Performance of Cache Memories for Vector Computers *

Qing Yang
Dept. of Electrical and Computer Engineering
University of Rhode Island,
Kingston, RI 02881
Tel. 401-792-5880

ABSTRACT

Recent studies [5] have shown that memory system is the major bottleneck of vector computers. Performance measurements on CRAY-2 computers indicate that as much as a factor of 4.4 performance degradation results from memory latency. In this paper, we study the performance of incorporating cache memories into vector computers. Two cache organizations are considered: direct-mapped cache and prime-mapped cache that we proposed [22]. We analyze the caching behavior of three typical blocked algorithms for numerical applications: matrix multiplication, Gaussian elimination and FFT. By analyzing the algorithm structures in conjunction with system architectures, we develop analytical models based on real applications rather than statistical estimate. Our performance models give the expected value of execution time of an algorithm averaged over a wide range of problem sizes. Performance measurements of the algorithms on a real machine are carried out to validate our analysis. Numerical results show that the prime-mapped cache minimizes cache miss ratio caused by line interferences that are critical for numerical applications.

*This research is supported in part by NSF grants #MIP-9208041.

1 Introduction

As the gap between memory cycle times and processor cycle times grows, memory system design has become a crucial challenge to supercomputer designers. High performance computers need sophisticated memory hierarchies that keep memory latency minimum. Several recent vector computers feature a cache-based memory hierarchy such as IBM 3090 [18], Alliant FX/8 [1] and Vax/600 [4]. Despite the essential importance of memory hierarchies to supercomputers, little has been done on performance evaluation of cache memories for such systems. In [22, 23], we have proposed a new cache design for vector computers [22] called prime-mapped cache by utilizing the special properties of a Mersenne prime. A simple probabilistic analysis assuming random stride vector accesses has shown that the new design has a great performance potential for future supercomputers [22]. The purpose of this paper is to analyze the caching behavior of real application algorithms on the architecture. We develop analytical models based on realistic algorithms to study the performance of memory hierarchy for supercomputers. Performance measurements on a real machine are carried out to validate our analysis. It is shown that our analysis is in a good agreement with the actual measurements.

Evaluating the performance of cache-based computer systems is a difficult task because of the complexity of program behaviors. One needs to consider the locality property of an application and reuse factors among other things. Traditional performance evaluation techniques can be broadly classified into three categories: trace-driven simulation[2, 9], event-driven simulation[20] and analytical modeling [21, 7, 19]. Trace-driven simulation is based on actual traces of programs running on a system. Therefore, it provides the most reliable and accurate performance estimates for given programs on a given system. Since the collection of traces is a time consuming process and requires special hardware or software supports, trace-driven simulation may not be effective in capturing the exact performance behavior of hypothetical architectures. Moreover, trace-driven simulation becomes prohibitively costly if one considers a wide range of problem sizes to which vector processing performance is sensitive. It is shown in [15] that an algorithm with one problem size can run at twice the speed of the same algorithm with a different size. Discrete-event driven simulation and analytical modeling, on the other hand, assume certain stochastic work load model[8, 21]. Performance estimates of a system can be derived fairly easily based on the assumed probability distributions of general program behaviors. However, actual program executions show, to a large extent, locality properties. No probability distributions can capture the exact behavior of program execution on the vector computers. Hence the probabilistic analysis may result in distorting the execution pattern of application algorithms. Our approach here is to analyze the performance of cache-based vector computers by considering actual algorithms running on the system. Through the analysis of the structural and behavioral properties of an algorithm, the precise effects of different

cache parameters on the system performance can be evaluated using analytical formulas that would otherwise be obtained from traces.

The application algorithms considered in this paper are matrix multiplication, Gaussian elimination, and FFT. The reasons for selecting these algorithms should be obvious. Matrix multiplication is a well understood matrix algorithm and is an interesting case study for cache memories because locality is carried in three different loops by three different variables; Gaussian elimination for solving linear systems of equations is the core of Linpack benchmark; and the Cooley and Tukey's Fast Fourier Transform (FFT) algorithm is a basic tool in various scientific and engineering disciplines, ranging from oil exploration to artificial intelligence. Efficient computation of these algorithms is desirable for any engineering/scientific computers. In order to show the potential benefits of the new cache organization, we consider all possible problem sizes and analyze the average performance.

In the next section, we present two simple vector architecture models based on which our performance analysis is carried out. Section 3 presents the analytical models based on the three algorithms. Performance measurements on the IBM ES/9000 vector computer are also presented in Section 3 to validate the analysis. We will carry out performance analyses and comparisons in Section 4. Our performance analysis on the application algorithms shows that the prime-mapped cache improves vector processing performance by 40% to a factor of 5 over conventional cache designs. Section 5 concludes the paper.

2 Architectural Models

2.1 Two Simple Vector Architectures

Two simplified vector processor models considered here are shown in Figures 1 and 2 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 being V_L words, and a set of 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. The second processor model (CC-model) shown in Figure 2 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 [4]. The number of words in each cache line is designated by L which is called cache line size. It is assumed that a line of data can be transferred on one of the buses within one cycle.

There are totally $M = 2^m$ low-order-bit interleaved memory banks. The memory access time

(reservation time) of these memory banks is assume to be t_m cycles. Memory accesses to different banks can be carried out concurrently provided that conflicts do not occur. The cache memory shown in Figure 2 is used to bridge the speed gap between processor and memories. In our performance analysis, we consider two cache mapping organizations as outlined below.

