Improving Cache Effectiveness through Array Data Layout Manipulation in SAC

Clemens Grelck

University of Kiel
Department of Computer Science and Applied Mathematics
24098 Kiel, Germany
e-mail: cg@informatik.uni-kiel.de

Abstract. SAC is a functional array processing language particularly designed with numerical applications in mind. In this field the runtime performance of programs critically depends on the efficient utilization of the memory hierarchy. Cache conflicts due to limited set associativity are one relevant source of inefficiency. This paper describes the realization of an optimization technique which aims at eliminating cache conflicts by adjusting the data layout of arrays to specific access patterns and cache configurations. Its effect on cache utilization and runtime performance is demonstrated by investigations on the PDE1 benchmark.

1 Introduction

SAC is a functional array processing language, which tries to combine generic, high-level program specifications with efficient runtime behaviour [20, 21]. Particularly in the field of numerical applications, the efficient utilization of the memory hierarchy plays a key role in achieving good performance [14]. However, for many numerical application programs it can be observed that small variations in problem sizes may have a significant impact on runtime performance. This is due to systematic cache conflicts which may occur for unfavourable combinations of array access patterns and array data layout in the presence of limited cache associativity [2].

Assuming the runtime performance of a program is poor for one problem size, but turns out to be significantly better for a marginally larger problem size, it is a rather straightforward idea to mimick the data layout associated with the larger problem size when actually dealing with the smaller one. In doing so, the originally dense representation of arrays is manipulated by the introduction of dummy elements in one or another dimension, so-called array padding [1]. The array padding optimization implemented in SAC basically consists of three steps. First, SAC code within WITH-loops, the predominant SAC language construct for the specification of aggregate array operations [7], is thoroughly analysed for array accesses, and the arrays involved are associated with accurate access patterns. Second, an inference heuristic estimates the cache utilization and identifies an appropriate amount of padding where necessary. Cache phenomena

such as spatial and temporal reuse are taken into account. Third, the data layout modification proposed by the inference heuristic is realized as a high-level transformation on intermediate SAC code.

The remainder of this paper is organized as follows. After a more detailed problem identification in Section 2, Sections 3, 4, and 5 describe the three steps of the implementation. Their effect on runtime performance is demonstrated by means of the PDE1 benchmark in Section 6. Section 7 sketches some related work while Section 8 concludes.

2 Problem identification

We have chosen the benchmark PDE1 as an example in order to investigate and quantify the potential impact of the problem size on runtime performance. PDE1 implements red/black successive over-relaxation on 3-dimensional grids. The benchmark itself as well as various implementation opportunities for SAC are discussed in [8]. In our experiments we have systematically varied the size of the 3-dimensional grid from 16^3 until 528^3 in uniform steps of 16 elements in each dimension. With double precision floating point numbers, this involves array sizes between 32KB and 1.1GB. All experiments have been done on a SUN Ultra Enterprise 4000 system. Fig. 1 shows the average times required to re-compute the value of a single inner grid element. It can be observed that these times significantly vary for the problem sizes investigated. While 155nsec are sufficient to update an inner element of a grid of size 16^3, it takes up to 866nsec to complete the same operation in a grid of size 256^3. Although exactly the same sequence of instructions is executed for each inner grid element regardless of the problem size, the time required to do so varies by a factor of 5.6.

Such extreme variations in runtime performance can only be attributed to different degrees of cache utilization caused by varying data layouts introduced by different problem sizes. In order to substantiate claims like this, the SAC compiler and runtime system are equipped with a tailor-made cache simulation feature. On demand, a trace of all array accesses during program execution is generated. This allows for a complete simulation of the cache behaviour, yielding statistical information regarding the effectiveness of cache utilization. Each processor of the SUN Ultra Enterprise 4000 multiprocessor system is equipped with a 16KB L1 data cache and a 1MB L2 unified cache. Both are direct-mapped and use cache lines of 32 and 64 bytes, respectively. Fig. 2 shows the percentage of L1 cache hits for the various problem sizes investigated as well as the percentage of memory requests satisfied by any of the two cache levels. It actually turns out that the extreme performance variations observed in Fig. 1 largely coincide with similar variations in the simulated cache hit rate.

The design of cache memories is essentially based on two assumptions: temporal locality and spatial locality [9]. A program exhibits temporal locality if it is likely that once a memory address is referenced in the code, it will be referenced again soon. Therefore, data is loaded into the fast cache memory in order to satisfy subsequent requests without slow main memory interaction. Spatial

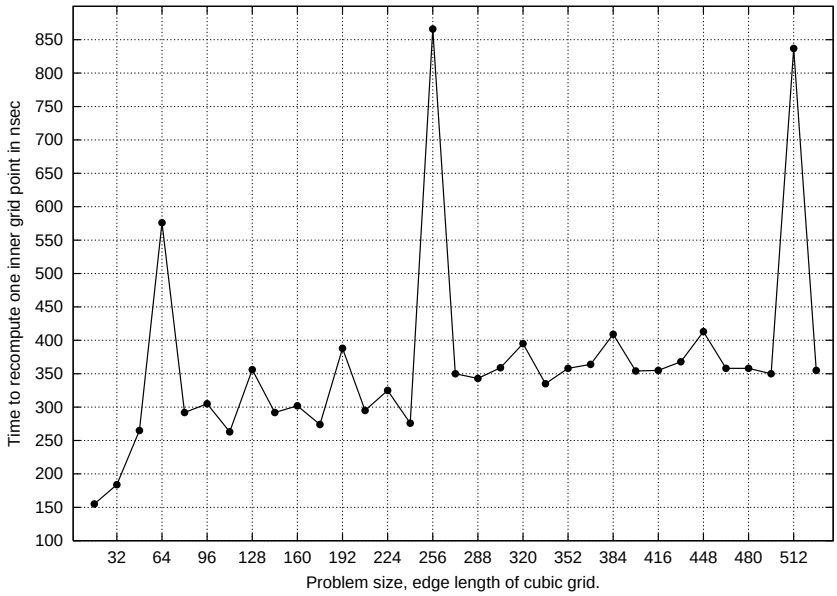

Fig. 1. PDE1: average time required to re-compute a single grid element.

locality means that once a memory address is referenced, adjacent addresses are likely to be referenced soon. For this reason, caches are internally organized in so-called *cache lines*, which typically comprise between 16 and 128 bytes of contiguous memory. All data transfers between main memory and cache involve entire cache lines rather than single bytes or words of memory. Application programs do only benefit from caches to the extent to which they exhibit spatial and temporal locality.

