Array Organization in Parallel Memories

Mayez Al-Mouhamed *

Abstract

The bandwidth mismatch between processor and main memory is one major limiting problem. Although streamed computations have predictable access patterns their references have little temporal locality and are generally too long to cache. A memory and compiler co-optimization aimed at reducing low-level memory accesses using software and hardware locality optimizations is presented. We propose a scalable and predictable parallel memory based on a compiler synthesis of storage schemes for multi-dimensional arrays that are accessed by an arbitrary but known set of data access patterns. Using algebra of non-singular Boolean matrices, we present analysis of conflict-free access to (1) parallel memories, and (2) alignment networks. Finding a multi-pattern storage scheme is one NP-complete problem. An effective compiler heuristic is proposed for finding a the storage matrix that minimizes overall memory access time. This applies to arbitrary linear patterns and arbitrary alignment networks. It is shown that the proposed storage scheme finds an optimal storage scheme for parallel (1) FFT, and (2) bitonic sorting. The proposed storage scheme outperforms statically optimized storages in the case of power-of-2 multi-stride access. The case of non power-of-2 strides is also addressed. The performance and scalability of the proposed parallel memory and its predictable access time are presented using numerical and multimedia algorithms. It is shown that a memory utilization above 83% is achieved by our storage scheme for 64 memories which largely outperforms previous proposals. Our approach provides a tool for matching the storage pattern with the data access patterns needed for embedded systems running streamed computations with predictable data access patterns.

Keywords: access patterns, embedded systems, compiler optimization, parallel memory, streamed computations.

1 Introduction

Effective utilization of bandwidth in hierarchical memory systems aims at exploiting compile time knowledge of a program to reduce unnecessary data transfer between processor and main memory. Compiler optimization that attempts maximizing *temporal* and *spatial* localities and minimizing mapping conflicts produced encouraging results [22]. To reduce memory conflicts in multiprocessors, compiler directed compaction-based data partitioning [27] improved performance from 13% to 40% for a class of synchronous dataflow computations. The compiler [6] knowledge of the access patterns of parallel applications is used to create *compiler directed page coloring* to direct run-time virtual memory page mapping. Here the compiler explicitly

*Computer Engineering Department, College of Computer Science and Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia (mayez@ccse.kfupm.edu.sa)

attempts predicting the *access patterns* of parallelized applications. A 50% improvement over a standard page mapping policy was achieved. Compile-time can also improve scalar access in parallel memories where scheduling of a very low number of data transfers proved that a very high percentage of memory access conflicts can be avoided.

While media processing is becoming the dominating force in desktop computing a substantial fraction of processor resource is used to hide the latency of an unpredictable hierarchical memory system [16]. The integration of high-speed logic and memory on a single chip provided large sequential and random memory bandwidth while making the delivered performance highly predictable. There is pressing need for innovative memory architectures and organization to reduce bandwidth imbalance between processor and main memory. The TERA MTA high-performance computer [4] uses multi-threaded execution pipelines to tolerate a 100-clock latency between the logic and the memory. The bandwidth mismatch between the processor and the main memory is expected to be more acute in the future. It is predicted that processor-memory latency will be 10^4-10^5 clocks in the Petaflops HTMT Computer [29] even with the use of exotic memory and optical network technologies.

For many synchronous dataflow multimedia computing a scalable and predictable parallel memory can be achieved if the data structure is allocated to separate memories whenever the compiler finds that data may be accessed in parallel, i.e. fine-grain data parallelism.

In high performance parallel memories, network contention and serialization of memory accesses are responsible for significant performance degradation [23, 24, 31, 32]. Memory interleaving causes non-uniform memory access, especially in the case of sequential addresses which differ by a constant amount, or *stride*, that is not relatively-prime to the number of memories. To reduce memory and network conflicts, q prime-number of memories [17] may be used, which generally outperform interleaving at the cost of some additional computation at the address translation level. The dynamic behavior of memory references has been extensively studied for both lock-step memories and shared-memory systems [28].

Static storage schemes are optimized to some fixed data patterns. Conflict-free access [5] to rows, columns, and diagonals of arrays have been proposed on the basis of row-rotation. Budnik and Kuck [5] proposed storage schemes based on row-rotation which have been statically implemented for conflict-free access to arbitrary rows and columns of arrays. Linial and Tarsi [19] studied the characterization of a shuffle-exchange network based on balanced matrices. Some approaches have proposed augmenting the topology of multistage networks to allow static conflict-free access to a set of non-linear data patterns [18].

To avoid run-time overheads due to row-rotation storage, Sohi [28] proposed bit-wise Boolean address transformations for vector processors in order to determine the memory number where a given array element should be stored. The scheme can be efficiently used for power-of-2 strides but other strides can also be accessed by using a few buffers at the memory inputs and outputs [12]. The buffers reduce the effects of transient degradation in pipelined memories. Deb [7] showed that storing an array with different linear schemes enables conflict-free access of nonlinear data patterns.

Boppana [3] proposed a storage scheme for conflict-free access to the row, column, main-diagonal, and square blocks. Norton [25] synthesized a transformation matrix that allows conflict-free access of the Baseline network for a number of power-of-2 strides. While all these approaches are useful, they either treat: 1) only the parallel memory characterization or, 2) both memory and network aspects but, consider only a reference set of static patterns. Unfortunately, the performance of static storages may vary widely depending on the accessed stride and its origin.

Our approach is a compiler technique that explicitly attempts predicting the *data access patterns* of compiler parallelized and restructured loops and compute a suitable storage scheme for effective access at run time. At run-time, the data structure is allocated to separate memories to favor parallel access of any instance of the predicted data patterns. However, the randomized routing methods or some static storages can still be used where access patterns cannot be found by the compiler or when computing a storage on-line is likely to take too much time. Many synchronous dataflow computations may benefit from this approach like multimedia compression/decompression algorithms, numerical algorithms, DSP functions, low-level vision processing, robotics, etc.

In this work, we find the necessary and sufficient conditions for accessing parallel memories without network and memory conflicts for a given set of power-of-two data patterns. Since finding a storage scheme that minimizes the memory access time for a given set of data access patterns is NP-complete we propose a systematic compiler heuristic. It is shown that our approach can easily be used by a compiler for synthesizing storage schemes for FFT, bitonic sorting, stride access, and a synthetic set of data patterns. Scalability of the proposed parallel memory is investigated by using a simulated memory model with a compiler directed storage scheme.

This paper is organized as follows. Section 2 presents a review of compiler techniques for data access analysis and presents our strategy to memory and compiler co-optimization. Section 3 presents some background on parallel memory and alignment networks. In Section 4, the characterization of parallel memory and network conflict-free access is presented. Section 5 presents our approach for finding a combined storage scheme for arbitrary but given sets of data patterns and summarizes the NP-completeness aspects. Section 6 presents a heuristic approach for finding multi-pattern storage schemes. In section 7, we present applications to FFT, bitonic sorting, arbitrary stride access, and application of parallel memory in hierarchical memory systems. In section 8 we evaluate performance of the proposed approach for the case of some numerical and multimedia algorithms. In section 9 we conclude.

2 Memory and compiler co-optimization

Two methods [33] are generally used for reducing latency of the memory: (1) latency tolerance, and (2) latency avoidance. Latency tolerance aims at overlapping computations with communications like in prefetching. We are concerned with latency avoidance approach which aimed at reducing low-level memory accesses using software and hardware locality optimizations. In SMP this reduces the number of accesses, memory contention, and network contention.

Unfortunately the access pattern of sparse or irregular data cannot be determined at compiler time. However, memory and compiler co-optimization have always been used to improve the efficiency of the memory based on analysis of the data and I/O locality. A latency avoidance compiler optimization [33] inserts application dependent tasks. At run time compiler tasks restructure parallel loops by shrinking and partitioning the memory-access space to minimize sharing among partitions and maximize partition reuse. Architecture and compiler co-optimization [24] provide a tool for reconfiguring and controlling memory hierarchy by a compiler that inserts hints based on classification of access features like reusability, consecutive access, stride, and regularity.

To improve bus utilization a memory controller can be used to re-map irregular or sparse memory accesses into dense accesses in the cache memory. For this a compiler optimiza-

tion [13] uses dependence and locality analysis (temporal and spacial) for finding cost models that determine when to use the controller based on reducing cache misses and cost of re-mapping.

Efficient shared-memory programming model in distributed-memory multicomputers have excessive overhead of run-time consistency maintenance. Regular access patterns can be precisely analyzed [8] and results in superior performance. To avoid broadcasting data in the case of irregular access patterns an "inspector-executer" precomputes the accessed data when executing a loop. For this a compiler strategy [8] analyzes shared-memory access patterns and transforms the code by inserting calls to the run-time system directing it to implement bulk data transfer.

To improve I/O performance the compiler analyzes I/O access patterns of individual applications and determines suitable file storage patterns and I/O strategies. Specifically, the compiler [14] identifies cases where array storage and access patterns do not match. In this case the compiler-directed collective I/O reads the data as stored and then redistribute the data among the processors to meet the access patterns. A compiler access pattern analysis strategy is proposed [15] for detecting and isolating definite cache hits in a given application. The compiler substitutes these cache references with energy-efficient loads that access only data instead of both data and tags.

Hierarchical memory has long been used to close up the gap between fast processors and slow memory, but as the gap increases the effectiveness of this approach is diminishing even if it is still a valid solution for general-purpose computing. Specifically vector computing in streamed applications is generally too long to cache and mostly used once without temporal locality. Memory systems can be made more efficient by exploiting the above access features and potential for parallel memories.

For many high performance computing applications the memory cannot supply operands at the right processor speed which may result in quite disappointing performance [30]. Memory bandwidth is a major limiting factor in streamed computations like scientific vector processing or multimedia compression and decompression, encryption, signal and image processing, and graphics processing, DNA sequence matching. These applications have predictable access patterns [23, 24]. To improve performance of DRAM memory a stream memory controller is proposed to combine compile-time detection of stream access with execution-time selection of the access order and issue. Evaluation shows that performance improved by a factor of 13 over traditional access methods.

The performance of Internet and Multimedia computations are increasingly limited by their data accessing capabilities. A configurable parallel memory (CPM) [32] having a (1) a hardwired address computation unit, (2) a set of parallel memories, (3) a crossbar-based permutation unit that shuffles the request and requested data in the right order. The CPM is intended to be part of a multimedia application specific embedded system. Evaluation shows that CPM has significant advantages over traditional DRAMs.

2.1 Compiler optimization

We are concerned with a compiler technique for finding the access patterns of regular streamed computations [31, 32]. Streamed computations have predictable access patterns [23, 24]. Generally, iterative parallel loop consists of an outer sequential loops and inner parallel loops. The outer loop enforces synchronization at each outer loop iteration. The iteration of inner loops can generally execute in any order (LID). For example the solution of partial differential

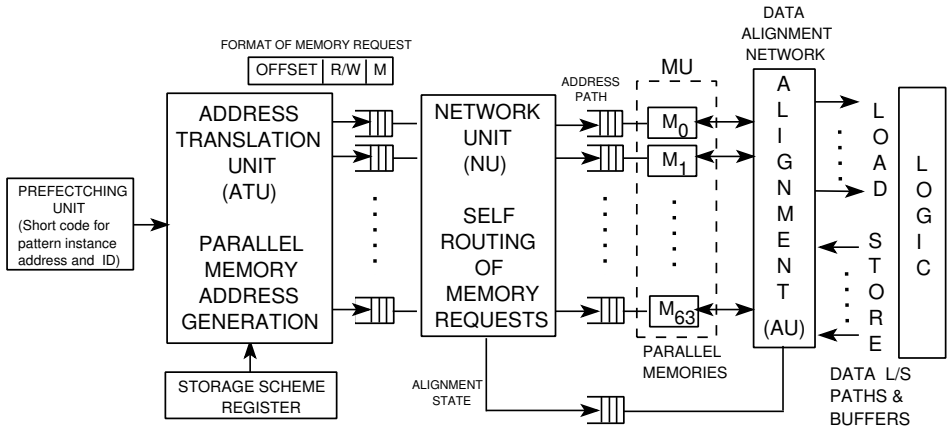

Figure 1: Pipelined memory: address translation, network unit, memory, and alignment

equations through relaxation. The collection of array offsets is referred to as the stencil. Consider the statement

$$a(i,j) = F(a(i-1,j), a(i,j-1), a(i,j+1), a(i+1,j))) \tag{1}$$

The associated stencil is $\{(0,0), a(-1,0), a(0,-1), a(0,1), a(1,0)\}$. Now assume 2^n processors executing successive innermost iterations in parallel in a lock-step fashion. If the innermost loop updates the first array dimension, we may load a sub-column data pattern corresponding to each element of the stencil. For example the stencil element $(-1,0)$ leads to load the pattern $\{(0,1),(1,1),(2,1),\ldots,(2^n-1,1)\}$ which represents a set of 2^n data elements. A spacial locality of reference called data access pattern can be identified for this loop. For example, the values in the first dimension of the pattern $\{(0,1),(1,1),(2,1),\ldots,(2^n-1,1)\} = \{(x, constant) : 0 \leq x \leq 2^n - 1\}$ takes all possible combinations of n-bit while the value in the other dimension is a constant. This means that the basis for the access pattern is the set of first canonical vectors f_{n-1}, ..., f_0. This defines a sub-column access pattern with basis $\{f_{n-1}, \ldots, f_0\}$. For each parallel access index $x = x_{n-1}f_{n-1} + \ldots + x_0 f_0$ takes all possible binary combinations, where multiplications are "and" and additions are modulo 2. Since $(x_{n-1}, \ldots, x_0)$ takes all possible 2^n binary combinations we may cause the data elements of the access pattern be stored in distinct 2^n memories if we can find a one-to-one mapping from $(x_{n-1}, \ldots, x_0)$ into storages.