2.2 Direct-Mapped cache

Direct-mapped cache is widely understood and successfully implemented in many general purpose computers [13]. The essential idea of the direct-mapped cache is that it consists of $C = 2^c$ lines and a data item with address A is mapped into line # $A \ mod \ 2^c$ in the cache. Since the modulus is a power of 2, the line number of a data item in the cache is a subfield of its memory address. As a result, cache access and address generation are greatly simplified. Cache hit time is minimal since it does not need associative search, which makes it very attractive for computer implementations. However, as observed by Lam et al. [15], cache line interferences increase drastically and dominate cache misses in a direct-mapped cache for numerical applications. In order to maintain a reasonable cache performance, the cache utilization (the fraction of cache that is used to hold useful data) has to be extremely small (about 3%).

2.3 Prime-Mapped Cache

We have recently proposed a cache design with a novel mapping scheme called **prime-mapping** that attempts to minimize interference misses [22, 23]. The main idea of the prime-mapped cache is that memory data are mapped into cache according to a Mersenne prime. The cache access logic of the 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 accessing the prime-mapped cache is done in parallel with normal address calculations resulting in no performance penalty. In other word, the prime-mapped cache does not increase the length of the critical path of a processor. Since Mersenne number is a prime, a vector access to the cache with this mapping scheme can be made conflict-free. Thus, it has the advantages of both direct-mapped cache and fully associative cache if replacement is not considered. The detailed description of the *Prime-mapped* scheme can be found in [22, 23].

3 Performance Analysis

3.1 Basic Model for Vector Performance

As mentioned in the introduction, we shall use analytical models to study the performance of the two vector architecture models (MM-model and CC-model). Consider a simple SAXPY operation on two vectors, X, Y and a scalar a:

$$Y := aX + Y.$$

Assuming that the vector length of X and Y is B, the execution time of this operation can be expressed as [5, 12]

$$T_b^x[B] = \lceil \frac{B}{V_L} \rceil T_{start}^x + BT_{elemt}^x, \tag{1}$$

where

B is the vector length,
T_{elemt}^x the time for processing one vector element,
T_{start}^x the strip startup time,
V_L the vector register length of the machine.

The superscript x can be M or C to distinguish MM-model (Figure 1) and CC-model (Figure 2) vector processors, respectively. For instance, T_{elemt}^M is the time for processing one vector element with the MM-model whereas T_{elemt}^C is the time needed to process one vector element with CC-model. In addition, the execution time, $T_b^x[\cdot]$, should also include the startup overhead for each outer loop which will be considered latter with respect to each realistic application algorithm.

The start-up time, T_{start}^x, depends on the memory access time and the type of arithmetic operations to be performed. It should not be difficult to determine its value for a given set of system parameters. If the functional unit is fully pipelined, T_{elemt}^x should be one in an ideal case. In reality, T_{elemt}^x is often greater than one because of stalls resulting from memory accesses, data dependencies and control dependencies. Since our primary interest here is memory system performance, we will concentrate on memory stalls only. In the following subsections, we will analyze memory stalls precisely based on the three application algorithms. The effects of the memory stalls on the vector performance are incorporated in T_{elemt}^x.

3.2 Matrix Multiplication

Consider the multiplication of an $N_1 \times N$ matrix X with an $N \times N_2$ matrix Y. Based on the same blocking strategy as the one described in [15], we partition X into N/B_1 submatrices of size $N_1 \times B_1$

and Y into submatrices of size $B_1 \times B_2$, as shown in Figure 3. Let $X[i, j \cdots j+B]$ designate a row vector in the ith row of X. It consists of $B+1$ elements with the second subscript values lying between j and $j+B$. The same notation holds for column vectors except that the subscript range is placed in the first subscript position. With this notation, the vectorized version of the blocked matrix multiplication algorithm is shown below.

Blocked Matrix Multiplication Algorithm

1. $j := 1$;
2. for $jj = 1$ to $\frac{N_2}{B_2}$ do begin /* jj-loop */
3. for $kk = 1$ to $\frac{N}{B_1}$ do /* kk-loop */
4. for $i = 1$ to N_1 do /* i-loop */
5. for $k = B_1(kk-1)+1$ to kkB_1 do /* k-loop */
6. $Z[i, j \cdots j+B_2-1] := Z[i, j \cdots j+B_2-1] + X[i,k] \cdot Y[k, j \cdots j+B_2-1]$
7. $j := j + B_2$;
8. end jj.

For clarity, we assume that B_1 divides N and B_2 divides N_2. At the beginning of each i-loop, vectors $X[i, kk \cdots kk+B_1-1]$ and $Z[i, j \cdots j+B_2-1]$ are loaded to vector registers. In each iteration of the inner most loop (k-loop), a sub-row vector of Y is loaded from the main memory into a vector register before the calculation begins.

3.2.1 MM-Model

Let us first consider the execution time of each iteration of i-loop of the above algorithm on the MM-model, denoted by T_i^M, which is given by

$$T_i^M[B_1] = T_o + B_1 T_b^M[B_2], \tag{2}$$

where $T_b^M[\cdot]$ is defined in Equation (1) and T_o is the overhead for vector startup of i-loop. T_o includes the time needed to load $X[i, kk \cdots kk+B_1-1]$ and $Z[i, j \cdots j+B_2-1]$, the overhead for computing the starting addresses, setting up vector controls, and executing scalar code for strip-mining. According to the algorithm, X and Z are loaded only once after every B_1 vector operations, which is much less frequently than Y being loaded. Therefore, the effect of memory stalls of loading

X and Z is relatively insignificant compared to the effects of loading Y. Notice also that as soon as the first elements of X and Z are successfully loaded in the registers, the arithmetic operations can start. Therefore, T_o can be considered as the sum of memory cycle time, t_m, and the startup cost. The total execution time of multiplying an $N_1 \times N$ matrix X with an $N \times N_2$ matrix Y, $T_{Matrix}[N_1, N, N_2]$, is given by

$$T_{Matrix}[N_1, N, N_2] = \frac{N_1 N N_2 \cdot T_i[B_1]}{B_1 \cdot B_2}. \tag{3}$$