However, spatial and temporal locality are mainly characteristics of a given program, and hence, do not explain the observed performance variations. In fact, it is a limitation in cache memory hardware that is responsible for this: very limited set associativity. In order to efficiently distinguish cache hits from cache misses, any given memory address can only be mapped to one of very few locations in the cache, which are directly derived from the memory address itself. Today's caches usually provide set associativities between one and four. As a consequence, data may be flushed from the cache before potential reuse is actually exploited, although the cache is sufficiently large to allow the reuse in principle. These so-called *conflict misses* may seriously limit cache utilization, as can be seen in Figs. 1 and 2. Since concrete memory addresses decide over cache conflicts, they are extremely sensitive against memory layout variations, in particular, whenever regularly structured data is accessed in regular patterns, which is typical for numerical codes involving large arrays.

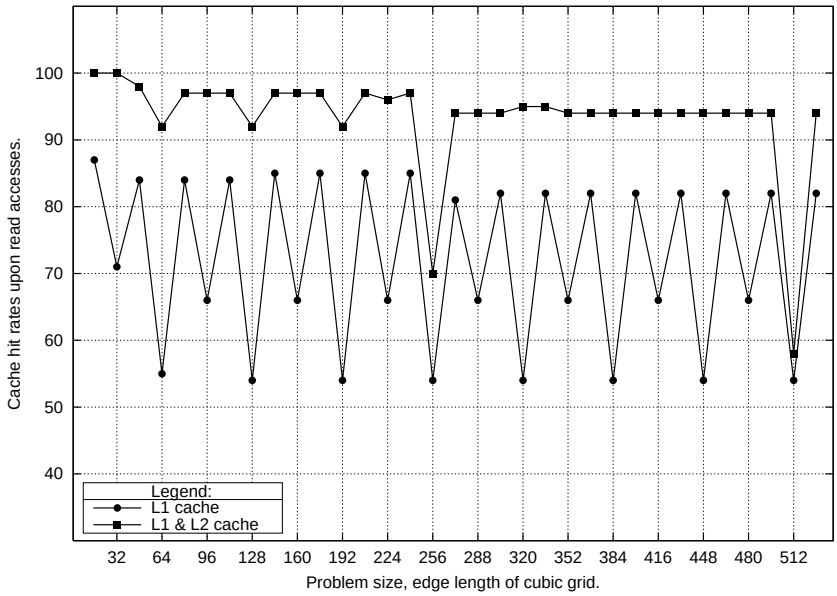

Fig. 2. PDE1: simulated cache performance.

Various different cache effects have been identified [22], e.g., a *spatial reuse conflict* occurs whenever not all array elements referenced in a single iteration of an inner loop can simultaneously be held in the cache. The number of different array elements which are mapped to the same cache set exceeds the cache's set associativity and, hence, cache lines are flushed from the cache before potential reuse can be realized in the following iteration. A *temporal reuse conflict* occurs when potential reuse between two references to the same array element cannot be exploited because another array reference interferes and causes the first one to be flushed from the cache before the potential reuse actually occurs. Conflicts are classified as either arising from references to the same array, so-called *self-interference* conflicts, or to different arrays, so-called *cross-interference* conflicts.

Thorough elimination of cache conflicts is crucial for keeping the runtime performance consistent over a range of problem sizes [13]. This can be achieved by a well-aimed manipulation of the data layout of arrays. Self-interference conflicts can be eliminated by modifying the internal representation of arrays, cross-interference conflicts by adjusting array base addresses. The latter approach is very difficult to realize in a language like SAC, which allocates and de-allocates all data structures dynamically. Therefore, we concentrate on self-interference conflicts in the following. One way to manipulate the internal representation of arrays is *array padding*, a well-known optimization technique that adds dummy elements to an array in one or another inner dimension [1]. For example, an array whose original shape is `[100,100]` may be transformed into an array of shape

[100,102] by adding two columns of dummy elements. Padding an array alters the memory addresses of different elements in different ways and, hence, allows to indirectly manipulate their associated relative cache locations.

However, applying array padding manually has some serious drawbacks. It requires both a lot of effort and expert knowledge by programmers, who in this case are solely responsible to identify where which amount of padding might have a positive impact on runtime performance. Moreover, explicit array padding increases program complexity and makes programs less readable and more error-prone. Last but not least, array padding renders program specifications machine-dependent because each combination of problem size, access pattern, and cache configuration typically requires a different amount of padding.

In contrast, array padding as a compiler optimization may be well-suited to achieve more consistent performance over a wide range of problem sizes and cache configurations. However, things are not as simple in low-level languages such as C or FORTRAN. Since these languages' semantics guarantee a certain (unpadded) data layout, thorough program analysis is required in order to prove that padding does not alter the meaning of a program. Here, the design of high-level languages like SAC pays off. Since they completely abstract from any concrete data layout, language implementations are free to exploit the benefits of varying data layouts as an additional optimization technique.

3 Array access analysis

Accurate analysis of array access patterns is one of the prerequisites for reasoning about cache conflicts. Severe cache conflicts typically arise from regular array references within loops, i.e., two or more references systematically conflict with each other in every iteration of the loop. Therefore, the analysis described in this section focusses on regular array references in WITH-loops. The WITH-loop is a SAC-specific language construct for the specification of aggregate multi-dimensional array operations; a thorough description may, for instance, be found in [7]. An array reference is considered being regular if and only if it can be written in the form

$$\text{val = Array [} \boldsymbol{s} * \boldsymbol{i} + \boldsymbol{d} \text{] ;}$$

where $\boldsymbol{s}$ denotes a constant stride vector, $\boldsymbol{d}$ a constant offset vector, and $\boldsymbol{i}$ the WITH-loop's index variable. Note that $*$ here denotes the elementwise product of two vectors. In other words, locations of regular array references are defined by dimension-wise affine functions of the WITH-loop's index variable. Fig. 3 shows an example WITH-loop featuring a few different regular array references. All array references that cannot be converted to this affine pattern, are considered irregular. They are likely not to conflict in a systematic way with other references, irregular or regular. Therefore, they are just ignored in the sequel.

