A Comparative Analysis of Cache Designs for Vector Processing *

Tong Sun and Qing Yang
Dept. of Electrical and Computer Engineering
University of Rhode Island,
Kingston, RI, U.S.A. 02881

ABSTRACT

This paper presents an experimental study on cache memory designs for vector computers. We use an execution-driven simulator to evaluate vector cache performance of a set of application programs from Perfect Club and SPEC92 benchmark suites. Our simulation results uncover a few important facts which were unknown before: First of all, the *prime-mapped cache* that we newly proposed shows great performance potential in vector processing environment. Because of its conflict-free property, the prime-mapped cache performs significantly better than conventional cache designs for all applications considered. Secondly, performance results on the benchmarks indicate that data locality in vector processing does exist although the effects of line size, associativity, replacement algorithm and prefetching scheme on cache performance are very different from what has been commonly believed. A medium size vector cache (e.g. 128Kbytes) eliminates the necessity of a large number of interleaved memory banks in vector computers. Our experiments show that the vector computer that has a medium size prime-mapped cache with small cache line size and limited amount of prefetching provides significant speedup over conventional vector computers without cache. Performance results reported in this paper can also provide a guidance to general-purpose computer designers to enhance cache performance for numerical applications.

1 Introduction

The growing speed gap between memories and processing elements makes the memory system design become a crucial challenge to computer designers. High performance computers need sophisticated memory hierarchy to bridge the speed disparity between processors and the main memory. In most existing vector computers, memory hierarchies are implemented by a combination of a large register file and a system of highly interleaved memory modules. A register file is usually small that can hardly hold the working set of a program. Register file also requires extra efforts in software to manage it. 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 [1, 11]. In addition, the interconnection network between processors and memories adds additional delay to each memory access.

*This research is sponsored by National Science Foundation under grants No. MIP-9208041 and MIP-9505601.

Recently, a few researchers [4, 3, 10, 8, 15] have started looking at using cache memories in vector computers as an enhancement towards a smooth memory hierarchy. All these previous studies on vector caches concentrated on conventional cache organizations that are for scalar computers. Because of the difference in data access patterns of numerical applications from that of general purpose computations, the effectiveness of cache memories for vector computers remains to be studied. In this paper, we present a simulation study on vector cache performances. An execution driven simulator has been developed that takes vectorized assembly code programs generated by the IBM3090 Fortran compiler as inputs. The simulator simulates the execution of each assembly instruction in a program and produces performance results. Using the simulator, we evaluate the cache performance of a set of vectorized numerical programs from Perfect Club and SPEC92 benchmarks. Through our experiments, the following observations are in order.

First of all, the major cause of high miss ratio in vector caches is cache line conflicts. All existing cache designs (conventional cache designs), even with high associativity, suffer from large amount of line conflicts in vector processing environment. This is because a vector processor usually accesses data with certain *stride* which is the difference between addresses associated with consecutive vector elements. If the stride is not relative prime to the number of sets in a cache, several cache lines may map to the same set giving rise to cache line conflicts. Our newly proposed *prime-mapped cache* minimizes cache line conflicts by allowing a prime number of logical sets in the cache [15]. As a result, the chance of line conflicts is significantly reduced. It is shown through the experiments that the new mapping scheme performs constantly better than the conventional caches. In many cases, it performs several times better than conventional caches.

Secondly, in general purpose computation environment large cache line sizes within a certain range (e.g. 32 or 64 bytes) usually give better hit ratio [14] because of spatial locality. This is not true in the vector processing environment. As observed by Fu and Patel [10], cache line sizes have unpredictable impacts on vector cache performance since the best cache line size for one program may be the worst for another program. This observation is reproduced by our experiments on conventional cache designs. In addition, when the prime-mapped cache is considered, the impacts of cache line size on cache performance are better unveiled because of the reduction in line conflicts which contribute to the major portion of cache misses. We observed that vector caches prefer small line sizes to large line sizes provided that prefetching is performed. With limited amount of stride-directed prefetching [10] on each cache miss, one or two long-words in each cache line result in optimal performance for almost all application programs considered.

Another interesting observation is that the most-recently-used (MRU) replacement algorithm (i.e. replacing the data that is most recently used when cache is full) performs better than LRU algorithm on the conventional set-associative cache. The reason to this phenomenon is the following. Vector access with a stride that is not relative prime to the number of sets may result in several elements of the vector being mapped to the same set. Therefore, serial access to the vector may dictate against the LRU algorithm since it may keep replacing the data to be used next. With the prime-mapped cache, on the other hand, the situation is quite opposite. Because of the prime-mapping, the chance that data elements in one set are co-related (i.e. belong to a same vector) is small. As a result, we observed that LRU performs better than MRU in the prime-mapped cache.

We further observed through our experiments that high degree of associativity in conventional caches does little in reducing cache line conflicts in the vector processing environment, which is opposite to common believes. For the same cache size, increasing associativity results in decreased number of sets that data can be mapped to. As a result, we will not see significant reduction in terms of conflict misses. As an example, consider 2 alternative designs of an eight-line cache memory: 1-way and 2-way set-associative. Suppose that a vector is loaded into the cache with a

stride that is an even number (in unit of cache lines). No matter which one of the two designs one wishes to consider, only four or less number of cache lines can be placed in the cache. In other words, line conflicts occur in either case if the vector has more than 4 elements. While this example is simple, it demonstrates a potential problem that exists in any existing cache design except for a fully associative cache. The prime-mapped cache proves to perform better than high associativity on conventional caches in reducing number of cache line conflicts.

In summary, vector caches can benefit from prime-mapping, small cache lines, moderate set associativity, limited amount of stride-directed prefetching [10], and LRU replacement algorithm. Experimental results show that cache miss ratios can be controlled at as low as 0.3% to maximum of around 5% on a moderate size cache for all applications considered in this paper. It is observed that the vector computer having a 128K bytes prime-mapped cache and 8 interleaved memory modules performs better than the uncached vector computers having 256 interleaved memory modules. This is true even for highly vectorizable programs. The performance gain is getting higher as the speed gap between processor and memory increases. Therefore, our prime-mapped vector cache is a cost-effective approach to high performance memory system for vector computers.

The rest of this paper begins with a brief description of the base architecture model upon which our experiments are carried out. Section 3 presents our performance evaluation methodology, namely the execution-driven simulator. Different vectorization schemes and memory reference characteristics of the benchmarks in the vector processing environment are discussed in Section 4. Numerical results and performance evaluations are presented in Section 5. Section 6 concludes the paper.

2 Architecture Model

The base architecture in our simulation consists of a vector processor, a set of registers and a memory system. The registers are organized in the same way as the IBM3090 Vector Facility. There are 16 vector registers with each register having maximum length of 128 vector elements. Vector instructions take operands either from registers or from the main memory. The resultant vector always goes to a vector register. Strip-mining (Sectioning) is automatically performed when the vector length of a program exceeds the maximum vector length of the register.

Depending on the memory organization, we define two types of vector computers: *cache-based vector computer* and *main-memory-only vector computer*.

Cache-Based Vector Computer

In the Cache-based vector computer or CC-Model for short, the memory system contains a cache memory that holds both vector data and scalar data. The cache memory can be implemented in two different organizations: the conventional cache and the prime-mapped cache. We call them *conventional CC-model* and *prime-mapped CC-model*, respectively. In CC-model architectures, memory references are first issued to the cache memory. If the data element is found in the cache, it will be ready for ALU operations within one CPU cycle. Otherwise a cache miss occurs and the memory request goes to the main memory that is interleaved with a small number of memory banks. After the memory access is complete, the requested data goes directly to the processor for processing. Meanwhile a copy of the data is made in the cache memory for later reuse. It is assumed here that the processor stalls after a cache miss and does not continue processing until the missing data item has been returned to the processor. Multiple outstanding misses are not considered in this paper.

Conventional Cache

The conventional cache is any cache organization that is in existence today. It generally consists of 2^s sets, for some positive integer s. Each set in the cache has a unique identification number ranging from 0 through $2^s - 1$ which is called set number. Each set has d cache lines for some d that is a power of 2. We call d the degree of associativity. Given a line address A, the associated memory data is mapped into set-number

$$A \ mod \ 2^s \quad for \ \ some \ positive \ integer \ s \tag{1}$$

in the cache.

A line of memory data can be placed in the cache only in one (for direct-mapped cache) or d (for d-way set associative cache) cache locations. If more than d blocks of data referenced by a program are mapped into a same set, cache line interference occurs. As a result, some useful data may be replaced giving rise to a high miss ratio. Extensive research has been reported in the literature aiming at minimizing cache line conflicts. The most straightforward approach is to increase the degree of associativity of a set-associative cache so that a data item can be potentially placed in a large number of places thereby decreasing the chance of conflicts. However, increasing the degree of associativity results in complicated hardware for associative data search in the cache and also possibly large cache access time as compared to direct-mapped cache [5]. In addition, in many situations, high degree of associativity may not help in reducing cache line conflicts as will be evidenced later in this paper.

Prime-mapped Cache

In spite of many previous efforts in reducing cache line conflicts, the line conflict problem has yet to be solved since most of existing approaches were only able to achieve the cache performance close to that of 2-way set-associative cache [9]. It has been shown in [9, 12] that 2-way set-associativity does little in reducing cache line conflicts. In order to explore the possibility of eliminating cache line conflicts, we made an effort to analyze where line conflicts come from.