Memory stalls due to bank conflicts when loading row vectors of Y are incorporated into the time for processing one vector element, T^M_{elemt}. The major source of memory stalls for MM-model is memory conflicts [11, 17]. We first consider column-major storage scheme while row-major scheme is considered later in this section. Based on the storage scheme, any two consecutive vector elements in a row of Y are N locations apart. Let $s = N \ mod \ M$. Then s is the access stride in terms of memory bank number. Since the stride is directly related to the matrix size which can be any value, it is reasonable to assume that the stride is uniformly distributed between $(1, 2 \cdots, M)$. That is, s takes any integer in $(1, \cdots, M)$ with probability $\frac{1}{M}$. It should be noted that strides greater than M need not be considered due to the modular operation.

Let us consider the number of possible stall cycles of accessing vector $Y[k, j \cdots j + B_2 - 1]$ from the main memory. When the vector access stream traverses across all memory banks (one sweep) with stride s, the number of memory banks visited by the access stream is given by $M/gcd(M, s)$, where $gcd(M, s)$ is the greatest common divisor of M and s. $M/gcd(M, s)$ is also called return number meaning the number of memory banks visited before the same memory bank is revisited again. If this return number is less than both the vector length to be accessed and the memory cycle time, memory conflicts occur. As a result, the first conflicting vector element will be delayed by $t_m - M/gcd(M, s)$ cycles. There are $B_2/(M/gcd(M, s))$ such conflicts as seen by a contended memory bank. For a special case where $gcd(M, s) = 2^m = M$, all element requests go to a single memory bank, and each of B_2 elements will be delayed by $t_m - 1$ cycles. Since M is a 2's power, $gcd(M, s)$ is in the form of 2^i for some $i = 0, \cdots m-1$. For each 2^i, there may be a number of values of s within M such that $gcd(M, s) = 2^i$. In order to find the average number of conflicts, we need to find out the number of integers within M that share the same $gcd(M, s) = 2^i$. This number is clearly the same as the number of values of s such that $gcd(\frac{M}{2^i}, \frac{s}{2^i}) = 1$, which is Euler's function [16] given by

$$\phi(\frac{M}{2^i}) = \frac{M}{2^{i+1}} = 2^{m-i-1}, \quad for \ i < m.$$

That is, the probability of the stride s being such a value that makes $gcd(M, s) = 2^i$ is $\frac{2^{m-i-1}}{M}$. For each such s, $\frac{B_2}{M/2^i}$ sweeps are delayed by $(t_m - M/2^i)$ cycles. Therefore the average number of stall

cycles resulting from accessing a vector of length B_2, I_{st}^M (the letter I is for memory Interference), is given by

$$I_{st}^M = \sum_{i=\lceil log_2 \frac{M}{t_m} \rceil}^{m-1} \frac{2^{m-i-1}}{M}(t_m - \frac{M}{2^i})\frac{B_2}{M/2^i} + \frac{1}{M}B_2(t_m - 1),$$

assuming that $t_m \leq M$. The lower limit of the sum in the above equation is determined based on the condition $t_m \geq M/2^i$. The second term of above expression takes care of stride being a multiple of M. Simplifying the above equation, we have

$$I_{st}^M = \frac{B_2}{M} \cdot [t_m + \frac{t_m}{2}\lfloor log_2 t_m \rfloor - 2^{\lfloor log_2 t_m \rfloor}].$$

The time for processing one element of the vector ignoring the start-up time, T_{elemt}^M, can now be expressed as

$$\begin{aligned} T_{elemt}^M &= 1 + average\ stall\ cycles\ per\ element \\ &= 1 + \frac{1}{M} \cdot [t_m + \frac{t_m}{2}\lfloor log_2 t_m \rfloor - 2^{\lfloor log_2 t_m \rfloor}]. \end{aligned} \quad (4)$$

Substituting Equation (4) into Equation (1) and then Equation (2), we obtain the execution time for each i-loop which is used in Equation (3) to obtain the total execution time of the algorithm on the MM-model.

3.2.2 Direct-Mapped CC-Model

Observing the blocked matrix multiplication algorithm, we find that a sub-row of Y is loaded at each iteration of the inner most loop. These memory accesses are the most time consuming part of the algorithm. Therefore, we can expect substantial speedup by putting a block of Y, $Y[kk \cdots kk + B_1 - 1, j \cdots j + B_2 - 1]$, in the vector cache memory. Once the block is loaded into the vector cache, it should be reused for N_1 times according to the algorithm in an ideal situation. In reality, however, cache misses may occur. These cache misses may have significant effect on system performance since each cache miss results in t_m stall cycles.

In general, cache misses can be classified into three categories [12]: 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. We call such conflicts *cache line interferences*.

Now let us consider the CC-model of Figure 2 that has a direct-mapped cache of size $C = 2^c$ sets. Assuming that Y is stored in column-major, then element $Y[i,j]$ is stored in memory location $Y_b + (j-1)N + (i-1)$, where Y_b is the starting address of the matrix. With the column-major storage scheme, elements of a column of Y are in consecutive memory locations. As a result, a cache line of size L words contains an L-element subcolumn of Y. We will use this subcolumn length as one blocking factor B_1 (see Figure 3), i.e. the cache line size L equals B_1.