All array references in the example shown in Fig. 3 are regular with respect to the above definition. This can be inferred during a rather simple bottom-up

```
int[100,100] A;
int[200,150] B;
int[120,120] C;
...
A = with ([1,1] <= iv < [100,100])
    {
      a   = B[ iv - 1];
      b   = C[ iv];
      c   = B[ iv + 2];
      d   = C[ [42, 42]];
      e   = B[ [2, 1] * iv];
      tmp = iv + [1, 1];
      f   = B[ [2, 1] * tmp];
      val = a + b + c + d + e + f;
    }
    genarray([100,100], val);
```

Fig. 3. Examples of regular array references in a WITH-loop.

traversal of the WITH-loop body. Compact array access information is accumulated, as outlined in Fig. 4. The array access pattern $\mathcal{AP}$ is a set of triples; each triple represents exactly one regular array reference found in the WITH-loop body. The access triples themselves consist of the name of the referenced array, the stride vector $\boldsymbol{s}$ and the offset vector $\boldsymbol{d}$.

As already pointed out, the technique presented in this paper focusses on self-interference cache conflicts, i.e. conflicts between references to the same array. References to different arrays, although occurring in a single WITH-loop, may be handled separately. Furthermore, only array references which are characterized by identical stride vectors $\boldsymbol{s}$ may actually interfere with each other in a systematic and, hence, expensive manner. These considerations lead to the division of an access pattern into disjoint so-called *conflict groups*. Each conflict group then contains exactly one subset of array references which are likely to systematically interfere with each other.

The example access pattern $\mathcal{AP}$ in Fig. 4 results in the introduction of four conflict groups, as outlined in Fig. 5. Each conflict group is represented by a pair consisting of the type of the referenced array and a sequence of offset vectors. The stride vectors are no longer needed. Whether or not two references of the same

$$
\begin{aligned}
\mathcal{AP} = \{\ & < \texttt{B},\ \texttt{[1, 1]},\ \texttt{[-1,-1]} >, \\
& < \texttt{C},\ \texttt{[1, 1]},\ \texttt{[0, 0]} >, \\
& < \texttt{B},\ \texttt{[1, 1]},\ \texttt{[2, 2]} >, \\
& < \texttt{C},\ \texttt{[0, 0]},\ \texttt{[42,42]} >, \\
& < \texttt{B},\ \texttt{[2, 1]},\ \texttt{[0, 0]} >, \\
& < \texttt{B},\ \texttt{[2, 1]},\ \texttt{[2, 1]} > \}
\end{aligned}
$$

Fig. 4. Array access pattern derived from example WITH-loop in Fig. 3.

$\mathcal{CG}_1 = <$ `int[200,150]` $, <$ `[-1,-1], [ 2, 2]` $> >$
$\mathcal{CG}_2 = <$ `int[120,120]` $, <$ `[ 0, 0]` $> >$
$\mathcal{CG}_3 = <$ `int[120,120]` $, <$ `[42,42]` $> >$
$\mathcal{CG}_4 = <$ `int[200,150]` $, <$ `[ 0, 0], [ 2, 1]` $> >$

Fig. 5. Conflict groups derived from access pattern $\mathcal{AP}$ in Fig. 4.

conflict group cause a cache conflict solely depends on their relative distance in memory, which is invariant against their strides. Last but not least, no cache conflicts may occur in conflict groups consisting of a single array reference only. As a consequence, all such conflict groups, e.g. $\mathcal{CG}_2$ and $\mathcal{CG}_3$ in Fig. 5, are simply ignored. The number of conflict groups can further be reduced by the elimination of multiple occurrences of identical ones and of those that are subsets of others.

4 Padding inference heuristic

This section presents the central padding inference algorithm. It associates each array type occurring in a SAC program or module with a padding recommendation appropriate for avoiding spatial and temporal self-interference cache conflicts. The basic idea is to pad all arrays of a given type (consisting of base type and shape) in a uniform way if at all. This helps to avoid costly transformations between unpadded and padded or even differently padded representations of arrays which originally had identical types and, hence, data layouts. Such transformations are limited to module boundaries, providing programmers with some means of control over array padding.

In addition to the conflict groups implicitly derived from SAC code, as described in Section 3, the inference scheme presented here is based on the specification of a cache configuration, which must explicitly be stated at compile time. It consists of the cache size and the cache line size, both in bytes, as well as the cache's set associativity. Furthermore, an upper limit must be set on memory consumption overhead caused by array padding.

When focussing on a single array type, which consists of a scalar base type and an original shape SHP, we may easily compute the cache size CS and the cache line size CLS in array elements. These figures, rather than the external specifications in bytes, are used by the inference scheme. Moreover, we compute the number of cache sets, $NSET := CS/(CLS * CA)$ where CA denotes the cache's set associativity. With this internal cache specification at hand, all conflict groups associated with the array type under consideration are then successively analysed with respect to potential cache conflicts. Padding recommendations are accumulated in a vector PAD, which is initially set to $\mathbf{0}$, i.e., we start out with recommending no padding at all.

First, spatial reuse conflicts are addressed. Let us consider a conflict group $\mathcal{CG}$ representing array references $R_1, \ldots, R_n$. For each reference R_i, the offset

vector D_i is converted into a scalar offset with respect to the array shape SHP extended by the padding vector PAD recommended so far:

$$\forall\ i \in \{1,\ldots,n\} \quad : \quad OFFSET_i \ := \ ADDR(\ D_i\ ,\ SHP + PAD) \quad ,$$

where $ADDR(vec, shp)$ is a function that computes the offset of vec in the row-major unrolling of an array with shape shp, i.e.

$$ADDR(vec, shp) \ := \ \sum_{k=0}^{|shp|} (vec_k \ * \ \prod_{m=k+1}^{|shp|} shp_m) \quad .$$

For reasons of simplicity it is desirable to avoid negative offsets. Since our interest is also limited to relative distances of cache locations, computed offsets can easily be shifted by a constant value. The easiest way to avoid negative offsets is to generally arrange the elements of a conflict group in ascending lexicographical order with respect to their offset vectors, and to subtract $OFFSET_0$ from each scalar offset, i.e.

$$\forall\ i \in \{1,\ldots,n\} \quad : \quad OFFSET_i := OFFSET_i - OFFSET_0 \quad .$$

With the shifted offsets at hand, we now determine the respective cache sets

$$\forall\ i \in \{1,\ldots,n\} \quad : \quad SET_i := (OFFSET_i / CLS) \bmod \mathrm{NSET} \quad .$$

