A Novel Cache Design for Vector Processing*

Qing Yang and Liping Wu

Dept. of Electrical Engineering
The University of Rhode Island
Kingston,RI 02881
Tel. 401-792-5880(0) 789-6085(h)
e-mail: qyang@ele.uri.edu

ABSTRACT

This paper introduces an innovative cache design for vector computers, called prime-mapped c ache. By utilizing the special properties of a Mersenne prime, the new design does not increase the critical path length of a processor, nor does it increase the cache access time as compared to a direct-mapped cache. The prime-mapped cache minimizes cache miss ratio caused by line interferences that have been shown to be critical for numerical applications by previous investigators. We show that significant performance gains are possible by adding the proposed cache memory into an existing vector computer provided that application programs can be blocked. The performance gain will increase with the increase of the speed gap between processors and memories. We develop an analytical performance model based on a generic vector computation model to study the performance of the design. Our preliminary performance analysis on various vector access patterns shows that the prime-mapped cache can provide as much as a factor of 2 to 3 performance improvement over the conventional direct-mapped cache in the vector processing environment. Moreover, the additional hardware cost introduced by the new mapping scheme is negligible.

KEY WORDS: *Cache Memory, Memory Bandwidth, Vector Processing, Vector Cache, Performance Analysis*

*This research is sponsored by Computer Systems Architecture Program of Computer and Computation Research, National Science Foundation.

ABSTRACT

This paper introduces an innovative cache design for vector computers, called prime-mapped cache. By utilizing the special properties of a Mersenne prime, the new design does not increase the critical path length of a processor, nor does it increase the cache access time as compared to a direct-mapped cache. The prime-mapped cache minimizes cache miss ratio caused by line interferences that have been shown to be critical for numerical applications by previous investigators. We show that significant performance gains are possible by adding the proposed cache memory into an existing vector computer provided that application programs can be blocked. The performance gain will increase with the increase of the speed gap between processors and memories. We develop an analytical performance model based on a generic vector computation model to study the performance of the design. Our preliminary performance analysis on various vector access patterns shows that the prime-mapped cache can provide as much as a factor of 2 to 3 performance improvement over the conventional direct-mapped cache in the vector processing environment. Moreover, the additional hardware cost introduced by the new mapping scheme is negligible.

1 Introduction

While cache memories have been successfully used in general purpose computers to boost system performance [1], their effectiveness for vector processing has not been established. Most of existing supercomputer vector processors typically do not have cache memories because of perceived poor performance for vectorized numerical algorithms. This common believe results primarily from three considerations: First, numerical programs generally have data sets that are too large for the current cache sizes. Sweep accesses of a large vector may result in complete reloading of the cache before the processor reuses them. Secondly, address sequentiality which has been an important assumption in conventional caches may not be as good in vectorized numerical algorithms that usually access data with certain stride (unit or non-unit). And third, register files and highly interleaved memories have been commonly used to achieve high memory bandwidth required by vector processing. It is not clear whether cache memories can significantly improve the performance of such systems. However, with the rapid advances in device technology and increased speed gap between processor and memory [2], it has become increasingly important to study the performance of cache memories for vector processors [3]

The first concern about possible poor performance of vector cache memories (we will call a cache for vector processor a *vector cache*) has been studied by a number of researchers [4, 5, 6]. It is well known that the memory hierarchy can be better utilized if numerical algorithms are *blocked*. Instead of operating on an entire vector of large size, we can block the vector into several segments

and perform computations on these segments. Blocking is a general program optimization technique that promotes data reuse in high speed memories. It has been shown that blocking is very effective for many algorithms in linear algebra [7]. Lam et al. have recently studied the cache performance of a blocked matrix multiply algorithm [4]. It is shown in their study that blocking factor (the size of inner loop) has significant impact on cache performance. In [5], So and Zecca presented an interesting performance study for vector caches by means of trace driven simulations [5]. They have shown that although the program locality of vector executions is significantly different from that of scalar executions their cache hit ratio is high enough to take advantage of a cache. Their study is based on traces of a set of fixed size programs that are either cache-optimized subroutines from the machine library or highly vectorized for vector machines. It is not clear how the cache behaves for generally blocked programs with different problem sizes.

Sequentiality of vector addresses depends on vector access stride that varies widely in numerical algorithms. Since the basic storage unit in a cache is a cache line that consists of a group of consecutive memory words, cache pollution may result if the access stride is not one. Large cache lines may exploit the spatial locality of vector accesses with small stride but may lead to poor cache performance for large strides. Small cache lines, on the other hand, may increase the number of cache misses depending on the vector stride. Fu and Patel [8] have recently presented a comprehensive study on the effects of cache line sizes on the performance of vector caches. They proposed two prefetching schemes, sequential-prefetching and stride-prefetching, for vector caches to reduce the influence of long stride vector accesses. As shown in [8], certain performance improvements as result of the two prefetching schemes are obtained. However, the cache miss ratios for some applications considered in [8] are still as high as over 40% in some cases. This is because that not only does the poor sequentiality of vector data result in cache pollutions but also a large amount of *interference misses* [4].

In spite of the fact that most supercomputer vector machines use interleaved memory and large register files to speed up memory accesses, the increased speed gap between processor and memory may still favor a cache memory to bridge this gap. This is because not only is a register file relatively small that can hardly hold the working set of a program but also it requires extra efforts in software to manage it. Cache memory, on the other hands, is transparent to programmers. Highly interleaved memory may provide enough bandwidth to single stream vector accesses, but the memory speed has to be extremely fast and the number of interleaved memory modules has to be excessively large in order to provide enough bandwidth for multiple stream vector accesses. Baily has shown in his study [9] that hundreds and even thousands of interleaved memory modules are needed to achieve a reasonable memory performance for multiple stream vector accesses. Furthermore, vector processing has become a mainstream form of computing ranging from superminis

to workstations. Vector processors have also been incorporated into mainframes as built-in accelerators for computationally intensive applications. For these types of machines, a cache memory can be a cost-effective enhancement towards a smooth memory hierarchy [5, 10, 8]. Several recent vector computers feature a cache-based memory hierarchy such as IBM 3090 [5], Alliant FX/8 [10] and Vax/600 [11].