At the beginning of each kk-loop ($i = 1$), the processor loads a subblock of Y into the cache while finishing the first i-loop for $i = 1$. The process of the initial cache loading will take the amount of time given by Equation (2). After the block of Y is loaded into the cache, the remaining $N_1 - 1$ iterations of i-loop are expected to be performed in the cache if miss does not occur. The total execution time of the matrix multiplication algorithm on the CC-model (Figure 2) is given by

$$T^C_{Matrix} = \frac{N_2 N}{B_1 B_2}\{T^M_i[B_1] + (N_1 - 1) \cdot (T_o + B_1 T^C_b[B_2])\}. \tag{5}$$

where $T^M_i[B_1]$ (given in Eq.(2)) covers the effects of compulsory and capacity misses as discussed above. The interference misses will be taken care of by T^C_{elemt} in $T^C_b[B_2]$.

Suppose that the processor tries to load B_2 cache lines of B_1 words each into the vector cache. Based on the storage scheme, any two cache lines (any two consecutive columns of a subblock of Y) are separated by N memory locations. Let $s = N \ mod \ C$. Then s is the access stride in terms of cache line number. The number of cache lines occupied as a result of the load sequence is given by $C/gcd(C,s)$, where $gcd(C,s)$ is the greatest common divisor of C and s. If the vector length B_2 is greater than $C/gcd(C,s)$, cache line conflicts (interference) occur. The number of cache line-interferences is clearly $B_2 - C/gcd(C,s)$. In order to be able to reuse the $C/gcd(C,s)$ line data that remain in the cache, we suppose that the cache is lockup free meaning that cache accesses and memory accesses can be done concurrently. In this case, the data already in the cache can be loaded to a register before they are overwritten by new conflicting cache lines. Thus, every cache access to a vector of length B_2 will result in $B_2 - C/gcd(C,s)$ cache misses based on the access pattern of the algorithm.

The number of cache misses is clearly dependent on the relative value of cache size C and and access stride s. We again assume that the stride is uniformly distributed between $(1, \cdots C)$. That is, s takes any integer in $(1, \cdots C)$ with probability $1/C$. Similarly, strides greater than C need not be considered due to the modular operation. Because C is a 2's power, $gcd(C,s)$ is in the form of 2^i for some $i \in (0, 1, \cdots, c-1)$. For each 2^i, there may be a number of values of s within C such that $gcd(C,s) = 2^i$. The number of integers within C that share the same $gcd(C,s) = 2^i$ is the same as the number of values of s such that $gcd(\frac{C}{2^i}, \frac{s}{2^i}) = 1$, which is Euler's function [16] given by $\phi(\frac{C}{2^i}) = \frac{C}{2^{i+1}} = 2^{c-i-1}, \quad for \ i < c$. For each such s, $B_2 - C/2^i$ line interferences occur.

If $gcd(C, s) = C$, then there would be $B_2 - 1$ conflicts. Therefore the average number of stall cycles resulting from accessing a vector of length B_2, I_{st}^C, is given by (STalls due to Cache line Interference),

$$I_{st}^C = \frac{1}{C}[\sum_{i=\lceil log_2 \frac{C}{B_2} \rceil}^{c} (B_2 - \frac{C}{2^i})2^{c-i-1} + B_2 - 1] \cdot t_m, \tag{6}$$

assuming that the cache is lockup free. The lower limit of the above summation is determined in such a way that $B_2 - C/2^i$ is always positive. For every line-interference miss the processor stalls for t_m cycles. Simplifying the above expression, we have

$$I_{st}^C = \frac{1}{3C}(3B_2 2^{\lfloor log_2 B_2 \rfloor} - 2 \cdot 2^{2\lfloor log_2 B_2 \rfloor} - 1) \cdot t_m. \tag{7}$$

For B_2 being a power of 2, we have

$$I_{st}^C = \frac{1}{3C}(B_2^2 - 1) \cdot t_m.$$

Thus, the time for processing one vector element is given by

$$T_{elemt}^C = 1 + \frac{I_{st}^C}{B_2}. \tag{8}$$

Substituting Equation (8) into Equation (1) and then Equation (5), we obtain the total execution time of the matrix multiplication algorithm on the direct-mapped CC-model.

To this point, one may argue that Y can be stored in row-major so that the cache would do better. The fact is, however, row-major storage will end up with the similar performance behavior as the column-major storage because the data stored in the cache is a two dimensional subarray. With row-major storage, a cache line will contain a subrow of Y. A subblock of Y will then have a number of subrows that are separated by the row length of Y, which will not make any difference as far as the number of line conflicts is concerned since we consider random matrix size. Although it may appear that row-major storage results in less number of loads from the main memory per vector operation, the data mapped to cache lines with potential conflicts can never be reused making the cache virtually useless. This fact is evidenced by our performance measurements on the IBM ES/9000 vector computer. We run the algorithm for two different matrix sizes of Y: 512 by 512 and 511 by 511. The former gives memory access stride of 512 and the latter gives the stride of 511 which is relative prime to the number of sets in the cache. By interchanging the two inner most loops, we obtain vectorized code either along columns or along rows of Y. As shown in Figure 4, we see performance difference between the two problem sizes no matter how we pick up vectors. The reason why we consider column-major will be evidenced shortly in the next subsection when we analyze the performance of the Gaussian elimination algorithm.

3.2.3 Prime-Mapped CC-Model

Since the modulus in the prime-mapped cache is a prime number, interference misses are minimum. For a random stride, the line-interference misses occur only when the stride is an integral multiple of the cache size. Therefore, I_{st}^{C} is simply given by

$$I_{st}^{C} = \frac{(B_2 - 1)}{C} t_m, \tag{9}$$

assuming that the stride is uniformly distributed between 1 and C. Substituting Equation (9) into Equation (8) we obtain the time for processing one vector element with prime-mapped cache. The total execution time is calculated using Equation (5).