Consider a cache memory that has the size of S sets. Let b and t_d be the starting address and the access stride of a vector, respectively. A vector access stream will issue the following sequence of addresses: $b + i \cdot t_d$, for $i = 1, 2, 3 \cdots$, to the memory. These addresses will be mapped to cache locations: $(b + i \cdot t_d) \ mod \ S$. Since there are only S possible integers modulo S, there will be repetitions when i increases. In other word, we have $(b + i \cdot t_d) \ mod \ S = (b + j \cdot t_d) \ mod \ S$ for some integers i and j, or $(i - j)t_d = 0 \ mod \ S$. As a result, the jth element of the vector will be mapped into the same set in the cache as the ith element. For a direct mapped cache, a cache line conflict occurs. The number of cache line conflicts in a vector access depends on the minimum value of $i - j$ that satisfies $(i - j)t_d = 0 \ mod \ S$. For example, if $i - j = 2$, then every other element will be mapped into the same set. If S is the minimum value satisfying $(i - j)t_d = 0 \ mod \ S$, then a sequence of S vector elements will be mapped into S distinct sets, which is the ideal situation for an S-set cache. We will see next how we can achieve this ideal situation.

From number theory, we know that $(i-j)t_d = 0 \ mod \ S$ is equivalent to $(i-j)t_d = kS \ mod \ S$. Let $t_d^p = \frac{t_d}{GCD(t_d,S)}$ and $S^p = \frac{S}{GCD(t_d,S)}$, respectively, where $GCD(t_d, S)$ is the greatest common divisor of t_d and S. Then, t_d^p is relative prime to S^p. There is an integer, h, such that $(i - j) = h \cdot S^p/t_d^p$. Because t_d^p and S^p are relative prime, to make $h \cdot S^p/t_d^p$ an integer not a fraction, h must be a multiple of t_d^p. The minimum integer solution for $i - j$ is, therefore, S^p when $h = t_d^p$. For a vector access with stride t_d, $S^p (= \frac{S}{GCD(t_d,S)})$ is the number of vector element accesses before an access to an element that is mapped to the same set again. This number is often called *return number*.

The main objective of our prime-mapped cache memory is to maximize the return number. In order to maximize the return number, $\frac{S}{GCD(t_d,S)}$, we want to minimize $GCD(t_d, S)$. If S is a prime number, then $GCD(t_d, S)$ is always minimum. Therefore, vector accesses for all strides except for

multiples of S are conflict-free. The idea behind the prime-mapped cache is simple. Instead of having 2^s sets in the cache, we allow only a prime number of logical sets to reside in the cache. Our approach here is to utilize the special properties of a class of prime numbers: the Mersenne primes. A Mersenne prime number is of the form $2^s - 1$ for some s that makes $2^s - 1$ a prime. Examples of such s are 2, 3, 5, 7, 13, 17, and 19 etc.. A cache line with address A is mapped into set number $A \ mod \ (2^s - 1)$ in the cache. Since the modulus is a prime, a vector access to the cache with this mapping scheme can be made conflict-free.

Given a line address A of a memory data and a cache that has $2^s - 1$ sets, the cache mapping function is defined as

$$A \ mod \ (2^s - 1); \quad for \ some \ integer \ s, \ s.t. \ 2^s - 1 \ is \ a \ prime, \tag{2}$$

instead of $A \ mod \ 2^s$. Suppose that the binary representation of a data address A contains i s-bit subfields: $A_1, \cdots A_i$. Then reduction of $A \ modulo \ 2^s - 1$ can be done very easily by noting that $2^s = 1$, which is given by

$$A \ mod \ (2^s - 1) = \sum_{k=1}^{k=i} A_k. \tag{3}$$

When two integers I_s and D_s, defined $modulo \ 2^s - 1$, are added together, we obtain an $(s+1)$-bit result, which is reduced to $modulo \ 2^s - 1$ by simply adding the most significant carry bit to the least significant bit position (since $2^s = 1$). Thus arithmetic operations $modulo \ 2^s - 1$ are equivalent to the familiar s-bit one's complement arithmetic. Therefore, if we know the current index (I_s) and the displacement (D_s), the next cache index can be derived easily by performing a one's complement addition operation. It is important to note that such one's complement address calculation does not increase the critical path length of a processor since it can be done in parallel to the normal address calculation. The cache access time also keeps the same as the conventional cache. A detailed discussion of the design can be found in [15].

Main-Memory-Only Vector Computer

In the Main-Memory-Only Vector Computer or MM-Model for short, the memory system consists of a number of interleaved memory modules (banks) to increase bandwidth. No cache memory is present in the MM-model. All memory references are directed to the interleaved memory system. We assume for the purpose of simplicity that the vector processor and interleaved memory banks are connected via a single pipelined bus with one double-word (64 bits) in width. One 8 bytes line of data can be transferred on the bus in one bus cycle provided that the bus is available. We define *memory cycle time* as the time between the start and completion of a memory operation in a memory module. This memory cycle time is the same for both CC-model and MM-model, and is denoted by t_m. Once a memory operation starts, a memory bank cannot be accessed again for a period called the *memory cycle time* which is the same as the *bank reservation time* in the terminology of Cray supercomputers. Note that the memory cycle time is different from the *memory access time* which is the time between the issue of a memory access request and the arrival of the requested data at the processor register. A vector access stream to contiguous banks does not encounter memory bank conflict in which case each requested vector element is available to the vector processor at every CPU cycle. Due to multiple vector access streams or nonunit stride accesses, element requests for memory access may go to the same memory bank. If several requests are issued to a particular bank less than the memory cycle time apart, we have bank conflicts and delays. Similar to CRAY X-MP and Y-MP, we assume in this paper that all operands have to arrive at CPU in the same

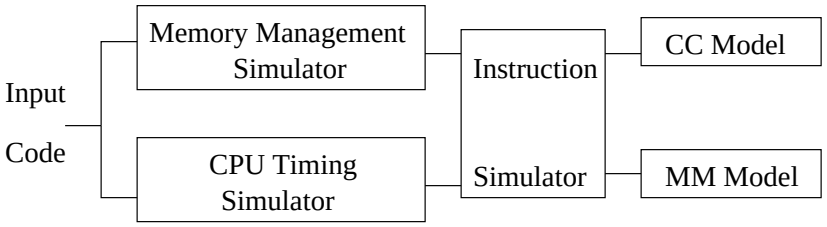


Figure 1: Block diagram of the simulator

sequence as they are requested. If the memory access of a single vector element is delayed because of the bank conflict, all subsequent elements of the vector are delayed by at least that amount. The executions of consecutive vector instructions including memory operations can be overlapped (i.e. vector chaining) as long as the dependency and the vector sequence order are preserved.

3 The Simulator

All results presented in this paper are obtained by using an execution-driven simulation at assembly instruction level of the IBM3090 VF. The instruction level simulator simulates the IBM S/370 scalar and vector instructions including actual ALU operations, flag settings, instruction fetchings, and main memory accesses. The overall structure of our simulator is depicted in Figure 1, which consists of three major parts: (1) an instruction-level simulator, (2) a memory simulator, and (3) a machine timing simulator. The simulator takes the assembly code of an application program generated by the IBM3090 FORTRAN Version 2.5 compiler with vectorization and optimization options, simulates the execution of each instruction of the input code, and finally produces cache miss ratio, total execution time and memory traffic for simulated programs.

Instruction-Level Simulator : It executes each assembly instruction according to the IBM3090 vector architecture. The instruction-level simulator contains functions and procedures for all instruction simulations, such as identifying opcode, generating the effective addresses, fetching operands, executing operations, resetting and checking condition codes and so on. In the instruction-level simulator, we collect all instruction-level statistics, including memory reference type, memory access format, the distributions of vector stride and vector length etc..

Memory Simulator: The memory simulator consists of two independent components which simulate the CC-model and the MM-model, respectively. For the CC-model, the memory system consists of a cache memory and a small set of interleaved memory banks. The cache memory in the CC-model can be either a conventional cache or the prime-mapped cache [7]. The cache design parameters can be specified directly in the CC-model. For the MM-model, the memory system consists of a number of low-order-bit interleaved memory banks. A memory access at a memory bank can start only if the bank is idle. Otherwise, the memory request waits in a queue for all earlier requests to finish. All memory banks act independently and concurrently so that memory accesses at different banks can be carried out in parallel across the memory banks.

Timing Controller: The timing controller maintains a global clock to keep track of all instructions executed and all memory references. In case of a cache miss or a memory bank conflict, the CPU stalls waiting for operands. The number of stall cycles depends on cache miss ratio, the number of memory bank conflicts, and memory cycle time. Cache miss penalty consists of bus transaction time, bus busy waiting time, write-back delays of dirty cache lines, memory stall time for bank conflict, and memory cycle time t_m. The total execution time is defined as the total elapsed

vectori-zations	vectori-zation rate	stride distribution	
		unit stride	1024bytes
V (1)	87.55%	0	100%
V (2)	96.67%	14.28%	85.72%
V (3)	94.30%	84.96%	15.04%
V (4)	98.70%	100%	0

Cache size: 16K bytes; Cache line size: 128 bytes; Set size: 1;

Matrix Size: Z(128,128)=X(128,128)*Y(128,128)

time to simulate a program in terms of CPU cycles.