Similar analysis on the use of the same array in other loops allows finding a set of access patterns and their bases. The pattern basis is the logic representation of the access pattern. The knowledge of pattern bases enables using our proposed method for finding an optimized storage scheme. Given an array, the task of the compiler is to estimate the usage (weight) of each access pattern in the loops and build its corresponding weighted conflict graph representation. This optimization problem can be solved by coloring the above graph and finding the storage scheme for a given array. The storage scheme is a linear or non-linear mapping from the array addresses onto the storages. The storage scheme causes the array data be uniformly distributed over all the memories for each of its data access patterns.

2.2 Parallel memory optimization

The parallel memory consists of (1) an address translation unit, (2) a set of parallel memories, and (3) a network unit used for data alignment. The parallel memory engine is shown on Figure 1. The pre-fetching unit generates requests for accessing a given data pattern and its location in the array. Using XOR operations, the product of the storage scheme (a rectangular binary matrix) by an array address gives the storage number where to find the corresponding array element. The address translation unit (ATU) uses the storage scheme and pattern parameters in generating the parallel memory request. A request consists of a vector of memory numbers (M), controls, and memory offsets.

The network unit (NU) is assumed to be a MIN made of 2×2 SEs. Due to data skew, self-routing [1] of the requests within the NU allows forwarding each request to its corresponding memory. If two requests conflict within an SE one request is correctly routed and the other is discarded and a negative feedback is returned, using a reverse path, to the input buffer from where the request was issued. Such an input buffer re-submits its previously discarded message in the next cycle. Following each network cycle the routed requests submit their R/W control and offset to the memory unit (MU) and an access occurs. Due to the data skew, we should align the outputs of parallel memories with the logic units, i.e. the ith logic unit receives the ith data element in the data pattern instance. The data alignment unit (AU) uses the state of previously established NU paths to align the data from the memories to the logic unit. The switching state is sent from NU to AU and used to create reverse paths over which the requested data is sent (aligned) to the logic units.

In summary the parallel memory serves request for data streams according to a given set of data patterns (sub-row, sub-column, block, etc.). For this it retrieves a cached storage scheme for a specific array, generates memory offsets, route the requests to memories, retrieve data, align the data, and forward to the logic unit.

3 Parallel memory access - Background

Consider a computer that consists of 2^n processing elements (PEs) which are interconnected to 2^n memories by using 2^n input/output multistage interconnection network (MIN).

An access pattern is a parallel access to a set of 2^n array elements (data pattern) lying in a row, column, square or rectangular block, a power-of-2 stride, etc. Considering a 2D array structure, a data pattern is formally a set of 2^n elements identified by the following array addresses $\{(b_r, b_c) + (k_r \times 2^{rs}, k_c \times 2^{cs})\}$, where (b_r, b_c) is a pattern instance identifier or pattern origin, k_r and k_c are used to index the row and column elements with the range $0 \leq k_r \leq 2^r - 1$, $0 \leq k_c \leq 2^c - 1$, and $2^n = 2^{r+c}$, and 2^{rs} and 2^{cs} are the row and column strides, respectively.

Assume a 2-dimensional array $A = \{a(i, j) : 0 \leq i, j \leq 15\}$. Consider 8 parallel memories and an access pattern of 8 array elements defined by $2^r = 2$ rows each has $2^c = 4$ columns with strides of $2^r s = 1$ and $2^c s = 2$ along the row and column, respectively. The first instance of this pattern is the set $\{(0, 0) + (k_r \times 2^0, k_c \times 2^1) : 0 \leq k_r \leq 1, 0 \leq k_r \leq 3\} = ((0, 0), (0, 2), (0, 4), (0, 6), (1, 0), (1, 2), (1, 4), (1, 6))$. Different instances of the same pattern can be defined as $\{(b_r, b_c) + (k_r \times 2^0, k_c \times 2^1) : 0 \leq k_r \leq 1, 0 \leq k_c \leq 3\}$ through assigning values to the pattern origin (b_r, b_c) such as $b_r = 0, 2, 4, \ldots, 14$ and $b_c = 0, 8$.

We may define other relevant data patterns for the same array. The objective of designing parallel memory systems is to store the array into the parallel memories so that the array

elements of any instance of a given data pattern be uniformly distributed over all the memories. A conflict free parallel memory enables any pattern instance be retrieved from the memories in parallel in one single access, i.e. one-to-one mapping from storages to elements of any pattern instance. The elements at output of memories need to be aligned (storage is skewed) to the PEs through the MIN. A conflict free MIN can align any instance of a given data pattern without internal conflicts. Data alignment in MIN means that first element is self-routed to PE_0, second element to PE_1, etc. Self-routing in MIN of elements of a given data pattern allows the ordered set of elements to be forwarded to the ordered set of requesting PEs. Therefore, we need a storage scheme that is conflict free with respect to both parallel memory (external conflicts) and network (internal conflicts). Given a set of data patterns our objective is to find a storage scheme that is optimized with respect to memory and network conflicts.

The memory utilization is maximum when all the 2^n array elements of a given data pattern instance are fetched in parallel, each element is from a distinct memory. Assume a 1-dimensional array $A = \{a(i) : 0 \leq i \leq 2^k - 1\}$, where $k \geq n$. This array is to be accessed by using a storage scheme defined by $d(i) = Mi$, where binary of i is $i = (i_{k-1}, \ldots, i_0)$ that represents the address of $a(i)$, M is some $n \times m$ Boolean matrix, and $d(i)$ is an n-bit value that represents the memory number where array element $a(i)$ should be stored, and m is the number of distinct vectors in all the pattern bases. The binary of i is restricted to $i = (i_{m-1}, \ldots, i_0)$ in mapping elements to storages defined by $d(i) = Mi$, where $m \leq k$. In other words, element $a(i)$ is stored into memory $M_{d(i)}$. Addition and multiplication of Boolean matrices ($d(i) = Mi$) are modulo 2. The binary of i is $i = v_{m-1}i_{m-1} + \ldots + v_0 i_0$, where $v_{m-1}, \ldots, v_0$ are m canonical vectors of Z_2^m that are $v_0 = (0, \ldots, 0, 1)$, $v_1 = (0, \ldots, 1, 0)$, and $v_{m-1} = (1, \ldots, 0)$.

Suppose we know a priori the memory access patterns of a program and let's consider the following mapping of elements to memories by using a storage matrix M:

$$d(i) = \begin{pmatrix} d_2 \\ d_1 \\ d_0 \end{pmatrix} = \begin{array}{c} \begin{matrix} c_3 & c_2 & c_1 & c_0 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array} . \begin{pmatrix} i_3 \\ i_2 \\ i_1 \\ i_0 \end{pmatrix} \tag{2}$$

where c_j denotes the jth column of M and i is restricted to its 4 least-significant bits, i.e. $i = (i_3, i_2, i_1, i_0)$. Mapping of 64 array elements to 8 memories ($n = 3$ and $k = 6$) is shown in the tables of Figure 2 which maps the 64 array elements to 8 memories by using the storage scheme defined in Eq. 2. The array elements that fall into column M_i ($0 \leq i \leq 7$) are stored into memory M_i. This scheme allows accessing four data patterns P_0, P_1, P_2, and P_3. We will see that these patterns are useful for parallel sorting. The first instance of pattern P_0 consists of accessing an 8-element row specified by the indices $\{0, 1, 2, \ldots, 7\}$ and the second instance is $\{8, 9, \ldots, 15\}$. Accessing these two instances enables the PE_i to load elements i and $i + 8$, for $0 \leq i \leq 7$. In Figure 2-(a) we marked the first and second instances of P_0 by putting their elements in squares and circles, respectively. The same notation is also used in Figure 2-(b), (c), and (d) for the first and second instances of pattern P_1, P_2, and P_3, respectively. Each pattern instance is a locality of data elements that can be translated within the array which leads to access different instances of that pattern. Ideally the elements of each instance of a given pattern are accessible in parallel, each element is stored into a distinct memory. Pattern P_1 allows accessing two groups of 4-element with a stride of 4 such

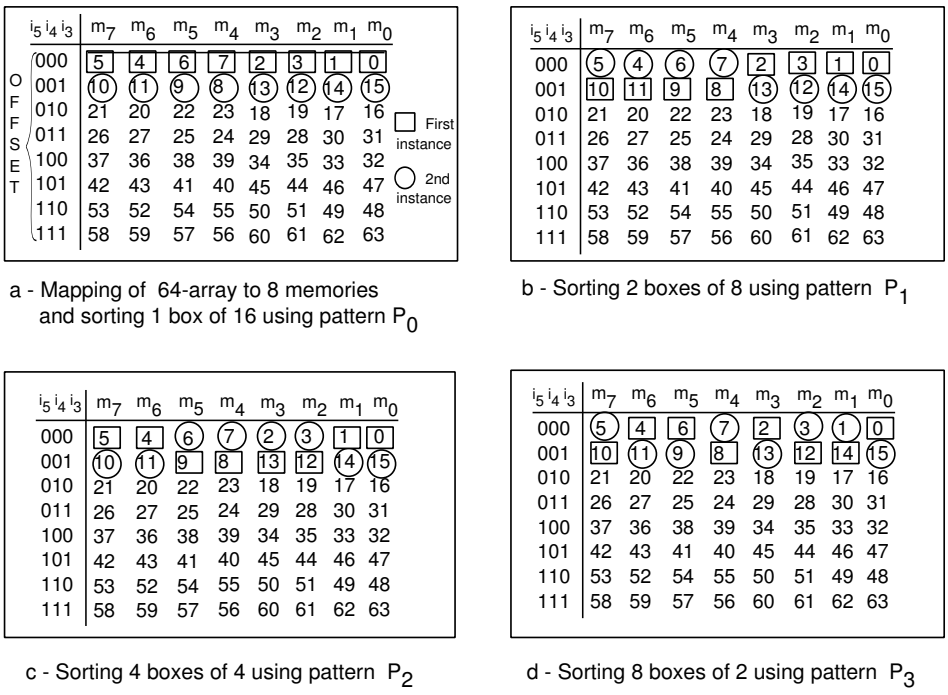

Figure 2: Mapping a 64-element array to 8 memories and first two pattern instances

as $(0, 1, 2, 3, 8, 9, 10, 11)$. Pattern P_2 allows accessing four groups of 2-element with a stride of 2 such as $(0, 1, 4, 5, 8, 9, 12, 13)$. Pattern P_3 allows accessing eight groups of 1-element with a stride of 2 such as $(0, 2, 4, 6, 8, 10, 12, 14)$.

Figure 3-(a) shows the access to first two instances of P_0 which is useful for sorting elements across two 8-elements. Figures 3-(b), (c), and (d) show the access to first and second instances of patterns P_1, P_2, and P_3, respectively.

Figure 4 shows the columns of array addresses that fall into of the eight memories by using the above storage scheme, i.e. $d(i) = Mi$ indicates that the ith array element falls into memory $M_{d(i)}$. In Figure 2-(a) a 64-element array can be partitioned into 8 rows each represents one possible instance of P_0. In the example the identifier of pattern instance is (i_5, i_4, i_3) which is used as the memory offset. To access a given instance of pattern P_0, each PE_k is to access the kth element of the pattern instance. For this PE_k $(0 \leq k \leq 7)$ generates the array address $(i_5, i_4, i_3, k_2, k_1, k_0)$ where $k = (k_2, k_1, k_0)$. and (i_5, i_4, i_3) identifies one pattern instance out of 8 possible instances of P_0. As part of address translation PE_k finds the memory where the kth element of the pattern instance is stored by evaluating $d(k) = M.(i_3, k_2, k_1, k_0)^t$ as shown in Figure 4-(a). Each PE_k issues its the storage number $d(k)$ to the kth input of the MIN which is used as a header in the process of self-routing to establish a route from PE_k to $M_{d(k)}$. Note that the jth output of the MIN is connected to the jth memory M_j. In this way each PE_k maps to the memory where the kth element of the pattern instance is stored as shown in Figure 4-(b). Next each PE_k sends the offset (i_5, i_4, i_3) to $M_{d(k)}$ over to established route through the MIN. The memories are simultaneously accessed and the accessed data is returned back over the same route the requesting PE.