It is clear from the above discussions that cache memories have a good performance potential for future vector processors. However, the question as to whether or not a cache memory can provide similar benefits for vector processors as it does for general purpose computers remains to be answered. A single miss in the vector cache results in a number of processor stall cycles equal to the entire memory access time, while the memory accesses of a vector processor without cache are fully pipelined. In order to benefit from a vector cache, the miss ratio must be kept extremely small. In general, cache misses can be classified into three categories [2]: compulsory, capacity, and conflicts. The compulsory misses are the misses in the initial loading of data, which can be properly pipelined in a vector computer. The capacity misses are due to the size limitation of a cache to hold data between references. If application algorithms are properly blocked as mentioned previously, the capacity misses can be attributed to the compulsory misses for the initial loading of each block of data provided that the block size is less than cache size. The last category, conflict misses, plays a key role in the vector processing environment. Conflicts can occur when two or more elements of the same vector are mapped to the same cache line or elements from two different vectors compete for the same cache line. The former is called *self-interference* whereas the latter is called *cross-interference* [4]. The recent study on blocked matrix multiply algorithm [4] shows that the self-interference misses increase drastically and dominate cache misses after the fraction of a 16K-word cache being used exceeds 3%. It is also shown in [4] that an algorithm with one problem size can run at twice the speed of the same algorithm with a different size.

Since conflict misses that significantly degrade vector cache performance have a lot to do with vector access stride, one may wish to adjust the size of an application problem to make a good access stride for a given machine. However, not only does this approach give a programmer a burden of knowing architecture details of a machine but also infeasible for many applications. It is known that the number of conflicts is minimum if the stride of accessing a vector is relative prime to the number of cache lines (or sets for set-associative cache) which is a power of 2 for direct or set-associative cache. Note that the stride required to access the major diagonal of a matrix is one greater than the stride required to access a row of the matrix stored in a column-major. Therefore, it is not possible to make both row access and major diagonal access efficient because one stride or the other is not relative prime to the cache size of any direct or set-associative cache.

In this paper, we propose a new cache design, called $prime-mapped$ cache, in an attempt to

minimize cache miss ratio. The main idea of the design is that memory data are mapped into cache lines according to a Mersenne prime. The cache access logic of the new mapping scheme keeps virtually the same as direct-mapped cache, resulting in no additional delay for cache accesses. At the meantime, the address generation takes no longer than the normal address calculation time due to the special properties of the Mersenne number. Moreover, generating addresses for cache access is done in parallel with normal address calculations resulting in no performance penalty. Thus, it has the advantages of both direct-mapped cache and fully associative cache if replacement is not considered. We also present an analytical model for two simple vector processor models: memory-register vector processor model and cache-based vector processor model. The advantage of an analytical modeling is that it provides us with a quick and insightful performance estimate of a given design without the limitation of application problem sizes. Based on the analytical model, one can easily pinpoint where the potential performance bottleneck is. Furthermore, most numerical algorithms are highly structured, which makes analysis possible and fairly accurate.

Our performance analysis on several typical vector access patterns shows that the new cache design improves vector processing performance by a factor of 2-3 over conventional cache designs with negligibly additional hardware cost. For vector accesses in which random multistrides are used to access two vector streams, the performance improvement of the prime-mapped cache over direct-mapped cache ranges from 40% to a factor of 3. By properly selecting blocking factors, subblock (or submatrix) accesses can easily be made conflict free even if the cache utilization is close to 1 for matrices of arbitrary sizes. For FFT access pattern, performance optimization is guaranteed as long as the blocking factor is less than the cache size.

The paper is organized as follows. Section 2 presents the cache design and discussions on hardware issues. In Section 3, we present an analytical performance model for two vector processors with and without a cache. We will carry out performance analyses and comparisons for various approaches in Section 4. We consider three types of typical vector access patterns: random multi-stride access, sub-block access, and FFT access in our performance analysis. Section 5 concludes the paper.

2 Minimizing Vector Cache Misses

In this section, we consider several alternatives to reduce vector cache misses. By vector cache, we mean, in this paper, the cache that holds vector data accessed by vector load or store instructions. We assume that scalar data have a separate cache [11]. The new mapping scheme that minimizes interference misses of the vector cache is presented in Subsection 2.3.

2.1 Can Associativity Help?

Since cache misses in a vector cache result mainly from line conflicts as discussed in the introduction, one would naturally think of set-associative-mapped cache with higher associativity or fully associative cache. It is well known that higher associativity has the advantages of less line conflicts and the flexibility of implementing a replacement algorithm such as LRU. Lam et al [4] have shown that the average miss rates of a set-associative cache are relatively lower than that of direct-mapped cache. However, the fraction of cache used in case of set-associative cache still remains very small (in the range of 20% to 60%) though it is larger than that of the direct-mapped cache. Moreover, the standard deviation increases steadily with the blocking factor implying that the execution time for some problem sizes can be significantly worse than others. For the same cache size, increasing associativity results in decreased number of sets that data can be mapped to although several cache lines can be mapped to the same set. As a result, we will not see significant reduction in terms of interference misses. Furthermore, increasing the associativity also increases the cache hit time [12] which has negative effects on system performance. As to the replacement algorithm, due to the nature of numerical algorithms, we may not be able to take this advantage as we do for most other applications since serial access to vectors dictates against LRU replacement [3]. Whether there exists a better replacement algorithm needs further study.

2.2 Effects of Line Size

Cache line size is one of the most important parameters in a cache design. Larger cache line sizes reduce compulsory misses if an application has a high spatial locality. At the meantime, larger cache lines also reduces the number of lines in the cache, giving rise to more conflicts. In addition, non-unit access strides may also result in cache pollutions since the loaded excess data may never be used before being replaced. Cache pollution degrades cache performance significantly because not only do the unused data use cache space but also memory bandwidth [8]. Fu and Patel [8] observed that cache line sizes have unpredictable impacts on vector cache performance. The best cache line size of one program may be the worst for another program in a given cache design. For some applications, increasing cache line sizes even results in increased miss ratio, which is contradictory to common believe. Their observations suggest that an optimal cache line size for vector processing be difficult to determine unless all application programs have the same stride characteristics. There are also other considerations in deciding cache line size such as bus width, degree of memory interleaving etc.. Cache line size selection for the vector cache is more complicated than general purpose computers. In this paper, we will concentrate on fixed line size of 8 bytes or one double precision word.

2.3 Prime-Mapped Cache

We propose a cache design with a novel mapping scheme called **prime-mapping** that attempts to minimize interference misses. The idea behind the new design is simple. Instead of having 2^c cache lines in the cache, we allow only a prime number of cache lines to reside in the cache. Using a prime number in a memory system to avoid memory access conflicts was attempted before. In the early 70's, Budnik and Kuck suggested the use of prime number of memory modules in the context of a parallel computer [13]. The idea was latter developed by Burroughs in the design of the BSP computer [14]. However, addressing for such prime-number memory systems is much more complex than for memory systems in which the number of memory modules is a power of 2. This complicated addressing is more critical in cache design since we can not afford to allow any additional delay for cache accesses. Any increase in cache hit time will affect not only the average access time but also the clock rate of the CPU. Therefore, it is of essential importance that any attempt in designing a new cache must ensure no increase in the length of the critical path of a processor.