4 Benchmark Characteristics

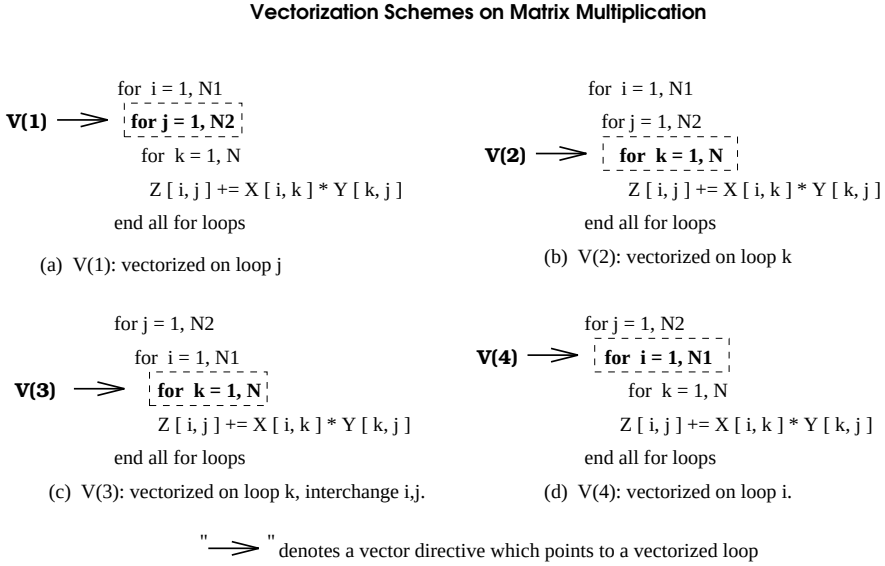
The sixteen programs chosen in this paper for performance evaluation represent a wide variety of applications in scientific computation. They have diversed memory reference characteristics. In this section, we will take a close look at these benchmark programs. In particular, we take the well-understood matrix multiplication algorithm as an example to show how different vectorization schemes affect the memory access behavior. We then classify all other benchmark programs according to their memory access characteristics.

4.1 Vectorization schemes on matrix multiplication program

In the vector processing environment, different ways of vectorizing a program may result in quite different memory reference patterns even for the same program. Matrix multiplication is a particular interesting case study for cache memories because data locality is carried in three different loops by three different variables. Different vectorization schemes can be realized depending on which one of these loops to be vectorized giving rise to different vector access patterns (i.e. different vectorization rates and vector stride distributions). By inserting vector directives at different loop levels, four vectorization schemes (i.e. V(1), V(2), V(3) and V(4)) of the matrix multiplication ($Z[N_1,N_2] = X[N_1, N] * Y[N, N_2]$) are shown as below.

As shown in Table 1, memory reference characteristics of these four vectorization schemes are considerably different. The vectorization rate is defined as the ratio of vector data references to total data references of a program.

To show how the different memory reference characteristics resulting from different vectorization schemes affect the overall vector processing performance of cache-based vector computers, we collected cache miss ratios of these four vectorization schemes of the matrix multiplication for two different cache organizations: the conventional cache and the prime-mapped cache, as shown in Figure 2. Cache performance varies drastically with different vectorization schemes in the conventional cache. The difference in cache miss ratio can be as high as a factor of five. This observation indicates that the proper vectorization of a program is essential to the performance of the conventional cache. Such a proper vectorization, however, is not always straightforward. For this particular algorithm, for example, the vectorization scheme V(4) that has maximum amount of vectorization rate and unit stride accesses does not necessarily give the optimal performance. This is because the

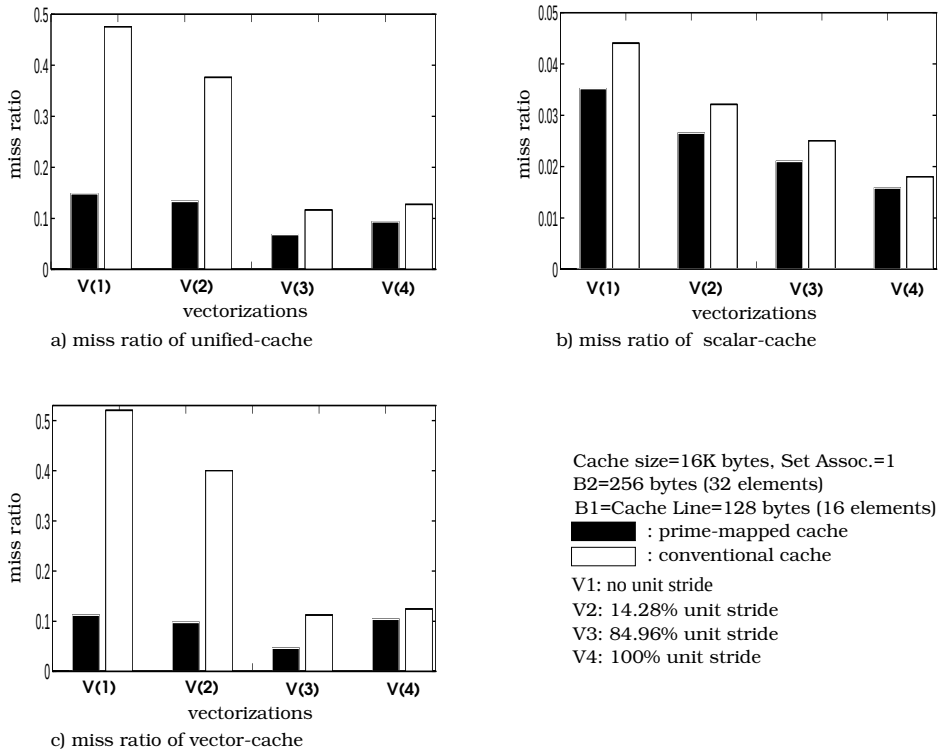


capacity misses rather than conflict misses dominate the cache misses in V(4). The vectorization scheme $V(3)$ gives the best performance among the four vectorization schemes. To compare the performance differences between the conventional cache and the prime-mapped cache, we listed in Figure 2 the speedup that is the ratio of the total execution time of the conventional cache organization to that of the prime-mapped cache organization. The speedup due to the prime-mapped cache organization ranges from 1.16 to 3.98 times (for split-cache) and from 1.29 to 2.89 times (for unified-cache).

4.2 Benchmarks' Characteristics

The memory reference characteristics of selected programs are listed in Table 2. From this table, we observe that most benchmark programs issue up to hundreds of millions of memory references and have data set sizes greater than 1Mbytes. The data set sizes of two programs from Perfect Club suite are larger than 100 Mbytes, and one program from SPEC92 is close to 65 Mbytes. The distribution of vector stride and the average vector length are shown in Table 3 in terms of bytes. Stride-0 accesses are scatter/gather operations that read or write to vector elements. A unit stride means that vector elements are located in consecutive memory locations. In terms of bytes, unit stride actually equals 4 bytes for single precision data accesses and 8 bytes for double precision data accesses. Vector length is a measure of the number of elements accessed by a single vector instruction rather than the length of vectors in the data set.

Based on Tables 2 and 3, we classify these sixteen programs into five groups. Group A is a collection of programs with high vectorization rate ($\geq$ 90%) and good spatial locality, large percentage of unit stride, such as FLO52, HYDRO2D, SU2COR and SWM256. Group B includes programs with high vectorization rate($\geq$ 90%) and a wide range of stride distributions, such as ARC2D, BDNA and MATRIX. Group C contains programs with vectorization rate less than 90% and good spatial locality, such as ADM, DYFESM, QCD, TRFD and TOMCATV. Group D is a collection of programs with vectorization rate less than 90% and a wide range of stride distributions, such as NASA7 and TRACK. Group E has programs with a very low vectorization rate or even no vector references, such as CFFT2D and VPENTA. We expect programs from the same group to have similar performance characteristics, which helps in directing our simulation experiments.



Speedups	
V(1)	2.89
V(2)	2.54
V(3)	1.52
V(4)	1.29

(a) unified cache

Speedups	
V(1)	3.98
V(2)	3.57
V(3)	1.93
V(4)	1.16

(b),(c) split caches

Figure 2: Cache Performance of different vectorization schemes on matrix multiplication

5 Performance Results

In this section, we present performance results obtained from our simulation experiments. We first evaluate and compare the performance of the cache-based vector computers (CC-model) with that of the main-memory-only vector computers (MM-model). For this purpose, we measure the total execution time and memory traffic of the simulated programs since cache hit/miss ratios have no significance. After comparing the CC-models and the MM-model, we concentrate on the effects of cache design parameters such as mapping function, cache size, degree of associativity, cache line size, prefetching scheme, and replacement algorithm on cache performance of vector computers. Both cache miss ratio and execution-time speedup are used as performance measures in comparing different cache configurations.