During an access to an instance of P_0, the group of bits (i_2, i_1, i_0) takes all possible binary combinations and the remaining bits of i are constant. We define the basis B_0 of pattern P_0 as the set of canonical vectors $B_0 = \{v_2, v_1, v_0\}$ whose components identify the elements

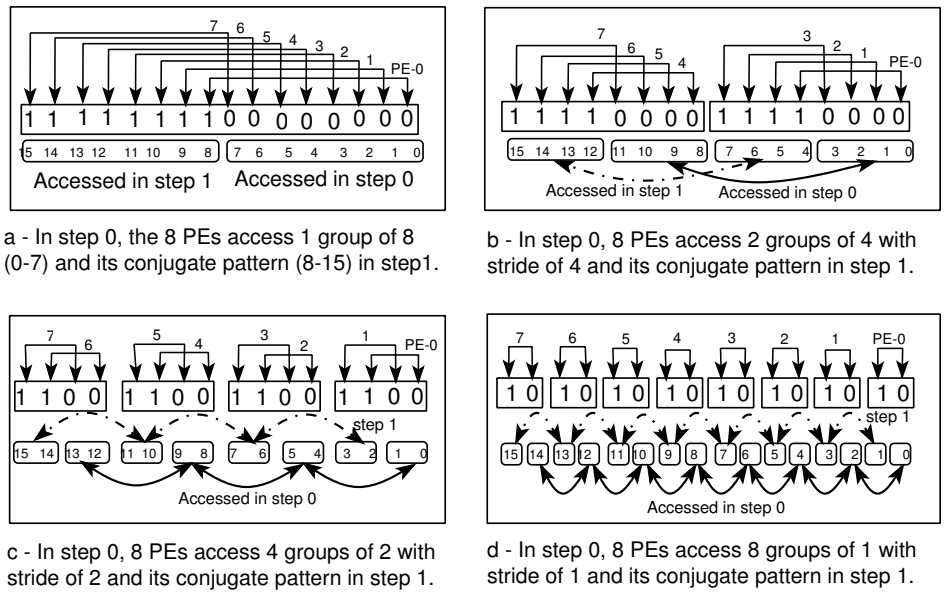

Figure 3: Sorting 16 items using 8 PEs and 4 data access patterns

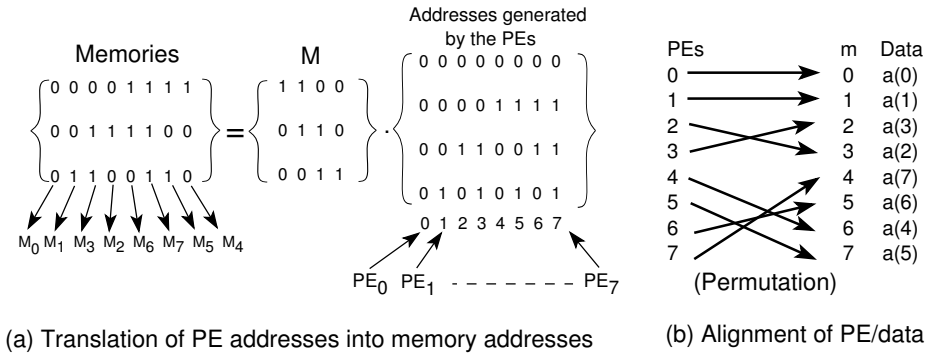

Figure 4: Mapping array elements to memories (a) and required permutation (b)

of a given pattern instance. Similarly, the same idea applies to groups (i_3, i_1, i_0), (i_3, i_2, i_0), and (i_3, i_2, i_1) when accessing an arbitrary instance of P_1, P_2, and P_3, respectively. For each accessed pattern, the PE numbers identifies the array element within the accessed pattern instance and the remaining address bits represent the pattern instance offset. In other words, patterns P_1, P_2, and P_3 have bases $B_1 = \{v_3, v_1, v_0\}$, $B_2 = \{v_3, v_2, v_0\}$, and $B_3 = \{v_3, v_2, v_1\}$, respectively.

Since array element $a(i)$ is stored into memory $m_{d(i)}$ defined by $d(i) = Mi$ for all patterns. The union of all pattern bases is $\{v_0, \ldots, v_3\}$ which indicates that the storage matrix M is 3×4. In general the storage matrix M is an $n \times p$ matrix, where p is the number of distinct canonical vectors in the union of all pattern bases.

4 Analysis of parallel memory organization

Networks are one of the fundamental factors in the design of multiprocessor systems which connect processors to parallel memories or provide data links among the processors. Conflicts within the network are responsible for performance degradation because serializing accesses leads to an increase in processor idle time. The efficiency of processor-memory networks requires: 1) that processors' requests be uniformly distributed over the memories and, 2) that network must align the accessed/retrieved data with respect to the processors.

We study a class of dynamic, full access, unique path, multistage networks that use 2×2 switching elements (SEs), and $N = 2^n$ inputs and outputs for each of the n stages. Inter-stage interconnection represents some permutation. There are 2^{n-1} switches in each stage. Each switch has two states, *straight* and *exchange* and, therefore, can perform $(N^N)^{1/2}$ permutations each corresponding to one state of the $n2^{n-1}$ switches. Such networks are called *blocking* because they cannot perform all possible $(2^n)!$ permutations of the inputs.

A network can perform a given permutation π if there exists a setting of the switches in the network such that the ith input is connected to the $\pi(i)$th output, where π is defined over the integers $\{0, \ldots, 2^n - 1\}$. A crossbar switch can achieve all possible permutations but its drawback is the cost of the $(2^n)^2$ switches required.

Self-routing a source $s = s_{n-1} \ldots s_0$ to destination $d = d_{n-1} \ldots d_0$ consists of finding a path of switching elements that connect s to d. Each SE (straight or exchange) receives two incident sources s and s', with destinations d and d', on its upper and lower inputs, respectively. In the ith stage, the routing bit (d_i) directs the message to either the upper $(d_i = 0)$ or lower output $(d_i = 1)$. Each network uses a specific destination bit for self-routing at a specific stage. A collision occurs at the switch when both incident messages require exiting the switch at the same output.

In an n-stage Omega network (Ω_n), the yth output link O_y of each stage is connected to the xth input link I_x of the next stage such that $y = \sigma(x)$, where $x = x_{n-1}x_{n-2} \ldots x_0$ and $\sigma()$ is the perfect shuffle permutation defined by $\sigma(x_{n-1}x_{n-2} \ldots x_0) = x_{n-2} \ldots x_0 x_{n-1}$. This finds the position of a message at the output of the ith stage:

$$pos_i(s, d) = s_{n-i-1} \ldots s_0 d_{n-1} \ldots d_{n-i+1} d_{n-i} \tag{3}$$

We can similarly find the position of the message after the ith stage for other multistage networks such as Baseline, Cube, Delta, etc. Note that for all unique path multistage networks the position of a message at the output of the ith stage is a combination of bits of the source and destination.

A network input (source) is denoted by $s = (s_{n-1}, \ldots, s_0) \in S$ and a network output (destination) is denoted by $d = (d_{n-1}, \ldots, d_0) \in S$, where $S = \{0, 1, \ldots, 2^n - 1\}$. A linear permutation [3] is an $n \times n$ NS Boolean matrix $M : S \rightarrow S$ for which sources and destinations are one-to-one. In particular, each destination d is the image of one single source s where $d = Ms = (d_{n-1}, \ldots, d_0)$.

We wish to know under what condition an Ω_n network can perform permutation M, where M is an $n \times n$ Boolean matrix. From Equation 1 we find:

$$pos_i(s, Ms) = (s_{n-i-1}, \ldots, s_0, d_{n-1}, \ldots, d_{n-i}) \tag{4}$$

Vector $pos_i(s, d)$ is the position of the message going from s to $d = Ms$ following the ith stage. We shall abbreviate $pos_i(s, d)$ to $pos_i(s)$ which can be written as a matrix product

$pos_i(s) = \tilde{M}[i]s$, where $\tilde{M}[i]$ defines the permutation of the inputs s to the outputs $pos_i(s)$ of the ith stage. Using Equation 4 we have:

$$pos_i(s) = \begin{pmatrix} s_{n-i-1} \\ \vdots \\ \vdots \\ s_0 \\ d_{n-1} \\ \vdots \\ \vdots \\ d_{n-i} \end{pmatrix} = \begin{pmatrix} 0 & \cdots & 0 & 1 & \cdots & 0 \\ . & \cdots & . & 0 & \cdots & . \\ . & \cdots & . & . & \cdots & . \\ . & \cdots & . & . & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 1 \\ a_{n-1,n-1} & \cdots & a_{n-1,n-i} & a_{n-1,n-i-1} & \cdots & a_{n-1,0} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-i,n-1} & \cdots & a_{n-i,n-i} & a_{n-i,n-i-1} & \cdots & a_{n-i,0} \end{pmatrix} . \begin{pmatrix} s_{n-1} \\ \vdots \\ \vdots \\ s_{n-i} \\ s_{n-i-1} \\ \vdots \\ \vdots \\ s_0 \end{pmatrix} \tag{5}$$

where the $(n-i) \times i$ matrix in the upper-left corner is formed by 0s, the $(n-i) \times (n-i)$ matrix in the upper-right corner is the identity, the $i \times (n-i)$ matrix in the lower-right corner will be denoted by $B[i]$, and the $i \times i$ matrix in the lower-left corner will be denoted by $M[i]$:

$$M[i] = \begin{pmatrix} a_{n-1,n-1} & \cdots & a_{n-1,n-i} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ a_{n-i,n-1} & \cdots & a_{n-i,n-i} \end{pmatrix} \tag{6}$$

The permutation matrix M that gives the position $pos_n(s) = d = Ms$ of the message at the output of the nth stage is defined as follows:

$$M = \begin{pmatrix} a_{n-1,n-1} & \cdots & a_{n-1,n-i} & a_{n-1,n-i-1} & \cdots & a_{n-1,0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-i,n-1} & \cdots & a_{n-i,n-i} & a_{n-i,n-i-1} & \cdots & a_{n-i,0} \\ a_{n-i-1,n-1} & \cdots & a_{n-i-1,n-i} & a_{n-i-1,n-i-1} & \cdots & a_{n-i-1,0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{0,n-1} & \cdots & a_{0,n-i} & a_{0,n-i-1} & \cdots & a_{0,0} \end{pmatrix} \tag{7}$$

Note that $M[i]$ is the $i \times i$ sub-matrix in the upper-left corner of M. It can be easily shown that all inputs of the ith stage map one-to-one to all outputs of the same stage if and only if $M[i]$ is NS [18, 25]. The reason is that $\tilde{M}[i]$ is NS if and only if $M[i]$ is NS. In other terms two messages issued at distinct sources exit the ith stage at distinct output links if $M[i]$ is NS. We now characterize the linear permutations M which the Omega network can perform. An $n \times n$ permutation matrix M is said to be *strongly-non-singular* (SNS) when sub-matrix $M[i]$ is NS for arbitrary i, where $0 \leq i \leq n-1$.

The set of sources S maps one-to-one to all the outputs of every stage i because $M[i]$ is NS, where $1 \leq i \leq n$. In other words, if all messages exit all stages at distinct outputs, then all sub-matrices of M, starting from the upper-left corner, are NS. Therefore, Ω_n can achieve permutation M without conflict whenever M is SNS for Ω_n.

The above results can be generalized [2] to other multistage networks such as Baseline, Cube, Delta, etc. The reason is that for each of these networks the position of a message at the output of some stage admits a formulation that is similar to Equation 3 which gives

similar results to Equations 5, 6, and 7. Therefore, an arbitrary, dynamic, full access, unique path, multistage network (Γ) can achieve an arbitrary linear permutation defined by an $n \times n$ Boolean matrix M if and only if M is SNS for Γ.

The location of sub-matrices $M[i]$, within M, can be different for each type of network [2]. For an Omega network, sub-matrices $M[1]$, $M[2], \ldots, M[n]$ are square matrices that start at the upper-left corner of M and move along the diagonal down to the lower-right corner. Then Ω_n can achieve the identity permutation ($M = I_n$). For a Baseline network, the same sub-matrices start at the upper-right corner of M and expand along the anti-diagonal up to the upper right corner of the matrix.

Generally, to each $n \times n$ permutation matrix M and each $n \times 1$ constant vector x we can define a *complement-permutation* (CP) [3, 20] which is of the form $\tilde{d} = Ms \oplus x$, where s and $\tilde{d}$ represent the $n \times 1$ source and destination vectors, respectively. The permutation $d = Ms$ can then be seen as a special case of CP. A network can achieve permutation $d = Ms$ whenever M is SNS for that network. It has been established [3, 20, 2] that a necessary and sufficient condition for an Ω network to achieve an arbitrary CP defined by $\tilde{d} = Ms \oplus x$ exists if and only if M is SNS for Ω.