Our approach here is to utilize the special properties of a class of prime numbers: the Mersenne primes. A Mersenne number is of the form $2^c - 1$ for some c that makes $2^c - 1$ a prime. Examples of such c are 2, 3, 5, 7, 13, 17, and 19 etc.. A cache line with address A_l is mapped into line number $A_l \ mod \ (2^c - 1)$ in the cache. Since Mersenne number is a prime, a vector access to the cache with this mapping scheme can be made conflict-free. Meanwhile, the address space is also well utilized since the number of lines in the prime-mapped cache is just one less than a 2's power.

A detailed description of the *Prime-mapped* scheme is as follows: Each memory address, same as conventional cache-based computer system[12], is partitioned into three fields: $W = log_2(line\ size)$ bits of word address in a line (offset); $c = log_2(number\ of\ lines+1)$ bits of index; and the remaining tag bits of tag. The access logic of the prime-mapped cache consists of three components: data memory, tag memory, and matching logic. Same as a direct-mapped cache, the data memory contains an address decoder and cached data; the tag memory stores tags corresponding to the cached lines; and the matching logic checks if the tag in an issued address matches the tag in the cache. The cache lookup process is exactly the same as the direct-mapped cache and hence takes the same amount of time as the direct-mapped cache. However, the index field used to access the data memory is not just a subfield of the original address word issued by the processor since the modulus for cache mapping is not a 2's power any more. It is the residue of the line address modulo a Mersenne number. How efficient this modular operation can be done is essential to the cache performance. In the following, we show how the index conversion can be done without adding additional delay in the address mapping process.

Figure 1 shows the block diagram of the address computation logic. It consists of two parallel parts: one normal address calculation unit that generates addresses for accessing the main mem-

ory and the other address generator that calculates addresses for accessing the vector cache. For notational convenience, we call the address for accessing the main memory *as memory address* and the one for accessing the vector cache as *cache address*. The normal address calculation unit is a part of address control logic or functional unit that any existing vector computers should have. The memory address of a vector element is calculated based on a vector stride and the address of the previous element of the vector. For generating a cache address, the tag field and the word (offset) field are the same as that for memory address. It is only the index field (c bits) that needs to be calculated in a Mersenne number form through the additional address generator. Since a Mersenne number is defined as an integer modulo 2^c-1, arithmetic operations are greatly simplified by noting that $2^c = 1$. Consider any two integers represented in c-bit binary form. If we add the two numbers we obtain a $(c+1)$-bit sum with the most significant bit being carry bit. But $2^c \equiv 1 \ mod \ (2^c - 1)$. Thus, addition modulo a Mersenne number is performed very simply by using a conventional full binary adder of c bits and by folding the most significant carry bit output back into the least significant carry bit input. Therefore, a c-bit full adder is sufficient to perform the address generation. The index field of the cache address of a vector element is obtained by adding the stride to the index field of the cache address of the previous element of the vector. Note that both the stride and the previous index value should be in the Mersenne number form as explained shortly. Because the addition is performed on a c-bit field which is a portion of a memory word, this process should take no longer than the normal address calculation process. Two addresses are therefore generated concurrently: one for cache access and the other for memory access in case of a cache miss.

It remains to consider how to generate the cache address of the first element of a vector and the vector stride in a Mersenne number form efficiently. Let the starting line address of a vector be A_{start} that consists of two fields: tag_A and $index_A$ with the lengths of each field being tag and c, respectively. We ignore the offset field of an address since a line is the basic unit for cache mapping. Suppose the least significant c bits of tag_A are represented by tag_{A1} and the second c bits are represented by tag_{A2} and so forth. In order to map the starting line of the vector into the prime-mapped cache, we need to perform a modular operation $A_{start} \ mod \ (2^c - 1)$. Since

$$A_{start} = index_A + 2^c tag_{A1} + 2^{2c} tag_{A2} + \cdots,$$

and $2^c \ mod \ (2^c - 1) = 1$, we can write

$$A_{start} \ mod \ (2^c - 1) = index_A + tag_{A1} + tag_{A2} + \cdots.$$

Therefore, to derive the index field of the starting line address of a vector, what we need to do is to perform a sequence of c bit additions. If the length of tag_A (tag) in bits is less than or comparable to the length of $index_A$ (c), then we just need one c-bit addition. In practice, this is often the case.

For example, the cache size of the Alliant FX/8 is 128K bytes (16K double words) [10] giving rise to an index length of 14 bits if the line size is 8 bytes. If the address length is 32 bits, the remaining tag field consists of maximum of 15 bits. We then need to perform a 14-bit addition and add back the most significant bit (32th bit from left) of the starting address into the least significant bit of the sum. Similarly, the VAX6000 Model 400 vector processor has a cache of 1 Mbytes that can easily make the tag field comparable to the index field. If it is desired in a design that the tag field be longer than the index field, we would need at most one more level of addition of c bits in calculating the index field of the starting address of a vector.

The address conversion logic for the starting element of a vector is also shown in Figure 1 assuming $tag \leq c$. Two multiplexors are used to select addends for the $c-$bit adder. The multiplexors will select two fields (tag and index) of the memory address word to do addition if it is for the starting element. For all other elements of the vector, the multiplexors will select index value of the cache address of the previous element and the stride to do addition. The generation of the vector starting address for cache access can be done in parallel with the memory access of the vector element if the vector is accessed first time. It is save to assume that a couple of stages of c bit additions can finish before the loaded memory data arrives in the vector cache. Thus, the loaded data can be written into the vector cache using the newly calculated index value. In this case, there is no performance penalty resulting from the address conversion. The converted cache address of the starting element can be stored in a special register for future reuses if the vector is going to be accessed again. Subsequent accesses to the elements in the same vector can be done using two possible addresses: cache addresses as discussed above or memory addresses if a miss occurs. The calculation of these addresses does not incur any additional delay compared to existing machines as evidenced previously.

Converting an integer stride into a Mersenne number can be done in the same way as for the starting address. That is, just additions are needed. This process can be done at the time when the vector stride is loaded into the vector stride register. However, a special register of c bits is needed to keep the converted stride as shown in Figure 1.

