Array Padding in the Functional Language SAC

Clemens Grelck
Department of Computer Science
University of Kiel
24098 Kiel, Germany

Abstract SAC *is a functional array processing language that tries to combine generic, high-level program specifications with efficient runtime behavior. Being particularly designed for numerical applications, runtime performance critically depends on the effective utilization of the memory hierarchy. For many programs, however, it can be observed that the achieved performance significantly changes with small variations in the problem size.*

Array padding is a well-known optimization technique that adjusts the data layout of arrays in order to make better usage of caches. The paper presents an algorithm that derives a customized data layout from an array access pattern and a cache specification. Cache phenomena such as spatial and temporal reuse are taken into account as well as different cache architectures. The effectiveness is demonstrated by investigations on the runtime performance of the PDE1 benchmark on a shared memory multiprocessor.

Keywords: array padding, spatial locality, temporal locality, compiler optimization, functional programming language, SAC

1 Introduction

SAC is a functional array processing language that tries to combine generic, high-level program specifications with efficient runtime behavior [19, 18]. Implicit compiler support allows to derive multithreaded host machine code that executes in parallel on shared memory multiprocessors [6]. Being designed with numerical applications in mind, the effective utilization of the memory hierarchy plays a key role in achieving good performance [13]. In the context of shared memory architectures, this is particularly important: failure to retrieve data from one of the processor private caches not only results in a slow main memory access — depending on the system architecture it also increases contention on the memory bus with the consequence of further delays on bus-based systems or may cause data to be fetched from a remote processing site via the interconnection network in the case of so-called *scalable* shared memory systems. Moreover, performance degradation due to poor cache utilization grows with increasing numbers of processors involved thus limiting scalability.

It is well-known that small changes in the problem size often have a significant impact on the runtime performance of numerical application programs. We have chosen the well-known benchmark PDE1 as an example in order to investigate and quantify this phenomenon. PDE1 implements red/black successive over-relaxation on 3-dimensional grids. Fig. 1 shows average wallclock execution times per inner grid point using 4 processors on a SUN Ultra Enterprise 4000. The egde length of the cubic grid is uniformly varied from 24 til 280 in steps of 8. Using double precision floating point numbers this involves array sizes between 108KB and 168MB. It can be observed that the time required to compute a single grid element differs from 45nsec for the best performing problem size up to 254nsec for the worst performing one; this means a factor of 5.6. Since the executed code itself remains unchanged, these variations in runtime performance can only be attributed to different degrees of cache utilization.

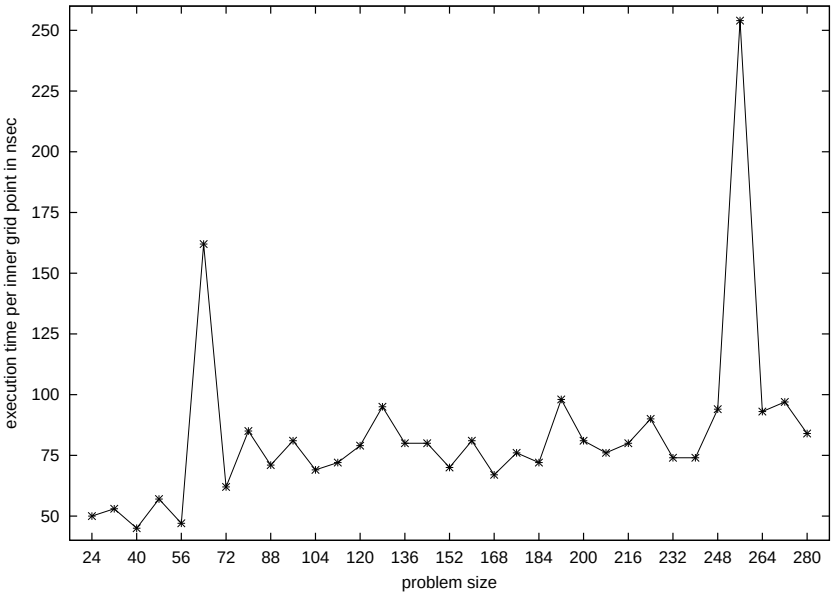

Figure 1: PDE1: average wallclock execution times per grid point for varying problem sizes.

