

An Efficient Semi-Hierarchical Array Layout *

N.P. Drakenberg (npd@it.kth.se) and F. Lundevall (f@it.kth.se)
Dept. of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, SWEDEN

B. Lisper (bjorn.lisper@mdh.se)
Dept. Computer Engineering, Mälardalen University, P.O.B. 883, S-721 23 Västerås, SWEDEN

Abstract. For high-level programming languages, linear array layout (*e.g.*, column major and row major orders) have *de facto* been the sole form of mapping array elements to memory. The increasingly deep and complex memory hierarchies present in current computer systems expose several deficiencies of linear array layouts. One such deficiency is that linear array layouts strongly favor locality in one index dimension of multidimensional arrays. Secondly, the exact mapping of array elements to cache locations depend on the array's size, which effectively renders linear array layouts non-analyzable with respect to cache behavior. We present and evaluate an alternative, semi-hierarchical, array layout which differs from linear array layouts by being neutral with respect to locality in different index dimensions and by enabling accurate and precise analysis of cache behaviors at compile-time. Simulation results indicate that the proposed layout may exhibit vastly improved TLB behavior, leading to clearly measurable improvements in execution time, despite a lack of suitable hardware support for address computations. Cache behavior is formalized in terms of conflict vectors, and it is shown how to compute such conflict vectors at compile-time.

1. Introduction

Present state-of-the-art compilers do not consistently deliver the performance reasonably expected by users. One of the most fundamental reasons for this is that existing high-level languages invariably induce a perception of memory as being flat, whereas actual computer architectures are being equipped with increasingly deep memory hierarchies to overcome the widening performance gap between processors and main memories.

Devising locality enhanced algorithms, suitable for hierarchical algorithms, is a creative process just as algorithm development in general, and it is not in general reasonable to expect compilers to automatically derive such algorithms from their non locality-enhanced counterparts.¹ Long-term success in the increasingly important problem of *consistently* obtaining high performance from hierarchical memories will most likely require performance models and languages where locality is somehow exposed. Ideally, this would enable a distinct division of responsibility between compilers and their users, where compilers perform all *architecture specific* tuning, and algorithmic aspects (including algorithmic locality) are managed by humans. For such a division of responsibilities to be meaningful, both parties must be given the means to perform their designated tasks. For a compiler this means, among other things, that logical locality should be reliably translated into effective lo-

cality. Consider the following fragment of Fortran 90 code:

```
function mmMpyAdd( A, B, C )
  real, dimension(:, :, intent(in)) :: A, B
  real, dimension(:, :, intent(in, out)) :: C

  integer L, M, N
  integer i, j, k

  M = size( C, 1 )
  N = size( C, 2 )
  L = size( A, 2 )
  do i = 1, M
    do j = 1, N
      do k = 1, L
        C(i, j) = C(i, j) + A(i, k) * B(k, j)
      end do
    end do
  end do
end function mmMpyAdd
```

In common with most other programming languages, the arrays passed to `mmMpyAdd` as `A`, `B` and `C` are likely to have been dynamically allocated with sizes depending on values input to the program, or being statically non-determinable for other reasons. Throughout this paper we assume that array sizes, for the most part, cannot be determined at compile time. When using linear array layouts both the direction and distance in the iteration space, between iterations which experience cache interference, depend on the sizes of the arrays involved.

In this paper we investigate an alternative, semi-hierarchical, array layout (called HAT), which addresses the issues raised above by being effectively dimension-neutral with respect to access distance in memory, and by mapping array elements to memory locations such that the possible cache-location of each element does not depend on the size of the corresponding array.

* This work was supported in part by the Swedish Research Council for Engineering Sciences (TFR, grant 97-722).

¹ Loop tiling and similar program transformations are successfully employed by many compilers, but several well known blocked forms of matrix algorithms can only be derived by exploiting algebraic properties of matrices [1].



The remainder of the paper is structured as follows: First, the details of our non-linear array layout are presented in Section 2 along with simulation statistics and measured performance results for a small set of computational kernels. Section 3 we show that when using the HAT layout for programs with regular control flow and data references (*i.e.*, nested loops with affine array index expressions), cache conflicts form repeating compact patterns which tessellate the iteration space, and whose exact contents is computable at compile-time. In Section 4 we present related work. Finally, in Section 5, we discuss our results, draw conclusions, and suggest future work.

2. Array Layout

Any implementation of a programming language which provides multi-dimensional arrays, must also decide upon ways in which to map multi-dimensional index-spaces to the linear address spaces used for memory. Such mappings of index spaces to address spaces may be formally specified as *layout* functions L [4], which should be interpreted such that L applied to tuples of array indices (i_1, \dots, i_m) yields the memory location of the corresponding array element relative to the starting memory location of the array.

2.1. LINEAR ARRAY LAYOUTS

The layout functions L_{linear} corresponding to linear array layouts may be expressed as

$$L_{\text{linear}}(i_1, \dots, i_m) = S_E \cdot \sum_{k=1}^m c_k i_k, \quad (1)$$

where S_E denotes the size of array elements. The ubiquitous row major and column major orderings are both linear array layouts which simply correspond to different choices of c_1, \dots, c_m . The most significant benefit of row major and column major layouts is due to their linearity (*i.e.*, $L(\alpha \mathbf{i} + \beta \mathbf{j}) = \alpha L(\mathbf{i}) + \beta L(\mathbf{j})$), as it enables *incremental* computation of memory addresses for sequences of index tuples. For the very common case of index tuple sequences with a constant difference between successive elements, incremental computation of addresses can be made particularly efficient, and is indeed performed by all modern optimizing compilers.

Sadly, the efficient address computations of linear array layouts are offset by their inclination to interact poorly with hierarchical memory systems when arrays

are large. Linear layout functions inevitably map array elements which are adjacent along some direction of the index-space to *consecutive* memory locations, whereas neighboring elements along remaining orthogonal directions tend to occupy widely separated memory locations. Separation of logically adjacent locations makes it unnecessarily difficult for compilers to transform logical locality into effective locality, and in combination with the different behavioral characteristics of caches and TLBs [16] and the influence of array sizes on the mapping of array elements to cache-locations, the task becomes near impossible.