For each reference R_i, we compute the number $NPSC_i$ of potential spatial reuse conflicts with other references. Two references R_i and R_j potentially conflict with each other if and only if

$$((|SET_i - SET_j| \ < \ 2 \quad \vee \quad (|SET_i - SET_j| \ = \ NSET - 1))$$
$$\wedge \quad (|OFFSET_i - OFFSET_j| \ > \ 2 * CLS) \quad ,$$

i.e., they reference non-adjacent memory addresses which are mapped to identical or directly adjacent cache sets. The latter serves as an additional buffer that allows to completely abstract from relative placements of references within cache lines. In a direct-mapped cache ($CA = 1$), any potential conflict actually is a real conflict. However, in general, a conflict occurs whenever the number of potential conflicts equals or exceeds the cache's set associativity CA, i.e., the number of spatial reuse conflicts associated with each array reference is defined as

$$\forall\ i \in \{1,\ldots,n\} \quad : \quad NSC_i := \max(0,\ NPSC_i - CA + 1) \quad ;$$

the total number of spatial reuse conflicts within the conflict group is defined as

$$NSC := \sum_{i=0}^{n} \ NSC_i \quad .$$

If there are no conflicts, i.e., $NSC = 0$, we are done and PAD is the recommended padding for this conflict group with respect to spatial reuse. If the number of conflicts is reduced relative to the best padding found so far, the

current padding and the number of spatial reuse conflicts associated with it are stored as new currently best solution. As long as there are still conflicts, we try to solve them with additional padding, i.e., the padding vector PAD is to be updated. For this purpose, we first identify dimensions that are eligible for padding. Assigning the index 0 to the outermost dimension and counting upwards, the minimum padding dimension is determined as $MINPADDIM := d + 1$, where d is the outermost dimension with $D_i[d] \neq D_j[d]$ for any pair of conflicting array references R_i and R_j. The maximum padding dimension is simply chosen as $MAXPADDIM := |SHP| - 1$. Among all eligible dimensions the outermost one is chosen, where $(SHP + PAD)[d]$ is maximal. This choice of $PADDIM$ guarantees that padding overhead grows in minimal steps. Padding is preferably applied to outer dimensions in order to reduce the negative impact of the loop overhead introduced by it.

The padding vector PAD is incremented by 1 in dimension $PADDIM$ and, assuming this additional padding does not exceed the given limit on memory consumption overhead, the cache behaviour is re-evaluated with this new padding vector as described so far. Otherwise, SHP is reset to 0 in dimension $MINPADDIM$ and, provided that $MINPADDIM < MAXPADDIM$, padding in the next dimension is increased by 1. The entire process is repeated until either all spatial reuse conflicts are eliminated or all padding vectors eligible with respect to the memory consumption overhead limit have been investigated. In the latter case, the best padding found during the process is stored as recommended padding.

With spatial reuse conflicts eliminated as far as possible, we may now focus on temporal reuse conflicts. As a first step, we determine for each reference R_i if there is a chance for temporal reuse from reference R_{i+1} in the presence of simple cache capacity constraints. This is the case if and only if

$$OFFSET_{i+1} - OFFSET_i \quad < \quad (NSET - 2) \ * \ CLS \quad .$$

Note here that all references are sorted with increasing offsets. For each pair of adjacent references R_i and R_{i+1} which may benefit from temporal reuse, we then compute the number of potential temporal reuse conflicts $NPTC$. An array reference R_j, $j \neq i \ \wedge \ j \neq i + 1$ represents a potential temporal reuse conflict if it is mapped to a cache set "in between" those associated with R_i and R_{i+1}, i.e.

$$(SET_i \leq SET_j) \ \wedge \ (SET_j \leq SET_{i+1}) \iff SET_i \leq SET_{i+1} \quad ,$$
$$(SET_i \leq SET_j) \ \vee \ (SET_j \leq SET_{i+1}) \iff SET_i > SET_{i+1} \quad .$$

In analogy to spatial reuse conflicts, the term "potential" is to be understood with respect to set associativity, i.e., the number of actual temporal reuse conflicts NTC is defined as

$$\forall \ i \in \{1, \ldots, n\} \quad : \quad NTC_i := \max(0, \ NPTC_i - CA + 1)$$

for each reference and in total as

$$NTC := \sum_{i=0}^{n} \ NTC_i \quad .$$

Whenever the current padding fails to eliminate all temporal reuse conflicts, a new padding vector candidate is determined in a similar way as for resolving spatial reuse conflicts. However, eligible padding dimensions are restricted in a slightly different way. The minimum eligible padding dimension is defined as $MINPADDIM := d + 1$, where d denotes the outermost dimension with $D_i[d] \neq D_j[d] \neq D_{i+1}[d]$ for any triple of conflicting array references R_i, R_j, and R_{i+1}. The maximum eligible padding dimension $MAXPADDIM$ is given as the outermost dimension d where $D_i[d] \neq D_{i+1}[d]$ holds for the same references R_i and R_{i+1} as above. The basic idea behind these choices for $MINPADDIM$ and $MAXPADDIM$ is to select a padding dimension which, on the one hand, is sufficiently large so that the relative cache locations of adjacent references with potential temporal reuse remain untouched, but, on the other hand, is sufficiently small, so that padding actually alters the relative cache locations between these adjacent references and the conflicting reference in between.

In contrast to the choice of a padding dimension for the elimination of spatial reuse conflicts, an eligible padding dimension to avoid temporal reuse conflicts not necessarily exists. In this case, array padding does not resolve this conflict, and the inference heuristic stops at this point. Otherwise, a new padding vector candidate is chosen exactly as in the context of solving spatial reuse conflicts and temporal reuse conflicts are re-evaluated iteratively until either all are eliminated or the padding overhead constraint is exhausted.

An alternative implementation different from the above inference heuristic is to evaluate all potential padding vectors eligible with respect to the given constraint on additional memory consumption. For each such padding vector, the number of spatial and temporal reuse conflicts as well as the associated overhead are computed. Afterwards, the padding vector which causes the minimal number of conflicts is selected. If there are several equally suitable padding vectors, the one which causes the least overhead is chosen. If there are still multiple candidates, the one which incurs the least padding in inner dimensions is taken eventually. While this alternative implementation is guaranteed to find the most suitable padding with respect to the number of cache conflicts, memory consumption overhead, and loop overhead, it generally requires considerably more computational effort. However, since this effort is made at compile time rather than at runtime, it may be tolerable in many situations.