Depending on the location of SNS sub-matrices within M, there exist specific SNS matrices [2] for each type of single path network for which the above necessary and sufficient condition is satisfied. Therefore, the above results are true for an arbitrary single path multistage network. Moreover, the set of all CPs ($\tilde{d} = Ms \oplus x$) associated with all SNS matrices M and all values of x are distinct [2]. The number of permutations $\tilde{d} = Ms \oplus x$ that an arbitrary n-stage multistage network can perform is $T_n = 2^{n^2}$, where M is an $n \times n$ SNS matrix and x is an $n \times 1$ arbitrary vector (see the appendix for a proof).

It can be shown [2] that the problem of finding an a memory and network conflict-free storage scheme for a given set of data access patterns is tractable for $n = 2$, but NP-complete for $n > 2$. The above problem is NP-hard for n=3. This problem corresponds to 2^{n-1}-coloring which is an NP-complete problem for $n \geq 3$.

5 Multi-pattern access using SNS storages

Recall the storage scheme that was defined by Eq. 2 in Section 3. When accessing P_0, function $d(i) = Mi$ can be decomposed into the following sum:

$$d(i) = \overset{\begin{matrix} c_3 & c_2 & c_1 & c_0 \end{matrix}}{\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}} . \begin{pmatrix} i_3 \\ i_2 \\ i_1 \\ i_0 \end{pmatrix} = \overset{\begin{matrix} c_2 & c_1 & c_0 \end{matrix}}{\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}} . \begin{pmatrix} i_2 \\ i_1 \\ i_0 \end{pmatrix} \oplus \overset{\begin{matrix} c_3 \end{matrix}}{\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}} . i_3 = M_{p_1} . \begin{pmatrix} i_2 \\ i_1 \\ i_0 \end{pmatrix} \oplus x \tag{8}$$

where M_{P_1} is the left-hand matrix of the sum and the right-hand term is constant (x) when accessing P_1. M_{P_1} is said to be M *restricted to pattern* P_1. Equation 8 is a complement-permutation whenever M_{P_1} is strongly-non-singular. The processors numbers $(s_0, s_1, \ldots, s_7)$ are identified with the values taken by (i_2, i_1, i_0), (i_3, i_2, i_1) and (i_4, i_3, i_2) during an access to some instance of P_1, P_2, and P_3, respectively. Figure 4 shows the mapping of data elements to memories by M_{P_0}. The ith element of an instance of P_0 is mapped to a unique memory $M_{P_0} i$ because M_{P_0} is NS (Figure 4-a). To align each PE_i with memory $M_{P_0} i$, where the

ith element is stored, the network should be able to achieve the permutation (Figure 4-b). The above permutation is conflict-free for Ω_3 because M_{P_0} is SNS for Ω_3. The permutation matrices associated with P_1, P_2, and P_3 are:

$$M_{P_1} = \begin{array}{c} \begin{matrix} c_3 & c_2 & c_1 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array} \quad M_{P_2} = \begin{array}{c} \begin{matrix} c_4 & c_3 & c_2 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \quad M_{P_3} = \begin{array}{c} \begin{matrix} c_4 & c_3 & c_2 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \tag{9}$$

Since (i_2, i_1, i_0) takes all possible binary values when accessing P_1, the product $M_{P_1} i$ also takes all possible binary values if and only if M_{P_1} is non-singular. This causes the array elements that belong to any instance of P_1 to be distributed in a skewed form over the memories because no two elements of a given instance are associated with the same memory. Mapping the array elements of each data pattern to the prescribed PEs requires realigning the elements within the network. The complement-permutation shown in Equation 8, for accessing P_1, guarantees proper alignment (conflict-free) over an Ω_3 because the permutations M_{P_1}, M_{P_2}, and M_{P_3} have been chosen to be strongly-non-singular for Ω_3. This shows that the storage scheme defined by M guarantees conflict free access to the network (realignment) and the memories in accessing any instance of patterns P_1, P_2, and P_3.

A multi-pattern storage scheme that uses SNS matrices for each power-of-2 data pattern can be synthesized according to the following steps:

- Restrict vector i to its components over the vectors of the union of all pattern bases. In the case of multi-dimensional arrays, block patterns of arbitrary power-of-2 size and power-of-2 strides can also be defined by restricting and intermixing the changing indices of different dimensions. In other words, accessing patterns in multi-dimensional arrays is the same as accessing multi-stride data patterns in 1-D arrays.

- Synthesize the restricted matrices $M_{P_1}, \ldots, M_{P_q}$ of M to each data pattern so that each matrix M_{P_i} is SNS for the given network.

Suppose M is a storage scheme for a given array which is to be accessed by a set $P = \{P_1 \ldots P_t\}$ of data patterns. Ideally, each instance of each pattern should be: 1) distributed over distinct memories and, 2) accessed by a given network without conflicts. Note that sub-matrix M_{P_i} must be SNS for each pattern P_i with respect to the network in order to map all elements of any instance of P_i into distinct network outputs (also memories). Then [2] M is conflict-free and network-contention-free for an arbitrary multistage network if and only if all M_{P_i}, for $1 \leq i \leq t$, are SNS for the network.

The performance of a storage scheme M depends on the rank of each of its sub-matrices associated to its data patterns. The rank of a restricted matrix M_P to pattern P gives the number of clocks needed to access each instance of P, where M_P is $n \times n$. In general, if $rank(M_P) = k \leq n$, then $2^{n-k} f(P)$ cycles will be required for $f(P)$ accesses to instances of P. The number of access cycles $C(M)$ for the combined storage scheme M is the sum of the access cycles of all of its q patterns $P_1, \ldots, P_q$. If each pattern P is accessed $f(P)$ times, then $C(M)$ is:

$$C(M) = \sum_{i=1}^{q} f(P_i) 2^{n - rank(M_{P_i})} \tag{10}$$

Therefore, the knowledge of the storage scheme allows computing the number of cycles needed for accessing a given instance of each of its data patterns, i.e. predictable access time for the set of data patterns.

6 Heuristic approach

We present an efficient algorithm for finding a linear storage scheme for a given pattern set, if one exists. We derive a practical algorithm for finding approximate linear schemes in which some patterns may not be accessed conflict-free.

The idea is to construct the $n \times p$ storage matrix M one row at a time, from top to bottom in the case of an Ω network. Note that M restricted to pattern P_i is an $n \times n$ which we denote by M_{P_i}. For pattern P_i, we denote by $M_{P_i}[j]$ the upper left $j \times j$ sub-matrix of M_{P_i}. We assume that, for each pattern P_i, the matrix $M_{P_i}[j]$ is SNS and attempt to construct the row $M_{j+1,*}$ so that each $M_{P_i}[j+1]$ is SNS. We the idea of the proof of Theorem 1 (see Appendix), in the construction.

We illustrate the algorithm by an example with $n = 3$. Suppose we are given patterns P_1, P_2, P_3, and P_4 with bases $\tilde{P}_1 = \{v_2, v_1, v_0\}$ $\tilde{P}_2 = \{v_3, v_2, v_1\}$ $\tilde{P}_3 = \{v_5, v_4, v_3\}$, and $\tilde{P}_4 = \{v_4, v_3, v_1\}$.

Now we construct the upper two rows of M. For each M_{P_i}, we wish to ensure that the 2×2 sub-matrix in its upper-left corner is SNS. We can accomplish this by constructing a matrix for the reduced pattern set $\tilde{P}'_1 = \{v_2, v_1\}$, $\tilde{P}'_2 = \{v_3, v_2\}$, $\tilde{P}'_3 = \{v_5, v_4\}$, and $\tilde{P}'_4 = \{v_4, v_3\}$.

Note that $\tilde{P}'_i$ is just $\tilde{P}_i$ with the lowest ordered vector removed. The vectors appearing first in some pattern basis $\tilde{P}'_i$ are $X = \{v_5, v_4, v_3, v_2\}$. The vectors appearing only second in some pattern are $Y = \{v_1\}$. Finally, v_0 does not appear first or second and thus can be assigned any value. Since v_0 does not appear in any $\tilde{P}'$, we arbitrarily assign it $\binom{1}{1}$. Let's denote by H-2 the above algorithm to assign the upper two rows of M.

The upper two rows of M have been determined. We let $x_5 \ldots x_0$ be the values in the lowest row:

$$M = \begin{array}{c} \begin{matrix} v_5 & v_4 & v_3 & v_2 & v_1 & v_0 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \end{pmatrix} \end{array} \qquad M_{P_1} = \begin{array}{c} \begin{matrix} v_2 & v_1 & v_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ x_2 & x_1 & x_0 \end{pmatrix} \end{array} \tag{11}$$

We must now ensure that each 3×3 matrix that corresponds to some pattern is NS. We must assign x_2, x_1, and x_0 in such a way that the matrix M_{P_1} is NS. The first step is to get the identity matrix in the upper right 2×2 sub-matrix of M_{P_1} using only row operations. This is satisfied for M_{P_1}. Using the notation of Theorem 1 (Appendix), we have $b_0 = x_1$, $b_1 = x_2$, $c = x_0$, $a_0 = 1$, and $a_1 = 1$. Matrix M_{P_1} is NS if and only if $b_2 x_2 \oplus b_1 x_1 x_0 = 1$, where $(b_2, b_1, x_0 = 1)$ is the right 3×1 column of M_{P_1}. In other words, matrix M_{P_1} will be non-singular, and thus strongly-non-singular, if and only if $x_2 \oplus x_1 \oplus x_0 = 1$.

Now consider M_{P_2}:

$$M_{P_2} = \begin{array}{c} \begin{matrix} v_3 & v_2 & v_1 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ x_3 & x_2 & x_1 \end{pmatrix} \end{array} \tag{12}$$

We need to get the identity matrix in the upper-left 2×2 sub-matrix, using only row operations. If column operations are used, then the values of the x's will be affected. We achieve this by adding row 1 to row 2 and, next, we add row 2 to row 1. Note that we do not want to change the original matrix, and so we denote this reduced matrix $\tilde{M}_{P_2}$. We now have the matrix in the format specified by Theorem 1:

$$\tilde{M}_{P_2} = \begin{array}{c} \begin{matrix} v_3 & v_2 & v_1 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ x_3 & x_2 & x_1 \end{pmatrix} \end{array} \tag{13}$$

For M_{P_2} to be non-singular, we must have $x_3 \oplus x_2 \oplus x_1 = 1$. The remaining conditions for M_{P_3} and M_{P_4} are $x_5 \oplus x_3 = 1$ and $x_4 \oplus x_3 \oplus x_1 = 1$, respectively.

One solution for this system of simultaneous equations is:

$$x_0 = 0, \quad x_1 = 1, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 0, \quad x_5 = 1 \tag{14}$$

So the final matrix is:

$$M = \begin{array}{c} \begin{matrix} v_5 & v_4 & v_3 & v_2 & v_1 & v_0 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{15}$$

The remaining conditions for M_{P_2}, M_{P_3}, and M_{P_4} are $x_3 \oplus x_2 \oplus x_1 = 1$, $x_5 \oplus x_3 = 1$, and $x_4 \oplus x_3 \oplus x_1 = 1$, respectively. One solution for this system of simultaneous equations is $x_0 = 0$, $x_1 = 1$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$, and $x_5 = 1$, which gives a solution for M for which all pattern sub-matrices are SNS for Ω_3. The permutations corresponding to M_{P_i}, where $1 \leq 4$, are all SNS matrices which indicates that they: (1) map into distinct storages, and (2) their permutations can be achieved in an Ω_3 network without conflict.

The general algorithm (H-n) is as follows:

1. Determine the upper two rows of the matrix using algorithm H-2.

2. Create each remaining row, working from top to bottom.
   For i in 2 to $n-1$ loop:

   (a) For each pattern P_j:

      i. Obtain a matrix $\tilde{M}_{P_j}$ by reducing the matrix M_{P_j} so that it has the identity matrix (Omega network) in its upper-left corner, using only row operations which do not affect the matrix M.

      ii. Use the ith column of this matrix to determine the equation associated with this pattern. Let the basis of P_j be $v_{\ell_{n-1}} \ldots v_{\ell_0}$, and $y_k = (\tilde{M}_{P_j})_{n-k-1,\ell_i}$. Then the equation is:

$$x_{\ell_i} \oplus \Sigma_{k=0}^{i-1} x_{\ell_k} y_k = 1 \tag{16}$$

   (b) Solve the system of simultaneous equations. Assign entry $M_{n-i-1,k}$ the value x_k.

In this algorithm, we do not need to perform row reduction from the original matrix in Step i. If we have available the result of the previous iteration, we can row-reduce this partially reduced matrix, and reduce the time complexity of this step from $O(n^3)$ to $O(n^2)$.