The additional hardware cost as result of this new mapping scheme includes 2 multiplexors, a full adder and a few registers as shown in Figure 1. Even a small percentage of performance improvement can justify such insignificant hardware cost. As will be evidenced latter in this paper, we expect to double and even triple the performance of direct-mapped cache and the performance of the machines without cache. However, there is still a trade-off between performance and hardware cost. The registers storing the starting addresses of vectors, for example, add more cost to the machine. If several vectors are accessed concurrently, we not only need more registers but also the control logic to control the access of these registers. If we can afford to spend 1 or 2 more cycles at

each vector start-up time, we may eliminate these registers. Each time a vector is accessed again, we recalculate its starting address even though the vector is already in the cache. This recalculation requires just an addition of subwords which takes small and limited amount of time, and it may also be pipelined. On the other hand, if performance is absolutely important one can pay for the registers.

3 An Analytical Model

In this section, we present an analytical performance model based on the simple vector performance model given in [2]. It should be noted that the analysis and the numerical results presented here are not meant to be predictors of performance of any realistic computer system. Rather, they are meant to give a quick insight into the simplified vector processor models and to show some potential problems that might exist in a vector cache memory. By varying different input parameters over a wide range, we use the analytical model to comparatively study the performance of different alternatives of cache designs.

3.1 A Simple Vector-Computational-Model

Two simplified vector processor models considered here are shown in Figures 2 and 3 which will be referred to as MM-model and CC-model, respectively. Both models have a vector processing unit, a set of vector registers with maximum vector length of MVL words, and a set of low order-bit interleaved memory modules (or banks) that are connected to the vector processor through three pipelined buses. Two of the three buses are read buses and the other is write bus. Each of these buses contains separate data bus and address bus. There are totally $M = 2^m$ interleaved memory banks each of which has the access time of t_m cycles. A line of data can be transferred on one of the buses within one cycle. The second processor model (CC-model) shown in Figure 3 differs from the first one (MM-model) only in that it has a cache memory of size C lines between the processor and memories. We assume that this cache memory is used purely for vector data similar to the split vector cache of VAX/6000 [11].

Cache miss ratio has been used by many researchers as a performance measure to evaluate cache performance. However, it is not a very good performance measure in this context because we intend to compare the performance differences between the MM-model and the CC-model. The performance measures that we will use in this paper are the *total execution time* of an application program and the *clock cycles per result* which is defined as the average number of clock cycles needed to produce one vector element result [2]. In order to estimate the total execution time or the clock

cycles per result, we present in the following a simple and generic vector computational model that attempts to cover a wide spectrum of numerical algorithms.

Our simplified computational model assumes that application programs are blocked into several segments. One of the vector data that an application program operates on is partitioned into several sub-blocks of size B each. We call this B a *blocking factor*. For example, if a matrix is blocked into several submatrices of size $b \times b$, then the blocking factor is b^2. Notice the slight difference of the blocking factor defined here from that in [4] which was the innermost loop length. A sub-block of data may be used several times by one program segment. We define a reuse factor, R, as the number of times a block of data is reused. During each vector operation, the processor may load two vectors from the memory simultaneously or load just one vector with the other operand available in a register. It is assumed that the probability that the processor accesses two vector streams simultaneously from the memory during one vector operation is P_{ds} (double stream). The probability that the processor accesses just a single vector stream with the other operand available in a register is P_{ss} $(= 1 - P_{ds})$. The system performs operations similar to the SAXPY operations. The processor first loads one or two vectors into its registers and then performs arithmetic operations on the two vectors or on one vector and a scalar in a scalar register.

We now determine the length of the second vector for double stream vector accesses. In order to simplify our analysis, we further assume that all single-stream vector accesses are evenly intercepted by one-double-stream accesses if $P_{ds} \neq 0$. In other words, every sequence of $\frac{BP_{ss}}{BP_{ds}} = \frac{P_{ss}}{P_{ds}}$ consecutive single-stream vector accesses is followed by one double stream vector access which is again followed by a statistically identical sequence of single stream accesses, and so forth. Thus, one can imagine one of the vectors of length B as a two dimensional matrix with $P_{ss}/P_{ds} + 1$ columns and BP_{ds} rows. The vector length of each column is BP_{ds}. After every P_{ss}/P_{ds} column accesses of the imagined matrix, the processor loads two vectors to perform vector operations on the first and the second vector. Therefore, the length of the second vector can be considered as BP_{ds}.

Although the assumed computation model is simplified, it is very close to many realistic applications. For example, the blocked matrix multiply algorithm in [4] has the blocking factor of b^2 since the matrices are blocked into submatrices of size $b \times b$. The reuse factor of each block is b and each sequence of $b-1$ single stream vector accesses is followed by a double stream access. Therefore, the fraction of time when the processor accesses a single vector stream is $(b-1)/b$ while the fraction of double stream accesses is $1/b$. The length of the first vector is then b^2 and the length of the second vector is b. Similarly, the blocked LU decomposition algorithm with a blocking factor of b^2 [15] has an average reuse factor of $3b/2$ and the blocked FFT algorithm [16] with a blocking factor of b has a reuse factor of $Log_2 b$.

The access strides of loading the two vectors are denoted by s_1 and s_2, respectively. Access strides

vary widely depending on application programs. In general, unit stride occurs more frequently than other strides in most numerical algorithms [8]. We therefore use $P_{stride1}$ to denote the probability that an access stride is 1. If a stride is not 1, we assume that it takes any other integer values equally likely. Since we are interested in memory bank conflicts and cache line conflicts, a stride is assumed to take an integer value in the range of $(1, 2, \cdots, M)$ for the MM-model and in the range of $(1, 2, \cdots, C)$ for the CC-model due to modular operations. We also assume that writing results into memory or cache will not delay the normal vector operations. This assumption is not a severe restriction because it can be realized in real machines by having write buffers, separate data bus for writing and separate write port for memories [2].

In summary, our simple vector computational model consists of the following seven-tuple:

$$VCM = [B, R, P_{ds}, s_1, s_2, P_{stride1}(s_1), P_{stride1}(s_2)].$$

By properly selecting these model parameters, the model can fit into a variety of numerical algorithms and various vector access patterns. For example, if we set $VCM = [b, r, 1, 1, P, 1, 1/C]$, we have double stream vector accesses to columns and rows of a $b \times b$ submatrix of a $P \times Q$ matrix stored in column-major. Each pair of column and row are used for r times. If $VCM = [b, r, 0, P+1, -, 1/C, -]$ [1], then we get a single vector access stream to a major diagonal of a $b \times b$ submatrix of the same $P \times Q$ matrix. FFT access and sub-block access can also be easily modeled with some minor modifications as will be discussed latter in the paper.