Two hardware characteristics of caches are mainly responsible for the varying performance observed: their organization in cache lines and their limited set associativity [7]. In conjunction, they make caches extremely sensitive to the data layout when regularly structured data is accessed in regular patterns both of which is typical for numerical codes involving large arrays. Many different effects have been identified [20]; for this paper, however, we focus on so-called *self-interference*, i.e. cache conflicts that arise from multiple references to a single array. A *spatial reuse conflict* occurs whenever not all array elements accessed in one loop instance can simultaneously be held in the cache. The number of different array elements that are mapped to the same cache set exceeds the cache's set associativity and, hence, cache lines are flushed from the cache before spatial reuse can be realized. 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 a possible reuse has occurred. Avoiding such conflicts is mandatory in order to achieve good performance [12].

The relative placement of array references in a cache can be influenced by modifying the array data layout. So-called *array padding* adds dummy elements to an array in one or the other inner dimension [1], e.g. an array whose original shape is `[100,100]`, may be transformed into an array of shape `[100,102]` adding two columns of dummy elements. Applying array padding manually, however, is hardly efficient as it requires both a lot of effort and expert knowledge on the programmer's side; it increases program complexity and makes programs less readable and error-prone. Moreover, array padding renders program specifications architecture-dependent since each problem size and cache configuration typically requires a different amount of padding.

However, as a compiler optimization array padding might be well-suited to achieve consistent performance over a wide range of problem sizes and cache architectures. Unfortunately, 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 these completely abstract from a concrete data

layout, they are free to choose the most suitable one with respect to access patterns and hardware characteristics. This includes array padding as an additional optimization technique.

While the technical part of applying array padding is one aspect, the more interesting question is which dimension(s) to pad by how many elements in order to avoid unnecessary cache conflicts for a given array access pattern on a given cache architecture. Section 2 presents a heuristic that successfully eliminates spatial and temporal reuse conflicts. Its effect on runtime performance is demonstrated by means of the PDE1 benchmark in Section 3. Section 4 sketches related work while Section 5 concludes.

2 Padding Inference

This section presents a padding inference algorithm that identifies spatial and temporal reuse conflicts and computes a vector PAD that specifies the recommended padding in each dimension. A cache configuration is specified as follows: let CS denote the cache size, CLS the cache line size, both in array elements, and CA the set associativity. We may then compute the number of cache sets, $NSET := CS/(CLS * CA)$. Moreover, let SHP denote a the original shape of the array under consideration as a vector. The array access pattern is represented as a set of array references. Each array reference R_i is characterized by two vectors, a stride vector SV_i and an offset vector OV_i, i.e. by an affine function on each dimension.

Array references that cannot be expressed in this way, are considered irregular. Since it is rather unlikely that irregular array references systematically generate cache conflicts, they can be ignored. Under the same consideration, the set of array references is partitioned into disjoint so-called *conflict groups*. Each conflict group contains references with identical or similar stride vectors as only these may systematically interfere with each other. Two references R_i and R_j belong to the same conflict group iff

$$ADDR(|SV_i - SV_j|, SHP + PAD) < CLS$$

where $ADDR(vec, shp)$ is a function that computes the offset of vec in the unrolling of an array with shape shp, i.e. $\sum_{k=0}^{n}(vec[k] * \prod_{l=k+1}^{n} shp[l])$. Note that n refers to the dimensionality of the array and that a contiguous, row-major storage order is assumed as the default data layout. The padding inference itself may then be applied separately for each conflict group. Within a conflict group stride vectors can simply be ignored. All references R_i are lexicographically sorted with respect to their offset vectors OV_i and multiple occurrences are eliminated.

First, spatial reuse conflicts are addressed. We start out with $PAD := \vec{0}$, i.e. with no padding. Then, for each reference R_i the offset vector OV_i is converted into a (scalar) offset with respect to the array shape SHP extended by the padding PAD inferred so far:

$$OFFSET_i := ADDR(OV_i, SHP + PAD) - ADDR(OV_0, SHP + PAD)$$

For reasons of simplicity we don't want to deal with negative offsets; since our interest is also limited to relative cache locations, all offsets are shifted by the same constant, i.e. by the offset determined for the first reference in lexicographical order, i.e. R_0. With the shifted offsets at hand, we now determine the corresponding cache sets

$$SET_i := (OFFSET_i/CLS) \bmod \text{NSET}.$$

For each reference R_i we compute the number $NPSCFL_i$ of potential spatial reuse conflicts with other references that are mapped to the same cache set. Two references R_i and R_j potentially conflict iff

$$(|OFFSET_i - OFFSET_j| \leq NSET * CLS) \wedge ((|SET_i - SET_j| < 2 \vee (|SET_i - SET_j| > NSET - 2))$$

In order to classify two references as non-conflicting at least one unused cache set in between is required. This additional buffer is required since we completely abstract from relative placements of references within a cache line. In a direct-mapped cache, each potential

conflict will actually result in a cache conflict at runtime. 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 is defined as

$NSCFL_i :=$
$\max(0, NPSCFL_i - CA + 1)$.

