. dimension* m, denoted by u_{k_m} for some i_m with $m \leq n$.

The second is the more general case; it includes innermost uniformity. Arrays $A_1, \ldots, A_l$ which satisfy condition u are in relation $u(A_1, \ldots, A_l)$. Now we are prepared to define the general notion of *togetherness* for arrays.

Definition: Arrays A_1, A_2 are accessed *together* in a loop nest with respect to the uniformity condition u, if they satisfy the uniformity condition, i.e. $A_1, A_2 \in u(A_1, \ldots, A_l)$. Again, we denote this by $t_u(A_1, A_2)$.

While togetherness wrt. the innermost dimension is obviously profitable if the traversal of the innermost loop coincides with the array layout in memory and stride $\leq B$, the use of the other kinds of uniformity is not so obvious. To transform these into k_n-uniformity, classical techniques such as *loop fusion* are applied [3]. However, innermost uniformity can lead to cache thrashing because accesses traverse the array in cadence.

4.2 A Conservative Heuristic

A general problem with data reordering is that it is also a *global* transformation: what is profitable for one loop nest, might be damaging cache behavior for subsequent loop nests. If only one array of a merge-set is accessed in another loop nest, spatial locality is violated.

There is no such thing as a low cost operation for copying data into a different layout pattern between loop nests, although some of this effort could be hidden behind high-latency floating point operations. Replication, too, is out of the question for large datasets. Thus, this paper chooses a conservative heuristic as a first approximation.

The number of merged arrays l for a l-way merging should always be a power of 2 of a multiple thereof to avoid unused space in a cache block. Most of the caches in out study are either direct mapped or 2-way associative. Thus, we restrict to *2-way merging*, hoping to make up for the benefits of 2-way associativity in a direct-mapped cache.

We maintain a list of loop nest levels and arrays that are accessed therein. The heuristic merges the arrays if they are used uniformly in all of the loops. In order to identify individual loops, we label them by tuples $id = i_{1,1}, \ldots, i_{s,p}$ respectively, i referring to the loop index, s referring to the nest level, p to the p-th instance of a loop over index i. By this, we capture the individual loop nests.

Heuristic: If two arrays A, A' are accessed uniformly wrt. dimension n in all individual loop nests id, i.e. $A, A' \in u_{k_n}(A_1, \ldots, A_l)$ holds for at least one k_n, then $t(A, \ldots, A') \leftrightarrow u_{k_m}(A, \ldots, A') \in \bigcap u_{k_n}$.

While the above works well with the codes we examined so far, a less conservative heuristic might want to allow for a less strict criterion than the set intersection. This necessitates a cost measure to assess the tradeoff between miss reduction and the possible loss of spatial locality. The cautious choice of 2-way merging will reduce the risk of violating spatial locality in that case: at most, $\frac{B}{2}$ elements are wasted.

4.3 Modifications to Code and Data

This is done now by preprocessing, will later be implemented at compile time. Like the analysis, this is rather easy and can be done efficiently. First, the compound array is given a new name which is replaced throughout the program. Then, memory for a merge-set is allocated together, simply changing the size of dimension N to 2*N. Finally, index calculations are replaced. Care must be taken not to incur too much additional cost — the FORTRAN compiler is very sensitive to index operations because it can highly optimize the usual expressions. Experience shows that doubling the stride and boundary of a loop is the cheapest way. Figure 3 shows the pseudo-code of the overall algorithm for analysis and source code transformation.

```
In: $maxid$ number of loop nests, $Vars$ list of variables in loop nest $k$
Out: 2-way merge-sets

For $k$= 1 to $maxid$
    For-all$Vars$
        analyze uniformity with standard algorithms
        distinguish $u_{k_m}$ for array dimensions $m = 1$ to $n$
    End for-all
End for
For $k$= 1 to $maxid$
    If $A, A' \in u_{k_m}(k)$ for any $k$
    Then $t(A, A')$                                          /*consider innermost uniformity*/
End for                                                      /*build tentative togetherness relations*/
For all pairs $A, A'$
    For $k$= 1 to $maxid$
        If $A, A' \in Vars \wedge A, A' \notin t'$
        Then delete $t(A, A')$
        Endif                                                /*delete unless together in every loop nest*/
        If $t(A, A')$, $t(A, A'')$ for $A \neq A' \neq A''$  /*in case two combinations of same array survive*/
            Then If $|t(A, A')| \leq |t(A, A'')|$
            Then delete $A, A'$ Endif                        /*keep the more frequently used pair*/
    End for
End for-all
For-all $t(A, A')$                                           /*apply transformations to FORTRAN code*/
    $A, A' \rightarrow$M$\circ A \circ A'$                   /*rename compound array prefixed by M*/
    [$N_1$]...[$N_n$]$\rightarrow$ [2*$N_1$]...[$N_n$]       /*double innermost dimension size*/
    DO 10 I = 1,N$\rightarrow$DO 10 I = 1,2*N,2              /*modify loop bounds and stride*/
    $A$($f(I_s)$, ... ,$f(I_1)$)$\rightarrow$ $A^*(f(I_s - 1), \ldots, f(I_1))$   /*modify array indexes within compound array $A^*$*/
End for-all
```