3.3 Model Validation Through Performance Measurements

In order to verify the accuracy of our analytical model presented here, we carry out performance measurements of the algorithm on the IBM ES/9000 vector computer that has cache memories. The function CPUTIME is used to measure the execution times of a program. All the system parameters used in the analytical formulas are chosen empirically based on our experiments. Memory cycle time t_m is assumed to be 30 CPU cycles. The start up times, T_{start} and T_o, are fixed at 9.5 μsec.

The blocked matrix multiplication algorithm of Section 3.2 was written in Fortran. The inner most loop of the Fortran code spans a subrow vector of Z and Y so that the vectorized code will have the same structure as the one presented here. We run the program for two different matrix sizes: one with column length of Y being 512 and the other with the column length being 511 which is relative prime to the number of sets in the cache. Since exhausting all possible matrix sizes in our experiments is impractical, we use the above two representative sizes to obtain approximate average (weighted-average) performance for the purpose of comparison with the analytical model for direct-mapped cache. Notice that the effect of having prime-mapped cache is equivalent to having a problem size (stride) that is relative prime to the number of sets in the cache. The details of the experiments can be found in [14]

Figure 5 shows the comparison between our analytical model and actual measurements for the matrix multiplication algorithm. The execution times measured in terms of second are plotted as functions of blocking factor B_1. It is shown in this figure that our analytical models for both direct-mapped cache and prime-mapped cache match excellently with the measurement results. Not surprisingly, when we change blocking factor B_2 while fixing B_1 similar agreements between analysis and measurements are observed as shown in Figure 6. In general, our analytical models agree well with all measurement results with different sample problem sizes (such as 128×128, $1k \times 1k$, etc.) [14].

3.4 Gaussian Elimination

In order to make efficient use of memory hierarchy, a number of blocked LU decomposition algorithms stands out in the literature. We consider here the block algorithm described in [3] which is similar to the BLAS3. Suppose the matrix A to be factorized is dimensioned $N \times N$ and is stored in a column-major. The block LU factorization algorithm consists of the following two steps. First, the algorithm computes the subblocks B_1 and B_2 (see Figure 7). Then B_1 and B_2 are used in the main algorithm as black blocks to do factorization.

Let the current stage of the reduction be k and the block size p which is equal to the cache line size L. We assume that p divides N for the sake of discussion. The main loop for computing B_1 and B_2 is executed for $j = k, \cdots k+p-2$. For each j, b_1 and b_2^t are computed as follows (see Figure 7)

$$b_1 := b_1 - B_1^{j-1} u_{j+1} \qquad b_2^t := b_2^t - l_{j+1}^t B_2^{j-1},$$

where blocks B_1 and B_2 at stage j can be mathematically expressed as follows

$$B_1^j = \left[B_1^{j-1} \; b_1 \right] \quad B_2^j = \begin{bmatrix} u_{j+1} & B_2^{j-1} \\ 0 & b_2^t \end{bmatrix},$$

as shown in Figure 7. The following vectorized algorithm computes $B_1 = A[k+p \cdots N, k \cdots k+p-1]$ and $B_2 = A[k \cdots k+p-1, k+p \cdots N]$ that will be used in the main algorithm.

Subroutine Computing B_1 and B_2

1. Pivot for stage k- primer for the algorithm
2. for $j = k$ to $k+p-2$ do begin
3. $\quad s := 1/A[j,j]$;
4. $\quad A[j+1 \cdots N, j] := s \cdot A[j+1 \cdots N, j]$;
5. $\quad$ for $i = k$ to j do $\quad$ /* compute $b_1 := b_1 - B_1^{j-1} u_{j+1}$ */
6. $\quad\quad A[j+1 \cdots N, j+1] := A[j+1 \cdots N, j+1] - A[j+1 \cdots N, i] \cdot A[i, j+1]$;
7. $\quad$ Pivot for stage $j+1$;
8. $\quad$ for $i = k$ to j do $\quad$ /* compute $b_2^t := b_2^t - l_{j+1}^t B_2^{j-1}$ */
9. $\quad\quad A[j+1, j+2 \cdots N] := A[j+1, j+2 \cdots N] - A[j+1, i] \cdot A[i, j+2 \cdots N]$;
10. end j

11. $s := 1/A[k+p-1, k+p-1]$;

12. $A[k+p \cdots N, k+p-1] := s \cdot A[k+p \cdots N, k+p-1]$;

The execution time of this subroutine at stage k, $T_{B_{1,2}}[k]$, is given by

$$T_{B_{1,2}}[k] = T_{piv} + T_b^{col}[N-k-p+1] + \sum_{j=k}^{k+p-2} \{T_{piv} + T_b^{col}[N-j] + \sum_{i=k}^{j} (2T_o + T_b^{col}[N-j] + T_b^{row}[N-j-1])\}, \tag{10}$$

where

T_{piv} is the time for pivoting and scalar operations such as statements 1 and 11, or statements 3 and 7.

T_o the vector startup overhead which is the same as the one used in Equation (2).

$T_b^{col}[\cdot]$ the basic processing time for a column vector defined in Equation (1).

$T_b^{row}[\cdot]$ the basic processing time for a row vector defined in Equation (1).

We distinguish processing time for a column vector from processing time for a row vector here because of the different memory access times due to stride difference. A column vector can be accessed conflict free from memory banks while a row vector accessed from the main memory may result in bank conflicts depending on the access stride which is related to the size of the original matrix.