5.1 Is Vector Cache Profitable?

It is commonly believed that the cache memory may not be beneficial to vector computers because the memory latency in such computers is amortized over pipelined streams of data references. To clarify this, we carried out experiments by adding a typical cache organization with two different mapping schemes: the conventional cache and the prime-mapped cache, into a vector computer to evaluate the potential performance. A 128K bytes, 2-way set associative cache with line size of

Program	Application Type	Vectorization Rate	Read%	Write%	# of references (M)	Input Data Size	Data Set Size (Mbytes)
MATRIX	matrix multiplication	94.3%	60.77%	39.23%	11.06	128*128	0.393
(Perfect Club)							
ADM	Air Pollution	62%	66.51%	33.49%	428.3	64*1*16 size, 720 steps	2.76
ARC2D	Finite difference	97%	69.24%	30.76%	395.6	X=385, ETZ=145, 1000 its	106.3
BDNA	Molecular dynamics	91%	66.81%	33.19%	637.7	635 modules, 20 counter ions	4.77
DYFESM	Structural dynamics	72.3%	63.97%	36.03%	272.8	4 elements, 1000 time steps	0.73
FLO52	CFD, transonic flow	95.3%	70.4%	29.6%	108.35	NX=40 NY=8 Mesh=3	2.42
QCD	Quantum chromodynamics	75.6%	64.3%	35.7%	93.47	16*16*16*16 lattice	115.2
TRACK	Signal Processing	68.3%	62.51%	37.49%	213.6	480 targets	1.59
TRFD	Molecular dynamics	87.1%	67.8%	32.2%	64.51	40 iteration times	12.88
(SPEC92)							
HYDRO2D	Hydrodynamics	96.10%	68.33%	31.67%	1402.3	400 timesteps	1.21
NASA7	Seven Kernels	71.20%	72.17%	27.83%	1767.6	100 iterations	18.11
SU2COR	Quantum physics	94.52%	65.82%	34.18%	1242.2	8*8*8*16 dims	4.89
SWM256	Weather prediction	96.90%	69.01%	30.99%	1082.5	grid size=256*256	4.13
TOMCATV	Meshgeneration	82.01%	61.17%	38.85%	666.0	1200 iterations , N=1024	61.58
CFFT2D	NASA Kernel	8.15%	52.42%	47.58%	12.91	100 iterations	0.34
VPENTA	NASA Kernel	0%	75.40%	24.60%	10.54	100 iterations	1.87

32 bytes is added between the vector processor and the interleaved memory system with 8 banks. Vector register files keep the same as those of IBM3090 VF in all the models considered. We intend to compare the performance of the CC-models with that of the MM-model which has 256 banks.

In Table 4, we show the speedups of the two CC-models, the conventional CC-Model (Conv.) and the prime-mapped CC-Model (New), over the MM-model for the sixteen benchmark programs as a function of memory cycle time t_m. The memory cycle time, t_m, varies from 16 CPU cycles through 96 CPU cycles. These numbers are reasonable selections reflecting both the current situation as well as the future trend. It can be seen from this table that most programs run faster on the CC-models than on the MM-model. The performance improvement of the CC-models is getting larger as the speed gap between processor and memory grows. The speedup of the prime-mapped CC-model over the MM-model goes up to as high as 4.88 when memory cycle time is 96 cycles. When the memory cycle time is 48 cycles, which is close to the case of Cray-2 [13], the speedup of the prime-mapped CC-model for all applications ranges from 1.15 to 2.68, though only 8 memory banks are present in the prime-mapped CC-model. Even with the fastest memory cycle time available in today's supercomputers (e.g. 4 cycles in Cray X-MP [13]), our prime-mapped cache organization still performs better than the MM-model having 256 memory banks as shown in Table 5.

It can be seen from Table 4 that the performance improvement of the two CC-models varies drastically among different programs due to their distinct localities and vectorizabilities. The conventional CC-model may not always perform better than the MM-model. For example, BDNA program has a 91% vectorization rate and a wide variety of stride distributions. For this program, the cache miss ratio of the conventional cache is close to 30% that results mainly from cache line conflicts. With such a high cache miss ratio, the conventional CC-model performs worse than the MM-model (see Table 4) when the t_m is smaller than or equal to 64 cycles. However, the prime-mapped cache memory helps greatly for such programs with poor spatial localities because of its conflict-free property. For example, the speedup of the prime-mapped CC-model over the MM-model for programs with a wide range of stride distributions(i.e. Groups B and D) ranges from 1.48

Program	Stride (in bytes) Distributions (%)								Average Vector Length (bytes)
	0-4	4-8	9-16	17-32	33-64	65-128	128-256	> 256	
MATRIX	0.0	84.96	0.0	0.0	0.0	0.0	0.0	15.04	1024
ADM	34.8	43.5	0.0	6.0	0.0	6.5	3.4	5.8	267.6
ARC2D	0.0	51.3	0.0	0.0	2.4	4.0	28.1	14.2	996.8
BDNA	2.3	14.7	4.1	9.4	43.0	8.7	12.3	5.5	487.6
DYFESM	27.6	52.7	3.0	0.0	8.1	0.0	7.2	1.4	332.0
FLO53	0.0	93.7	0.0	4.5	0.0	0.0	1.8	0.0	808.9
QCD	7.9	84.6	0.0	3.2	2.7	0.0	0.0	1.6	481.3
TRACK	2.3	44.2	0.0	16.2	25.56	0.0	8.1	2.5	753.4
TRFD	25.6	64.3	3.3	0.0	4.1	2.7	0.0	0.0	430.5
HYDRO2D	0.0	89.4	0.0	0.0	4.32	0.0	0.0	6.3	477.2
NASA7	1.6	65.6	0.24	18.78	0.30	1.5	0.5	12.0	445.4
SWM256	3.1	89.4	0.0	0.0	2.7	4.3	0.5	0.0	576.2
SU2COR	1.0	92.9	0.0	0.0	0.6	1.5	1.9	2.1	491.2
TOMCATV	2.1	93.2	0.0	1.21	0.0	0.0	1.6	1.9	870.4
CFFT2D	0.0	100	0.0	0.0	0.0	0.0	0.0	0.0	0.0
VPENTA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

to 2.68 when t_m is 48 cycles. But for programs in Group A (e.g. FLO52, HYDRO2D, SU2COR and SWM256) with good spatial localities and high vectorizabilities, the speedups of both CC-models over the MM-model are not significant. This is because the MM-model with a large of number of interleaved memory banks can provide enough memory bandwidth for such programs with good address localities [1]. Since conflict misses no longer dominate cache misses for these programs, the prime-mapped cache does not show great advantage. For the low vectorizable programs (CFFT2D and VPENTA), it is observed that the two CC-models significantly outperform the systems without cache.

Memory traffic measurements are summarized in Table 6 in terms of the number of memory accesses (Mbytes) going through the memory system. Both the conventional CC-model and the prime-mapped CC-model reduce memory traffic to a large extent compared to the MM-model. For all benchmark programs, the memory traffic reduction due to the cache memory ranges from a factor of 10 to 40.

In summary, for all benchmark programs, the prime-mapped CC-model outperforms the MM-model to different extents. Even though only 8 memory modules are used in the prime-mapped CC-model, we observed that the prime-mapped CC-model performs significantly better than the MM-model with 256 modules. In other words, a 128K bytes prime-mapped cache does the job several times better than 248 extra memory modules. In the following section, we will see that after fine tuning the cache design parameters on the prime-mapped cache, further speedup is possible.

5.2 How to Configure a Vector Cache?

From the initial experiments on the two CC-models, we observed that a properly designed cache can improve vector processing performance. In this section, we study the effects of various cache design parameters on vector processing performance. We assume in this section that the main memory cycle time is 48 cycles (i.e. t_m=48 cycles). First, we change the structural parameters of a vector

Group	tm / Program	16		48		64		80		96	
		New	Conv.	**New**	**Conv.**	New	Conv.	**New**	**Conv.**	New	Conv.
A	FLO52	1.12	1.07	1.31	1.22	1.37	1.28	1.45	1.35	1.63	1.44
	HYDRO2D	1.21	1.09	1.27	1.11	1.33	1.12	1.36	1.13	1.38	1.15
	SU2COR	1.10	1.03	1.29	1.13	1.37	1.14	1.49	1.17	1.58	1.19
	SWM256	1.08	1.00	1.15	1.09	1.20	1.11	1.25	1.16	1.31	1.18
B	ARC2D	1.28	0.97	1.48	1.10	1.52	1.14	1.56	1.15	1.60	1.17
	BDNA	1.31	0.89	1.50	0.93	1.54	0.96	1.60	1.09	1.68	1.11
	MATRIX	1.41	1.19	1.56	1.20	1.83	1.31	1.94	1.40	2.10	1.58
C	ADM	1.37	1.29	1.74	1.68	1.83	1.72	1.95	1.80	2.33	1.88
	DYFESM	1.27	1.23	1.64	1.60	1.72	1.69	1.87	1.84	2.14	2.12
	QCD	1.24	1.10	1.56	1.26	1.79	1.42	1.91	1.58	2.09	1.97
	TOMCATV	1.21	1.11	1.72	1.38	2.02	1.46	2.25	1.52	2.46	1.57
	TRFD	1.56	1.11	1.79	1.20	1.88	1.22	1.96	1.28	2.10	1.31
D	NASA7	1.89	1.37	2.68	1.76	2.91	2.00	3.35	2.08	4.88	2.54
	TRACK	2.38	1.53	2.47	1.59	2.51	1.64	2.60	1.73	2.71	1.80
E	CFFT2D	1.08	1.08	1.60	1.56	1.67	1.65	1.74	1.71	1.78	1.75
	VPENTA	1.69	1.67	1.89	1.88	1.96	1.96	2.09	2.09	2.18	2.17

New: New CC-model (with prime-mapped cache)

Conv.: Conventional CC-model (with conventional cache)

cache such as cache size, cache line size and the degree of associativity. Then we will examine the performance effect of different replacement algorithms. Finally, we discuss on what would be the optimal cache configuration for vector processing.