For each conflict the question is whether padding might or might not alter the situation and which dimension of the array should be padded by how many elements. Therefore, we determine $PADDIM := d + 1$ where d is the outermost dimension with $OV_i[d] \neq OV_j[d]$ for any pair of conflicting array references R_i and R_j. In other words, we select the outermost dimension where padding may solve any conflict. Eventually, the pading vector PAD is incremented by 1 in dimension $PADDIM$ and the cache behavior is re-evaluated. The entire process is repeated until either all actual spatial reuse conflicts have been solved or an upper limit for padding has been reached. The latter is needed in order to keep the extra amount of memory required for the representation of a padded array within acceptable bounds even for pathological cases.

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

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

Note here that all references are sorted with increasing offsets. For each pair of neighboring references R_i and R_{i+1} that might benefit from temporal reuse, we now compute the number of potential temporal reuse conflicts $NPTCFL$. 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 of R_i and R_{i+1}, i.e. $(SET_i < SET_j) \ \wedge \ (SET_j < SET_{i+1})$.

In analogy to spatial reuse conflicts, the term "potential" is to be understood with respect to set associativity, i.e.

$NTCFL_i :=$
$\max(0, NPTCFL_i - CA + 1)$.

Whenever potential conflicts turn into real conflicts due to limited set associativity, it must again be determined whether array padding might help and which array dimension should be used for padding. The basic idea is to select a padding dimension that is sufficiently small so that the relative positions of neighboring references with potential temporal reuse remain untouched. However, it must be sufficiently large so that padding alters the relative positions between these and the corresponding conflicting reference. Such a dimension may or may not exist. In the latter case, the temporal reuse conflict cannot be solved. In the former case, however, the padding vector PAD is incremented by 1 in the smallest suitable dimension and the temporal reuse is iteratively re-evaluated until all conflicts have been solved or have been found unsolvable through array padding or the prespecified upper limit for padding amounts has been reached.

3 Evaluation

Each processor of the SUN Ultra Enterprise 4000 used in the performance measurements in Section 1 is equipped with a 16KB L1 data cache and a 1MB L2 cache. Both are direct-mapped and use cache lines of 32 and 64 bytes, respectively. The array padding inference algorithm outlined in Section 2 is applied to the PDE1 benchmark with respect to the L1 data cache. From an overall number of 31 problem sizes investigated, the inference algorithm chooses to pad 4 by a padding vector of [0,2,0], 9 by [0,1,0], and 19 not to pad at all. For 5 problem sizes, the decision for padding is a result of analysing spatial as well as temporal reuse conflicts; for all others the final padding vector has already been determined after handling spatial reuse conflicts alone. Subsequent application of the inference algorithm with respect to the L2 cache characteristics does not yield additional padding requirements.

Fig. 2 shows the effect of array padding on

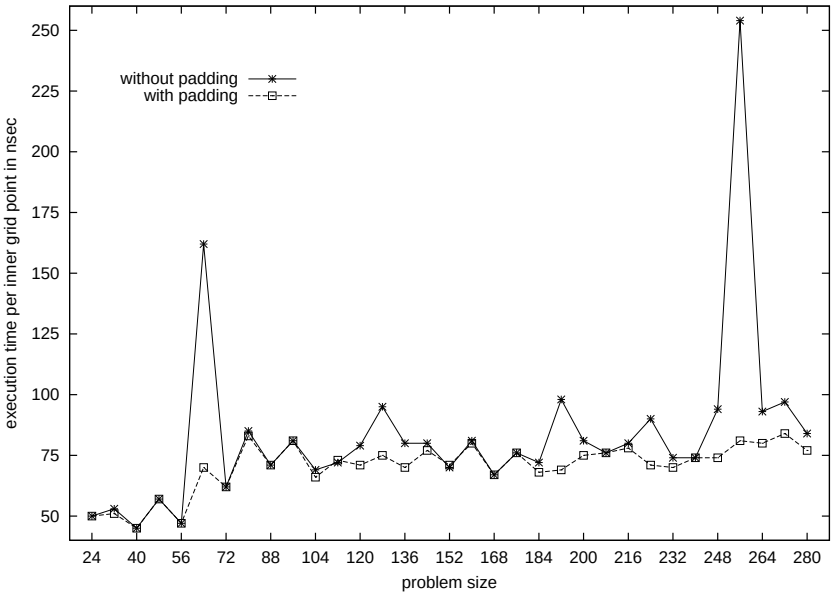

Figure 2: PDE1: average wallclock execution times per grid point for varying problem sizes with and without implicit application of array padding.