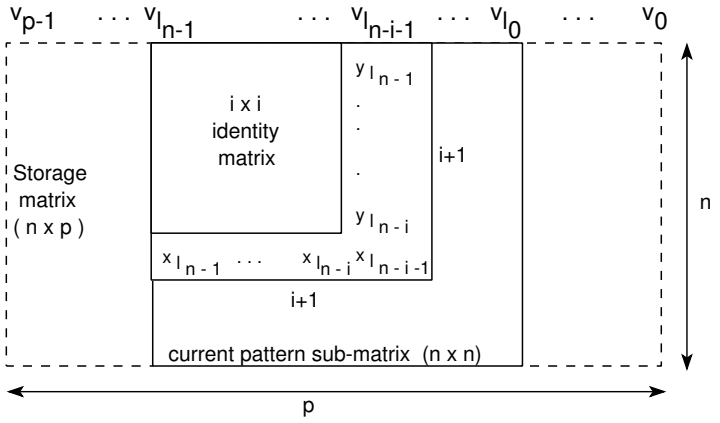

Figure 5: Progressive building of the $(i+1) \times (i+1)$ pattern sub-matrix after reducing the $i \times i$ previous sub-matrix

Note that in Step 1, there may be several different ways H-2 could color the conflict graph which affects the ability of this algorithm to find a solution in Step 2. In Step (2-b), the set of equations may not have a unique solution. The selection of a solution may affect the ability of the algorithm to find a solution for a later row. Since there are potentially several alternatives at Steps 1 and (2-b), one possibility is to use backtracking to search exhaustively for a solution. It is also possible to use heuristics to guide the selection at these steps, with or without backtracking.

We analyze the time complexity of algorithm H-n, in the case where no backtracking is allowed. Algorithm H-n makes an initial call to algorithm H-2, and then runs in $n-2$ phases. Let t be the number of patterns. Each phase constructs t equations. Constructing each equation requires $O(n^2)$ time, if we use the partially reduced matrix from the previous phase. The total construction time is $O(tn^2)$. Solving the resulting set of simultaneous equations can be done in $O(tp^2)$ time, since we have t equations in p unknowns. Since $n \leq p$, the total time required for a phase is $O(tp^2)$. The total complexity of algorithm H-n is therefore $O(ntp^2)$, where t, 2^n, and p, are the number of patterns, the number of memories, and the number of distinct vectors of the pattern bases, respectively.

7 Applications

In this section we present applications of the proposed scheme to the design of embedded memory systems for: (1) parallel FFT using arbitrary MINs, (2) parallel bitonic sorting, (3) multiple non power-of-2 stride access, and (4) improving cache data re-use using parallel memories.

7.1 Application to FFT

Assume 2^n memories interconnected to 2^n PEs by using an $2^n \times 2^n$ Ω network. Assume an $n \times n$ matrix M_Ω that is SNS for Ω and consider the storage scheme $j = M_\Omega i$ which maps array element $a(i)$ to memory m_j without memory conflicts or network conflicts (see Section 4). A complement permutation CP_x from the PEs onto the memories defined by

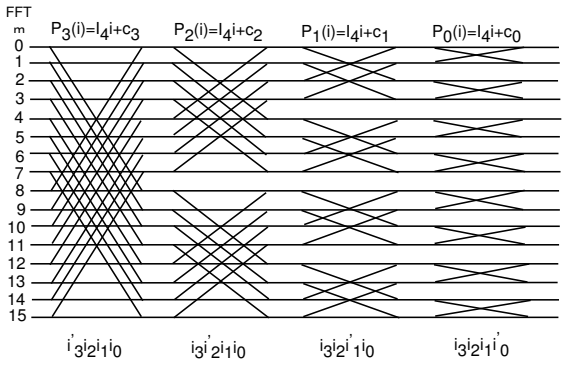

Figure 6: Permutations required for FFT over Ω_4

$J = M_\Omega k + P_\Omega$ maps each PE_k onto memory m_J, where P_Ω is a given $n \times 1$ constant vector. As stated in Section 4, the permutation $M_\Omega k + P_\Omega$ is achievable on Ω because matrix M_Ω is SNS for Ω. Since $a(i)$ belongs to m_j whenever $j = M_\Omega i$ then PE_k is mapped to array element $a(k + M_\Omega^{-1} P_\Omega)$ by permutation CP_x.

Consider a distinct $2^n \times 2^n$ MIN denoted by Γ and let $l = M_\Gamma q$ be the storage scheme that maps array element $a(q)$ to memories m_l, where M_Γ is SNS for Γ. In the following we show how CP_x can be made achievable for Γ. Consider the permutation from the PEs onto the memories defined by $L = M_\Gamma k + M_\Gamma M_\Omega^{-1} P_\Omega$ which maps PE_k to memory m_L. As $a(q)$ is stored into memory $l = M_\Gamma q$, then PE_k accesses m_L which contains array element $a(M_\Gamma^{-1} L) = a(k + M_\Omega^{-1} P_\Omega)$. Thus Γ can achieve the same CP_x permutation that was previously defined over Ω provided that $a(q)$ is stored into memory $l = M_\Gamma q$ and PE_k uses the permutation $M_\Gamma k + M_\Gamma M_\Omega^{-1} P_\Omega$. In other words, a CP defined for one MIN can also be equally achieved for other MINs by using the above scheme that converts achievable CPs for one MIN to equivalent CPs for another MIN.

In the following we consider one implementation of a 16-point FFT over $N = 16$ memories/PEs. The permutation needed for FFT are defined by $k + 2^u$ modulo N. Each of the $k + 2^u$ modulo N permutation is a CP defined by $I_4 k + b_u$, where $b_3 = (1000)$, $b_2 = (0100)$, $b_1 = (0010)$, and $b_0 = (0001)$ and I_4 is the 4×4 identity matrix. Assume data $a(i)$ is stored into memory m_j such that $j = M_\Omega i$, where $0 \leq i, j \leq 15$. This is shown on Figure 6. To access element $k + b_u$ the permutation to be performed by PE_k must be $P_u(k) = M_\Omega k + c_u$, where $c_u = M_\Omega b_u$ and $0 \leq u \leq 3$. One possible implementation of the storage matrix M_Ω and the corresponding values of constant vectors c_us are the following:

$$M_\Omega = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad c_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad c_2 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad c_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad c_0 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \qquad (17)$$

The scheme of Figure 6 is valid only for Ω_4 because I_4 is SNS for Ω_4. A 16-input Baseline network is defined by $B_4 = E_1 \sigma_4^{-1} E_2 \sigma_3^{-1} E_3 \sigma_2^{-1} E_4$, where E_i is the ith stage. This implies that the position of a message, issued at $s = s_3 s_2 s_1 s_0$, is $pos_1 = s_3 s_2 s_1 d_3$, $pos_2 = d_3 s_3 s_2 d_2$, $pos_3 = d_3 d_2 s_3 d_1$, and $pos_4 = d_3 d_2 d_1 d_0$ at the output of the stages. We note that I is not SNS for B_4. With the idea of algorithm H-n we can build boolean matrices that are SNS for arbitrary MINs so that $pos_i = M_i s$ (see Sections 4 and 6). To guarantee conflict-free

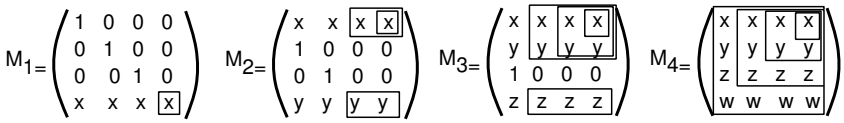

Figure 7: Non-singular matrices for B_4

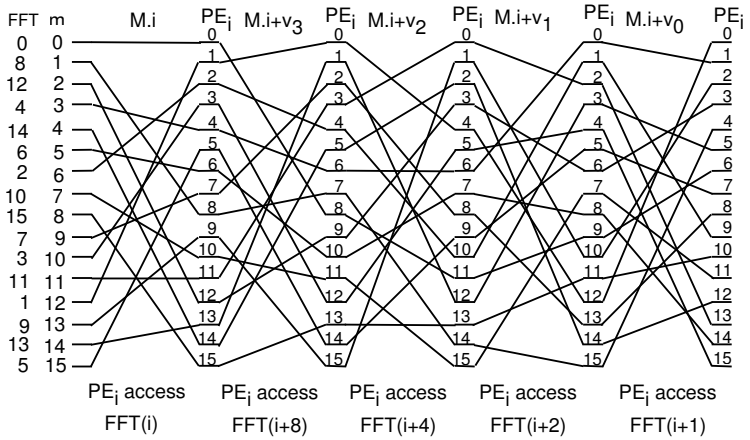

Figure 8: Permutations required for FFT over B_4

access to the network and the memories we need M_i to be an SNS matrix for the target network. The square sub-matrices within the frames shown in Figure 7 are to be chosen as NS. Matrix $M = M_4$ is SNS for B_4 if and only if all the framed sub-matrices are NS. Thus the permutations needed for B_4 are of the form $H_u(k) = Mk + v_u$, where M is SNS for B_4 and v_u is some constant vector. To achieve permutations on the B_4 which are identical to the above defined permutations for Ω_4, PE_k only needs to use the permutations $H_u(k) = M_{B_4}k + M_{B_4}M_\Omega^{-1}P_\Omega$, where M_{B_4} is any 4×4 Boolean matrix that is SNS for B_4. This finds $v_u = M_{B_4}M_\Omega^{-1}c_u$ for each c_u instance of P_Ω. In other words, PE_k is to use the permutations $H_u(k) = M_{B_4}(k + M_\Omega^{-1}c_u) = M_{B_4}(k + b_u)$. where $b_u = M_\Omega^{-1}c_u$.

One possible solution for achieving the four CPs $H_u(k) = M_{B4}(k + b_u) = M_{B4}k \oplus v_u$, for $0 \le u \le 3$, $0 \le k \le 15$, and $v_u = M_{B4}b_u$. With the idea of algorithm H-n we construct the storage matrix M_{B4}:

$$M_{B4} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad v_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad v_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \quad v_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \quad v_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad (18)$$

The permutations $H_u(k) = M_{B4}k + v_u$ are shown in Figure 8, where the columns denoted by (a(i)) and (m) denote the data elements and memory-PE numbers, respectively. Each PE_k accesses data element $a(k + 2^u)$ by using the CP $M_{B4}k + v_u$ which is also conflict-free for B_4 because M_{B4} is SNS for B_4. Data elements are mapped to distinct memories because B_4 is NS. All the CPs H_u are conflict-free for B_4 because M is SNS. One may extend the above solution to an inverse baseline IB_4 by setting $H_u(k) = M^{-1}k \oplus w_k = M^{-1}(k \oplus v_k)$ where $w_k = M^{-1}v_k$.

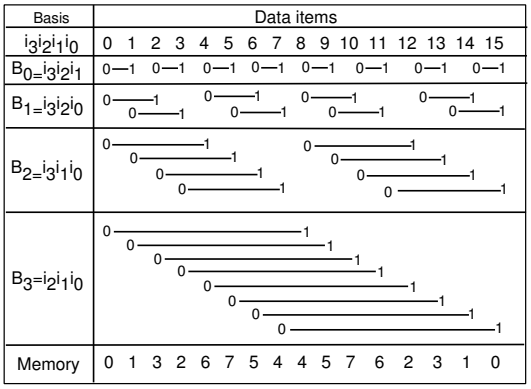

Figure 9: Permutations required for sorting 16-items over 8 memories

This example shows how one can convert linear permutations for implementing FFT from one network to another. Since some permutations can only be achieved on some networks, the benefit of our approach is to reallocate the data so that the needed permutations become achievable for each multistage network. The above approach can be used to convert FFT algorithms for one network to another by simply modifying the address generation scheme which can be supported by hardware or simply through software emulation.

7.2 Sorting

In the following we show how data patterns can be synthesized for sorting a 16-point array by using 8 memory/PE pairs and an Ω_3 network. Bitonic sorting can be implemented on the basis of the four access groups shown in Figure 9. For each group, every processor is to read a pair of items from memory, sort them, and store the sorted items back into memory. Each pair of items (0-1) involved in one group are connected by an edge in Figure 9. The problem is to find the mapping of data elements to memories and the implied access patterns.

Two accesses are needed for each group: 1) the left-hand items which are all accessed in step 0 and, 2) the right-hand items which are all accessed in step 1. This finds the bases of the access pattern for each group. The bases are $B_0 = \{v_3v_2v_1\}$, $B_1 = \{v_3v_2v_0\}$, $B_2 = \{v_3v_1v_0\}$, and $B_3 = \{v_2v_1v_0\}$ and correspond to access patterns (P_0, P_1, P_2, P_4) of the four groups which are shown in Figure 3 and 9. For an Ω_3 network, the leading NS matrices are located at the upper left edge of the storage matrix. With the idea of algorithm H-2 we may assign $\binom{11}{01}$ as the upper left corner of the storage matrix. We may use algorithm H-n to complete the previous sub-matrix and find a solution for which each of the restricted sub-matrices (M_{P_j}) is SNS:

$$M = \begin{array}{c} \begin{matrix} v_3 & v_2 & v_1 & v_0 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{19}$$

Each data element $a(i)$, for $0 \leq i \leq 15$ is stored into memory Mi at a selected offset $off = i_3$ which is constant for each access of an instance of P_3 because $B_3 = \{v_2v_1v_0\}$. This is shown in Table 2. The distribution of 64 elements into parallel memory is shown on Figure 2. The selected offset causes the offsets of all distinct array elements that fall into same memory to be distinct because M_{P_3} is SNS. The array address generated by the PE for

Items/Patterns	Each PE_k fetches
items 0 of P_0	$a(k_2k_1k_00)$
items 1 of P_0	$a(k_2k_1k_01)$
items 0 of P_1	$a(k_2k_10k_0)$
items 1 of P_1	$a(k_2k_11k_0)$
items 0 of P_2	$a(k_20k_1k_0)$
items 1 of P_2	$a(k_21k_1k_0)$
items 0 of P_3	$a(0k_2k_1k_0)$
items 1 of P_3	$a(1k_2k_1k_0)$

Table 1: Address generation for each pattern access

Mem	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
$a(i)$	0	1	3	2	7	6	4	5
	15	14	12	13	8	9	11	10
P_0	0	1	1	0	1	0	0	1
	1	0	0	1	0	1	1	0
P_1	0	0	1	1	1	1	0	0
	1	1	0	0	0	0	1	1
P_2	0	0	0	0	1	1	1	1
	1	1	1	1	0	0	0	0
P_3	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1

Table 2: Storage scheme and access patterns

each pattern is shown in Table 1.