We consider cache sizes from 8K bytes through 512K bytes, cache line sizes from 8 bytes to 128 bytes, and the set associativities from 1 to 8. Since most benchmark programs generate over hundreds of millions of memory references and some programs' data set sizes are even larger than 100M bytes, it takes at least 20 hours to get one point of performance result on the IBM mainframe and at least 35 hours on a workstation. Therefore, we did not exhaust all possible combinations of cache parameters. Instead, we carried out selected experiments based on previous experiments. In other words, each experiment we performed provides us with a guidance as to what is more important to do for the next experiment. Using the IBM mainframe as well as about 10 workstations that are often available, we were able to collect about 500 performance points for our performance evaluation purpose within 8 months.

5.2.1 Cache size, Associativity and Line size

Figure 3 shows the vector cache performance in terms of cache miss ratio and execution time speedup for different cache sizes ranging from 8K bytes to 512K bytes. In this subsection, the speedup is defined as the ratio of the execution time of the conventional cache organization to the execution time of the prime-mapped cache organization. Results of ten programs are shown in this figure, two from each of the five groups. As indicated in Section 4, we expect programs from the same group to have similar performance characteristics. Programs from group C such as ADM and DYFESM show very good cache performance as shown in Figure 3(c). The cache miss ratios for these two programs start from about 2% and 7% for the 8K bytes cache respectively to less than 1% for the 512K bytes cache. The reason why these two programs show such good cache performance is that

Perfect Club Benchmarks	ADM	ARC2D	BDNA	DYFESM	FLO52	QCD	TRACK	TRFD
New	1.28	1.16	1.15	1.15	1.03	1.12	1.37	1.10
Conv.	1.23	0.88	0.73	1.12	0.97	1.06	1.35	1.07

SPEC92 Benchmarks	HYDRO2D	NASA7	SWM256	SU2COR	TOMCATV	CFFT2D	VPENTA	MATRIX
New	1.09	1.52	1.05	1.08	1.15	1.06	1.52	1.18
Conv.	1.05	1.20	0.91	1.01	1.08	1.06	1.52	1.07

New: New CC-model (with prime-mapped cache)

Conv.: Conventional CC-model (with conventional cache)

they have very good spatial localities. As shown in Table 3, most data accesses (about 80%) of both programs are stride 0 and unit stride accesses. In this case, conflict misses do not dominate the cache misses. Therefore, the speedups due to the prime-mapped cache for these two programs are not significant (less than 10%). We also note in Table 2 that their vectorization rates are not very high, 62% in ADM and 72.3% in DYFESM.

When the vectorization rate becomes high and the range of stride distributions becomes wide, the caching behavior changes drastically. This is exemplified by the other six programs: HYDRO2D, SU2COR, ARC2D, BDNA, NASA7 and TRACK in Figures 3 (a),(b) and (d) respectively. The cache miss ratio of the BDNA is over 40% with 8K bytes conventional cache in contrast to 2% of the DYFESM with the same cache organization. Similarly, miss ratios of the SU2COR, ARC2D, NASA7 and TRACK are over 20% for the 8K bytes cache. Such high miss ratios would certainly make the cache memory useless for vector computers. It simply would not help much if such a conventional cache had been inserted in a vector computer. However, the prime-mapped cache shows significantly better cache performance for programs from Groups B and D (e.g. ARC2D, BDNA, NASA7 and TRACK) as shown in Figures 3 (b) and (d). The prime-mapped cache performs over 4 times as good as the conventional cache for the BDNA program with 8K bytes cache in terms of cache miss ratio, and the speedup of the prime-mapped cache over the conventional cache is up to a factor of 2.5. The speedup resulting from the prime-mapped cache for the Group A programs is also up to 40% as shown in Figure 3 (a). For programs with low vectorization rate such as Group E (CFFT2D and VPENTA) in Figure 3(e), the prime-mapped cache still outperforms the conventional cache because of its conflict-free property.

It is interesting to observe from Figure 3 that the miss ratios of the prime-mapped cache stay relatively flat across different cache sizes. Moreover, miss ratios of the 8K bytes prime-mapped cache are almost the same as miss ratios of the 512K bytes conventional cache, especially for highly vectorizable programs. These results imply that the usable portion of the conventional cache is just about 8K bytes even though the cache size is increased to 512K bytes. The results also indicate that cache misses in the conventional cache are not mainly due to capacity misses. Rather, they mainly result from cache line conflicts. We have seen in Table 2 that programs, such as ARC2D, BDNA, NASA7 and TRACK, access vector data with a high percentage of nonunit stride, which results in a large amount of cache line conflicts. Such cache line conflicts cause 2 to 4 times more cache misses as evidenced in Figures 3 (b) and (d).

Since the major source of cache misses comes from cache line conflicts, our next experiment is to increase the set associativity to examine cache performance for programs with a wide range of stride distributions. It is well known that a high associativity reduces cache line conflicts in

the conventional cache. Since programs with good spatial localities, such as Group A and Group C programs, will not show significant change in performance as set associativity increases, our experiments on set associativity are primarily based on programs from Groups B and D.

In Figure 4, we plotted the cache miss ratio and execution-time speedup respectively as a function of set associativity on four programs: ARC2D, BDNA from Group B, NASA, TRACK from Group D. It is surprising to note in Figure 4 that the increase in associativity does not improve cache performance significantly in vector processing environment. This is very contradictory to common believes since we have known that cache misses are mainly caused by line conflicts and it is believed that the high associativity reduces line conflicts. It can be seen from Figure 4 that even an 8-way set-associative conventional cache has about 30% higher miss ratio than the 1-way, prime-mapped cache. To better understand this phenomenon, let us consider the following example.

- Cache size: 8 lines;
- Line size: one vector element;
- Configuration 1: direct-mapped, i.e. there are 8 sets, sets 0, 1, ... 7;
- Configuration 2: 2-way set associative, i.e. there are 4 sets, sets 0, 1, 2, and 3, each having 2 lines.

Suppose that a vector of 8 elements is loaded into the cache starting from set 1. Assume further that the vector accessing stride is an even number, say 2. In configuration 1, the first element goes to set 1, the second element goes to set 3, and so on. When the fifth element is fetched, it is mapped to set 1 again giving rise to a line conflict. Similarly, all the subsequent three elements will go to sets 3, 5 and 7 resulting in total 4 line conflicts. Now consider configuration 2, the first element goes to set 1 and the second element goes to set 3. The third element will also be placed in set 1 but no conflict since each set can hold 2 lines. However, conflict starts from the fifth element onward resulting in totally 4 line conflicts. Therefore, high associativity does not improve cache performance in this particular example. The increase of the cache associativity also increases the cache access time which has negative effects on system performance.

In the previous experiments, the cache line size is fixed at 64 bytes without knowing whether this line size is a good choice. Figure 5 shows the cache performance as a function of cache line size. The eight representative programs are: FLO52 and SWM256 from Group A, ARC2D and BDNA from Group B, TOMCATV and TRFD from Group C, and NASA7 and TRACK from Group D. Because of the good sequentiality of FLO52, SWM256, TOMCATV and TRFD, the miss ratios of these programs decrease as the cache line size increases, as shown in Figures 5 (a) and (c). The situation is quite different for ARC2D, BDNA, NASA7 and TRACK because of their high percentages of nonunit stride accesses. Cache miss ratios of these programs are no longer monotonically decreasing as cache line size increases. The shape of cache miss ratio curves of the conventional cache for these programs is largely dependent on the program localities. It is difficult to predict the effects of cache line sizes on the performance of the conventional cache. It is important to observe, however, that the miss ratio curves of the prime-mapped cache corresponding to different programs have similar shapes (see solid lines in the miss ratio curves in Figure 5). In other words, the prime-mapped cache makes it possible to predict cache performance for different cache line sizes and therefore select an optimal cache line size.

5.2.2 Replacement Algorithms and Prefetching Schemes

In the last subsection, we have seen that the prime-mapped cache significantly outperforms the conventional cache. Further performance improvement is possible by means of a good replacement algorithm and a proper prefetching scheme. In this subsection, we evaluate the vector cache performance by taking into consideration of replacement algorithm and prefetching scheme. We consider three replacement algorithms, namely most recently used (MRU) which replaces the data that are most recently used, least recently used (LRU) that replaces the least recently used data, and Random algorithm. The prefetching scheme considered here is the *stride-directed prefetch* presented in [10] which shows great performance potential in vector cache designs. The stride-directed prefetching makes use of stride information for vector data. Let F be the fetch size and L be cache line size both in terms of bytes. Then F/L consecutive cache lines are fetched if stride is less than or equal to L. If the access stride is larger than cache line size, then we fetch cache lines that are separated by the stride (in units of cache lines). Since the major concern of this prefetching scheme is to improve the vector cache performance, in the following experiments, we select programs with high vectorization rate such as those from Groups A and B.