The main algorithm is as follows

1. $l := N/p$

2. $k := 1$

3. for $i = 1$ to l do

4. Compute B_1 and B_2;

5. $A_p := A_p - B_1 B_2$;

6. $k := k + p$

7. end i

The total execution time of the blocked LU decomposition algorithm can now be expressed as

$$T_{LU}[N] = (\frac{N}{p} + 2)T_{scal} + \sum_{i=1}^{N/p} (T_{B_{1,2}}[k] + T_{Matrix}[m, p, m] + mT_b^{col}[m]), \tag{11}$$

where $k = 1 + (i-1)p$ and $m = N - ip$. The first term in the above equation gives the execution time of scalar operations in statements 1, 2, and 6 of the main algorithm, and the second term gives the execution time of the main i-loop. In the main i-loop, B_1 and B_2 are first computed for each i, and then multiplied, followed by a matrix subtraction. The processing time for these three steps are specified by the three terms in the summation. Averaging over all possible values of $N \in (p, 2p, \cdots, pC)$, we have the average execution time

$$\overline{T}_{LU} = \sum_{n=1}^{C} \frac{T_{LU}[np]}{C}. \tag{12}$$

Both Equations (10) and (11) contain the basic vector processing time $T_b^{col}[\cdot]$ and $T_b^{row}[\cdot]$ that may differ depending on the architecture model (MM-model or CC-mode). With the MM-model, operand vectors are first loaded from the main memory to registers before the operations start. Since the matrix is stored in column-major, T_{elemt}^{M} used in calculating T_b^{col} can be considered to be one whereas the T_{elemt}^{M} used in calculating $T_b^{row}[\cdot]$ is given by Equation (4). The second term in the summation of Equation (11), T_{Matrix}, can be calculated using Equation (3) by setting $N_1 = m$, $N = B_1 = p$ and $B_2 = N_2 = m$.

Now let us consider the execution time of the algorithm on the direct-mapped CC-model. Similar to the analysis of matrix multiplication, suppose that the subblock B_2 is loaded into the vector cache while the factorization is in progress. As a result, the term $T_b^{row}[\cdot]$ in Equation (10) is replaced by $T_b^{C}[\cdot]$ defined by Equations (1), (7) and (8) with B and B_2 being substituted by $N - j - 1$. The term $T_{Matrix}[m, p, m]$ in Equation (11) is replaced by T_{Matrix}^{C} defined in Equation (5) by setting $N_1 = m$, $N = B_1 = p$ and $B_2 = N_2 = m$. For the prime-mapped CC-model, the execution time is estimated in the same way as for the direct-mapped CC-model except that all I_{st}^{C}'s are calculated using Equation (9).

3.5 Fast Fourier Transform

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 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 vector machines [6].

One way to implement the FFT algorithm in a memory hierarchical system is to map the input data into a two dimensional array [6]. Suppose an N-point data array to be transformed can be represented as $N = B_1 \times B_2$. Then the input data can be considered to be a matrix of B_1 rows and

B_2 columns stored in a column-major as shown in Figure 8. The algorithm proceeds by first doing B_2-point row FFTs. A group of L rows are transformed simultaneously with the vector length of L elements along the columns. Each row FFT contains $Log_2 B_2$ computation stages. Each stage has $B_2/2$ butterflies. The first stage loads data from the main memory, which takes amount of time given by

$$T_b^M[L] = \lceil \frac{L}{V_L} \rceil T_{start}^M + L T_{elemt}^M, \tag{13}$$

where T_{elemt}^M is derived as follows. Since each butterfly loads two vectors from the main memory with stride 1, there is no memory bank conflict among elements of the same vector. Memory stalls are mainly due to bank conflicts between elements of different vectors. Assuming that the starting elements of the two vectors are randomly placed in the memory, the probability that the two vectors collide is $\frac{2L}{M}$. If such collision occurs, the average distance of the first element of one vector from the first element of the other vector is $\frac{L}{2}$ resulting in $t_m - \frac{L}{2}$ stalls. The average number of stall cycles is therefore given by $\frac{2L}{M}(t_m - \frac{L}{2})$ if $t_m \geq L$. if $t_m < L$, memory stalls occur only when the distance between the first elements of the two vectors is less than t_m, which occurs with the probability $\frac{2t_m}{M}$. Again, the average distance is $\frac{t_M}{2}$ in this case giving rise to the average number of stall cycles being $\frac{t_m^2}{M}$ for $t_m < L$. Therefore, T_{elemt}^M is given by $T_{elemt}^M = 1 + \frac{2}{M}(t_m - \frac{L}{2})$ if $t_m \geq L$ and $T_{elemt}^M = 1 + \frac{t_m^2}{ML}$ if $t_m < L$. After L rows are loaded into the cache, the remaining $Log_2 B_1 - 1$ stages are expected to be done within the local cache. Each of these stages takes the following amount of time

$$T_b^C[L] = \lceil \frac{L}{V_L} \rceil T_{start}^C + L T_{elemt}^C, \tag{14}$$

where T_{elemt}^C is calculated by taking into account interference misses. Obviously, if $B_2 > C/gcd(B_1, C)$, then line conflicts occur in the direct-mapped cache and cache lines are replaced before they are reused due to the FFT access pattern. In case of prime-mapped cache, no line interference occurs as long as $B_2 < C$. Taking into account the outer loop overheads, the total execution time for the B_1 row FFTs is given by

$$T_{FFT}^{row} = \lceil \frac{B_1}{L} \rceil \{T_o + \frac{B_2}{2} T_b^M[L] + (log_2 B_2 - 1)\frac{B_2}{2} T_b^C[L]\}. \tag{15}$$

After all B_1 row FFTs are done, we multiply twiddle factors and perform column FFTs for B_2 times. Since the matrix is stored in column-major, several ($\lfloor \frac{C}{B_1} \rfloor$) consecutive columns can be held in cache without line conflict for both direct-mapped cache and prime-mapped cache. Vector operations are done along the rows of length ($\lfloor \frac{C}{B_1} \rfloor$). The total execution time for column FFTs is calculated in the same way as for row FFTs and is given by

$$T_{FFT}^{col} = \lceil \frac{B_2}{\lfloor C/B_1 \rfloor} \rceil \{T_o + \frac{B_1}{2} T_b^M[\lfloor C/B_1 \rfloor] + (log_2 B_1 - 1)\frac{B_1}{2} T_b^C[\lfloor C/B_1 \rfloor]\}, \tag{16}$$

where $T_b^M[\cdot]$ is given by Equation (13), and $T_b^C[\cdot]$ is given by Equation (14) with T_{elemt}^C being 1.