3.2 Execution Time of the MM-Model

Let us consider the MM-model of Figure 2 first. The vector performance can be characterized by the execution time for a sequence of operations on a vector of length B, T_B. This quantity depends on a number of factors including the overhead for computing the starting addresses and setting up vector controls, the overhead for executing scalar code for strip-mining (fine level of blocking), and the maximum vector register length, MVL. By assuming these parameters to be constant having the values given in [2], we have

$$T_B = 10 + \lceil \frac{B}{MVL} \rceil \cdot (15 + T_{start}) + B \cdot T^M_{elemt}, \tag{1}$$

where T_{start} is the start-up time for each inner most loop and T^M_{elemt} [2] is the time for processing one element of the vector ignoring the start-up time. The start-up time depends on the memory access time and the type of arithmetic operations to be performed. It should not be difficult to

[1] The symbol "-" means undefined

[2] For notational convenience, we use superscript M and C on a parameter to distinguish MM-model (Figure 2) and CC-model (Figure 3) vector processors, respectively.

determine its value for a given set of system parameters. In our discussions, we assume T_{start} to be constant for a given memory access time. If the system is fully pipelined with one functional unit, T^M_{elemt} should be one in an ideal case. In reality, T^M_{elemt} is often greater than one because of stalls resulting from memory accesses, data dependencies and control dependencies. Dependency analyses of numerical algorithms have been studied extensively in the literature. Since our primary interest here is memory system performance, we will concentrate on memory stalls only. The major source of memory stalls for vector processors is memory conflicts [17, 18, 9]. A number of storage schemes to minimize memory conflicts stands out in the literature. Some examples are [19, 17, 14, 13] to list a few. In our analysis, we consider only the low-order bit interleaving scheme although some conflict-free dynamic storage schemes can provide about 18% better performance than the simple interleaving [17]. In the following, we derive T^M_{elemt} by taking into account memory stalls.

Memory stalls can result from two types of bank conflicts: One is the conflicts between elements in the same vector (*self-interference*) denoted by I^M_s, and the other occurs between elements from two different vector access streams (*cross-interference*) denoted by I^M_c. Let us first consider the number of possible stall cycles of accessing one vector stream with stride s_1. When a vector access stream traverses across all the memory banks (one sweep), the number of memory banks visited by the access stream is given by $M/gcd(M, s_1)$ [18, 19], where $gcd(M, s_1)$ is the greatest common divisor of M and s_1. If $t_m > M/gcd(M, s_1)$, a memory conflict occurs and all remaining accesses of the sweep are delayed by $t_m - M/gcd(M, s_1)$ cycles. Since M is a 2's power, $gcd(M, s_1)$ is in the form of 2^i for some $i = 1, \cdots m-1$. If $gcd(M, s_1) = 2^m = M$, then each of $MVL - 1$ elements will be delayed by $t_m - 1$ cycles. The number of possible values of s_1 within M such that $gcd(M, s_1) = 2^i$ $(i \in (1, \cdots m-1))$ is equal to the divisor function of $M/2^i$ which is given by $\phi(\frac{M}{2^i}) = \frac{M}{2^{i+1}} = 2^{m-i-1}$ [20]. For each such s_1, $\frac{MVL}{M/2^i}$ sweeps are delayed by $(t_m - M/2^i)$ cycles. Recall that the probability of $s_1 = 1$ is $P_{stride1}$ and the probability of s_1 taking any other value $(2 \cdots M)$ is $\frac{1-P_{stride1}}{M-1}$. Therefore the total number of stall cycles resulting from accessing a vector of length MVL, I^M_s, is given by

$$I^M_s = \frac{1 - P_{stride1}}{M-1}[\sum_{i=\lceil log_2 \frac{M}{t_m} \rceil}^{m-1} (t_m - \frac{M}{2^i}) 2^{m-i-1} \frac{MVL}{M/2^i} + MVL(t_m - 1)].$$

The lower limit of the summation in the above equation is determined based on the condition $t_m \geq M/2^i$. Notice that the above equation assumes that $t_m < M$ so that unit stride does not incur any stalls. Simple algebraic manipulation leads to

$$I^M_s = MVL \frac{1 - P_{stride1}}{(M-1)} \cdot [t_m + \frac{t_m}{2} \lfloor log_2 t_m \rfloor - 2^{\lfloor log_2 t_m \rfloor}].$$

Next, let us consider memory stalls as result of cross-interferences between two vector access streams. Let s_2 be the stride of accessing the second vector stream and D be the memory bank differ-

ence between the starting addresses of the two vectors. We assume that D is uniformly distributed between 1 and M. If the memory bank address of the ith element of the first vector is is_1, the memory bank address of the jth element of the second vector is js_2+D. Thus whenever the congruence, $s_1 i \equiv s_2 j + D(mod\ M)$, has solutions for $i \in (0, 1, \cdots, MVL-1)$ and $j \in (0, 1, \cdots, MVL-1)$ such that $\mid i-j \mid < t_m$, conflicts between the two access streams (cross interferences) occur. The number of stall cycles due to such a conflict is $t_m - \mid i-j \mid$. Therefore, the cross-interferences can be calculated by solving the above congruence equation with two variables [20]. We have written a program of solving the congruence equation. The total number of stall cycles due to the cross-interference for a given set of $s_1,\ s_2$ and D is the accumulation of quantities $t_m - \mid i-j \mid$ for each pair of solution (i, j), where $i, j \in (0, 1, \cdots, MVL-1)$. The final cross-interference, I_c^M, is averaged over all possible values of $s_1,\ s_2$ and D based on the given distributions discussed in the beginning of this section.

T_{elemt}^M can now be expressed as

$$\begin{aligned} T_{elemt}^M &= 1 + average\ stall\ cycles\ per\ element \\ &= 1 + P_{ss}\frac{I_s^M}{MVL} + P_{ds}\frac{(2I_s^M + I_c^M)}{MVL}. \end{aligned} \tag{2}$$

Substituting Equation (2) into Equation (1) we get the execution time of one block. Assume that the total data size of a program is N which is blocked into several segments of length B. Taking into account the reuse factor, R, the total execution time T_N^M can be expressed as:

$$T_N^M = T_B \cdot R \cdot \lceil \frac{N}{B} \rceil, \tag{3}$$

where T_B is the execution time for a sequence of operations on a vector of length B given in Equation (1).

3.3 Execution Time of Direct-Mapped CC-Model

Now let us consider the CC-model of Figure 3 that has a direct-mapped cache of size $C = 2^c$ lines. We assume that a cache line contains one double precision word of data.