Data prefetching is to load data into cache before they are actually referenced to reduce cache misses. Prefetching may result in high memory traffic, which may in turn result in high miss penalty. Therefore, using cache miss ratio alone as a performance measure to compare with non-prefetched cache is not appropriate. The total execution time of a program is more appropriate. We will use the following speedup in our discussions.

$$Speedup = \frac{Execution\ time\ of\ non-prefetched\ cache}{Execution\ time\ of\ prefetched\ cache} \tag{4}$$

For the conventional cache, miss ratio decreases as fetch size increases as shown in Figure 6. However, the corresponding speedup is not as significant. The increase in memory traffic reduces the overall speedup due to prefetching. It is also noticed that MRU algorithm presents lower cache miss ratio than LRU and Random algorithms in the conventional cache. This is another unusual phenomenon in vector cache designs since we are used to the concept that LRU generally performs the best. The main reason is that the vector access stride results in uneven data distribution in the conventional cache. The large working set size for vector processing makes LRU keep replacing the data to be used next.

The situation changes completely when we simulate the prime-mapped cache, as shown in Figure 7. First of all, the cache performances are no longer monotonically decreasing with the increase of fetch size. After the fetch size exceeds certain value, the miss ratio rises and speedup drops in all three programs as shown in Figure 7. The LRU algorithm performs better than the MRU and Random algorithms. Our explanations to these results are as follows. With the prime-mapped cache, the cache utilization is much higher than the conventional cache as evidenced by Figure 3. In other words, the fraction of the cache being utilized to hold useful data is larger in the prime-mapped cache as compared to the conventional cache. Thus, the chance that prefetched data take the cache positions of some useful data is high. As a result, massive amount of prefetching may result in unnecessary replacement of some useful data. Therefore, cache miss ratio starts becoming larger when the fetch size is too large. As to the replacement algorithm, it is under our expectation that the LRU is better than the MRU. With a very high probability, data elements in one vector go to different sets in the prime-mapped cache due to its conflict-free property. As a result, the chance that data in one set are co-related is very small. If data in a set are not co-related, the caching behavior is similar to the conventional general purpose cache. Therefore, LRU performs better than MRU and Random algorithms.

In order to observe how cache line size and fetch size affect the prime-mapped cache performance with prefetching, we plotted cache performance as a function of both cache line size and fetch size in Figure 8 for the prime-mapped cache only. Prefetching on the prime-mapped cache improves the cache performance even further as shown in Figure 8 when proper cache line size and fetch size are selected. The advantage of prefetching diminishes as the line sizes become larger, because large cache lines cause cache pollution when data are accessed with nonunit strides. Large cache line also increases the chance of line conflicts for the same cache size and incurs more memory traffic. From Figures 7 and 8, the best performance points occur at 8 bytes cache line and 128 bytes (16 cache lines) fetch size.

5.3 Optimal Cache Designs

Now let us put all sixteen programs together to observe the performance of the prime-mapped cache. Based on our observations in the previous section, we select the 128K bytes, 4-way set-associative prime-mapped cache with line size of 16 bytes and LRU replacement. Because a moderate set-associativity shows a good performance, a 4-way set-associative prime-mapped cache instead of 1-way prime-mapped cache is used here. For prefetched cache, 128 bytes of data are stride-directed prefetched on each cache miss. Table 7 illustrates the cache miss ratios in the two CC-models and the speedups of the prime-mapped CC-model over the MM-model for all 16 benchmark programs. The speedups over the MM-model as a result of adding a non-prefetching prime-mapped cache into vector computers range from 1.15 to 2.68. The speedup range increases for a properly-designed prime-mapped cache with prefetching, namely ranging from 1.31 to 3.10. And cache miss ratios in this prime-mapped cache design are as low as 0.3% to the maximum 5.21%. We expect the speedup to be even larger as the speed gap between processors and memories increases, particularly in multiprocessor systems which may add additional memory delays due to interconnection network contention and more memory interferences. Therefore, the 4-way set-associative prime-mapped cache with small cache line (16 bytes) and limited amount of stride-directed prefetching is a good choice for vector cache designs.

6 Conclusions

In this paper, we have studied cache performance of vector computers by using execution-driven simulations. A set of scientific application programs from Perfect Club and SPEC92 benchmark suites are simulated on the vector computer simulator with different memory hierarchy configurations. Simulation results on vector cache performance are reported in this paper considering a variety of cache configurations by varying mapping function, cache size, line size, degree of associativity, as well as replacement algorithm. Cache miss ratio, total execution time and memory traffic are used as performance measures to evaluate the memory hierarchy performance. It is shown that the cache-based vector computers with our newly proposed prime-mapped cache outperform the vector computers without cache or with the conventional set-associative cache for all applications considered. Prime-mapped cache has proven to be cost-effective and it provides optimal vector cache performance. Numerical results indicate that vector caches prefer small cache line and moderate associativity. Vector caches can also benefit greatly from limited amount of stride-directed prefetching on each cache miss. The properly designed prime-mapped cache can double and even triple the overall performance of existing vector computers in terms of overall execution time. Our conclusion is that cache memory can improve the performance of vector processing and is a cost-effective enhancement towards a smooth memory hierarchy for vector computers.

References

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

[2] 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.

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

[4] W. Abu-Sufah, A. D. Malony, "Vector Processing on the Alliant FX/8 Multiprocessor", *Int. Conf. on Parallel Processing*, pp. 559-566, Aug., 1986.

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

[6] M. Berry, et al.,"The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l Journal for Supercomputer Applications*, Fall 1989.

[7] Mark D. Hill, "Dinero cache simulator," Copyright 1985, 1989, Univ. of Wisconsin.

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

[9] Qing Yang, S. Adina, "A one's complement cache", *Proceedings of 94' Int'l Conf. on Parallel Processing*, pp. 250-258, Aug., 1994.

[10] 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.

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

[12] 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.

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

[14] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache performance of the SPEC92 benchmark suite," *IEEE Micro*, Aug. 1993, pp. 17-27.

[15] Q. Yang and L. W. Yang, "A novel cache design for vector processing," *the 19th Int'l Symp. on Computer Architecture*, May 1992. Gold Coast, Australia.

[16] Q. Yang, "Introducing a new cache design into vector computers," *IEEE Trans. on Computers*, vol.42, December 1993.

[17] T. Sun and Q. Yang, "Performance of SPEC92 on Prime-mapped Vector Cache", *Sixth IEEE Symposium on Parallel and Distributed Processing*, Dallas, Oct. 1994.