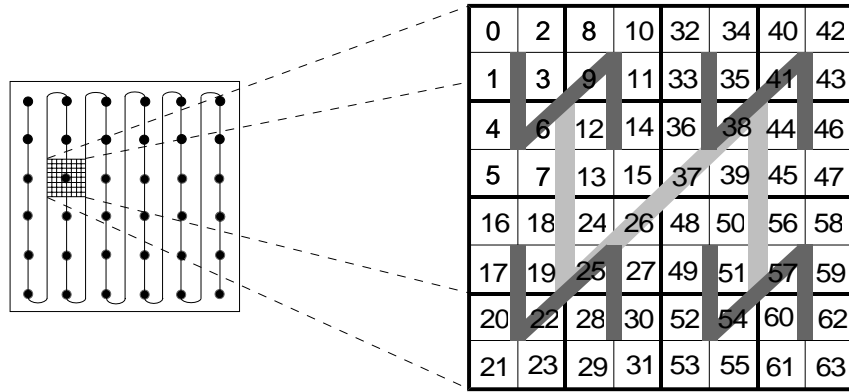


Figure 1. The semi-hierarchical array layout (HAT) applied to a two-dimensional array, and using 8-by-8 top-level tiles.

2.2. HIERARCHICAL ARRAY LAYOUTS

An m -dimensional array which is mapped to memory using a *hierarchical layout* can be seen as being recursively constructed from 2^m equally shaped subarrays. Different rules for the relative ordering of the constituent subarrays lead to globally distinct hierarchical array layouts, which are known by names such as C-order, U-order, Hilbert order and Z or Morton [17] order (an example of transposed Morton, or Z^T -layout is provided by the right hand-side of Figure 1). Hierarchical array layouts have been developed and used for various special purposes, such as in computational subroutine “libraries” [5, 4], load balancing of parallel computations [13, 18], and in image processing [9, 28].

Despite a non negligible volume of results on hierarchical storage layouts, such results have typically not become widely known. Several authors seem to have reinvented such storage layouts plus associated concepts and results themselves [D.S. Wise, personal communication], only to subsequently find them scattered among an unusually wide assortment of scientific publications [24, 17, 21]. Our motive for reinventing the Morton order was to enable accurate and precise compile-time analysis of cache behavior and to simplify simultaneous locality enhancement with respect to complete memory hierarchies (*e.g.*, both cache and TLB).

A downside of pure hierarchical array layouts based on globally constant subarray shapes, is that they potentially waste huge amount of address-space for arrays whose shape deviates from the subarray defining its layout.² To avoid this we have chosen to combine a

² In [26], Wise has recently pointed out that only address space, not actual storage, is being wasted. His conclusion that such waste

linear layout (column major order) with a hierarchical layout (transposed Morton order), by using the Morton order for subarrays up to one or several TLB pages in size, and let these subarrays in turn, be ordered according to the linear layout. The resulting *semi-hierarchical* array layout (illustrated in Figure 1) is called HAT (for “Hierarchical Array Tiling”), to emphasize its intended coexistence with locality enhancement and compiler optimizations.

2.2.1. Address Arithmetic

The computation of addresses from indices does look like a potential source of inefficiency for hierarchical array layouts. Fortunately, for a Morton (and transposed Morton) ordering based on $2 \times \dots \times 2$ subarrays (as used for HAT), the addresses of array elements are computable from indices through simple bit operations, leading to a definition of the layout function for transposed Morton order as:

$$L_{\text{Morton}}(i_1, \dots, i_m) = S_E(i_m \mathbb{M}^{m-1} (\dots (i_2 \mathbb{M}^1 i_1) \dots)) \quad (2)$$

where \mathbb{M}^k is an operator³ such that $r = (a \mathbb{M}^k b)$ is the interleaving of groups of k bits from b with single bits from a , as shown in Figure 2 for $k = 2$. From (2) it is easily seen that the mapping of array elements to memory is *independent of an array’s size* and that address computations may be performed through simple bit-operations, but current processor architectures rarely include \mathbb{M} -operations in their instruction sets, nor are these operations easily synthesized as short sequences

is harmless overall will, in our opinion, require more experimental evidence than given in [26].

³ The visual appearance of \mathbb{M} is intended to suggest the interleaving of bits from two sources.

of common instructions, and as a consequence direct evaluation of L_{Morton} will be rather expensive. Interestingly, this is not vastly different from linear layouts, for which the integer multiplications in L_{linear} makes its direct evaluation costly. Surprisingly, incremental updating of addresses can be efficiently done also for arrays using Morton order as is demonstrated by the following example:

EXAMPLE 1. Consider the index tuple $(3, 5)$ of a two dimensional array of single precision floats, whence

$$\begin{aligned} L_{\text{Morton}}(3, 5) &= 4 \cdot (3 \ll^1 5) \\ &= 4 \cdot (0000101 \ll^1 00000011) \\ &= 10011100 = 156, \end{aligned}$$

which may be decomposed into components corresponding to each index as:

$$10011100 = 00010100 \vee 10001000,$$

where \vee denotes the bitwise or of its two operands. Now, given the pair of values or:ed to form $L_{\text{Morton}}(3, 5)$, we may compute $L_{\text{Morton}}(3 + 3, 5)$ as follows:

$$\begin{array}{r} 00010100 \\ + 10101000 \\ \hline 10111100 \\ + 00010100 \\ \hline 11010000 \\ \wedge 01010100 \\ \hline 01010000 \quad \vee \quad 10001000 \\ = 11011000 = 216, \end{array}$$

Formal justification of the operations just performed is provided by the algebra of *dilated* integers [24, 21, 26] (p. 53–55, p. 222–226, and p. 779–781, respectively). For the convenience of our readers we now briefly summarize its most salient features from the references just cited.

DEFINITION 1. Let s be an integer whose binary representation is $s = s_r s_{r-1} \dots s_1 s_0$, $s_i \in \{0, 1\}$, and using two's complement representation for negative integers. The integer

$$s_{\ll^k} = s_r \underbrace{0 \dots 0}_{k \text{ bits}} s_{r-1} \underbrace{0 \dots 0}_{k \text{ bits}} \dots s_1 \underbrace{0 \dots 0}_{k \text{ bits}} s_0$$

is called a k -dilated version of s , or simply a dilated version of s for $k = 1$.