5 Padding transformation

The padding inference algorithm described in the previous section results in the definition of a function $\mathcal{P}ad\mathcal{T}ype$, which for each array type found in the program or module under consideration yields the recommended padded type. Types for which a manipulation of the internal data layout is not recommended are simply returned by $\mathcal{P}ad\mathcal{T}ype$ as they are. This section focusses on the actual realization of the padding recommendation, which in the sequel will be formalized by means of a transformation scheme $\mathcal{APT}$. It defines a high-level source-to-source transformation on simplified and type-annotated intermediate SAC code.

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{rettypes}\ \mathit{fun}\ (\ \mathit{args}\ )\ \{\ \mathit{vardecs}\ \mathit{instrs}\ \}\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{APT}[\![\ \mathit{rettypes}\ ]\!]\ \mathit{fun}\ (\ \mathcal{APT}[\![\ \mathit{args}\ ]\!]\ )\ \{ \\
\qquad\quad \mathcal{RepArgs}[\![\ \mathit{args}\ ]\!] \quad \mathcal{APT}[\![\ \mathit{vardecs}\ ]\!] \\
\qquad\quad \mathcal{APT}[\![\ \mathit{instrs}\ ]\!] \\
\qquad \}\quad \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \\
\mathcal{APT}[\![\ \mathit{type}\ ,\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{PadType}[\![\ \mathit{type}\ ]\!]\ ,\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \\
\\
\mathcal{APT}[\![\ \mathit{type}\ \mathit{argname}\ ,\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{PadType}[\![\ \mathit{type}\ ]\!]\ \mathit{argname}\ ,\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \\
\\
\mathcal{RepArgs}[\![\ \mathit{type}\ \mathit{argname}\ ,\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathit{type}\ \_\mathit{argname}\ ;\ \mathcal{RepArgs}[\![\ \mathit{Rest}\ ]\!] \quad | \quad \mathcal{ToBePadded}[\![\ \mathit{type}\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{RepArgs}[\![\ \mathit{Rest}\ ]\!] \quad | \quad \mathit{otherwise} \\
\\
\mathcal{APT}[\![\ \mathit{type}\ \mathit{varname}\ ;\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{PadType}[\![\ \mathit{type}\ ]\!]\ \mathit{varname}\ ; \quad | \quad \mathcal{ToBePadded}[\![\ \mathit{type}\ ]\!] \\
\qquad\quad \mathit{type}\ \_\mathit{varname}\ ;\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \\
\quad \Longrightarrow \quad \mathit{type}\ \mathit{varname}\ ;\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \quad | \quad \mathit{otherwise}
\end{array}
$$

Fig. 6. Transformation scheme $\mathcal{APT}$ on function definitions.

The former means that nested expressions are lifted to separate assignments to temporary variables; the latter provides a function $\mathcal{T}ype$, which associates each variable with a SAC data type. The transformation scheme $\mathcal{APT}$ is based on two additional auxiliary functions: $\mathcal{S}hape[\![\ \mathit{type}\ ]\!]$ yields the shape part of an array data type *type* as a vector, and $\mathcal{ToBePadded}[\![\ \mathit{type}\ ]\!]$ decides whether or not a padding is recommended for a given type, i.e.

$$\mathcal{ToBePadded}[\![\ \mathit{type}\ ]\!] \ := \ \mathcal{PadType}[\![\ \mathit{type}\ ]\!] \neq \mathit{type} \quad .$$

Fig. 6 shows the effect of the compilation scheme $\mathcal{APT}$ on function definitions. The formal parameters of a function are traversed, and whenever padding is recommended for a return or argument type, the original type specification is replaced by the respective padded type. A similar transformation is applied to the local variable declarations. As already pointed out in Section 4, the transformation of a padded array into its unpadded representation is necessary in certain situations, e.g. at module boundaries. Since we do not have any a priori knowledge as to whether or not such a transformation will actually be required, additional variable declarations are introduced for each padded original local variable. [1] The same is done for padded formal parameters by means of the auxiliary compilation scheme $\mathcal{RepArgs}$.

The effect of $\mathcal{APT}$ on applications of user-defined and of built-in functions is defined in Fig. 7. Whereas nothing is to be done in the case of locally defined functions, the application of an imported function may require a change in the

[1] Superfluous variable declarations are eliminated by subsequent optimization steps.

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{vars}\ \texttt{=}\ \mathit{fun}\ \texttt{(}\ \mathit{args}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{vars}\ \texttt{=}\ \mathit{fun}\ \texttt{(}\ \mathit{args}\ \texttt{);}\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{vars}\ \texttt{=}\ \mathit{module}\texttt{:}\mathit{fun}\ \texttt{(}\ \mathit{args}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathcal{UnPad}[\![\ \mathit{args}\ ]\!] \\
\qquad\quad \mathcal{R}\mathit{ename}[\![\ \mathit{vars}\ ]\!]\ \texttt{=}\ \mathit{module}\texttt{:}\mathit{fun}\ \texttt{(}\ \mathcal{R}\mathit{ename}[\![\ \mathit{args}\ ]\!]\ \texttt{);} \\
\qquad\quad \mathcal{P}\mathit{ad}[\![\ \mathit{vars}\ ]\!]\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{var}\ \texttt{=}\ \texttt{dim(}\ \mathit{array}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{var}\ \texttt{=}\ \texttt{dim(}\ \mathit{array}\ \texttt{);}\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{var}\ \texttt{=}\ \texttt{shape(}\ \mathit{array}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{var}\ \texttt{=}\ \mathcal{S}\mathit{hape}[\![\ \mathcal{T}\mathit{ype}[\![\ \mathit{array}\ ]\!]\ ]\!]\ \texttt{;} \qquad \mid\ \mathcal{T}\mathit{oBePadded}[\![\ \mathcal{T}\mathit{ype}[\![\ \mathit{array}\ ]\!]\ ]\!] \\
\qquad\quad \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{var}\ \texttt{=}\ \texttt{shape(}\ \mathit{array}\ \texttt{);}\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!] \quad \mid\ \mathit{otherwise}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{var}\ \texttt{=}\ \texttt{psi(}\ \mathit{array}\ \texttt{,}\ \mathit{vec}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{var}\ \texttt{=}\ \texttt{psi(}\ \mathit{array}\ \texttt{,}\ \mathit{vec}\ \texttt{);}\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{var}\ \texttt{=}\ \texttt{modarray(}\ \mathit{array}\ \texttt{,}\ \mathit{vec}\ \texttt{,}\ \mathit{val}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathit{var}\ \texttt{=}\ \texttt{modarray(}\ \mathit{array}\ \texttt{,}\ \mathit{vec}\ \texttt{,}\ \mathit{val}\ \texttt{);}\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{APT}[\![\ \mathit{var}\ \texttt{=}\ \texttt{reshape(}\ \mathit{vec}\ \texttt{,}\ \mathit{array}\ \texttt{);}\ \mathit{Rest}\ ]\!] \\
\quad\Longrightarrow\quad \mathcal{UnPad}[\![\ \mathit{array}\ ]\!] \\
\qquad\quad \mathcal{R}\mathit{ename}[\![\ \mathit{var}\ ]\!]\ \texttt{=}\ \texttt{reshape(}\ \mathit{vec}\ \texttt{,}\ \mathcal{R}\mathit{ename}[\![\ \mathit{array}\ ]\!]\ \texttt{);} \\
\qquad\quad \mathcal{P}\mathit{ad}[\![\ \mathit{var}\ ]\!]\ \mathcal{APT}[\![\ \mathit{Rest}\ ]\!]
\end{array}
$$