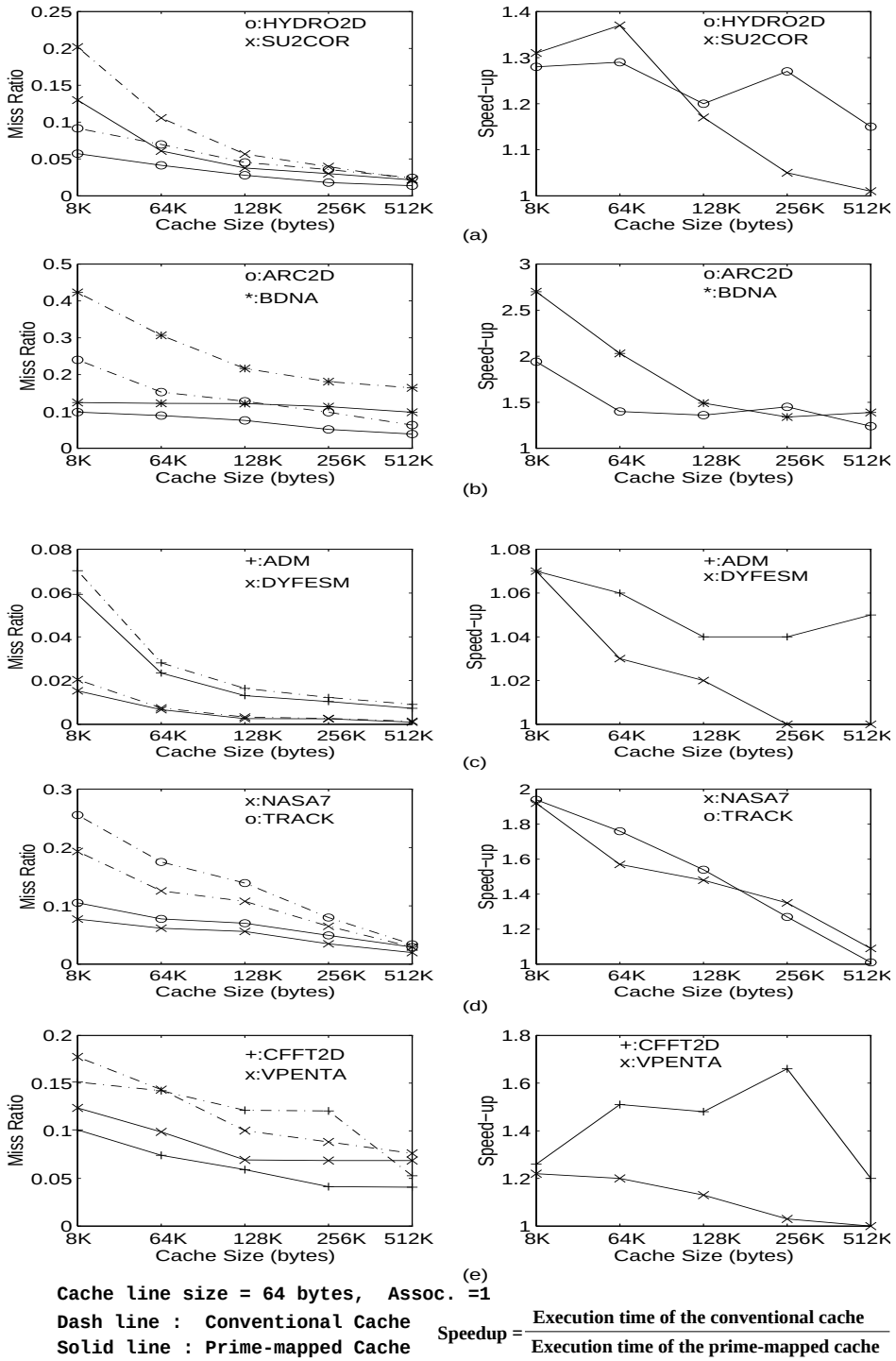
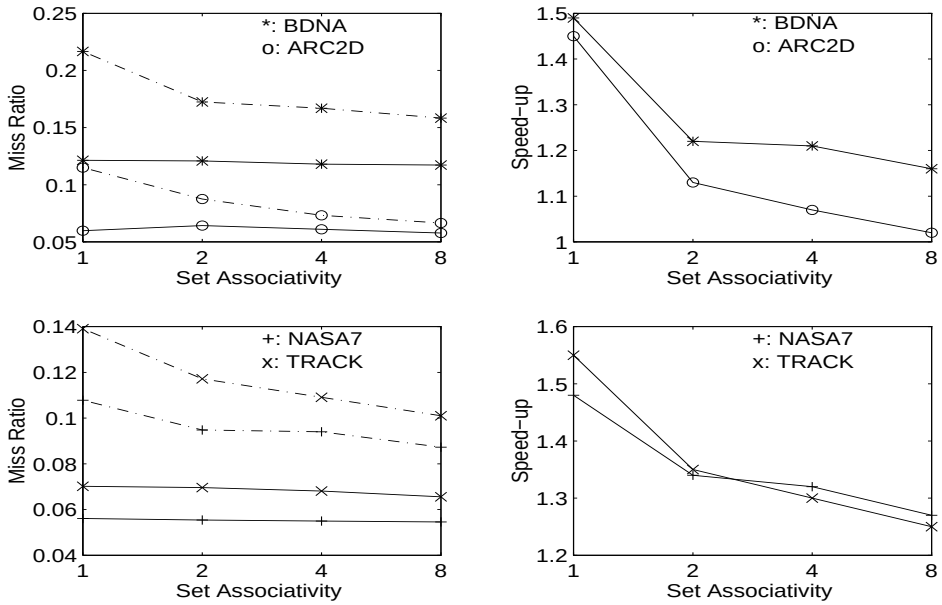


Figure 3: Cache performance (miss ratio & speedup) vs. cache size



Cache Size = 128K bytes, Cache Line Size = 64 bytes

Dash Line: Conventional Cache

Solid Line: Prime-mapped Cache

$$\textbf{Speedup} = \frac{\textbf{Execution time of the conventional cache}}{\textbf{Execution time of the prime-mapped cache}}$$

Figure 4: Cache performance (miss ratio & speedup) vs. set associativity

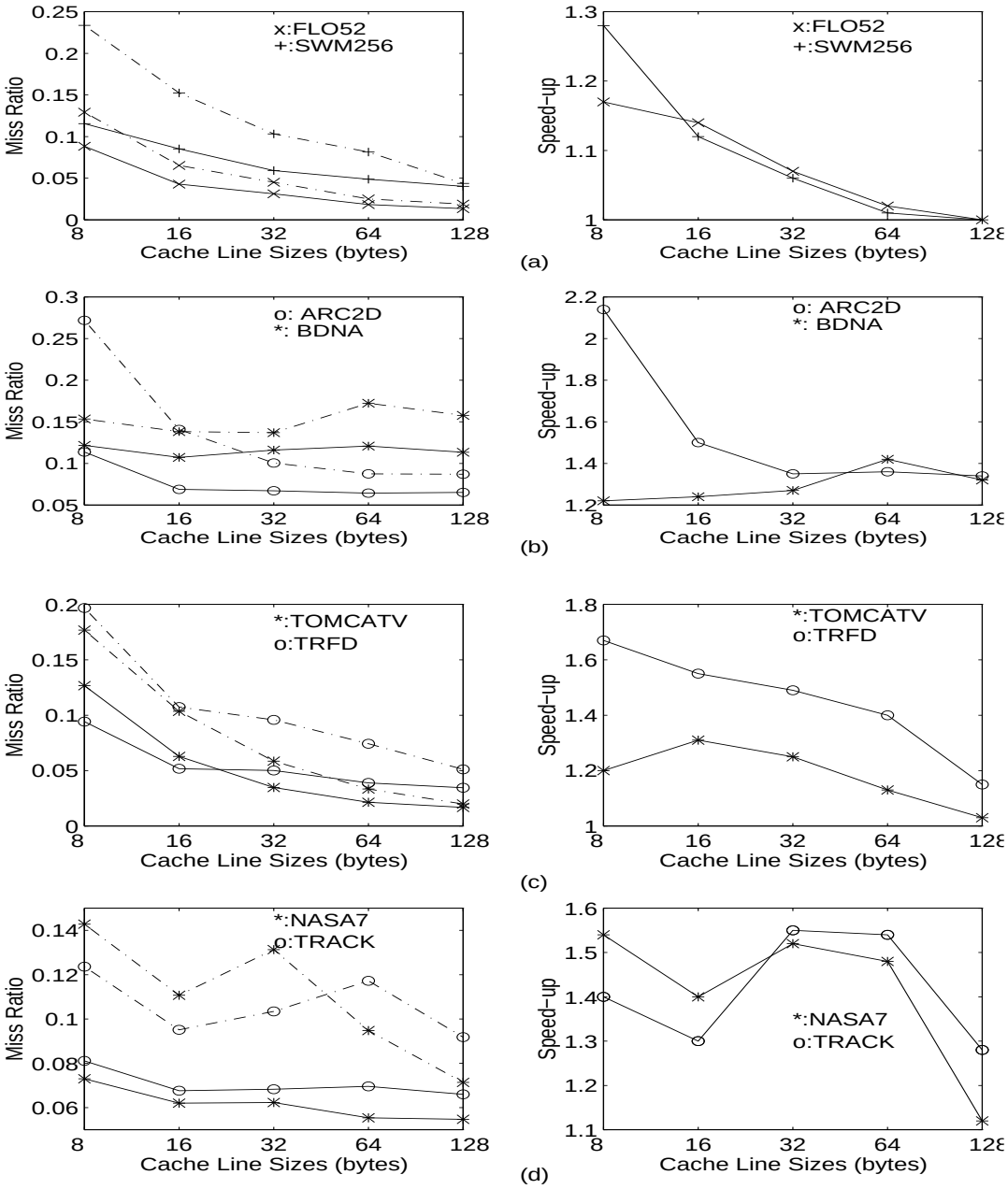
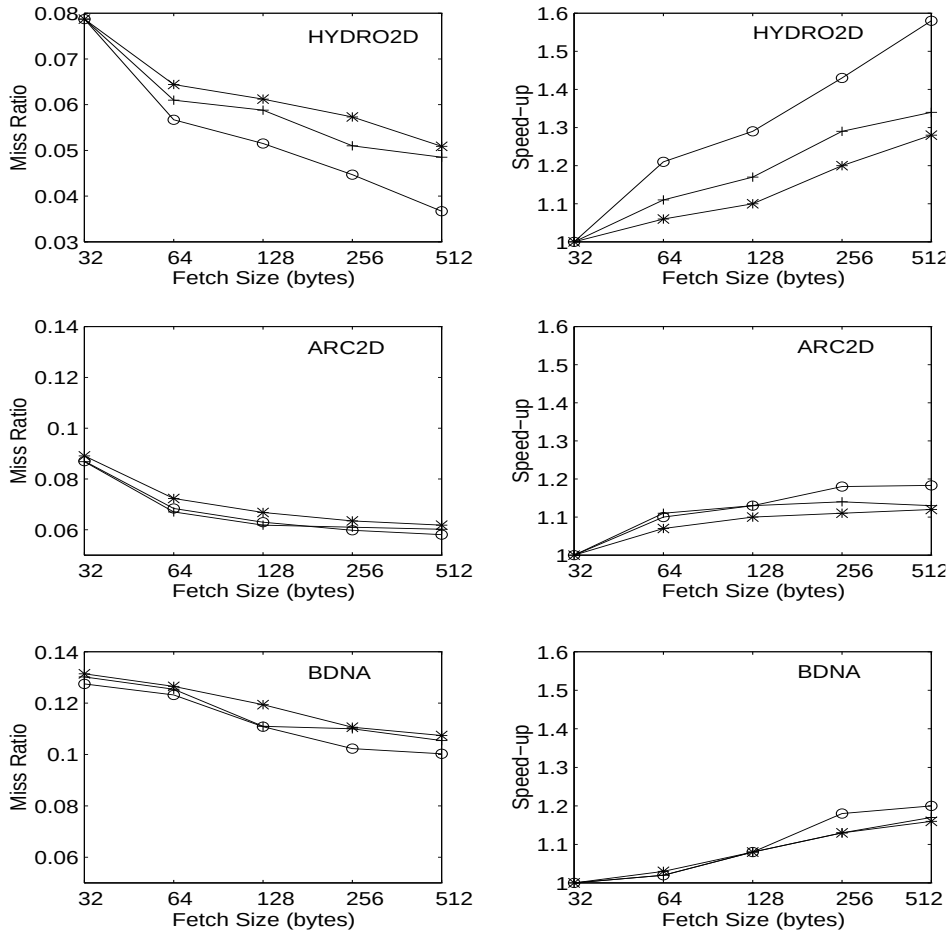