DEFINITION 2. The transposed Morton-order offset of array element (i, j) is given by $i_{\ll^1} \vee j_{\ll^1} \ll 1$, where $x \ll y$ denotes the left-shifting of x by y . Similarly, for an index tuple (i, j, k) the corresponding offset is given by $i_{\ll^2} \vee (j_{\ll^2} \ll 1) \vee (k_{\ll^2} \ll 2)$, and so on for higher dimensional arrays.

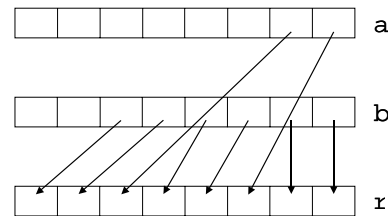


Figure 2. A \ll^2 operation, generating an 8-bit result.

THEOREM 1. Let m and n be two integers and let m_{\ll^k} and n_{\ll^k} be their k -dilations, for some k . Then, if m and n are both nonnegative or both negative,

$$\begin{aligned} m = n &\quad \text{iff} \quad m_{\ll^k} = n_{\ll^k}, \\ m > n &\quad \text{iff} \quad m_{\ll^k} > n_{\ll^k}. \end{aligned}$$

THEOREM 2. Let \oplus_k and \ominus_k denote the k -dilated addition and subtraction operators, such that $(i_{\ll^k} \oplus_k j_{\ll^k}) = (i + j)_{\ll^k}$ and $(i_{\ll^k} \ominus_k j_{\ll^k}) = (i - j)_{\ll^k}$, respectively. Then

$$i_{\ll^k} \oplus_k j_{\ll^k} = (i_{\ll^k} + f_{\ll^k} + j_{\ll^k}) \wedge m_{\ll^k}, \quad (3)$$

$$i_{\ll^k} \ominus_k j_{\ll^k} = (i_{\ll^k} - j_{\ll^k}) \wedge m_{\ll^k}, \quad (4)$$

where m_{\ll^k} is the k -dilated form of -1 or $111 \dots 111$, and f_{\ll^k} is the bitwise complement of m_{\ll^k} .

Note that when either of i_{\ll^k} or j_{\ll^k} are constants a costly run-time k -dilation may instead be done at compile-time, and in addition, either $i_{\ll^k} + f_{\ll^k}$ or $f_{\ll^k} + j_{\ll^k}$ may be evaluated at compile-time which further reduces the operation count of address computations.

Returning now to the HAT-layout, we see that the layout function of HAT may be written as

$$\begin{aligned} L_{\text{HAT}}(i_1, \dots, i_m) &= \\ &S_M \cdot L_{\text{linear}}(i_1 \text{ div } T_1, \dots, i_m \text{ div } T_m) + \\ &S_E \cdot L_{\text{Morton}}(i_1 \text{ mod } T_1, \dots, i_m \text{ mod } T_m), \end{aligned}$$

where S_M is the size of the largest Morton ordered subarray (e.g., one TLB-page) and T_1, \dots, T_m are the sizes along each axis of the array of the largest Morton ordered subarray. S_E denotes the size of individual array elements. The algebra of dilated integers may be used to incrementally update addresses within the Morton ordered subarrays of HAT.

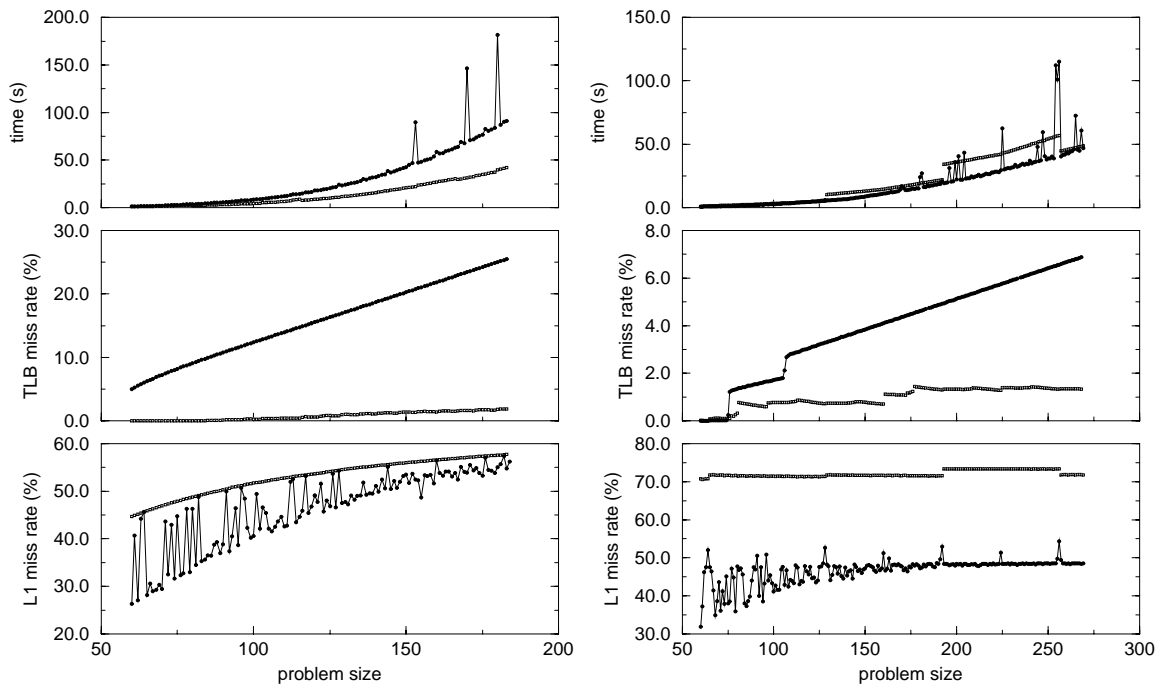


Figure 3. Execution time plus cache and TLB miss rates for the NAS GMTRY kernel (left) and the NAS VPENTA kernel (right), vs problem size. Filled dots represent column major results, and hollow squares represent HAT-layout results.