Fig. 7. Transformation scheme $\mathcal{APT}$ on function applications.

representations of argument as well as of result arrays. This is described by the three auxiliary compilation schemes $\mathcal{R}\mathit{ename}$, $\mathcal{P}\mathit{ad}$, and $\mathcal{UnP}\mathit{ad}$ defined in Fig. 8.

SAC supports only a very limited number of built-in operations on arrays. For instance, `dim` and `shape` retrieve an array's dimensionality and shape, respectively. Since padding has no effect on dimensionality, any application of `dim` may simply remain as it is. In contrast, an application of `shape` must be replaced by the shape corresponding to the original type of the argument array. The function `psi` selects the element of *array* specified by the index vector *vec*. The offset in memory specified by *vec* is computed using the function $ADDR(vec, shp)$ defined in Section 4. However, this function also computes the correct offset of an array element in a padded array representation when providing the padded shape as second argument. Hence, no code transformation is required for the selection of elements regardless of whether or not an array is padded. The built-in function `modarray` yields an array that is identical to its first argument except for the element denoted by the second argument, which is replaced by the third argument. Since $\mathcal{T}\mathit{ype}[\![\ \mathit{var}\ ]\!] = \mathcal{T}\mathit{ype}[\![\ \mathit{array}\ ]\!]$ and hence

$$\mathcal{P}\mathit{ad}\mathcal{T}\mathit{ype}[\![\ \mathcal{T}\mathit{ype}[\![\ \mathit{var}\ ]\!]\ ]\!] = \mathcal{P}\mathit{ad}\mathcal{T}\mathit{ype}[\![\ \mathcal{T}\mathit{ype}[\![\ \mathit{array}\ ]\!]\ ]\!]\ ,$$

`modarray` can be applied to padded arrays without additional measures. The last remaining built-in function is `reshape`, which creates an array that consists of the same elements as the argument *array*, but is associated with the new shape defined by the argument *vec*. Applications of `reshape` are restricted to arguments where the given array's original shape and the new shape are compatible, i.e., they refer to arrays with the same number of elements. However, as soon as one of the two shapes is padded, this restriction is violated. Even if both shapes are padded, it is rather unlikely that the padded shapes comply with the compatibility restriction. As a way out, both the argument array as well as the result array have to be converted between padded and unpadded representations.

The transformation of an array from a padded into an unpadded representation or vice versa is subject to the three auxiliary compilation schemes $\mathcal{R}ename$, $\mathcal{P}ad$, and $\mathcal{U}n\mathcal{P}ad$ defined in Fig. 8. Whenever a padded array is encountered where an unpadded representation is required, it is transformed by means of a predefined generic function `UnPad`. In a similar way, arrays which are created in an unpadded representation for some reason, but whose types are recommended to be padded according to $\mathcal{P}ad\mathcal{T}ype$, are transformed into the corresponding padded representation using the predefined generic function `Pad`.

Aggregate array operations are defined in one way or another by means of WITH-loops in SAC itself. The effect of the compilation scheme $\mathcal{APT}$ on WITH-loops is described in Fig. 9. Apart from recursively applying $\mathcal{APT}$ to the instructions within the body of a WITH-loop, only a single code transformation is actually required. The expression that defines the shape of the result array in a `genarray`-WITH-loop is replaced by the corresponding padded shape.

Assuming a generator depends in one way or another on the shape of a padded array, all applications of the built-in function `shape` would have been

$$
\begin{array}{l}
\mathcal{R}ename[\![\ var\ ,\ Rest\ ]\!] \\
\quad \Longrightarrow \quad \texttt{\_}var\ ,\ \mathcal{R}ename[\![\ Rest\ ]\!] \qquad \mid\ \mathcal{T}o\mathcal{B}e\mathcal{P}added[\![\ \mathcal{T}ype[\![\ var\ ]\!]\ ]\!] \\
\quad \Longrightarrow \quad var\ ,\ \mathcal{R}ename[\![\ Rest\ ]\!] \qquad \mid\ otherwise \\
\\
\mathcal{R}ename[\![\ const\ ,\ Rest\ ]\!] \\
\quad \Longrightarrow \quad const\ ,\ \mathcal{R}ename[\![\ Rest\ ]\!] \\
\\
\mathcal{P}ad[\![\ var\ ,\ Rest\ ]\!] \\
\quad \Longrightarrow \quad var\ \texttt{= Pad(\_}var\ \texttt{);}\ \mathcal{P}ad[\![\ Rest\ ]\!] \qquad \mid\ \mathcal{T}o\mathcal{B}e\mathcal{P}added[\![\ \mathcal{T}ype[\![\ var\ ]\!]\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{P}ad[\![\ Rest\ ]\!] \qquad \mid\ otherwise \\
\\
\mathcal{U}n\mathcal{P}ad[\![\ var\ ,\ Rest\ ]\!] \\
\quad \Longrightarrow \quad \texttt{\_}var\ \texttt{= UnPad(}\ var\ \texttt{);}\ \mathcal{P}ad[\![\ Rest\ ]\!] \qquad \mid\ \mathcal{T}o\mathcal{B}e\mathcal{P}added[\![\ \mathcal{T}ype[\![\ var\ ]\!]\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{P}ad[\![\ Rest\ ]\!] \qquad \mid\ otherwise \\
\\
\mathcal{U}n\mathcal{P}ad[\![\ const\ ,\ Rest\ ]\!] \\
\quad \Longrightarrow \quad \mathcal{P}ad[\![\ Rest\ ]\!]
\end{array}
$$

Fig. 8. Auxiliary schemes $\mathcal{R}ename$, $\mathcal{P}ad$, and $\mathcal{U}n\mathcal{P}ad$.