Table 2 shows the storage of the data elements by M and the first (marked by 0) and second (marked by 1) access to each pattern P_j. The first and second entry of the data $a(i)$ is addressed within each memory by its offset (0 or 1). The elements of each pattern P_j map to distinct memories because M_{P_j} is NS. The elements of each P_j can be aligned to the PEs without conflict because each M_{P_j} is SNS for Ω_3.

This approach can easily be adapted to sorting arbitrary power-of-2 numbers of data elements and arbitrary power-of-2 numbers of memories. The advantage of our storage scheme is the ability to synthesize the needed memory conflict-free mapping and conflict-free network access in one compact address transformation (M), a task that our algorithm can perform for arbitrary multistage network.

7.3 Optimizing for arbitrary stride access

In the following we examine finding storage schemes for accessing arbitrary strides. We first show that algorithm H-n can always find a combined storage scheme which is conflict-free for arbitrary groups of power-of-2 strides.

Let $b_k = \{i_k, i_{k+1}, \ldots, i_{k+n-1}\}$ be the basis of stride 2^k pattern with 2^n memory modules. Similarly, the basis of stride 2^{k+1} is $b_{k+1} = \{i_{k+1}, i_{k+2}, \ldots, i_{k+n}\}$. The bases b_k and b_{k+1} share $n-1$ vectors that are $i_{k+1}, \ldots, i_{k+n-1}$, which implies that the vectors of b_k and b_{k+1} are n-colorable. This indicates that H-n can find *optimum combined address transformations* for arbitrary groups of power-of-2 strides. Conflict-free access to arbitrary strides is harder to achieve in the general case. However, the use of specific linear storage schemes that minimize the degree of serialization in accessing arbitrary stride provide useful throughput for many multimedia SIMD engines.

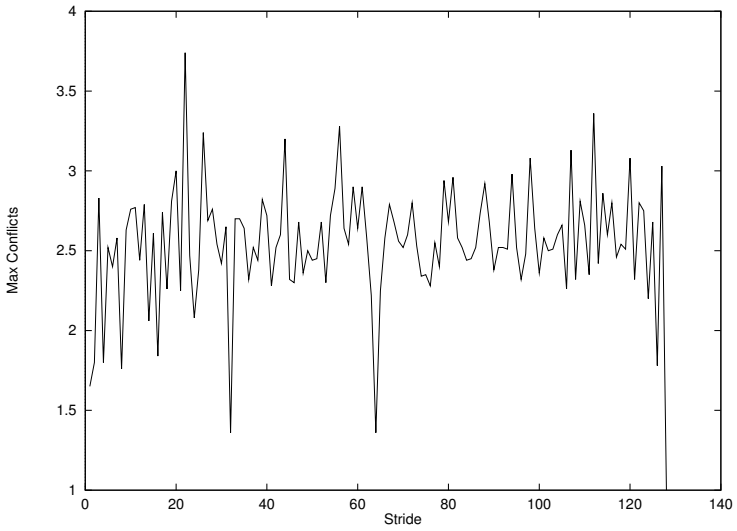

Figure 10: Sohi's maximum conflicts

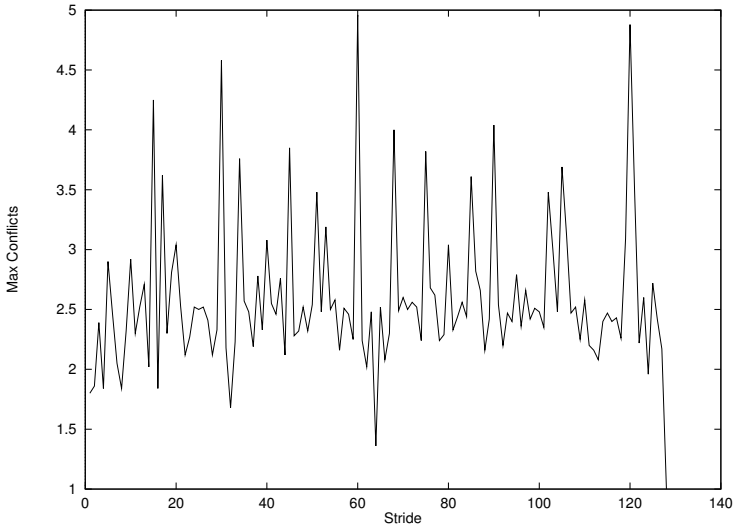

Figure 11: Norton's maximum conflicts

Harper [12] and Sohi [28] showed that high memory throughput can be achieved when a few buffers are used at the memory inputs and outputs. The buffers reduce the effects of transient degradation in pipelined memories due to memory conflicts in the case of non power-of-2 stride access. In this case, Sohi selected a storage matrix that allows a higher throughput than those obtained by using interleaving or row-rotation. Norton [25] proposed a specific scheme for the IBM-RP3 parallel memory. The linear storage schemes selected by Sohi and Norton for 8 memories are:

$$M_{sohi} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \quad M_{norton} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{20}$$

Sohi's matrix has an equal number of ones in each row and any group of 3 successive columns form a NS matrix. The latter condition allows conflict-free access to power-of-two strides but can also be used for arbitrary strides. Sohi's matrix is a particular case of CPs that can be found by using algorithm H-n. We present a scheme denoted by M_c which is:

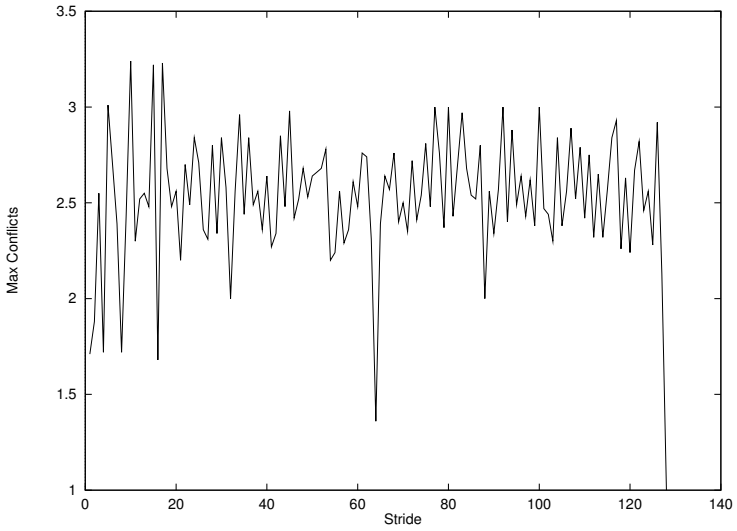

Figure 12: Maximum conflicts for a synthesized scheme

$$M_c = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{21}$$

The scheme can be easily generalized because the basis matrix (3×3) that appears in the right hand side is SNS [9]. The remainder of M_c is obtained by row rotating the basis matrix.

To compare these schemes we used the degree of conflict which is the maximum number of cycles required to access a given stride with a random origin address. The stride address is $a, a+s, a+2s, \ldots, a+(2^n-1)s$, where a is the origin, s is the stride, and 2^n is the number of memories. The origin was set randomly for each run and the results averaged over 100 runs. Figures 11, 12, and 13 show the plots of the degree of conflict versus the stride $(1 \leq s \leq 128)$ for M_{sohi}, M_{norton}, and M_c, respectively. All these schemes are fundamentally equivalent. Norton's scheme has many peaks which reach a degree of conflict of 5. Sohi's scheme has less fluctuation than the others. M_c achieves the lowest peak conflict.

7.4 Improving cache data re-use using parallel memories

In hierarchical memory systems the DRAM can deliver one single locality of reference (row-major or column-major) that is cached as such regardless of the regularity of code access (if any). This is suitable for irregular access patterns which cannot be predicted before the run-time.

Streamed computations have predictable access patterns [23, 24]. To improve performance of DRAM we propose a parallel DRAM memory in a framework that combines compile-time detection of data access patterns with proper integration of parallel memory and cache. The aim is to improve cache data re-use by caching relevant data locality references such as stride access over one or more array dimensions. A similar approach was proposed for re-mapping [13] of irregular or sparse memory accesses into dense accesses in the cache memory. In the following we present our approach which is illustrated by an example.

Consider a set of 8 memories and an 8×16 data array. Assume the following four data patterns $\{T_1, T_2, T_3, T_4\}$ defined by their bases $B(T_1) = \{g_2, g_1, g_0\}$, $B(T_2) = \{f_2, f_1, f_0\}$, and $B(T_3) = \{f_0, g_1, g_0\}$, and $B(T_4) = \{g_3, g_2, g_1\}$, where the fs and gs are the canonical vectors in the row and column array dimensions, respectively. The union of all base vectors is then

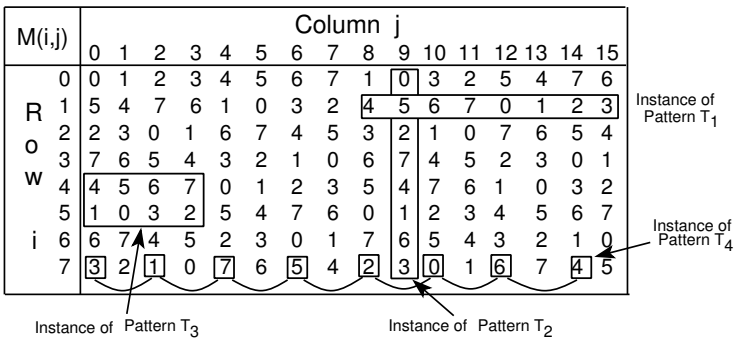

Figure 13: Mapping of 128 elemets (i, j) to 8 memories

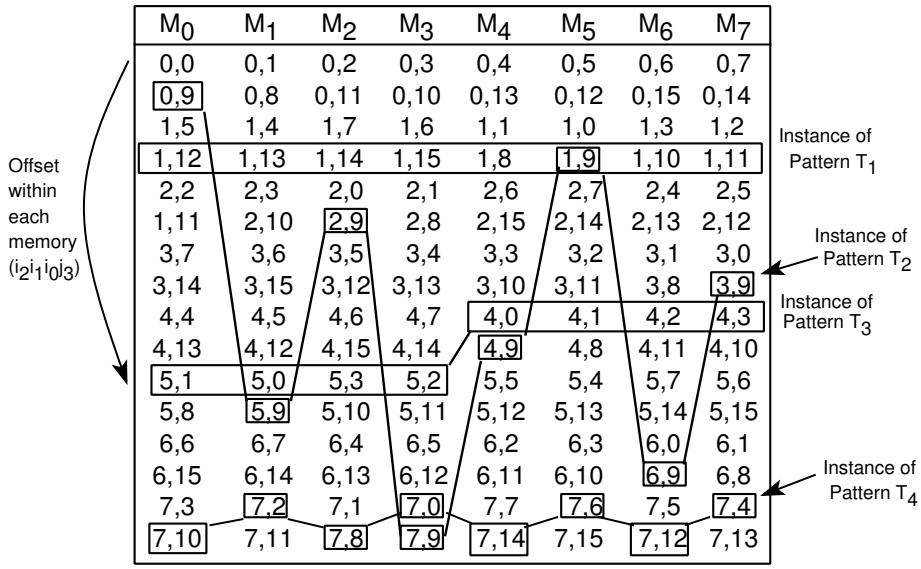

Figure 14: Mapping of 128 elements onto 8 memories and their offsets

$B = \cup_{1 \le k \le 4} B(T_k) = \{f_2, f_1, f_0, g_3, g_2, g_1, g_0\}$. Array element $(i, j) = (i_2, i_1, i_0, j_3, j_2, j_1 j_0)$ will be stored into memory M_k such that $k = M.(i, j)$. Suppose we choose M as follows:

$$M.(i,j) = \begin{array}{c} \begin{matrix} f_2 & f_1 & f_0 & g_3 & g_2 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array} \cdot \begin{pmatrix} i_2 \\ i_1 \\ i_0 \\ j_3 \\ j_2 \\ j_1 \\ j_0 \end{pmatrix} \tag{22}$$

Figure 14 shows the mapping of each array address (i, j) into the memory module number $M_{M.(i,j)}$ where array element $A(i, j)$ is stored, Here M is selected so that all four patterns can be accessed without conflicts because all corresponding pattern matrices are SNS. Four shown frame corresponds to one pattern instance and each frame contains eight distinct memory numbers. One may decide to set the offset in each memory according to a specific data pattern distribution, such as T_1 if M_{T_1} has full rank. Therefore all array elements that

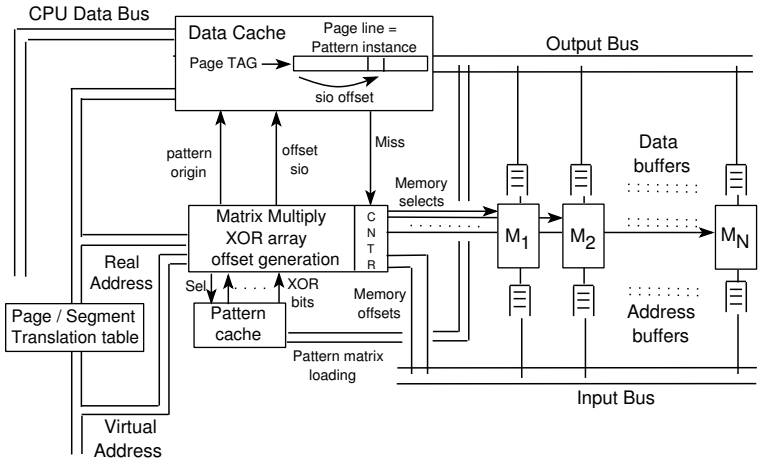

Figure 15: Translation of VA into real address, selection of storage matrix, generation of memory offsets, and packing of a pattern instance data in cache page.

belong to any instance of T_1 will be accessed with the same offset from all memories. The offset is formed by all address bits except those that are in the basis of T_1, i.e. the offset is (i_2, i_1, i_0, j_3). Note that other choices are also possible.