2.3. EXPERIMENTAL EVALUATION

To evaluate the performance and behavior of the HAT layout, we have rewritten a small set of Fortran kernels to take problem size as a command line argument and to use arrays of corresponding sizes. Compile-time switches are used to select either column major array layout or HAT-layout, where the latter forms have been explicitly devised to incrementally update addresses of array elements by using dilated integers. No other changes have been made to the programs which means that the reported results correspond to the performance of the HAT-layout for row and column traversals of the arrays (*i.e.* non-tiled code). All programs were compiled using the Sun Workshop 5.0 Fortran compiler using near maximum optimization⁴ and were run on Sun Ultra 10 workstations equipped with 333 MHz UltraSPARC-III microprocessors, 2Mb of unified 2nd level cache, 640Mb of main memory, which run the Solaris 2.6 operating system. Cache and TLB statistics were obtained using SpixTools/Shade [6] running identical binaries as used for timing measurements, configured to simulate 16Kb direct mapped data and instruction caches and 60-entry fully associative data and instruction TLBs. Reported L1 miss rate statistics were gathered under the assumptions of a perfect data TLB (no misses).

⁴ -xtarget=ultra2 -xarch=v8plusa -xO5 -depend.

The benchmark codes for which results are shown in Figure 3 (GMTRY and VPENTA from the NAS kernel benchmark) were chosen for not exclusively traversing arrays along the most favorable index dimension. These codes were however *not* designed for the HAT-layout. Upon inspection of the diagrams in Figure 3, it is immediately visible that the column major layout suffers from very high TLB miss rates whereas the HAT-layout suffers from rather high L1 miss rates. The TLB miss rates experienced by the column major layout is an immediate effect of consecutive accesses not being along the favored index dimension. The HAT-layout is inherently *more* likely to experience high cache miss rates for regular array references, than are linear layouts. However, as we show in the next section, the HAT-layout permits accurate and precise compile-time analysis of its cache behavior, which in turn enables compile-time elimination or reduction of poor cache behavior.

Looking at the execution time results for HAT and column major results we see that the column major layout clearly suffers from its high TLB miss rate also that relatively small variations in L1 miss rates may result in substantially increased execution times (some peaks in execution do not correspond to L1 or TLB miss-rate peaks).

For the HAT-layout, execution time forms a substantially smoother curve than for the column major layout,

which by itself might be taken as an indication of easier optimization problems, and distinct bump in execution time for the VPENTA kernel corresponds directly to an increase in L1 miss rates.

When comparing execution times, it must also be remembered that for the present experiments, the HAT layout is penalized to a varying degree by the lack of compiler support. The compiler we have used, is completely ignorant of the HAT-layout and therefore does not fully detect common subexpressions or do strength reduction of address computations.

3. Cache Interference & Conflict Vectors

3.1. CACHE INTERFERENCE

The results of this section rely on expressions used in array references being affine functions of loop-control variables from enclosing loops. For such index expressions the mapping of loop-control variables to array indices implied by a reference such as “ $A(l + 1, l + j + 2)$ ” may be expressed using matrix notation:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} l \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \quad (5)$$

Henceforth, such matrices of coefficients, originating from array references will be called *access matrices* and similarly constant vectors such as $[1 \ 2]^T$ in (5) are called *offset vectors*.

3.2. CONFLICT VECTORS

A pair of array references⁵ R_A and R_B will access memory locations which map to the same index in cache, and thereby possibly cause cache interference, only if:

$$\mathcal{M}(\langle R_A, \mathbf{i} \rangle) \equiv \mathcal{M}(\langle R_B, \mathbf{j} \rangle) \pmod{C_S}, \quad (6)$$

where C_S denotes the size of a cache-set, and where $\mathcal{M}(\cdot)$ denotes a function which maps memory locations to memory-lines⁶ (i.e., $\mathcal{M}(a) = a - (a \bmod L_S)$, where L_S is the cache line-size), and $\langle \cdot, \cdot \rangle$ denotes the address of a memory access implied by a specific combination of array reference (e.g., R_A) and iteration vector. For a direct-mapped cache, the index uniquely determines

⁵ Using the terminology of Ghosh *et al.* [10, 11], we refer to a static read or write in a program as a *reference*, whereas a particular execution of that read or write at runtime is a *memory access*.

⁶ A *memory line* refers to a cache line sized-and-aligned block in memory, while a *cache line* refers to the actual block in cache to which a memory line is mapped.

the possible location of each datum in cache which in turn implies that the condition in (6) is both necessary and sufficient. For set-associative caches, on the other hand, the index of a datum does not uniquely determine its possible locations in cache, and thus (6) is then reduced to only being a necessary condition. That is, for set-associative caches, cache interference also depends on the order of memory accesses and in such cases (6) is merely a conservative estimate, albeit one with fairly high precision.

One of the main advantages of the HAT-layout is that it allows the solutions to (6) to be determined and enumerated at *compile-time*, which in turn means that potential cache interference can be detected and accurately quantified during compilation. To capture and characterize this aspect of HAT-layout behavior, we introduce the notion of *conflict vectors*, which is formalized by the following definition:

DEFINITION 3. For a pair of (not necessarily distinct) array references R_A and R_B , a conflict vector (denoted by ξ^7) is said to exist for each pair of (by statement) iteration vectors \mathbf{i} and \mathbf{j} which satisfy equation (6) above, and its value is then defined by $\xi = \mathbf{j} - \mathbf{i}$, where the smaller vector of \mathbf{i} and \mathbf{j} is extended by zeros to the size of the larger. In cases when the values of ξ may have a dependence on an iteration vector, say \mathbf{i} , we write this as $\xi(\mathbf{i})$.

For comparison, *reuse vectors* [15, 10, 11] indicate the direction(s) in the iteration space along which one or several array references will access the *same* array element, rather than an array element that is potentially conflicting in cache.

3.3. COMPUTING CONFLICT VECTORS

The mapping of array indices to memory locations, as well as efficient address computations, for arrays that use the HAT-layout have been described in the previous section. However, for the purpose of computing conflict vectors it is more convenient to work with the reverse mapping, from storage locations to array indices. The mapping from storage locations, \mathbf{s} , to array indices, \mathbf{i} , implied by the Morton order for an m -dimensional array may be expressed as the matrix-vector product:

$$\mathbf{i} = [\mathbf{I}_m \ 2\mathbf{I}_m \ 4\mathbf{I}_m \ 8\mathbf{I}_m \ \dots] \mathbf{s}, \quad (7)$$

where \mathbf{I}_m is the identity matrix of order m , and where the vector \mathbf{s} is the *binary* encoding of the storage location. As an example, consider the case of a two-dimensional

⁷ The greek symbol ξ was chosen because of the acronym-like correspondence: $\xi = \text{xi} \approx \text{abbrev. cross interference}$.

array; the mapping of storage locations to array indices implied by (7) is then given by

$$\mathbf{i} = \begin{bmatrix} 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 & \cdots \\ 0 & 1 & 0 & 2 & 0 & 4 & 0 & 8 & \cdots \end{bmatrix} \mathbf{s},$$

and it is easily seen how the different components of \mathbf{i} are formed from even and odd components (bits) of the storage location. Naturally, mappings of storage locations to array indices have a fundamental role in the computation of conflict vectors, and for this reason, \mathbf{H}_m^c where 2^c is the size of Morton ordered subarrays, is consistently used to denote the matrix $[\mathbf{I}_m \ 2\mathbf{I}_m \ 4\mathbf{I}_m \ \cdots \ 2^{c-1}\mathbf{I}_m]$ throughout the rest of this text.