The processor initially loads one or two vectors into the cache while performing the designated vector operations. The process of the initial loading will take the amount of time given by Equation (1). After the vectors are loaded into the cache, the remaining operations on the same set of data are expected to be performed in the cache without accessing memories in an ideal situation. The total execution time of the CC-model (Figure 3) is given by

$$T_N^C = \{T_B + [10 + \lceil \frac{B}{MVL} \rceil \cdot (15 + T_{start} - t_m) + B \cdot T_{elemt}^C] \cdot (R-1)\} \cdot \lceil \frac{N}{B} \rceil, \tag{4}$$

where T_B covers the effects of compulsory and capacity misses as discussed above. Notice that the start up time after the initial vector loading is decreased by t_m since the reused vector is in the cache provided that it has not been replaced due to a conflict. Similar to the analysis of the MM-model, we use T^C_{elemt} to include memory stalls due to cache misses. Cache misses may have significant effect on system performance since for each cache miss the processor has to stall for t_m cycles due to the fact that cache misses may not be easily pipelined. We will consider in the following the effects of cache misses caused by line interferences on the system performance.

Suppose that the processor tries to load a vector of length B into the vector cache with a random stride s_1 having the distribution described in the beginning of this section. The number of cache lines occupied by the vector is given by $C/gcd(C, s_1)$. Thus, there will be $B - C/gcd(C, s_1)$ self-interferences if the vector length B is greater than or equal to $C/gcd(C, s_1)$. Again the number of s_1's such that $gcd(C, s_1) = 2^{(c-i)}$ is 2^{i-1} (divisor function) [20]. If $gcd(C, s_1) = C$, then there would be $B - 1$ conflicts. Therefore, the average number of stalls due to self-interference misses in the B-element vector is given by

$$I^C_s(B) = \frac{1 - P_{stride1}}{C - 1}[\sum_{i=1}^{c-\lceil log_2 \frac{C}{B} \rceil} (B - \frac{C}{2^{c-i}})2^{i-1} + B - 1] \cdot t_m, \tag{5}$$

where the upper limit of the summation is determined in such a way that $B - C/2^i$ is always positive. Notice that there is no self-interference if the access stride is one and the block size B is less than cache size. For every self-interference miss the processor stalls for t_m cycles. Simple algebraic manipulation leads to

$$I^C_s(B) = \frac{1 - P_{stride1}}{C - 1}\frac{1}{3}(3B2^{\lfloor log_2 B \rfloor} - 2 \cdot 2^{2\lfloor log_2 B \rfloor} - 1) \cdot t_m. \tag{6}$$

For B being a power of 2, we have

$$I^C_s(B) = \frac{1 - P_{stride1}}{3(C - 1)}(B^2 - 1) \cdot t_m.$$

The cross interference misses, I^C_c, can be calculated in a similar way as for the I^M_c or using the footprint model described in [3, 4] assuming that the two vectors are independent. The probability that any one element of the second vector falls into the footprint of the first vector is $\frac{B}{C}$. There are totally BP_{ds} elements in the second vector. Therefore, the total stalls due to cross-interferences are given by $I^C_c = \frac{B^2 P_{ds}}{C} t_m$. Thus, the time for processing one vector element, T^C_{elemt}, is

$$T^C_{elemt} = 1 + P_{ss}\frac{I^C_s(B)}{B} + P_{ds}\frac{(I^C_s(B) + I^C_c(BP_{ds}) + I^C_c)}{B}. \tag{7}$$

Substituting Equation (7) into Equation (4) we obtain the total execution time of the direct-mapped CC-model.

3.4 Analysis and Comparison of the Two Models

We now carry out a preliminary performance analysis for the two vector computers based on the simple analytical model presented above. We use the *average clock cycles per result* as a performance measure in our following discussions. It is obtained by dividing the total execution time by the product of N and R. In the following figures, we fix the values of MVL and T_{start} at 64 and $30+t_m$ [2], respectively. The probability of unit stride access, $P_{stride1}$, is also fixed at 0.25 which is the average value of the experimental data reported in [8].

Figure 4 shows the *average clock cycles per result* as a function of memory access time in terms of processor cycles. If memory access time is small, it can be seen from the figure that adding a direct-mapped cache memory does not help. The performance of the MM-model performs even better than the CC-model that has a cache memory. This is primarily due to the fact that any one of irregularly distributed cache misses results in a memory access that takes t_m cycles. The interleaved memory with pipelining may well satisfy the bandwidth requirement of the processor. However, as the speed gap between processor and memory increases, i.e. the number of cycles needed for accessing memory increases, the performance of the CC-model takes over. With the blocking factor of 4K, the CC-model outperforms the MM-model after the memory access time exceeds 20 cycles. If the blocking factor is 2K, the CC-model shows better performance when the memory access time is over 7 cycles.

It is well known that cache performance depends on how often data are reused after they have been loaded into the cache. In Figure 4, the reuse factor is fixed at B. In order to observe how the reuse factor affect the performance of vector caches we plotted *clock cycles per result* as a function of reuse factor R while fixing the blocking factor at $1K$ as shown in Figure 5. Two curves are plotted for CC-model corresponding to memory access times of 8 and 16 cycles, and two other similar curves are plotted for the MM-model. As expected, the performance of the two processor models is the same when the reuse factor is 1. Recall that the initial loading of the cache is pipelined and is done concurrently with the vector operation. With the system parameters indicated in the figure, whenever the reuse factor is more than one, the CC-model performs better than the MM-model. Moreover, it can be seen in this figure that after the reuse factor exceeds certain value, the performance change is not significant.

We noticed earlier in Figure 4 that different blocking factors show different performances. To observe this effects more closely, we plotted the performance vs. blocking factors with other parameters being fixed at the values shown in Figure 6. For memory access time of 16 cycles, the blocking factor can not be more than 4K after which the vector cache performs poorly. The performance cross point of the CC-model and MM-model is for the blocking factor to be about 5K if memory access time is 32 cycles. However, our cache size for the CC-model is 8K words implying that only

a small portion of the cache is utilized.

From the above three figures, the following observations are in order. First of all, as the memory access time increases with respect to the clock time of processors, vector cache memory can provide better performance than the machine without cache even with a small reuse factor; Secondly, the performance improvement of the direct-mapped cache for vector processor is limited as shown by these figures. Finally, the utilization of the vector cache is very poor in order to sustain a reasonable performance.

4 Performance of the Prime-Mapped Cache

From the performance analysis presented in the last section, we observed that the performance gain resulting from adding a conventional direct-mapped cache into a vector computer is not very impressive. In order to obtain a reasonable performance improvement over the MM-model, the cache utilization has to be very low. In this section, we will show that the newly proposed cache mapping scheme can provide significant performance improvement over the conventional cache designs. Moreover, the cache utilization can be close to one without significant performance degradations. We consider three typical vector access patterns that are commonly encountered in vector processing: random multistride access, subblock access and FFT access. Unless otherwise specified, all the numerical results reported in this section are computed by fixing MVL and T_{start} at 64 and $30+t_m$ [2], respectively.