In Figure 15 all array elements are mapped to distinct offset locations. For each instance of T_1, the array elements fall into the same offset within all memories. However, this is not the case of other patterns. Consider the instance of T_3 that is shown in Figure 14-(a) for which the address of each element address satisfies $(i, j) = \binom{i_2 i_1 i_0 j_3 j_2 j_1 j_0}{1\ 0\ -\ 0\ 0\ -\ -}$. The offset is $(i_2 i_1 i_0 j_3)$ which overlaps with the pattern basis $B(T_3) = \{i_0, j_1, j_0\}$ in component i_0. This means that four elements of each instance of T_3 will be found at offset $\binom{i_2 i_1 i_0 j_3}{1\ 0\ 0\ 0}$ and the other four will be found at offset $\binom{i_2 i_1 i_0 j_3}{1\ 0\ 1\ 0}$. This does not pose any problem because all offsets are distinct within each memory.

The storage scheme $M.(i, j) = k$ indicates that array element addressed as (i, j) will be output from memory M_k. Due to data skew, we need to permute the data in the right order before returning the data elements as a page to cache. For this output of M_k must be set in a cache page at offset (i_0, j_1, j_0). This way the processor will refer to elements by their offset (sio in Figure 16) in the page.

The schematic of the proposed multi-pattern caching is shown on Figure 16. A processor virtual address is translated into real address by a TLB. Similar to paging, the real address (page address in main memory) must include identifier of array, pattern, pattern instance, and data element offset within the pattern instance. If the corresponding page line is present (hit), then the cache the offset to retrieve the corresponding data element. For example a page of 8 array elements corresponding to a row data pattern with a stride of 2 would contains $(a(i, 0), \ldots, a(i, 14))$. If this page is referenced with an offset of 3 the cache returns $A(i, 6)$. If there is a miss, then the cache activate a parallel memory controller (see block Matrix Multiply) which uses a pattern identifier from the address to fetch the corresponding storage matrix. Using XOR matrix operations the controller determines the memories to be selected and their offsets for fetching the identified instance of the accessed pattern. It activates the memories and forward their offsets. The data elements at output of each parallel memory are accessed in parallel but stored into a cache line in sequence. The cache receives the data in

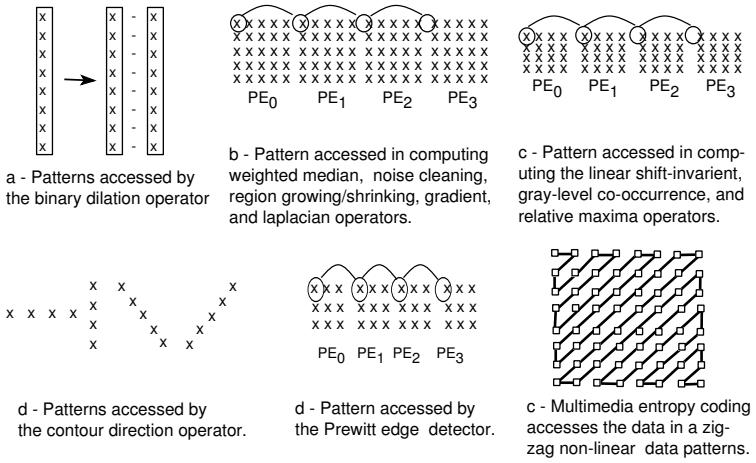

Figure 16: Typical access patterns in multimedia and image processing

the order described above and form a cache line.

A cache page contains array elements that belong to an instance of currently accessed data pattern for which the storage scheme was optimized for. A page is a compact storage to fit a well defined instance of a locality of reference such as a stride access or other.

For computations having predictable access patterns integrating a parallel DRAM memory in the hierarchical memory system guarantees that the storage pattern in the cache matches the actually accessed data patterns. This reduces bus traffic and enhances re-use of cached data.

8 Evaluation

We evaluate the proposed multi-pattern storage scheme in the case of some multimedia data access patterns. We also compare our approach to other contributions.

8.1 Performance under multimedia access patterns

In the parallel memory organization, the static storage schemes are intended to cause the data elements of some typical access patterns be uniformly assigned to the memories. Some static storages are intended for power-of-2 stride access [28, 25, 12] while other static schemes are optimized for some typical access patterns like the row, the column, and the blocks [10, 3]. Unfortunately, the performance of static storages may varies widely depending on the accessed stride and its origin.

Multimedia computing does not favor hierarchical memories [16] because cache misses are responsible of unpredictable memory access time. The need for predictable access with a large memory bandwidth is one key feature ensure *Quality-of-Service* for dynamic media and other hard real-time environments. Figure 17 shows some typical data access patterns that can be used in image [11] and multimedia [21] processing such as real-time motion estimation and *discrete cosine transform* (DCT/IDCT). In multimedia, the parallel memory [31] is investigated for its potential of providing fine-grain data parallelism as needed for video compression and decompression over SIMD engines such as the SSE2 multimedia unit of Pentium 4. For example the *binary dilation* (BD) operator, shown on Figure 17-(a), reads a column of 8 (or

4) pixels and produces two columns of pixels. Using 8 memories the basis of the accessed patterns is $B_{BD} = f_m, f_{m+1}, f_{m+2}$ for an $M \times M$ array of pixels, where $M = 2^m$ and the fs are the canonical basis vectors. The computation of neighborhood operators [11] like the weighted median operator can be implemented by assigning each logic unit a separate $k \times k$ frame of pixels as shown in Figure 17-(b), where k can be 3, 5, 7, or 9. Thus a pattern access with an odd stride is required. The accessed elements with stride $k = 5$ are shown on the same figure. Other functions like the *linear-shift invariant* requires assigning $k \times k$ frame of pixels, where k can be 4, 6, 8, or 10. This needs an even stride access as shown in Figure 17-(c). Similarly, contour segmentation, Prewitt edge detector and entropy scan require access to the patterns shown on Figures 17-(d), -(e), and -(f).

In the following we compare the performance of the proposed H-n storage with the classical interleaving and three typical storage schemes for stride and block accesses. These were proposed by Norton [25], Sohi [28], and Frailong [10]. Norton proposed a method to build a static storage matrix that minimizes both memory access conflicts and network alignment conflicts for a set of power-of-2 strides. Sohi's storage is based on eliminating conflicts for power-of-2 strides while ensuring that each bit in the memory number computes parities by using approximately the same number of address bits. Frailong optimized his storage by manually combining the storages of a given set of block patterns.

Our approach is based on the compiler which finds an optimized storage scheme based on loop dependence analysis and identification of the accessed data patterns as depicted in Section 6. For this we manually optimized the parallel execution of computations representing some well known numerical and vision algorithms. These are: (1) *sorting* (S) [26], (2) *LU decomposition* (LU) [26], (3) *matrix multiply* (MM) [26], (4) *cyclic reduction, FFT, and DCT* (CR/FFT/DCT) [21], (5) *vision algorithm with odd stride* (V-odd) [11], and (6) *vision algorithm with even stride* (V-even) [11]. All used data array are $N \times N$, where $N = 1024$. S requires access to the regular data patterns shown on Figure 9. LU uses a column access pattern, i.e. the sequence of reads has a constant stride of N that is the size of the row. MM has a mixing of row and column access patterns. The CR/FFT/DCT are fine-grain computations with a unit stride but requires linear permutations for aligning the data elements with the logic units. V-odd and V-even are the *weighted median* and the *linear shift invariant* operators which access the strides $s = (4, 6, 8, 10)$ and $s = (3, 5, 7, 9)$, respectively. Some instance of their patterns are shown on Figures 17-(b) and -(c).

The parallel memory engine is shown on Figure 1. The setting consists of 64 memories and PEs with the following latencies: (1) 2 clocks for the ATU, (2) 6 clocks for the NU, (3) 30 clocks for MU, and (4) 1 clock for the passive AU. The objective function measured through the simulation is the *Parallel Memory Utilization* (U) which is the ratio of average number of busy memories over the number of available memories. The average is taken over all the parallel load/store operations of the data pattern instance accesses needed for a given computation.

The results of the simulation are displayed on Figure 18 which plot the average memory utilization U. All studied storages perform very well for the unit stride if no alignment is needed (not shown). Generally, the static schemes are not well optimized for block access like in S, V-even, and V-odd. The memory access time of static storage schemes varies widely depending on the accessed data patterns and can be considered as unpredictable for practical considerations. In many cases (S, LU, MM, and CR/FFT/DCT) synthesizing a storage scheme that meets the application requirements provided a near-optimum memory utilization. Norton's static scheme is excellent for quite distinct combinations of power-of-2

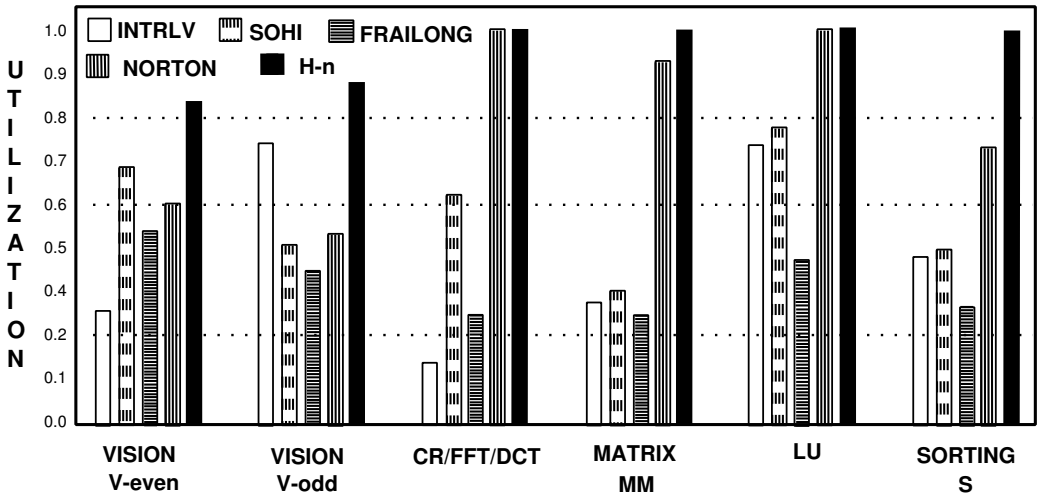

Figure 17: Utilization of 64 parallel memories

strides like those needed in CR/FFT/DCT, MM, and LU. We repeated the above experiment for 128 memories and found that a near optimum utilization is obtained for the studied combination of power-of-2 strides (S, LU, MM, and CR/FFT/DCT). In this case, each access to a given pattern instance has predictable number of memory cycles as explained in Section 5. A memory utilization above 83% is achieved for 64 memories. For non power-of-2 strides we obtain similar results if the stride is prime to the number of memories. Non power-of-2 strides cause a degradation of the utilization by about 15% as we increase the number of memories from 64 to 128.

The compiler overhead in finding the program data access patterns and optimizing the corresponding storage can be largely rewarded by many repetitive accesses. This is particularly suitable for synchronous dataflow computation which are generally compiled once and run many times on different data sets. Many of these computations can be found in DSP and dynamic media processing where there is a sharp need for large memory bandwidth together with predictable memory access time.