In section 2 it is shown how, after reaching a predefined and carefully chosen size-limit, the HAT-layout changes from using Morton order to using column major order. As a consequence, the mapping of storage locations to array indices given in (7) is incomplete and needs to be augmented with a linear term corresponding to the transition to a linear layout. Doing so, and simultaneously replacing the left hand-side of equation (7) with the mapping from iteration space to index space implied by some reference, say R_B , yields an expression for the correspondence between iteration space coordinates and storage locations for that array reference:

$$\mathbf{B}\mathbf{j} + \mathbf{b} = \mathbf{H}_{d_B}^c \mathbf{s} + \mathbf{L}_{d_B}^c \mathbf{q}. \quad (8)$$

In (8), \mathbf{j} is an iteration vector, \mathbf{B} and \mathbf{b} are the access matrix and offset vector respectively of reference R_B and d_B denotes the dimensionality of array accessed through reference R_B . The meaning of $\mathbf{H}_{d_B}^c$ is given above and $\mathbf{L}_{d_B}^c$ is a diagonal matrix with elements identical to the sizes of Morton ordered subarrays along corresponding dimensions. As an example, given Morton ordered regions of 8192 bytes ($c = 10$), and an array reference $B(l+1, l+J+2)$ where B is a two-dimensional array of eight-byte elements, the corresponding instance of (8) is:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \mathbf{j} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 & 16 & 0 \\ 0 & 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 & 16 \end{bmatrix} \mathbf{s} + \begin{bmatrix} 32 & 0 \\ 0 & 32 \end{bmatrix} \mathbf{q},$$

where $\mathbf{j} \in \mathbb{Z}^2$, $\mathbf{s} \in \{0, 1\}^{10}$, and $\mathbf{q} \in \mathbb{Z}^2$.

As indicated in section 2 it is wise to chose transition points between linear and non-linear layouts such that an integer multiple (≥ 1) of Morton ordered regions fit and are aligned in each cache-set. The usefulness of such a choice is due to the modulo- C_S indexing of typical caches, which causes the values of \mathbf{q} -vectors in (8) to become largely unrelated to the cache behavior of reference R_B . The binary vector \mathbf{s} in (8) indicates

the offset of an array element in some Morton-ordered region. Since the size and alignment of Morton ordered regions is tailored to the cache parameters, and assuming temporarily that all array elements and cache lines are of equal size (how to remove this restriction is discussed on pages 14 and 16), the values of \mathbf{s} for any pair of conflicting accesses must be equal. Thereafter, and if keeping equation (8) in memory, it is easily realized that any conflict vector $\boldsymbol{\xi}(\mathbf{j})$ of reference R_B w.r.t. R_A must satisfy

$$\begin{aligned} \mathbf{A}(\mathbf{j} + \boldsymbol{\xi}) + \mathbf{a} &= \mathbf{H}_{d_A}^c \mathbf{s} + \mathbf{L}_{d_A}^c \mathbf{p} \\ \mathbf{B}\mathbf{j} + \mathbf{b} &= \mathbf{H}_{d_B}^c \mathbf{s} + \mathbf{L}_{d_B}^c \mathbf{q}, \end{aligned} \quad (9)$$

where $\mathbf{s} \in \{0, 1\}^c$, $\mathbf{p} \in \mathbb{Z}^{d_A}$, $\mathbf{q} \in \mathbb{Z}^{d_B}$, and where \mathbf{A} , \mathbf{B} and \mathbf{a} , \mathbf{b} are the access matrices and offset vectors of references R_A and R_B respectively.

Cache interference phenomena are conventionally categorized as being due either to *self-interference* or *cross-interference* [15]. Self-interference represents the case when it is memory accesses of the same reference that interfere in cache, and cross-interference represents all other cases of interference. To simplify the presentation, and to remain “compatible” with existing literature, the computation of conflict vectors is described separately for self-interference and for cross-interference.

3.4. SELF INTERFERENCE

When references R_A and R_B are not distinct, we have $\mathbf{A} = \mathbf{B}$ and $\mathbf{a} = \mathbf{b}$, and of course $d_A = d_B$ in equation (9), which may then (because of $\mathbf{H}_{d_A}^c = \mathbf{H}_{d_B}^c$) be rewritten as:

$$\mathbf{B}\boldsymbol{\xi} + \mathbf{B}\mathbf{j} + \mathbf{b} - \mathbf{L}_{d_B}^c \mathbf{p} = \mathbf{B}\mathbf{j} + \mathbf{b} - \mathbf{L}_{d_B}^c \mathbf{q},$$

which in turn is easily reduced to

$$\mathbf{B}\boldsymbol{\xi} - \mathbf{L}_{d_B}^c \mathbf{r} = \mathbf{0}, \quad (10)$$

where $\mathbf{r} = (\mathbf{p} - \mathbf{q}) \in \mathbb{Z}^{d_B}$. Equation (10) is a system of linear Diophantine equations, and as such it may be solved by any one of the existing methods for solving such systems of equations, see for example, [22] (p. 52–59) or [27] (p. 106–117). By the theorem below, the set of integer solutions to (10) can be represented by a set of linearly independent integer vectors:

THEOREM 3. *For any matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and vector $\mathbf{b} \in \mathbb{Q}^m$ such that $\mathbf{A}\mathbf{x}_0 = \mathbf{b}$ for $\mathbf{x}_0 \in \mathbb{Z}^n$, a set of integral vectors $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_t$, exist, such that*

$$\{\mathbf{x} | \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \in \mathbb{Z}^n\} = \{\mathbf{x}_0 + \lambda_1 \mathbf{x}_1, \dots, \lambda_t \mathbf{x}_t | \lambda_1, \dots, \lambda_t \in \mathbb{Z}\}, \quad (11)$$

where $\mathbf{x}_1, \dots, \mathbf{x}_t$ are linearly independent, and $t = n - \text{rank}(\mathbf{A})$.