Figure 3: Conservative Algorithm for 2-way array merging

Note that while possibly endangering spatial locality if used too aggressively, merging can on the other hand actually enhance spatial locality in cases where k_n-uniformity cannot be achieved: By merging l accesses that are k_m-uniform for $m < n$, i.e. they are accessed in the same loop iteration, spatial reuse can be improved by a factor of $\lfloor \frac{B}{l} \rfloor$.

5 Case study: Mesh Generation with Thompson Solver

Mesh Generation with Thompson Solver, better known as `tomcatv` within the SPEC92 and SPEC95 floating point benchmarks, is infamous for its bad cache behavior and thus has become a favorite target for cache optimization. Though smallest in code size of the entire benchmark suite, it behaves extremely badly in the cache, both of which make it suitable for presentation here. `Tomcatv` is a typical *stencil code* (see section 2): It has loop nest depth 2 (not counting the outermost iteration loop) and exhibits four-point stencil patterns and a simple 2-point stencil. The code is made up of seven individual loop nests ($id = 1, \ldots, 7$), six of which iterate

1:	$J_{1,1}$	$I_{2,1}$:	X,Y,RX,RY,AA,DD
2:	$J_{1,2}$	$I_{2,2}$:	RX,RY
3:		$I_{1,1}$:	D
4:	$J_{1,3}$	$I_{2,3}$:	AA,DD,RX,RY,D
5:	$J_{1,4}$	$I_{2,4}$:	RX,RY,D
6:	$J_{1,5}$	$I_{2,5}$:	RX,RY,AA,D
7:	$J_{1,6}$	$I_{2,6}$:	X,Y,RX,RY

1:	$J_{1,1}$	$I_{2,1}$:	X,Y,RX,RY	
2:	$J_{1,1}$	$I_{2,2}$:	X,Y,RX,RY,AA,DD	
3:	$J_{1,1}$	$I_{2,3}$:	RX,RY	
4:		$I_{1,1}$:	D	
5:	$J_{1,1}$	$I_{2,4}$:	AA,DD,RX,RY,D	$N \to 1$
6:	$J_{1,1}$:		RX,RY,D	
7:	$J_{1,2}$	$I_{2,5}$:	RX,RY,AA,D	$N \to 1$

Figure 4: (left) original loops (right) after loop fusion, reuse distance improvement

both over index variables I and J (Figure 4). The first step is to apply loop fusion order to nest a sequence of I-loops within a J-loop, so that several traversals of one array column are completed before the next is accessed. This changes uniformity from type J to type I and improves temporal locality. The danger of cache thrashing has however be increased: what before was only erratic, now becomes imminent because accesses run in cadence.

Figure 4 shows an abstraction of the code. On the left, loops are shown before, on the right after loop fusion, In both cases there are seven distinct loop nests (denoted by lines 1: to 7: in the first column). We abstract from the actual loop structure by denoting the loop indices I and J in the second and third column. The first subscript shows the depth (usually 1 for J, 2 for I). The second subscripts are only used to distinguish between the independent instances of I and J. $I_{2,5}$ for example means the 5th instance of the loop index I at depth 2. Loop 3 (loop 4 after fusion) goes over I only, so here I has depth 1. All arrays that occur in a loop nest are shown in the fourth column. The biggest change occurs in loop 5 and 7 where uniformity of all references with respect to the inner loop is achieved. $N \to 1$ stands for the improvement of reuse distances through loop fusion: instead of in two iterations of the outer loop which are N iterations of the inner loop apart, accesses happen in the same iteration.

Figure 5 shows the analysis result for the accesses before and after loop fusion. All arrays that are uniform with respect to I and J, are placed together in a set u_I or u_J, respectively. Arrays that share a set u_I are innermost uniform.