Random Stride Accesses

Since the modulus in the prime-mapped cache is a prime number, interference misses are minimum. For the random stride access pattern, the self-interference misses occur only when the stride is an integral multiple of the cache size. Therefore, I_s^C is simply given by

$$I_s^C(B) = \frac{(1 - P_{stride1})(B-1)}{C-1} t_m, \tag{8}$$

assuming that the stride is uniformly distributed between 2 and C if it is not 1. For cross-interferences, the footprint model similar to the direct-mapped cache can be used to estimate I_c^C. Substituting Equation (8) into Equation (7) we obtain the time for processing one vector element for prime-mapped cache. The total execution time is calculated using Equation (4).

In Figure 7, we plotted the clock cycles per result as a function of memory access time. Three curves are drawn corresponding to the MM-model, direct-mapped CC-model and prime-mapped CC-model respectively. Compared to the previous figures, we increased the number of memory banks from 32 to 64. As the memory access time increases with respect to the processor cycle time,

the performance of the MM-model is getting worse. The performance of the direct-mapped cache catches up with the MM-model after the memory access time exceeds about 24 cycles. However, average cycles per result of the CC-model keeps increasing though with a smaller slope than that of the MM-model. The prime-mapped cache, on the other hand, shows little change in performance as memory access time increases. It outperforms both the MM-model and the direct-mapped CC-model over entire range of memory times considered. When the memory access time matches the number of memory modules (64), the prime-mapped CC-model runs three times faster than the direct-mapped CC-model and almost five times faster than the MM-model.

As we observed in the last section, the blocking factor affect to a large extent the performance of direct-mapped vector cache. In order to observe this effect on prime-mapped cache, Figure 8 shows the performance as a function of blocking factor for the three machine models with the memory cycle time being half of the number of banks. As shown in the figure, average cycles per result for the direct-mapped cache quickly cross over that of the MM-model after the blocking factor is about 3K, while the average cycles per result for the prime-mapped cache remains flat. Thus, the prime-mapped cache is not very sensitive to the blocking factor for the random stride access.

In the above two figures, we have fixed the probability of accessing a vector with stride 1, $P_{stride1}$, at 0.25 which is the average value taken from the experimental data reported in [8]. To examine the effect of this parameter on system performance, we plotted clock cycles per result by varying $P_{stride1}$ as shown in Figure 9. High probability of unit stride will reduce both memory conflicts and cache line conflicts. As shown in the figure, as this probability increases, the performance difference between the two mapping schemes comes close. When the probability of having unit stride is 1, the performance of the two mapping schemes is the same. Whenever this probability is nonunit, the prime-mapped cache performs better than the direct-mapped cache.

Figure 10 shows the effect of proportion of double access streams on the performance of the three models. When the proportion of double access streams is large, cross-interferences between the two vectors in the cache becomes large. It is shown in the figure that the cycle time increases with the increase of the proportion of double stream accesses. Notice that the cross-interference in the prime-mapped cache is severer than that in direct-mapped cache because the footprint of the former is larger than the footprint of the latter. Nevertheless, the prime-mapped cache still performs better than the direct-mapped cache over all possible fractions of double stream accesses. The performance difference ranges from 40% to a factor of 2.

If we fix the stride of one vector at 1 and assume the other stride to be random, then we have accesses to rows and columns of a random sized matrix. Figure 11 shows the performance of such access patterns. If columns of the matrix were accessed more often than rows, we would expect small number of interferences in the direct-mapped cache because the stride is one for a column

access. If rows were accessed more often, on the other hands, we would expect more interferences in the direct-mapped cache as shown in Figure 11. In both cases, however, the prime-mapped cache shows the same performance which is better than that of direct-mapped cache.

Sub-block Accesses

Sub-block accesses have proven to be important in many vector processing applications. Blocked matrix multiply is an example that needs to access a submatrix of a large matrix. Let the original large matrix be dimensioned $P \times Q$, and the sub-block be dimensioned $b_1 \times b_2$. A sub-block access can be characterized as b_2 stride-1 accesses to column vectors of length b_1. The distance between the starting addresses of two successive column vectors is P.

Sub-block accesses can be easily made conflict free for any arbitrary two dimensional matrix by properly selecting blocking factors. Notice that this is either impossible or prohibitively costly for the MM-model since the modulus is a power of 2 [17]. Consider the above matrix with dimensions $P \times Q$ that is stored in a column-major. To make accesses to a sub-block of dimension $b_1 \times b_2$ conflict free in the prime-mapped cache, what we need to do is to select b_1 and b_2 in such a way that the following conditions are satisfied.

$$b_1 \leq min(P \ mode \ C, \ C - P \ mod \ C), \ and$$
$$b_2 \leq \lfloor \frac{C}{b_1} \rfloor.$$

A row-major storage can also be realized similarly by interchanging the corresponding dimensions. It can be easily shown that a sub-block with b_1 and b_2 satisfying the above conditions can be stored in the prime-mapped cache without self-interference. This is because that b_1 elements of any column vector are stored in consecutive memory locations and hence no conflict can occur among them. Moreover, since $(P \ mode \ C) \geq b_1$ the first elements of any two consecutive column vectors are mapped into two cache locations that are at least b_1 lines apart as long as $b_2 \leq \lfloor \frac{C}{b_1} \rfloor$. The fraction of the cache being used is $b_1 b_2 / C$. If we let $b_1 = min(P \ mode \ C, \ C - P \ mod \ C)$, and $b_2 = \lfloor \frac{C}{b_1} \rfloor$. then the cache utilization is close to 1. In other words, conflict free access is possible to the submatrix even with the cache utilization approaching 1.

FFT Accesses

The FFT accesses result from the Cooley and Tukey's Fast Fourier Transform (FFT) algorithm. The FFT is a basic tool in various scientific and engineering disciplines, ranging from oil exploration to artificial intelligence. Efficient FFT computation is desirable for any engineering/scientific computers. While the FFT algorithm performs well on scalar computers, it does not perform as well on vector computers because the access stride changes after each stage of computation. Moreover,

other than the final stage all strides are 2's powers, which results in a lot of line conflicts in a direct-mapped cache. The FFT in its original form can only keep a very small amount of data in the cache if the data size that has to be a power of 2 is greater than the cache size. A tremendous amount of efforts has been made during the past 20 years to optimize the performance of FFT on a vector machine [16].