Proof: See Corollary 4.1c in [22] (p. 45–48).

Note that solutions to (10) where $\boldsymbol{\xi}$ belongs to the nullspace or kernel of \mathbf{B} (i.e., $\boldsymbol{\xi} \in \{\boldsymbol{\alpha} | \mathbf{B}\boldsymbol{\alpha} = \mathbf{0}\}$, commonly denoted by $\boldsymbol{\xi} \in \text{null}(\mathbf{B})$ or $\boldsymbol{\xi} \in \text{ker}(\mathbf{B})$) represent temporal reuse rather than potentially conflicting accesses. The components of solution vectors corresponding to \mathbf{r} lets us distinguish between reuse and conflict without computing the nullspace of \mathbf{B} . Any solution representing a potential conflict must have some non-zero \mathbf{r} -component, since otherwise the pair of accesses are in fact accessing the same location which by definition is a case of reuse. Thus, the set of self-conflict vectors is chosen as the $\boldsymbol{\xi}$ -components of the basis indicated by Theorem 3, for which the \mathbf{r} -components are non-zero. For multi-word cache-lines, potential conflicts exist that do not satisfy (10). An augmented set of self-conflict vectors corresponding to multi word cache lines of some known constant length can be obtained by solving variants of (10) with different right hand sides corresponding to how the Morton order in question maps array elements to cache lines [8]. Doing so yields sets of slightly different conflict vectors for adjacent iterations, which may be more compactly represented by the union of all such sets at the loss of some precision.

Under the assumption that base-addresses of multi-dimensional arrays that use the HAT-layout are aligned on cache-set boundaries, an identical derivation of conflict vectors applies also to references which access different arrays, but which use otherwise identical access expressions. However, in the latter case, solutions for which $\boldsymbol{\xi} \in \text{ker}(\mathbf{B})$, including $\boldsymbol{\xi} = \mathbf{0}$ do represent true conflicts. Alternatively, assume the same constraints as immediately above, with the exception that arrays using the HAT layout are *not* constrained to cache-set boundaries, but that the relative alignment of such arrays is somehow known. Then the expression corresponding to equation (10) will (typically) have a non-zero right hand-side, corresponding to the difference in alignment of the arrays.

3.5. CROSS INTERFERENCE

For a pair of distinct references R_A and R_B , possibly to arrays of differing dimensionalities, (9) can not be simplified in the manner done for self-interference above. Furthermore, cross-interference behavior is inherently more varied and more complex to characterize. By trying to solve (9) directly, the regularity and structure which exist among solutions easily becomes obscured.

Instead it is useful to study a “reduced” system of equations, corresponding to the iterations \mathbf{j}' such that the references R_A and R_B without offset vectors, would access conflicting locations:

$$\begin{aligned} \mathbf{A}\mathbf{j}' &= \mathbf{H}_{d_A}^c \mathbf{s}' + \mathbf{L}_{d_A}^c \mathbf{p}' \\ \mathbf{B}\mathbf{j}' &= \mathbf{H}_{d_B}^c \mathbf{s}' + \mathbf{L}_{d_B}^c \mathbf{q}', \end{aligned} \quad (12)$$

where $\mathbf{s}' \in \{0, 1\}^c$, $\mathbf{p}' \in \mathbb{Z}^{d_A}$, $\mathbf{q}' \in \mathbb{Z}^{d_B}$, and where \mathbf{A} and \mathbf{B} are the access matrices of references R_A and R_B , respectively, and \mathbf{j}' is an iteration vector. The significance of (12) is that for any pair of solutions $\{(\mathbf{j}, \mathbf{s}, \mathbf{p}, \mathbf{q}), (\mathbf{j}', \mathbf{s}', \mathbf{p}', \mathbf{q}')\}$ to (9) and (12) such that $\mathbf{s}^T \cdot \mathbf{s}' = 0$, we clearly have $\boldsymbol{\xi}(\mathbf{j}) = \boldsymbol{\xi}(\mathbf{j} + \mathbf{j}')$ since $\boldsymbol{\xi}(\mathbf{j}') = \mathbf{0}$, thus specifying *translational symmetry* in the solutions to (9). As shown below, the sets of \mathbf{j}' satisfying (12) are easily obtained for typical index expressions. Space limitation prevent us from including proofs, which may instead be found in Chapter 6 of [8].

To establish the results just stated, we begin with definitions which distinguish between types of \mathbf{s}' -vectors satisfying (12), which motivates us to rephrase (12) slightly, and label equation instances corresponding to different dimensionalities of \mathbf{s}' :

$$\begin{aligned} \mathcal{S}_c &= \{\mathbf{s}' \in \{0, 1\}^c \mid \exists \mathbf{j}', \mathbf{p}', \mathbf{q}' : \\ &\quad \mathbf{H}_{d_A}^c \mathbf{s}' = \mathbf{A}\mathbf{j}' - \mathbf{L}_{d_A}^c \mathbf{p}' \wedge \mathbf{H}_{d_B}^c \mathbf{s}' = \mathbf{B}\mathbf{j}' - \mathbf{L}_{d_B}^c \mathbf{q}'\} \end{aligned} \quad (13)$$

DEFINITION 4. A vector $\mathbf{s}' \in \mathcal{S}_c$ is said to be reducible (in \mathcal{S}_c) whenever $\mathbf{s}' = \mathbf{s}'_1 + \mathbf{s}'_2$, $\mathbf{s}'_1, \mathbf{s}'_2 \in \mathcal{S}_c$ and $\mathbf{s}'_1 \neq \mathbf{0}$, $\mathbf{s}'_2 \neq \mathbf{0}$. Vectors $\mathbf{s}' \in \mathcal{S}_c$ which are not reducible are said to irreducible, among which the nullvector, $\mathbf{0}$, of appropriate dimensionality is always present due to it being a trivial solution to (12).