1:	$J_{1,1}$	$I_{2,1}$:	$u_J(X,Y), u_I(X,Y),$ $u_J(RX,RY,AA,DD),$ $u_I(RX,RY,AA,DD)$
2:	$J_{1,2}$	$I_{2,2}$:	$u_J(RX,RY), u_I(RX,RY)$
3:		$I_{1,1}$:	D
4:	$J_{1,3}$	$I_{2,3}$:	$u_J(AA,DD,RX,RY,D)$
5:	$J_{1,4}$	$I_{2,4}$:	$u_J(RX,RY,D), u_I(RX,RY,D)$
6:	$J_{1,5}$	$I_{2,5}$:	$u_J(RX,RY,AA,D), u_I(AA,D)$
7:	$J_{1,6}$	$I_{2,6}$:	$u_J(X,Y,RX,RY),$ $u_I(X,Y,RX,RY)$

1:	$J_{1,1}$	$I_{2,1}$:	$u_J(X,Y), u_I(X,Y),$ $u_J(RX,RY), u_I(RX,RY)$
2:	$J_{1,1}$	$I_{2,2}$:	$u_J(X,Y), u_I(X,Y),$ $u_J(RX,RY,AA,DD),$ $u_I(RX,RY,AA,DD)$
3:	$J_{1,1}$	$I_{2,3}$:	$u_J(RX,RY), u_I(RX,RY)$
4:		$I_{1,1}$:	D
5:	$J_{1,1}$	$I_{2,4}$:	$u_I(AA,DD,RX,RY,D)$
6:	$J_{1,1}$:		$u_J(RX,RY,D), u_I(RX,RY,D)$
7:	$J_{1,2}$	$I_{2,5}$:	$u_I(RX,RY,AA,D), u_J(AA,D)$

Figure 5: uniformity and togetherness (left) original (right) fused loop

As a result of applying the algorithm in Figure 3, only $t(X,Y)$ and $t(RX,RY)$ remain as merge-sets, because they are alway used together. Other candidates like $t(AA,D)$ are evicted because D is also used alone in loop nest 4. In [14] only X and Y are merged, whereas RX and RY are accessed much more frequently. On the left of the tabular, the loop identifiers are shown, on the right the access structure for the arrays. AA and D are not merged in this conservative heuristic because it would incur penalty in loops where only D and not AA is accessed (as is the case for loop 4). In the case of `tomcatv` indexing is most efficiently hidden within the loop index. Figure 6 shows modified code pieces for declaring, reading in, and accessing the new merged array MXY.

Figure 6.a shows the applied techniques, where *none* denotes the original code taken as is from SPEC. In addition we examine merging X and Y without loop fusion (*merge2*) and *merge2+2*, where additionally RX and RY are merged, as would be the result of our algorithm. The abbreviation *fusion2* corresponds to the technique applied in [14], where only X and Y are merged in addition to loop fusion, For merging X, Y and RX, RY in addition to fusing, *fusion2+2*, the best results for direct-mapped cache and TLB are expected. As shown in [18], only inter-array padding is applicable to the fused loops (*fpad*), while the original loop structure is viable to both inter- and intra-array padding (*pad*).

(a)

```
REAL*8  MXY(2*NMAX,NMAX)
...
DO   10  J = 1,N
  DO   10   I = 1,2*N,2
    READ(...) MXY(I-1,J),MXY(I,J)
...
DO   50  I = 2,2*N-3,+2
         XX = MXY(I+2,J)-MXY(I-2,J)
         YX = MXY(I+3,J)-MXY(I-1,J)
...
```

(b)

name	modifications
none	original program
merge2	merge RX,RY
merge2+2	merge RX,RY and X,Y
pad	inter- and intra padding
fusion	loop fusion
fusion2	loop fusion + merge2
fusion2+2	loop fusion + merge2+2
fpad	loop fusion + inter-padding

Figure 6: (a) Code for array merging (b) Variants of code for various optimizations

6 Experimental Results

With the goal to get an insight into the response of different architectures to merging, we present a comparison for one benchmark, `tomcatv` from section 5, in full detail. Table 1 summarizes the characteristics of the arhitectures used in this study. Memory hierarchy level capacity C_i is always a power of two; the number of

Machine	Alpha 500au	Alpha DS10	Ultra-10	SGI Origin*
CPU	Alpha 21164	Alpha 21264	UltraSparc-II	MIPS R12000
Clock rate	500MHz	466MHz	300MHz	300MHz
L1 (A/B/C)	1/32B/8KB	2/64B/64KB	1/32B/16KB	2/32B/32KB
L2 (A/B/C)	3/64B/96KB	1/64B/2MB	1/64B/512KB	1/64B/8MB
L3 (A/B/C)	1/64B/4MB	-/-/-	-/-/-	-/-/-
TLB (A)	64	128	64	64
MM (B**/C)	*8K/256MB	8K/512MB	8K/320MB	16K/16GB
compiler (option)	f77 (-fast)	f77 (-O4)	f77 (-fast)	f77 (-O3)
Operating System	Digital Unix 4.0D	FreeBSD4.0	SunOS5.7	Irix 6.5.3

Table 1: Machine configurations, *single processor mode, **virtual memory page size. Cache sizes refer to the data cache hierarchy only

arrays in loop nests of the codes we surveyed is usually large, indicating the imminent threat of cross interference and even more potential for merging.