One way to implement the FFT algorithm in a memory hierarchical system is to map the input data into a two dimensional array [16]. Suppose an N-point data array to be transformed can be represented as $N = B_2 \times B_1$. Then the input data can be considered to be a matrix of B_1 columns and B_2 rows. The matrix is stored in a column-major. The algorithm proceeds by first doing B_1-point row FFTs for B_2 times each of which is expected to be done within the local cache. After all B_2 row FFTs are done, we multiply twiddle factors and perform column FFTs for B_1 times. Again it is expected that each of the column FFTs is done inside the cache. If B_2 is less than the cache size, each of the column FFTs can be done in the cache without misses except for the compulsory misses since the array is stored in a column-major. However, the row FFTs of the first step may not be guaranteed to be done inside the cache without misses. Depending on the value of B_2, conflict misses may occur. The number of interference misses is obviously given by $B_1 - C/gcd(B_2, C)$ for $B_1 > C/gcd(B_2, C)$.

The total execution time of this algorithm on the direct-mapped cache can be computed by using Equation (4) twice. First, we substitute B and R in (4) by B_1 and $log_2 B_1$, respectively. T^C_{elemt} should include the above self interferences for a given B_2. Notice that $P_{ds} = 0$ if we assume that twiddle factors are available in the registers. Second, Equation (4) is used again with B and R being replaced by B_2 and $log_2 B_2$, respectively. T^C_{elemt} can be considered to be 1 assuming that $B_2 < C$. The final total execution time is the sum of the above two times. The execution time of prime-mapped cache can be done similarly by noting that there are no self-interference misses unless $B_2 = C$. Compulsory misses (T_B) should also be adjusted based on the FFT stride characteristics.

Figure 11 shows the performance comparison of the FFT algorithm on the two different cache organizations. The curves are the average clock cycles per point which is the total execution time divided by N. In the figure, we fix one dimension while varying the other. It is observed from these curves that the prime-mapped cache outperforms the direct-mapped cache by a factor of more than 2. The improvement is valid over all possible values of the blocking factor B_2. Agarwal [16] has designed an optimal FFT algorithm for IBM 3090 vector processor. It is also a two dimension FFT that has relative small B_1. A group of rows is loaded into the cache at each step of computation. After all the rows are transformed, B_1 B_2-point column FFTs are performed. The access pattern of this optimized FFT algorithm is somehow similar to the sub-block access since it essentially loads a submatrix into the cache. In this algorithm, the selection of B_2 is tricky in order to maximize

cache hit ratio since improper B_2 can make the cache performance very poor. A good choice of B_2 may lead to some other complicated algebraic manipulations. Moreover, if B_2 is greater than the cache size, it is unknown whether the algorithm is still optimal. With the prime-mapped cache, the FFT algorithm can be implemented very easily. Optimization is guaranteed as long as the block size is less than the cache size.

5 Conclusions

In this paper, we have proposed an innovative cache mapping scheme for vector computers, called prime-mapped cache. The cache lookup time of the new mapping scheme keeps the same as direct-mapped cache. Generation of cache addresses for accessing the prime-mapped cache can be done in parallel with normal address calculations. This address generation takes shorter time than the normal address calculation due to the special properties of the Mersenne prime. Therefore, the new mapping scheme does not result in any performance penalty as far as the cache access time is concerned. The hardware cost introduced by the prime mapping scheme includes 2 multiplexors, a full adder and a few registers.

We have also developed an analytical performance model for a generic vector computational model that covers a wide spectrum of numerical algorithms. The analytical model is simple and precise for a given set of vector access patterns. Three typical vector access patterns have been considered in our performance analysis: random multistride, submatrix and FFT accesses. It is shown that the prime-mapped cache outperforms both the vector computer without cache and the vector computer with a direct-mapped cache for all vector access patterns considered. The performance improvement ranges from 40% to a factor of 3 depending on the memory cycle time, blocking factor and access patterns.

Our conclusion is that cache memory can improve the performance of vector processing provided that application programs can be blocked. With the new mapping scheme, the cache memory can provide significant performance improvement which will become larger as the speed gap between processor and memory increases. Further studies are needed to collect experimental data for the new design.

References

[1] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, pp. 472–530, Sept. 1982.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 1990.

[3] H. S. Stone, *High Performance Computer Architecture.* Addison-Wesley, 1990.

[4] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of Arch. Supp. for Prog. Lang. and Opr. Sys.*, pp. 63–74, April 1991.

[5] K. So and V. Zecca, "Cache performance of vector processors," in *Proc. 15th. Int'l Symp. on Comp. Arch.*, pp. 261–268, 1988.

[6] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," in *Int'l Conf. on Supercomputing*, 1987.

[7] J. Dongarra and et al., "A set of level 3 basic linear algebra subprograms," *ACM Trans. on Math. Soft.*, vol. 16-1, pp. 1–17, March 1990.

[8] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th. Int'l Symp. on Comp. Arch.*, pp. 54–63, 1991.

[9] D. H. Bailey, "Vector computer memory bank contention," *IEEE Trans. on Computers*, vol. C-36, pp. 293–298, MARCH 1987.

[10] W. Abu-Sufah and A. D. Malony, "Vector processing on the Alliant FX/8 multiprocessor," in *Int. Conf. on Parallel Processing*, pp. 559–566, Aug. 1986.

[11] D. Bhandarkar and R. Brunner, "VAX vector architecture," in *Proc. 17th. Int'l Symp. on Comp. Arch.*, pp. 204–215, 1990.

[12] M. D. Hill, "A case for direct-mapped caches," *IEEE Computer*, pp. 25–40, Dec. 1988.

[13] P. Budnik and D. J. Kuck, "Organization and use of parallel memories," *IEEE Trans. on Computers*, pp. 1566–1569, Dec. 1971.

[14] D. H. Lawrie and C. R. Vora, "The prime memory system for array access," *IEEE Trans. on Computers*, vol. C-31, pp. 435–441, May 1982.

[15] J. Armstrong, "Algorithm and performance notes for blocked LU factorization," in *Int. Conf. on Parallel Processing*, pp. III–161–164, Aug. 1988.

[16] J. W. Cooley, *The Structure of FFT and Convolution Algorithms.* IBM T. J. Watson Research Center, 1990. Research Report.

[17] D. T. HarperIII, "Block, multistride vector, and FFT accesses in parallel memory systems," *IEEE Trans. on Parallel and Distributed Systems*, Jan. 1991.

[18] W. Oed and O. Lange, "On the effective bandwidth of interleaved memories in vector processor systems," *IEEE Trans. on Computers*, vol. C-34, pp. 949–957, Oct. 1985.

[19] R. Raghavan and J. P. Hayes, "On randomly interleaved memories," in *Proceedings of Supercomputing-90*, pp. 49–58, Nov. 1990.

[20] A. J. Pettofrezzo and D. R. Byrkit, *Elements of Number Theory.* Prentice-Hall, 1970.