8.2 Comparison to other contributions

Minimizing memory and network contention for accessing rows, columns, diagonals, and square blocks was proposed in [3]. In this case, the storage matrix is optimized for a given set of reference patterns. In [25], network contention was analyzed with respect to conflict-free access to a fixed set of vector strides.

Our proposed approach assembles the requirements of the network, the memories, and access patterns in one single storage scheme that attempts to minimize overall access time. This approach is presented in a general framework. It can be applied to arbitrary multistage networks, arbitrary power-of-2 memories, and arbitrary data patterns. The compiler cost of finding set of data access patterns is nearly equal to the cost of carrying out the data dependence analysis. However, a randomized storage or a static storage can still be used as a default solution to our approach in the case where the access patterns cannot be found by the compiler. We have presented an algorithm to automate the major steps of synthesizing the dynamic storage matrix (for each array) and indicated how this approach can be used

for different networks including conversion of permutations from one network to another. For streamed computations [31, 32] which have predictable access patterns [23, 24] this approach can be implemented as a memory and a compiler co-optimization.

Our approach allows formulating and combining linear data patterns but cannot handle non-linear patterns, which represents the main limitation of this approach. For vector processors that inherently use pipelined bus architecture, Sohi [28] proposed a linear storage scheme to improve access of arbitrary strides in parallel memories. These approaches deal with memory organization and buffering in order to maximize the memory throughput. The issue is to find a linear address transformation that minimizes the degree of conflict for arbitrary strides and arbitrary origins. The result is that memory buffering smooth out the transient behavior of the memory, due to unresolved conflicts, and the memory throughput becomes close to optimum. We have proposed a scheme based on SNS conditional-exchange matrices and showed that it has a comparable performance to those proposed by Sohi and Norton but can be easily generalized to an arbitrary data patterns and arbitrary number of memories. Though this approach does not directly apply to non-linear data patterns, its choice of SNS matrices enables minimizing the global degree of conflict (also access time) in the case of arbitrary strides.

9 Conclusion

A memory and compiler co-optimization aimed at reducing low-level memory accesses using software and hardware locality optimizations is presented. Given an arbitrary set of power-of-2 data patterns, we have addressed the problem of storing arrays in parallel memories so that any instance of a pattern can be: 1) accessed without conflicts through an arbitrary multistage network and, 2) uniformly distributed over the memories following a linear address translation.

We have presented necessary and sufficient conditions for accessing parallel memories without network and memory conflicts for a given set of power-of-two data patterns. We presented a compiler strategy for finding multi-pattern storage schemes for streamed applications for which the data access patterns are predictable at compile-time. Our analysis indicates that this approach is applicable to an arbitrary sets of power-of-two data patterns and arbitrary networks. Our formulation allows the conversion of linear permutations from one network to another which enables the finding of equivalent memory mappings for each type of network. In the case of a combination of power-of-2 strides, our approach can be used by a compiler for synthesizing near-optimum memory utilization together with predictable memory access times for the cyclic reduction, FFT, DCT, bitonic sorting, LU, and matrix multiply. It outperforms static storage schemes for non power-of-2 strides. Our approach provides a tool for matching the storage pattern with the data access patterns needed for embedded systems running streamed computations with predictable data access patterns.

10 Appendix

Theorem 1 *The number of permutations $\tilde{d} = Ms \oplus x$ that an arbitrary n-stage multistage network can perform is $T_n = 2^{n^2}$, where M is an $n \times n$ SNS matrix and x is an $n \times 1$ arbitrary vector.*

Proof We first find the number R_n of $n \times n$ SNS matrices and evaluate the number of permutations $\tilde{d} = Ms \oplus x$ for all values of x.

Suppose we are given an arbitrary $n \times n$ matrix M. Assume that $M[i]$ is SNS. Then by performing row and column operations, we can transform M so that $M[i]$ is the identity matrix, thus M and $M[i+1]$:

$$M = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & a_{i-1} & \cdots \\ 0 & 1 & \cdots & 0 & 0 & a_{i-2} & \\ \vdots & \vdots & & \vdots & \vdots & & \\ 0 & 0 & & 1 & 0 & a_1 & \\ 0 & 0 & \cdots & 0 & 1 & a_0 & \\ b_{i-1} & b_{i-2} & \cdots & b_1 & b_0 & c & \cdots \\ \vdots & & & & & \vdots & \end{pmatrix} \qquad M[i+1] = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & a_{i-1} \\ 0 & 1 & \cdots & 0 & 0 & a_{i-2} \\ \vdots & \vdots & & \vdots & \vdots & \\ 0 & 0 & & 1 & 0 & a_1 \\ 0 & 0 & \cdots & 0 & 1 & a_0 \\ b_{i-1} & b_{i-2} & \cdots & b_1 & b_0 & c \end{pmatrix} \tag{23}$$

We examine the matrix $M[i+1]$. We denote the entry in the lower-right corner as c, the entries above c as $a_0 \ldots a_{i-1}$, and the entries to the left of c as $b_0 \ldots b_{i-1}$. We can determine whether $M[i+1]$ is NS. If all a_i and b_i are zero and c is one, it is easily seen that $M[i+1]$ is NS. For the general case, the fact that $M[i]$ is the identity makes it easy to cancel the a_i's and b_i's using row and column operations. $M[i+1]$ will be NS if and only if after these operations $c = 1$. If $a_j = 1$, add the column containing b_j to the column containing c of M. This changes a_j to zero and c becomes $c \oplus b_j$. Similarly, if $b_j = 1$ and we add the row containing a_j to the row containing c of M, this changes b_j to zero and c becomes $c \oplus a_j$. These operations may affect c as follows: 1) c does not change if $a_j = b_j = 0$ or $a_j \oplus b_j = 1$ or, 2) c is flipped if $a_j = b_j = 1$.

In the last case, both a_j and b_j are one. If we choose to cancel a_j first, the value of $b_j = 1$ is added to c, changing it from a one to a zero, or vice-versa. If we choose to cancel b_j first, the value of $a_j = 1$ is added to c, and c is again changed. In all other cases, we can cancel a_j and b_j without affecting c. The non-singularity of $M[i+1]$ will therefore depend on two factors: the initial value of c and the number of flips. In other words, $M[i+1]$ is non-singular if:

$$a_0 b_0 \oplus a_1 b_1 \oplus \ldots \oplus a_{i-1} b_{i-1} \oplus c = 1 \tag{24}$$

Counting the number of ways we get 0 flips, we find that we can do so in 3^i ways, because there are three ways each a-b pair can be assigned without causing a flip. There are $i3^{i-1}$ ways we can get one flip, and $i(i-1)3^{i-2}/2$ ways we can get two flips. The total number of ways is simply $\sum_{j=0}^{i} \binom{i}{j} 3^{i-j}$.

If there are R_i ways that $M[i]$ can be NS, then there are

$$R_i \cdot \sum_{j=0}^{i} \binom{i}{j} 3^{i-j} = R_i \cdot (3+1)^i = R_i \cdot 4^i \tag{25}$$

ways that $M[i+1]$ can be NS. Combining this with our value for R_1 we have: $R_1 = 1$, and $R_{n+1} = R_n 4^n$. The number of SNS matrices is then $R_n = 4^{(n-1)n/2} = 2^{(n-1)n}$. For each SNS matrix M, we can find 2^n distinct vectors for x. Therefore, the number of permutations $\tilde{d} = Ms \oplus x$ is $T_n = R_n 2^n = 2^{n^2}$. ∎

References

[1] M. Al-Mouhamed and M. Kaleemueddin. Evaluation of pipelined switch architecture for ATM networks. *IEEE/ACM Trans. on Networking*, No 5, Vol 7:724–740, Oct 1999.

[2] M. Al-Mouhamed and S. Seiden. Minimization of memory and network contention for accessing arbitrary data patterns in SIMD systems. *University of California Irvine, ICS-UCI Technical report 93-29*, Jun 1993.

[3] R. V. Boppana and C. S. Raghavendra. Optimal self-routing of linear-complement permutations in Hypercubes. *Proceedings of the 5th Dist. Mem. Comput. Conf.*, pages 800–808, 1990.

[4] P. Briggs and J. Feo. The tera programming workshop. *Inter. Conference on Parallel Architectures and Compilation Techniques,, Paris, France*, October 1998.

[5] P. Budnik and D. Kuck. The organization and use of parallel memories. *IEEE Trans. on Computers*, C-20, No 12:1566–1569, Dec 1971.

[6] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler directed page coloring for multiprocessors. *Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, 1996.

[7] A. A. Deb. Multiskewing - a novel technique for optimal parallel memory access. *IEEE Trans. on Parallel and Distributed Systems*, 7, No 6:595–604, Jun 1996.

[8] S. Dwarkadas, Honghui Lu, A.L. Cox, R. Rajamony, and W. Zwaenepoel. Combining compile-time and run-time support for efficient software distributed shared memory. *Proceedings of the IEEE*, 97, No 3:476–486, 1999.

[9] A. Edelman, S. Heller, and S. L. Johnson. Index transformation algorithms in a linear algebra framework. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 12:1302–1309, Dec 1994.

[10] J. M. Jalby W. Frailong and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proceedings of the International Conference on Parallel Processing*, pages 276–283, 1985.

[11] R. M. Haralick and L. G. Shapiro. Computer and robot vision. *Addison Wesley*, 1992.

[12] D. T. Harper III. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Trans. on Parallel and Distributed Systems*, 2, No 1:43–51, Jan 1991.

[13] Xianglong Huang, Zhenlin Wang, and K.S. McKinley. Compiling for the impulse memory controller. *Inter. Conf. on Parallel Architectures and Compilation Techniques*, pages 141–150, 2001.

[14] M. Kandemir. Compiler-directed collective-I/O. *IEEE Trans. on Parallel and Distributed Systems*, 12, No 12:1318–1331, 2001.

[15] M. Kandemir, U. Sezer, and V. Delaluz. Improving memory energy using access pattern classification. *IEEE/ACM Computer Aided Design*, pages 201 –206, 2001.

[16] C. E. Kozyrakis and D. A. Patterson. A new direction for computer architecture research. *IEEE Computer*, Nov. 1998.

[17] D. Lawrie and C.R. Vora. The prime memory system for array accesses. *IEEE Trans. on Computers*, C-31, 12:435–442, May 1982.

[18] D.-L. Lee. Scrambled storage for parallel memory systems. *IEEE Symp. on Computer Architecture*, pages 232–239, 1988.

[19] N. Linial and M. Tarsi. Interpolation between bases and the shuffle-exchange network. *European Journal of Combinatorics*, 10:29–39, 1989.

[20] Z. liu, J. You, and X. Li. Conflict-free routing on hypercubes. *Inter. Conf. on Computers and Information*, pages 153–158, 1992.

[21] M. Mattavelli, S. Brunetton, and D. Mlynek. A parallel multimedia processor for macroblock based compression standards. *Proceedings of the 1997 International Conference on Image Processing (ICIP '97)*, 1997.

[22] S. McFarling. Program optimization for instruction caches. *Third Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[23] S.A. McKee, W.A. Wulf, J.H. Aylor, R.H. Klenke, M.H. Salinas, S.I. Hong, and D.A.B. Weikle. Dynamic access ordering for streamed computations. *IEEE Trans. on Computers*, 49, No 11:1255–1271, 2000.

[24] H. Nakamura, M. Kondo, T. Ohneda, M. Fujita, S. Chiba, M. Sato, and T. Boku. Architecture and compiler co-optimization for high performance computing. *Inter. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 50–56, 2002.

[25] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. *Proceedings of the International Conference on Parallel Processing*, pages 247–254, 1987.

[26] M. Quinn. Designing efficient algorithms for parallel computers. *McGraw-Hill Inter., Second Edition*, 1988.

[27] M. Saghir, P. Chow, and C. Lee. Exploiting dual data-memory banks in digital signal processors. *Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, 1996.

[28] G. S. Sohi. High-bandwidth interleaved memories for vector processors–A simulation study. *IEEE Trans. on Computers*, 42, No 1:34–44, Jan 1993.

[29] T. Sterling. A hybrid technology multithreaded computer architecture for petaflops computing. *MS 159-79, J.P.L., California Institute of Technology*, January 1997.

[30] T. Stricker and T. Cross. Global address space, non-uniform bandwidth: a memory system performance characterization of parallel systems. *Inter. Symp. on High-Performance Computer Architecture*, pages 168–179, 1997.

[31] J. Tanskanen and J. Niittylahti. Parallel memnories in video encoding. *Proceedings of the Data Compression Conference, Snowbird, Utah.*, page 552, March 1999.

[32] J. Vanne, E. Aho, K. Kuusilinna, and T. Hamalainen. Enhanced configurable parallel memory architecture. *Euromicro Symposium on Digital System Design*, pages 28–35, 2002.

[33] Yong Yan, Xiaodong Zhang, and Zhao Zhang. A memory-layout oriented run-time technique for locality optimization on smps. *Proc. Inter. Conf. on Parallel Processing*, pages 189–196, 1998.