$\mathcal{APT}$[[*var* = `with ( generator ) { instrs } genarray( shp , val );` *Rest*]]
$\Longrightarrow$ *var* = `with ( generator ) {` $\mathcal{APT}$[[*instrs*]] `}`
`genarray(` $\mathcal{S}hape$[[$\mathcal{T}ype$[[*var*]]]] `,` *val* `);` $\mathcal{APT}$[[*Rest*]]

$\mathcal{APT}$[[*var* = `with ( generator ) { instrs } modarray( old , iv , val ); Rest`]]
$\Longrightarrow$ *var* = `with ( generator ) {` $\mathcal{APT}$[[*instrs*]] `}`
`modarray(` *old* `,` *iv* `,` *val* `);` $\mathcal{APT}$[[*Rest*]]

$\mathcal{APT}$[[*var* = `with ( generator ) { instrs } fold( fun , neutral , val );` *Rest*]]
$\Longrightarrow$ *var* = `with ( generator ) {` $\mathcal{APT}$[[*instrs*]] `}`
`fold(` *fun* `,` *neutral* `,` *val* `);` $\mathcal{APT}$[[*Rest*]]

Fig. 9. Transformation scheme $\mathcal{APT}$ on WITH-loops.

abstracted out of the generator itself. These applications are then replaced by the original shapes of the arrays they refer to (see Fig. 7). As a consequence, array padding does not alter the generators of WITH-loops in any way. Should padding apply to the result array of a `genarray`-WITH-loop or `modarray`-WITH-loop, the additional dummy elements are automatically initialized according to the default rule of the WITH-loop without any additional measures required.

While the padding transformation of WITH-loops, as outlined in Fig. 9, is simple and elegant on a conceptual level, it unfortunately introduces superfluous and avoidable runtime overhead. Initializing dummy array elements according to the WITH-loop's default rule leads to additional memory accesses that, by definition, do not contribute to the program result. This observation gives way to an additional optimization which distinguishes between dummy and regular array elements in the intermediate representation of WITH-loops. The internal format of multi-generator WITH-loops, as described in [7], provides a suitable framework for this purpose.

6 Performance evaluation

Fig. 10 shows the effect of applying the array padding optimization outlined in Sections 3, 4, and 5 to the PDE1 benchmark. Given the same problem sizes as in the initial investigations described in Section 2 and the upper limit on memory consumption overhead set to 10%, the padding inference heuristic decides to pad 25 out of the total of 33 problem sizes under consideration. In 16 cases, it recommends a padding of `[0,1,0]` (32^3, 96^3, 160^3, 224^3, 272^3, 288^3, 304^3, 336^3, 368^3, 400^3, 416^3, 432^3, 464^3, 480^3, 496^3, 528^3) and in 7 cases a padding of `[0,2,0]` (64^3, 128^3, 192^3, 256^3, 320^3, 384^3, 448^3). For the problem size 352^3 a padding of `[0,22,0]` and for 512^3 a padding of `[0,5,1]` is chosen. Fig. 10 shows the effect of array padding on the simulated cache performance of the PDE1 benchmark. In fact, array padding succeeds in keeping the L1 cache hit rate on a consistently high level between 84% and 88% across all problem sizes. It also manages to avoid the sharp drops in the overall cache hit rate, which can be observed for the problem sizes 256^3 and 512^3 in the original figures.

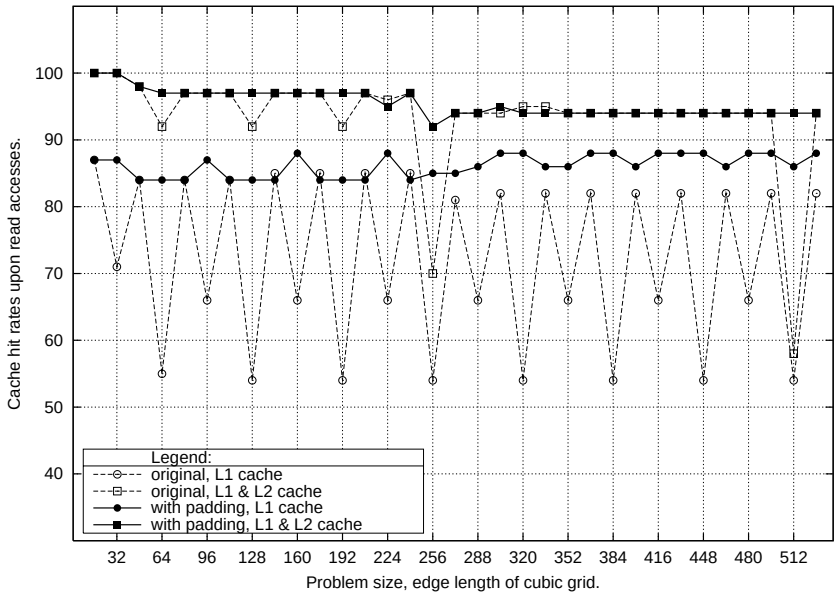

Fig. 10. PDE1: simulated cache performance with and without array padding.

Fig. 11 shows the effect of array padding on the runtime performance of the PDE1 benchmark. First of all, it can be observed that for none of the problem sizes the padding heuristic yields a performance degradation. In contrast, improvements can be observed whenever the padding transformation actually is applied, some of them being quite considerable. In particular, for the problem sizes 64^3, 256^3, and 512^3 the average time needed to re-compute a single grid element can be reduced by 53%, 64%, and 63%, respectively. Also, the variance in runtimes is significantly decreased. With array padding consistent runtimes are achieved over the whole range of problem sizes investigated.

7 Related work

In most functional programming languages, lists rather than arrays are the predominantly used data structure. The most prominent exception is the language SISAL. However, SISAL represents arrays as vectors of vectors rather than as contiguous data, and this storage format renders optimizations like array padding obsolete. So, we are not aware of any similar optimization technique in the area of functional languages.