the runtime performance of the PDE1 benchmark; wallclock execution times per grid element achieved with and without array padding can directly be compared. Significant improvements in runtime performance can be observed whenever array padding actually is applied. In particular, for the problem sizes 64^3 and 256^3 average program runtimes per grid element can be reduced by 56% from 162nsec to 70nsec and by 68% from 254nsec to 81nsec, respectively. Furthermore, the variance in runtimes is decreased to a factor of 1.8 between the best and the worst performing problem size. When compared to the original factor of 5.6 without padding enabled, program runtimes are much more predictable.

4 Related Work

In high-level functional programming languages, lists rather than arrays form the predominant data structure. The most prominent exception is the language SISAL [10]. However, SISAL represents arrays as vectors of vectors rather than as contiguous data which renders array padding useless. So, we are not aware of any similar optimization technique in this area.

However, in high-performance computing, mostly based on FORTRAN, data locality has long been identified as an important issue [21]. Much research has been focussed on program transformations that reorder the sequence of computations in loop nestings [4, 17, 11]. 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 skewing and subsequent permutation. It seeks to improve temporal locality in loop nestings by reducing the iteration distance between subsequent accesses to the same array element [8, 3]. Moreover, loop fusion allows to exploit locality of reference across single loop nestings [9].

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 [2] or that of array padding with tiling in order to increase tile sizes and thus reduce the

additional overhead inflicted by tiled code [14]. While these approaches mostly focus on capacity misses, conflict misses due to limited set associativity have been identified as another important source of performance degradation [20]. Their quantification has been achieved by counting the number of integer solutions to 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 [5]. Due to the complexity and expense of such accurate investigations, simpler heuristics have been proposed [15, 16]. The padding inference algorithm presented in Section 2 extends this work with respect to self-interference in several aspects. It does not assume a direct-mapped cache but explicitly supports set-associative caches. In addition to spatial reuse conflicts, temporal reuse conflicts are also taken into account. Furthermore, padding can be performed in any dimension and even in multiple dimensions; an additional algorithm selects the most appropriate padding dimension for each conflict to be solved.

5 Conclusion

This paper presents an algorithm that successfully eliminates spatial and temporal reuse conflicts in SAC programs by implicitly applying array padding where necessary. Runtime performance investigations on the PDE1 benchmark show that this optimization technique allows to substantially reduce program runtimes for various problem sizes and, moreover, achieves much more consistent runtime performance over a large 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*, 26(4):345–420, 1994.

[2] 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.

[3] 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*, pages 279–290, 1995.

[4] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, 1988.

[5] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the ACM International Conference on Supercomputing (ICS'97), Vienna, Austria*, 1997.

[6] C. Grelck. Shared Memory Multiprocessor Support for SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 38–54. Springer-Verlag, 1999. ISBN 3-540-66229-4.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann Publishers, 1995. ISBN 1-55860-329-8.

[8] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California*, pages 63–74, 1991.

[9] N. Manjikian and T.S. Abdelrahman. Fusion of Loops for Parallelism and Locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[10] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.

[11] K. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[12] 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.

[13] 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*, pages 62–73, 1992.

[14] 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*, pages 587–592. IEEE Computer Society Press, 1997.

[15] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. Conference on Programming Language Design and Implementation (PLDI'98), Montréal, Canada*, pages 38–49. ACM SIGPLAN Notices, 33(5), 1998.

[16] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia*, 1998.

[17] 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*, pages 175–187, 1992.

[18] 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 Array Processing Language Conference (APL'98), Rome, Italy*, pages 40–45. ACM Press, 1998.

[19] S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Mgrid Benchmark in SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 216–228. Springer-Verlag, 1999. ISBN 3-540-66229-4.

[20] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, Tennessee*, pages 261–271. ACM Press, 1994.

[21] 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)*, pages 30–44, 1991.