THEOREM 4. For any pair of array references R_A, R_B such that the row sums of $[\mathbf{A}^T \mathbf{B}^T]$ are less than the minimum diagonal element of $\mathbf{L}_{d_A}^c$ and $\mathbf{L}_{d_B}^c$, the set of binary vectors \mathcal{S}_c is generated by a unique subset of \mathcal{S}_c consisting only of irreducible vectors.

First, each irreducible element of \mathcal{S}_c corresponds to an irreducible element of \mathcal{S}_{c-1} . Thus, given the set of irreducible elements of \mathcal{S}_{c-1} , the irreducible elements of \mathcal{S}_c may be found by extending each irreducible element $\mathbf{s} \in \mathcal{S}_{c-1}$ (beginning with $\mathbf{s} = \mathbf{0}$) to $[\mathbf{s}^T 1]^T$ and testing for membership in \mathcal{S}_c .

Having obtained a complete set of irreducible $\mathbf{s}' \in \mathcal{S}_c$, we may solve (12) for each such \mathbf{s}' :

$$\begin{bmatrix} \mathbf{A} & \mathbf{L}_{d_A}^c & 0 \\ \mathbf{B} & 0 & \mathbf{L}_{d_B}^c \end{bmatrix} \begin{bmatrix} \mathbf{j}' \\ \mathbf{p}' \\ \mathbf{q}' \end{bmatrix} = \begin{bmatrix} \mathbf{H}_{d_A}^c \\ \mathbf{H}_{d_B}^c \end{bmatrix} \mathbf{s}',$$

using standard methods [22, 27]. By doing so, a set of solutions for \mathbf{j}' , on the form $\mathbf{j}' = \{\mathbf{u} | \mathbf{u} = \mathbf{v}_0 + \lambda_1 \mathbf{v}_1 + \dots + \lambda_k \mathbf{v}_k\}$, is determined for each irreducible \mathbf{s}' , where in turn, $\xi(\mathbf{j}) = \xi(\mathbf{j} + \mathbf{j}')$ for each \mathbf{j} such that $\mathbf{s}^T \mathbf{s}' = 0$, where \mathbf{s} is the location corresponding to \mathbf{j} . Note that when the conditions of theorem 4 are not satisfied by a pair of references, the members \mathbf{s}' of \mathcal{S}_c may be determined by enumeration of all elements in $\{0, 1\}^c$ and attempting to solve (12) for each such $\mathbf{s}' \in \{0, 1\}^c$. Typically, however, the elements in access matrices of multi-dimensional arrays are small, giving theorem 4 wide applicability in practice.

Finally a full cross-interference characterization may be determined as the solution to (9) for each $\mathbf{s} \notin \mathcal{S}_c$, which minimizes $\|\xi\|_1$. The cross-interference behavior of the pair of references is then perfectly described for each location \mathbf{s} , by its conflict vector ξ and the sets of iteration vectors obtained by solving (12) for each irreducible $\mathbf{s}' \in \mathcal{S}_c$. Note that for cross conflicts between arrays of equal dimensionality and equal-size elements, storage locations are sufficiently well identified by their corresponding array indices, and thus (9) may be simplified to

$$\begin{aligned} \mathbf{A}(\mathbf{j} + \xi) + \mathbf{a} &= \mathbf{l} + \mathbf{L}_{d_A}^c \mathbf{p} \\ \mathbf{B}\mathbf{j} + \mathbf{b} &= \mathbf{l} + \mathbf{L}_{d_B}^c \mathbf{q}, \end{aligned} \quad (14)$$

in such cases.

For multi-word cache lines, an augmented set of conflict vectors is obtained by solving several instances of (9) wherein $\mathbf{H}_{d_A}^c \mathbf{s}_1$ and $\mathbf{H}_{d_B}^c \mathbf{s}_2$ are used instead of $\mathbf{H}_{d_A}^c \mathbf{s}$ and $\mathbf{H}_{d_B}^c \mathbf{s}$, respectively, with \mathbf{s}_1 \mathbf{s}_2 being different locations belonging to same cache line [8].

3.6. EXAMPLES

The following program fragment originates from the SPEC95 TOMCATV-benchmark:

```
DO J = 3, N-1
  DO I = 2, N-1
    R = AA(I,J) * D(I,J-1)
    D(I,J) = 1.0/(DD(I,J) - AA(I,J-1)*R)
    RX(I,J) = RX(I,J) - RX(I,J-1)*R
    RY(I,J) = RY(I,J) - RY(I,J-1)*R
  END DO
END DO
```

wherein all arrays (AA, D, DD, RX, RY) have double precision elements. Since the access matrices are identical for all array references above and all arrays are two-dimensional with double precision elements, the matrices \mathbf{B} and $\mathbf{L}_{d_B}^c$ in (10) are identical for all array references leading to identical self-interference ξ -vectors.

For values of \mathbf{B} and $\mathbf{L}_{d_B}^c$ given by

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ and } \mathbf{L}_{d_B}^c = \begin{bmatrix} 64 & 0 \\ 0 & 32 \end{bmatrix},$$

corresponding to the access matrices of the references above, and a cache set-size of 16Kb ($c = 14$), the corresponding set of self-interference ξ -vectors, as determined by solving (10)

$$\left\{ \begin{bmatrix} 64 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 32 \end{bmatrix} \right\}$$

as expected. The set of ξ -vectors between references to distinct arrays that use identical index expressions becomes

$$\left\{ \begin{bmatrix} 64 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 32 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

and the set of cross-interference ξ -vectors for pairs of the two distinct index expressions present in the program fragment above is obtained by solving (14), which yields ξ -vectors

$$\xi = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \xi = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

respectively.