6.1 Methodology

For a realistic assessment of the benefits, it is necessary to select a high level compiler optimization. Compiler optimization may well interact badly with the program and data layout transformations employed. Optimization level is -fast for Alpha 21164 and SPARC, -O4 for Alpha 21264 (highest available under freeBSD) and -O3 for the SGI (as recommended for most reliable use of the tool set). The benchmark uses 513*513 arrays of `REAL` type 8 Byte elements, and the maximal number of iterations is set to 750 (both the original SPEC95 settings). All in all, we examine eight variants of `tomcatv`.

As to measuring the cache miss rates on the ALPHAS, we used a simulation based on ATOM [21], a tool that instruments binaries. The TLB is evaluated directly via the hardware counters with the help of Digital's DCPI [2]. Fast-cache on SPARC also uses a binary-driven simulation was developed within the Wisconsin tool set, together with the tools used in [14] for profiling. The Mips R1200 features a variety of tools [20] that allow a convenient inspection of the hardware counters.

6.2 Run Times

For reasons of tradeoff as discussed in the previous section, it is crucial to also give run times; a correlation should be observable. We now show run times for the entire program (750 iterations as in the SPEC95 suite). Merging has more potential when combined with loop fusion because it avoids thrashing. The results are good

Table 2: Run time results (seconds) for `tomcatv` size 513*513

Architecture	Alpha 21164	Alpha 21264	UltraSparc	R12000
none	180	127	237	135
merge2	168	119	227	136
merge2+2	162	111	248	128
pad	180	108	245	133
fusion	144	104	173	95
fusion2	137	91	217	95
fusion2+2	129	81	198	110
fpad	131	94	198	94

wrt. the Alphas' memory hierarchies (up to 28% improvement on the 21164, 36% on the 21264). Intra-array padding (as part of *pad*) incurs high run time penalties on all architectures as FORTRAN cannot optimize the additional index operations appropriately. With loop fusion, *fpad* can only use inter-array padding which has practically no additional run time cost (a dummy array is introduced during declaration[18]), but also no benefit except in the rare cases where several arrays would have mapped to a starting address that maps to the same cache line. The UltraSparc run time degrades with merging, unexpectedly for a purely direct-mapped architecture with comparatively small caches. Also, this is out of proportion to the observed miss rates. With this architecture, our results indicate that pure loop fusion performs best, but more research is required to dig out the reasons. On the SGI Origin, featuring a powerful memory hierarchy for high-performance graphics demands, *fusion2+2* improves run time only 18.5%, while *fusion2* improves it by over 29%.

6.3 Miss Rates

For validation, we show also the miss rates. Table 3 shows detailed miss rate results for all code variants and architectures. The Alpha 21164 has a very small, direct mapped L1 data cache, which badly interacts with the total of eight relatively small arrays. Here, Lebeck and Wood's *fusion2* beats all other methods, except that merging both X, Y and RX, RY exhibit significantly fewer TLB misses. Alpha 21264 shows the most significant benefits from merging, particularly for *merge2+2* and *fusion2+2* by our heuristic. This is surprising because it features a 2-way associative L1 data cache and a very large L2 cache (see Table 1). Actually SPEC chose an array size of 513, accounting for a loop index range of 512. A power of two, this interacts pathologically with the power-of-two cache size. Intra-array padding is specialized in avoiding this case, which is reflected by the good results. The SPARC interacts badly with both padding and merging, which is equally surprising for a memory hierarchy with comparatively limited facilities as the comparatively large profits of Alpha 21264. For the SGI, D1 miss rates increase for all optimized variants. Loop fusion without merging and *fusion2* yield the best results, in spite of suffering from worse L1 cache behavior. For nearly all merged variants however, TLB misses are reduced.

In summary, the run time improvements do not alway show a perfect correlation to the change in miss rates. Where they improve significantly stronger however, this coincides with the TLB miss reduction.