Figure 5: Cache performance (miss ratio & speedup) vs. cache line size

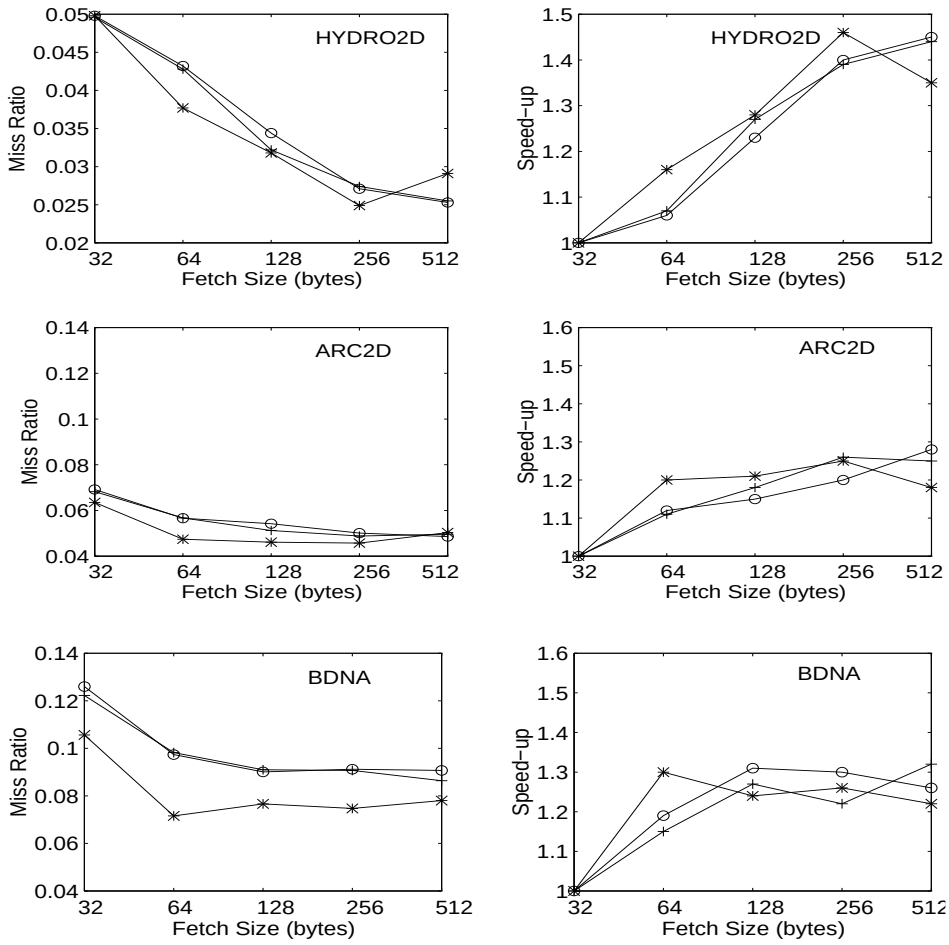


Cache Size = 128K bytes Cache Line = 32 bytes Set Asso = 4

*** : LRU replacement o: MRU replacement +: Random replacement**

$$\text{Speedup} = \frac{\text{Execution time of non-prefetched cache}}{\text{Execution time of prefetched cache}}$$

Figure 6: Cache performance (miss ratio & speedup) of the conventional cache vs. fetch size.

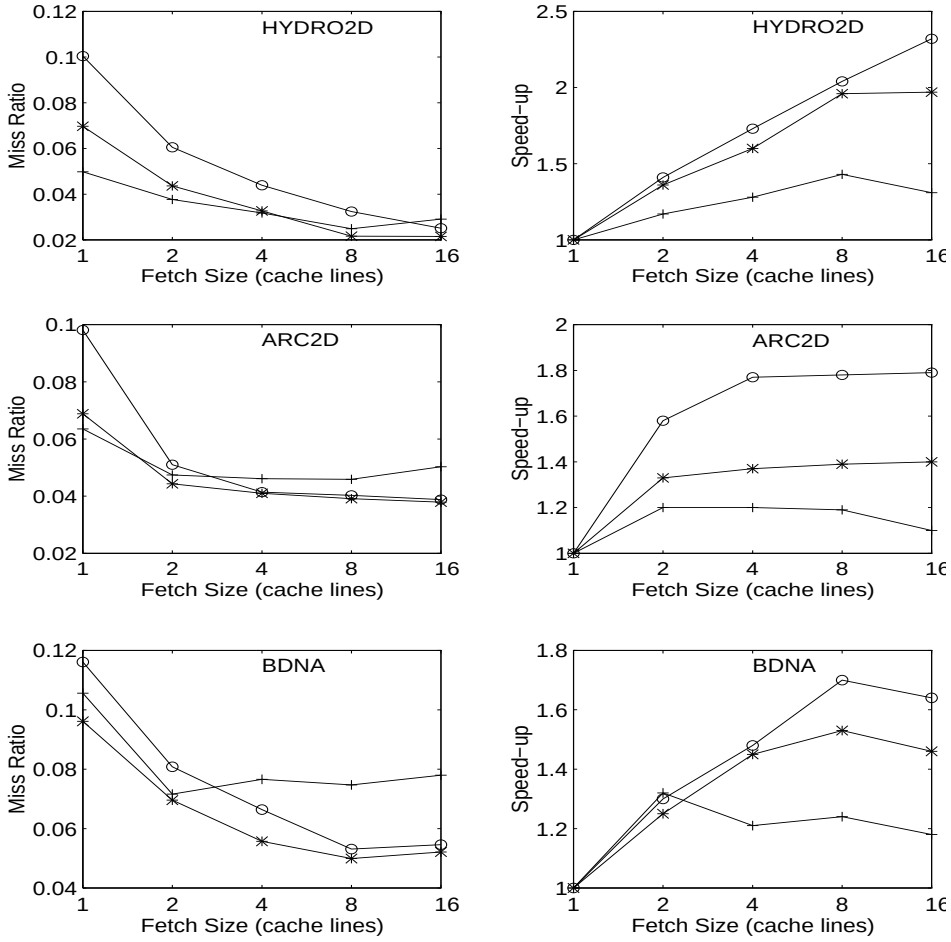


Cache Size = 128K bytes Cache Line = 32 bytes Set Asso = 4

***: LRU replacement o: MRU replacement +: Random replacement**

$$\text{Speedup} = \frac{\text{Execution time of non-prefetched cache}}{\text{Execution time of prefetched cache}}$$

Figure 7: Cache performance (miss ratio & speedup) of the prime-mapped cache vs. fetch size.



Cache Size = 128K bytes Set Assoc = 4
o: cache line = 8 bytes, *: cache line = 16 bytes, +: cache line = 32 bytes

$$\textbf{Speedup} = \frac{\textbf{Execution time of non-prefetched cache}}{\textbf{Execution time of prefetched cache}}$$

Figure 8: Prime-mapped cache performance (miss ratio & speedup) vs. fetch size and cache line size.

PROGRAMS		Conventional cache	prime-mapped cache (noprefecth)		Optimal Cache	
		miss ratio	*miss ratio*	*speedup*	*miss ratio*	*speedup*
PEFECT CLUB	ADM	1.48%	1.34%	1.74	1.01%	2.07
	ARC2D	7.01%	6.13%	1.48	3.81%	1.87
	BDNA	10.60%	9.15%	1.50	5.01%	1.84
	DYFESM	0.31%	0.27%	1.64	0.23%	2.03
	FLO52	2.11%	4.25%	1.31	1.04%	1.69
	QCD	2.47%	6.28%	1.56	1.43%	2.12
	TRACK	4.98%	5.11%	2.47	2.13%	3.10
	TRFD	1.34%	1.96%	1.79	0.37%	2.37
SPEC92	HYDRO2D	2.61%	4.87%	1.27	1.30%	1.65
	NASA7	2.75%	8.75%	1.29	1.59%	1.70
	SU2COR	3.87%	7.43%	1.15	1.98%	1.31
	SWM256	1.68%	4.19%	1.29	1.01%	2.56
	TOMCATV	6.68%	6.01%	2.68	4.21%	3.05
	CFFT2D	7.53%	6.28%	2.43	4.44%	2.21
	VPENTA	6.21%	5.81%	2.26	5.21%	2.48
	MATRIX	4.15%	3.11%	1.19	2.04%	1.68

$$\text{Speedup} = \frac{\text{Execution time of the MM-Model}}{\text{Execution time of the CC-Model}}$$