In high-performance computing based on imperative languages, still predominantly FORTRAN, data locality has long been identified as an important issue [23]. Much research has been focussed on program transformations that reorder the sequence in which single iterations within a nesting of loops are actually

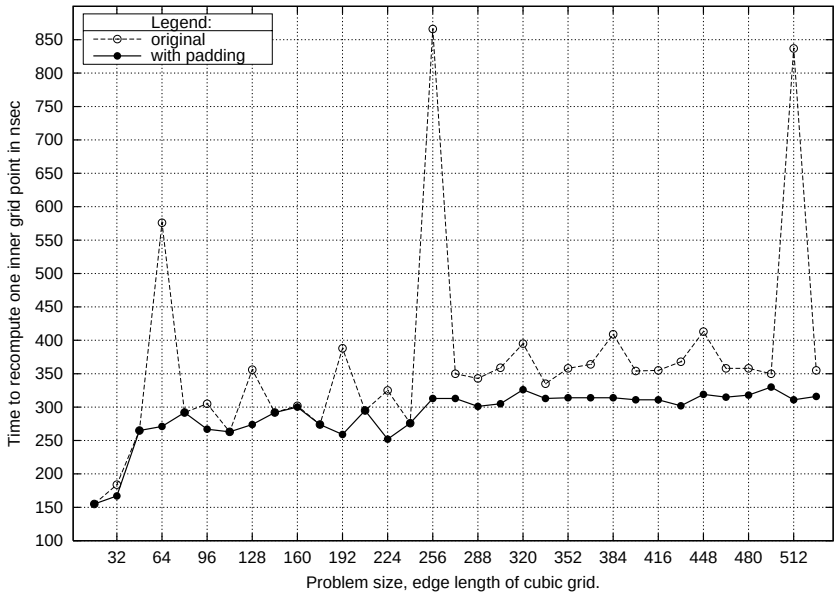

Fig. 11. PDE1: average time required to re-compute a single grid element.

executed [5, 19, 12]. Loop transformations such as permutation, reversal, or interchange, are used to adjust the iteration order to a given array data layout in order to achieve unit stride memory accesses in inner loops and, hence, to exploit spatial locality. Loop tiling, also called loop blocking, is a combination of loop skewing and subsequent loop permutation. It seeks to improve temporal locality in loop nestings by reducing the iteration distance between subsequent accesses to the same array element [10, 4, 18]. Moreover, loop fusion allows to exploit locality of reference across multiple adjacent loop nestings [11].

Often, superior cache performance can be achieved if both the iteration order as well as the memory layout are subject to compiler transformations. Examples are the combination of array transposition with loop permutation [3] or that of array padding with tiling in order to increase tile sizes and, thus, to reduce the additional loop overhead inflicted by tiled code [15]. Whereas these approaches mostly focus on capacity misses, conflict misses due to limited set associativity have been identified as another important source of performance degradation [22]. Their quantification has been achieved by so-called *cache miss equations*, i.e. linear Diophantine equations, that specify the cache line to which an array reference in a loop will be mapped [6]. Due to the complexity and expense of such accurate investigations, simpler heuristics that address both self-interference as well as cross-interference cache conflicts in FORTRAN loop nestings, have been proposed recently [16, 17].

8 Conclusion

This paper presents an algorithm that successfully eliminates spatial and temporal reuse conflicts in SAC programs by implicitly adjusting array data layouts to access patterns and cache configurations. Cache simulation as well as runtime performance investigations on the PDE1 benchmark show that this optimization technique allows for substantial reductions in program runtimes for certain problem sizes and, moreover, achieves a decidedly more consistent runtime performance over a wide range of problem sizes.

References

1. D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, vol. 26(4), pp. 345–420, 1994.
2. B. Bershad, D. Lee, T. Romer, and B. Chen. Avoiding Conflict Misses in Large Direct-Mapped Caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), San José, California, USA*, 1994.
3. M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Design and Implementation (PLDI'95), La Jolla, California, USA*, 1995.
4. S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), La Jolla, California, USA*, pp. 279–290, 1995.
5. D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, vol. 5(5), pp. 587–616, 1988.
6. S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, vol. 21(4), pp. 703–746, 1999.
7. C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, selected papers*, *Lecture Notes in Computer Science*, vol. 1868, pp. 77–94. Springer-Verlag, 2000.
8. C. Grelck and S.-B. Scholz. HPF vs. SAC — A Case Study. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the 6th International Euro-Par Conference on Parallel Processing (Euro-Par'00), Munich, Germany, Lecture Notes in Computer Science*, vol. 1900, pp. 620–624. Springer-Verlag, 2000.
9. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1995.
10. M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Palo Alto, California, USA*, pp. 63–74, 1991.

11. N. Manjikian and T.S. Abdelrahman. Fusion of Loops for Parallelism and Locality. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8(2), pp. 193–209, 1997.
12. K. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, vol. 18(4), pp. 424–453, 1996.
13. K. McKinley and O. Temam. A Quantative Analysis of Loop Nest Locality. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), Boston, Massachusetts, USA*, 1996.
14. T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, Massachusetts, USA*, pp. 62–73, 1992.
15. P.R. Panda, H. Nakamura, N.D. Dutt, and A.Nicolau. A Data Alignment Technique for Improving Cache Performance. In *Proceedings of the International Conference on Computer Design VLSI in Computers and Processors, Austin, Texas, USA*, pp. 587–592. IEEE Computer Society Press, 1997.
16. G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'98), Montréal, Canada, ACM SIGPLAN Notices*, vol. 33(5), pp. 38–49. ACM Press, 1998.
17. G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proceedings of the ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia*. ACM Press, 1998.
18. G. Rivera and C.-W. Tseng. A Comparison of Compiler Tiling Algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands, Lecture Notes in Computer Science*, vol. 1575, pp. 168–182. Springer-Verlag, 1999.
19. V. Sarkar and R. Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), San Francisco, California, USA*, pp. 175–187, 1992.
20. S.-B. Scholz. On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities. In S. Picchi and M. Micocci, editors, *Proceedings of the International Conference on Array Processing Languages (APL'98), Rome, Italy*, pp. 40–45. ACM Press, 1998.
21. S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, selected papers, Lecture Notes in Computer Science*, vol. 1595, pp. 216–228. Springer-Verlag, 1999.
22. O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Nashville, Tennessee, USA*, pp. 261–271. ACM Press, 1994.
23. M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pp. 30–44, 1991.