As our second example of ξ -vector computation, we consider the following program fragment (from the NAS GMTRY kernel) which performs gaussian elimination:

```
DO I=1,MATDIM
  RMATRIX(I,I) = 1.0/RMATRIX(I,I)
  DO J=I+1, MATDIM
    RMATRIX(J,I) = RMATRIX(J,I) * RMATRIX(I,I)
    DO K=I+1, MATDIM
      RMATRIX(J,K) = RMATRIX(J,K) - RMATRIX(J,I) * RMATRIX(I,K)
    END DO
  END DO
END DO
```

As in the previous example, all arrays are two-dimensional and have double precision elements so that a simpler form of (12) derivable from (14) may be used for computing cross-interference. For the pair of references RMA-TRX(J,I), RMATRIX(I,K), the linear system to solve becomes

$$\begin{bmatrix} 1 & 0 & 0 & -64 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -32 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -64 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -32 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ p_1 \\ p_2 \\ q_1 \\ q_2 \end{bmatrix} - \begin{bmatrix} u \\ v \\ u \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

for which one finds that

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \lambda_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \lambda_2 \begin{bmatrix} 64 \\ 0 \\ 64 \end{bmatrix} + \lambda_3 \begin{bmatrix} 0 \\ 0 \\ 32 \end{bmatrix}$$

gives the translational symmetry of the complete set of solutions. The region in which to determine individual solutions is therefore given by

$$\{(i, j, k) | 0 \leq k < 32, 0 \leq i+k < 128, 0 \leq i+j+k < 3\}$$

An enumeration of all individual solutions here would clearly require too much space and is easily obtained by scanning the region given above. Hence we conclude our examples by saying that the process is repeated for each distinct pair of references in a loop nest in order to obtain complete information on the cache behavior.

4. Related Work

Array layout has received much attention in the context of automatic array alignment and distribution for distributed memory machines [3, 12, 14]. In the context of uni-processor memory hierarchies, the work of Chatterjee *et al.* [4, 5] is the most similar to ours. They investigate and evaluate an array layout which is essentially identical to the HAT-layout, but which uses a set of smallest tile-sizes within which linear layouts. Significant performance gains are reported for some hand-tailored tiled algorithms using their layout. The basic properties of Morton ordering have been provided multiple times in the literature, the most recent such report being that of Wise [26].

With respect to program analysis, our work is most closely related to the Cache Miss Equations [10, 11]. The cache miss equations (CME) framework, developed by Ghosh *et al.*, identify cache-misses as integer solutions to specific equations. The CME solution count could possibly be used to select between a limited number of transformations, but it is costly to compute even though faster, approximative methods have recently been suggested [25]. Furthermore, unknown array bounds turn up in the equations and counting the solutions would then imply solving the CME symbolically which seems prohibitively difficult. This, however, is not a fault of the CME-framework, but a direct consequence of linear array layouts.

Lam *et al.* point out the destructive effects of self-interference for tiled algorithms, and show how tile-sizes may be selected at run-time to avoid self-interference. Subsequently Coleman and McKinley [7] generalized

and improved the techniques of Lam *et al.*. Carter *et al.* [2] discuss hierarchical tiling schemes for a hierarchical shared memory model. Rivera and Tseng [19, 20] evaluate different heuristics for intra- and inter-array padding as a means of avoiding conflicts. Temam *et al.* have studied at data copying as a means to avoid cache interference in tiled loops [23]. Kodukula *et al.* have developed a seemingly flexible data-centric approach to loop tiling, called “shackling”, which handles imperfect loop nests and may be used to tile for multiple levels of a memory hierarchy. The common focus on storage suggests that data shackling might be a suitable starting-point for developing more comprehensive optimization frameworks for the HAT-layout.

5. Conclusions

We have investigated a hierarchically tiled array layout, HAT, from a compiler perspective. The main advantage of this layout is that logical data locality in multi dimensional arrays consistently results in effective data locality at run-time, and that it makes compile-time analysis with respect to memory system performance feasible. This makes it possible to construct compilers which perform automatic tiling, and other compile-time optimizations, with a higher degree of accuracy and precision than allowed by linear array layouts.

With respect to cache hit rates, HAT inherently has a disadvantage for codes with regular array accesses. On the other hand, the HAT-layout makes codes with regular array accesses analyzable so that poor cache behavior can be detected and avoided at compile-time. A consequence of this is that the HAT-layout also could be interesting for applications where predictability is more important than average performance, such as in real-time systems.

Future work includes developing and evaluating automatic methods for selecting loop tile sizes and data copying, based on the information provided by the conflict vectors.

References

1. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, pages 114–124, Nov. 1992.
2. L. Carter, J. Ferrante, and S. Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, Apr. 1995.

3. S. Chatterjee, J. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical report, XEROX PARC, Dec. 1992.
4. S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proc. 1999 ACM Int. Conf. on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.
5. S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proc. Eleventh ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.
6. R. F. Cmelik. Spixtools user's manual. Technical Report SMLI TR-93-6, Sun Microsystems Labs, Mountain View, CA, Feb. 1993.
7. S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, 1995.
8. N. P. Drakenberg. *Hierarchical Array Tiling*. Licentiate thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, 2001. In preparation. <http://www.it.kth.se/~npd/lic-thesis.ps>.
9. I. Gargantini. Linear octrees for fast processing of three-dimensional objects. *Comput. Graphics Image Process.*, 20:365–374, 1982.
10. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. 1997 International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.
11. S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Vienna, Austria, Oct. 1998.
12. M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992.
13. Y. Hu, S. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proc. Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.
14. K. Knobe, J. D. Lucas, and W. J. Dally. Dynamic alignment on distributed memory systems. In *Proc. 3rd Workshop on Compilers for Parallel Computers*, pages 394–404, July 1992.
15. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.
16. N. Mitchell, L. Carter, and J. Ferrante. A compiler perspective on architectural evolutions. In *Workshop on Interaction between Compilers and Computer Architectures*, San Antonio, Texas, Feb. 1997.
17. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, Mar. 1966.
18. J. Pilkington and S. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. on Parallel and Distributed Systems*, 7:288–300, Mar. 1996.
19. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.
20. G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proc. 1998 International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998.
21. G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.*, 55(3):221–230, May 1992.
22. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.
23. O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proc. Supercomputing '93*, Portland, OR, Nov. 1993.
24. K. Tocher. The application of computers to sampling experiments. *J. Roy. Statist. Soc.*, 16(1):39–61, 1954.
25. X. Vera, J. Llosa, A. Gonzalez, and C. Ciuraneta. A fast implementation of cache miss equations. In *Proc. 8th Workshop on Compilers for Parallel Computers*, pages 321–328, Aussois, France, Jan. 2000.
26. D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In A. Bode et al., editors, *Proc. Euro-Par 2000*, pages 774–783. Springer-Verlag, 2000.
27. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
28. J. R. Woodwark. The explicit quadtree as a structure for computer graphics. *Comput. J.*, 25(2):235–238, 1982.