The total execution time of this algorithm on the CC-model is the summation of Equation (15) and Equation (16). The execution time on the MM-model is the sum of the following two expressions.

$$T_{FFT}^{rowM} = \lceil \frac{B_1}{V_L} \rceil \{T_o + (log_2 B_2)\frac{B_2}{2} T_b^M[V_L]\},$$

$$T_{FFT}^{colM} = \lceil \frac{B_2}{V_L} \rceil \{T_o + (log_2 B_1)\frac{B_1}{2} T_b^M[V_L]\}.$$

4 Numerical Results and Discussions

In this section, we present some numerical results based on the performance models developed in the previous section and compare the performance of the direct-mapped cache with that of the prime-mapped cache. 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 vector processing environment due to pipelining of memory accesses as mentioned earlier. We will use *speedup* resulting from adding cache memories into the MM-model vector computer. The *speedup* is defined as the ratio of the execution time of a given algorithm on the MM-model to the execution time of the same algorithm on the CC-model. Except where indicated otherwise, all of the following figures are for the case $V_L = 64$, $T_{piv} = 15$, $T_{scal} = 1$, $T_o = t_m + 15$, and $T_{start} = t_m + 15$. These parameters were chosen to be close to the commonly accepted values appeared in the literature [12]. The cache size is assumed to be 8K lines in our discussion although many larger cache sizes are available in commercial computers. The actual cache performance is dependent on the relative values of cache size, problem size and blocking factors. What is important is to keep the blocking factors less than cache size in order for the vector cache to be useful.

Figure 9 shows the *speedup* 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, particularly when blocking factor is large. 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 1K, the speedup of the CC-model over the MM-model is about 1.6 for memory access time being 32 cycles. It is clear from the figure that adding the prime-mapped cache memory triples the vector performance of the matrix multiplication algorithm after the memory time exceeds

28 cycles.

Now let us consider Figures 5 and 6 again. These figures show the effects of blocking factors (B_1, B_2) on the vector performance. Recall that matrix Y is blocked into submatrices of size $B_1 \times B_2$. During the matrix multiplication, we wish to store in the cache a submatrix which needs B_2 cache lines. With direct-mapped cache organization, some of the B_2 columns of a submatrix may be mapped to a same cache line resulting cache line interferences. With the prime-mapped cache, on the other hand, all B_2 subcolumns will be mapped to B_2 different cache lines without any cache line conflicts. As a result, the prime-mapped cache performs about 3 times better than the direct-mapped cache for all values of B_1 as shown in Figure 5.

We noted in Figure 5 that the two performance curves remain relative flat when we change the blocking factor B_1 (note that $B_1 = L$). This is because matrix Y is stored in column-major, i.e. elements along any column are in consecutive memory locations. As a result, changing subcolumn size (cache line size) does not change number of line conflicts as long as B_2 keeps the same. However, when B_1 is fixed while B_2 changes, the execution time of the algorithm on the direct-mapped cache organization increases drastically as shown in Figure 6. This increase in execution time results mainly from the increase of the number of cache line conflicts as more lines are loaded into the cache. The prime-mapped cache shows great performance advantage over the direct-mapped cache in this case. The performance keeps unchanged as long as the submatrix size is less than the cache size

Figure 10 shows the vector speedup of the Gaussian elimination algorithm as a function of memory access time, t_m. It can be seen from the figure that the performance difference between the two cache mapping schemes is dramatic. The direct-mapped CC-model does not improve the vector performance but worsens the performance, i.e. the speedup is less than 1. The main reason for this is the following. When we analyze the performance of the matrix multiplication algorithm, we noticed that the blocking factor has significant impact on the cache miss ratio. After the blocking factor exceeds certain value, the cache performs very poorly. In the Gaussian elimination algorithm, the submatrix size changes after every reduction step starting from the maximum size which is dimensioned $p \times (N - p)$. Therefore it is unavoidable that the cache has to hold a subblock of bad size during the course of factorization of a sufficiently large matrix. In addition, both column and row accesses are needed in the algorithm, which makes it virtually impossible to have a good access tride. Furthermore, as mentioned previously, the fact that cache misses are not easily pipelined makes the performance of direct-mapped CC-model worse than that of MM-model. The prime-mapped cache, on the other hand, performs up to 5 times better than the MM-model for large memory cycle times as evidenced in Figure 10.

The effects of cache line size on performance are shown in Figure 11. It is interesting to note that

the situation here is quite different from that of matrix multiplication (compare with Figure 5). As the line size changes, so does the performance of the Gaussian elimination algorithm. Recall that line size L is also the blocking factor p of the algorithm. For small line sizes, the locality is not very strong resulting in lower cache benefit. For very large line sizes or blocking factors, on the other hand, we end up with more arithmetic operations [10], which implies that the unenhanced portion of the execution time is increased. As a result, speedup reduces according to Amdahl's law. The optimal cache line size or blocking factor is at about 64 words that gives a factor of 5.5 performance improvement. Notice, however, that we observe a factor of 3 performance improvement even with line size of one word as shown in the figure. In other words, the proposed cache organization can triple the vector performance even for level 1 BLAS subroutines as long as the column or row length is less than cache size which is 8K-1 in our discussion.