7 Conclusions and Future Work

This paper presents, to our knowledge, the first systematic compiler approach to array merging. We have shown that for a large class of scientific codes characterized by stencil accesses, merging can be profitably combined with standard techniques, meanwhile avoiding some of the dangers of other data layout approaches. While it is known that padding interacts well with tiling [16], merging can exploit the benefits of loop fusion to a larger extent. As shown in the case study, systematic merging can significantly reduce misses on all levels of

Table 3: Miss rates for `tomcatv` on the four target architectures

Alpha21164					UltraSparc-10			
MH level	D1	D2	D3	TLB	MH level	D1	D2	TLB
none	62.1	13.0	3.83	0.052	none	14.6	3.62	0.029
merge2	53.0	11.3	4.10	0.046	merge2	11.6	3.65	0.028
merge2+2	45.4	5.27	3.71	0.041	merge2+2	19.9	3.86	0.030
pad	48.8	5.81	3.85	0.052	pad	14.7	3.84	0.039
fusion	62.9	5.05	2.64	0.057	fusion	14.5	3.17	0.020
fusion2	28.8	7.36	2.99	0.048	fusion2	15.6	2.43	0.017
fusion2+2	41.3	3.56	3.75	0.042	fusion2+2	16.7	2.56	0.018
fpad	47.6	5.17	2.85	0.056	fpad	14.3	2.16	0.019
Alpha21264					R12000			
MH level	D1	D2		TLB	MH level	D1	D2	TLB
none	22.6	11.3		0.049	none	15.8	1.09	0.027
merge2	20.9	11.3		0.038	merge2	15.9	1.59	0.026
merge2+2	14.7	11.1		0.021	merge2+2	15.0	1.60	0.026
pad	14.1	4.58		0.058	pad	10.9	1.22	0.031
fusion	26.1	5.3		0.055	fusion	17.8	1.26	0.009
fusion2	18.6	3.03		0.044	fusion2	18.2	1.25	0.008
fusion2+2	17.4	4.15		0.036	fusion2+2	16.3	1.70	0.009
fpad	13.4	4.50		0.049	fpad	15.4	1.31	0.009

the memory hierarchy including the TLB. Although much more work is required to gain full insight into the interaction with current high-performance memory hierarchies, compilers and operating systems, the overall experimental results so far are extremely encouraging.

Although compile time is not an issue in this paper, it is obvious that merging is cheaper than padding: at most the same analysis power is required, and merging does not require a compile-time search for pad size. Merging can be easily handled with today‘s compiler techniques. Encouraged by the experimental results, we are currently integrating the analysis and heuristic into a large, widely used research compiler framework. TLB misses due to scattering of data over pages are a major problem also non-scientific code, where first approaches exist for merging of simple data structures [10]. Inter-loop analysis techniques, such as the matrix-based work of Ahmed et al. [1] can likely be used to extend inter-loop-nest analysis beyond the current heuristic.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the International Conference on Supercomputing*, Sante Fe, New Mexico, May 8-11 2000.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. CAS-CON'94*, pages 270–282, Toronto, Ontario, 1994.

[5] G. J. Burnett and Jr. Coffman, E.G. A study of interleaved memory systems. In *Proc., AFIPS 1970 Spring Jt. Computer Conf.*, volume 36, pages 467–474, Montvale, NJ, 1970.

[6] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *ACM SIGPLAN Notices*, 33(11):139–149, November 1998.

[7] Michal Cierniak and Wei Li. Interprocedural array remapping. In *Proc. PACT*, pages 146–155, San Francisco, CA, November 10–14, 1997. IEEE Computer Society Press.

[8] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 30(6) of *ACM SIGPLAN Notices*, pages 279–290, June 1995.

[9] Daniela Genius. A case for array merging in memory hierarchies. Technical report, University of North Carolina at Chapel Hill, 2000. Technischer Bericht.

[10] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An Approach to improve Locality using Sandwich Types. In *Proceedings of the 1998 Types in Compilation workshop*, pages 194–215, Kyoto, Japan, March 1998. Springer LNCS 1473.

[11] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM TOPLAS*, 21(4):703–746, July 1999.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufman, 2. edition, 1996.

[13] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991.

[14] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

[15] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.

[16] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and A. Nicolau. Improving cache performance through tiling and data alignment. In *IRREGULAR 1997*, pages 167–185. Springer LNCS 1253, 1997.

[17] Jai Rawat. Static analysis of cache performance for real-time programming. Technical Report IASTATECS//TR93-19, Iowa state university, Nov 1993.

[18] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, Juni 1998.

[19] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. In *Proc. of the 2001 International Conference on Computational Science (ICCS 2001)*, San Francisco, CA, May 2001. to appear.

[20] Silicon Graphics Inc. Performance analysis using the MIPS R10000 performance counters. In *http://techpubs.sgi.com*, 1996.

[21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[22] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.