The performance for the FFT algorithm is shown in Figure 12 as a function of memory cycle time. When memory access time is small (less than 6 cycles), adding cache memory does not improve vector performance. Notice that on the MM-model, all vector operations utilize the maximum vector register length ($V_L = 64$) while on the CC-model vector operations for row FFTs only use half of the maximum register length (32 words). The latter results in more strips as compared to the former. This is why the CC-model does not perform better than the MM-model for small memory cycle time. When the memory cycle time becomes larger, the CC-model takes over. The performance improvement of both mapping schemes increases as the memory time increases. As shown in the figure, the prime-mapped cache performs better than the direct-mapped cache The maximum performance improvement observed for the parameters considered here is about 40%.

Figure 13 shows how the performance changes with the change of blocking factor B_1. Initially, the speedup increases gradually with the increase of the blocking factor. However, when the blocking factor exceeds about 128, the speedup drops down. Recall that B_1 is the number of rows or the column length of the input data array. As the column length increases, the number of columns that can be placed in the cache is reduced for a given cache size. As a result, the vector length of each vector operation at the second step of the algorithm becomes small, implying more strips and more startup overheads. For B_1 being 1K, less than 8 columns can be placed in the cache and a vector strip contains only 8 elements as compared to 64 in the MM-model.

Notice that the FFT algorithm used in our analysis is the two dimensional, cache-optimized vector code developed in [6]. A lot of efforts has been made in optimizing the algorithm. If B_1 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 by means of multi-dimensional FFT. The cache access is conflict free for all stages of the transform. Optimization is guaranteed as long as the block size is less than the cache size.

5 Conclusions

In this paper, we have presented a comparative performance analysis of alternative cache designs for vector processing. Since the performance benefits from adding a direct-mapped cache into vector computers is limited, we consider an innovative cache mapping scheme, called prime-mapped cache. Analytical performance models have been developed for the vector computer architectures with or without cache memory based on three numerical algorithms: matrix multiplication, Gaussian elimination and FFT. By analyzing the algorithm structures in conjunction with the architecture models, execution times of the three algorithms on the architectures are precisely expressed in analytical formulas. Our analytical models are validated through actual performance measurements on a real vector machine. It is shown that analytical results and measurement results are in a good agreement. Numerical results that are averaged over all problem sizes show that vector computers with a prime-mapped cache (CC-model) outperform both the vector computer without cache (MM-model) and the vector computer with a direct-mapped cache for all three algorithms. The performance improvement of the prime-mapped CC-model over the MM-model goes up to a factor of 3.4 for matrix multiplication, a factor of 5.5 for Gaussian elimination which is the core of Linpack benchmark, and 70% for cache optimized FFT. Similar performance improvements are observed as compared to the direct-mapped CC-model.

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.

Acknowledgements

The author would like to thank the anonymous referees for their valuable comments that helped in improving the quality of the paper. Mr. J. Huang helped in converting the matrix multiplication algorithm into Fortran code for IBM ES/9000 and collecting measurement data.

References

[1] 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.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherences," in *The 15th Ann. Int. Symp. on Comp. Arch.*, pp. 280–289, June 1988.

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

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

[5] I. Y. Bucher and M. L. Simmons, "Measurement of memory access contentions in multiple vector processor systems," in *Supercomputing'91*, Nov. 1991.

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

[7] M. Dubois and J.-C. Wang, "Shared data contention in a cache coherence protocol," in *Proc. 1988 Int. Conf. on Paral. Proc.*, pp. 146–155, Aug. 1988.

[8] M. Dubois and F. Briggs, "Effect of cache coherency in multiprocessors," *IEEE Tran. on Comput.*, vol. C-31, pp. 1083–1099, Nov. 1982.

[9] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherence protocols," in *Proc. of 16th Int. Symp. on Comp. Arch.*, pp. 2–15, June 1989.

[10] K. Gallivan, W. Jalby, and U. Meier, "Use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory," *SIAM J. Sci. Stat. Comput.*, NOV. 1987, pp. 1079-1084.

[11] D. T. HarperIII, "Block, multistride vector, and FFT accesses in parallel memory systems," *IEEE Trans. on Parallel and Distributed Systems*, Jan. 1991, pp. 43-51.

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

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

[14] J. Huang, "Analysis of cache memories for vector and multiprocessors," *M.S. Thesis*, Dept. of Ele. Engg. University of Rhode Island, Feb. 1993.

[15] 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.

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

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

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

[19] D. F. Thiebaut and H. S. Stone, "Footprints in the Cache," *ACM Tran. on Comput.Syst.*, vol. 5, No. 4, pp 305-329, Nov. 1987.

[20] Q. Yang, G. Thangadurai, and L. Bhuyan, "Design of an adaptive cache coherence protocol for large scale multiprocessors," *IEEE Trans. on Parallel and Distributed Systems* Vol. 3 No. 3, May 1992, pp.281-293.

[21] Q. Yang, L. Bhuyan, and B.-C. Liu, "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor," *IEEE Trans. on Comput..*, vol. 38, pp. 1143–1153, August 1989. Special Issue on Distributed Computer Systems.

[22] Q. Yang and L. W. Yang, "A novel cache design for vector processing," *The 19th Int'l Symposium on Computer Architecture*, MAY 1992. Gold Coast, Australia, pp. 362-371.

[23] Qing Yang, "Inroducing a new cache design into vector computers," *IEEE Trans. on Computers*, in press.