Program locality of vectorized applications running on the IBM 3090 with Vector Facility

by K. So
V. Zecca

An instruction-level simulator is used to study the program locality of large scientific applications. The simulator, which models an IBM 3090 processor with Vector Facility and a cache, was developed to help a programmer improve the performance of an application through better understanding and use of the Vector Facility and the memory hierarchy of the IBM 3090 system. Our main observations on a set of scientific applications are as follows: (1) although the applications have different characteristics of memory accesses and vectorization, their program locality is high enough to take advantage of conventional cache structures; (2) the cache hit ratio of the vector execution can be quite different from (but not significantly lower than) that of the scalar execution of the same application; and (3) the application programs that are written to optimize the use of the memory hierarchy in the system generally result in higher cache hit ratios than the others. The cache performance of these applications with respect to various cache parameters is also presented. In particular, our study finds that the cache structure of the IBM 3090 is well suited for large scientific applications.

During the last decade, the tremendous growth of computationally intensive applications has led to the rapid development of vector supercomputers in many aspects. The so-called "second-generation supercomputers"[1] have advanced to multiprocessing, e.g., from the Cray-1[2] to the Cray-XMP.[3] In contrast, vector processors have been incorporated into mainframes as built-in accelerators for computationally intensive applications, e.g., the Vector Facility (VF) in the IBM 3090 system.[4] These new and much more powerful processor (CPU) organizations require a well-matched high-performance memory organization to keep them fully utilized; otherwise a longer memory access time could result in a memory bottleneck and limit the overall system performance.

The use of cache memories in mainframe computers has proved to be very effective in reducing the memory access time.[5,6] A cache is a small but high-speed buffer for keeping the recently used data of a CPU accessible within one or two cycles. Its effectiveness relies mainly on the principle of *program locality*,[7] which states that an executing program tends to use memory locations that were either recently referenced or near recent references. In other words, memory references tend to be clustered in space and time.

A *line* of a cache storage is a group of data of consecutive addresses in the memory that are loaded and replaced as a logical unit. Spatial and temporal locality in conventional applications tends to produce multiple references to a cache line over a short period of time. The design of the cache involves the selection of cache-structure parameters such as line size so that the cache performs well for typical applications. However, the use of data structures and the memory reference pattern of vector instructions

in a scientific program can be quite different from conventional (or nonnumeric) applications.

Compared to the use of memory in conventional applications, computationally intensive applications can be characterized by

1. The use of large data sets of multidimensional matrices
2. The use of DO loops, which can address almost randomly several large data sets at a time
3. Addressing the operands of a vector instruction on a *stride* basis, where the access pattern of a vector instruction is said to be *stride i* if any two successive data references are a distance of *i* words or *i* double words apart

A vector execution that addresses elements sparsely located in multidimensional matrices can result in rapid replacement of cache lines and thus poor cache performance. Without an understanding of the program locality of this new area of applications, it is not known if any conventional cache structures would still be effective.[8]

In the past, cache memories had never been implemented in supercomputers; instead, big register files and memory-interleaving techniques had commonly been used to speed up the memory accesses.[9,10] A cache is not a replacement for any of these components but is a very good enhancement for further reducing memory access time in the following three ways:

1. A vector register file is so small that it can hardly hold all of the data needed by the vector processor. A cache can serve as an overflow buffer for data re-use with an access time of only one or two cycles.
2. In a tightly coupled multiprocessor system, where a main memory is shared among a set of scalar and vector processors, the highly interleaved memory will become a bottleneck for multiple memory accesses generated by the processors, even if it is fast enough to satisfy a single memory request. A cache, which stages the momentarily repeated accessed data of a CPU, can alleviate memory contention.
3. While both register files and memory are architectural contexts that require a complex software effort to manage them, a cache can be transparent for any program.

Cache design and caching techniques for vector processors are areas remaining to be fully explored. Understanding the characteristics of memory accesses from real applications is the first and an important step. Prior work in the performance analysis of vector processors and applications has been limited to taking measurements from live systems (see, for example, Jordan[11]) or to an analysis of the algorithmic performance of specific computations. A trace-driven simulation of an experimental vector processor is described in Paul,[12] with an emphasis on the vector architecture and its instruction set. Only recently have we seen a limited amount of performance data from real and complete vectorized applications published.[13]

To help programmers improve the performance of programs on an IBM 3090 equipped with Vector Facility (VF), we have developed a Performance Analyzer (PA), which includes an IBM 3090 VF instruction-level uniprocessor simulator with a cache model. The analyzer accepts scalar and vectorized applications running on IBM 3090 systems and provides levels of execution information of the applications, including the VF and its memory hierarchy (register file, cache, and memory).

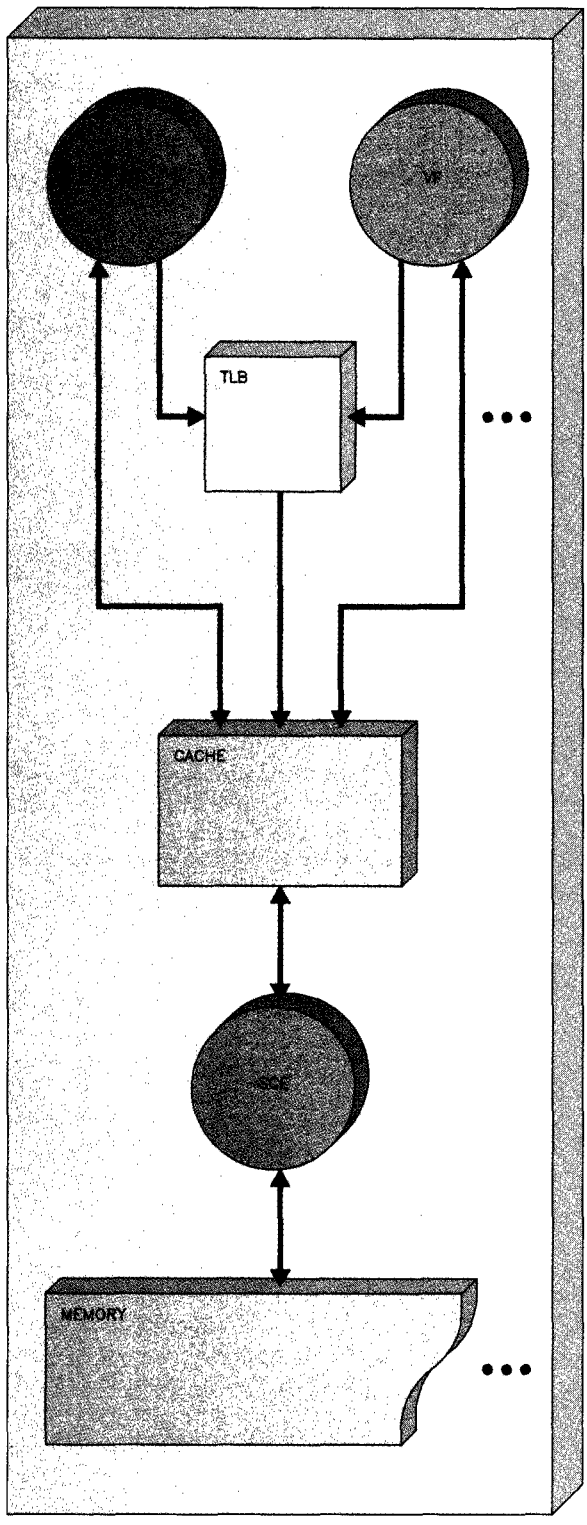
In our study, the cache model, which accepts memory reference traces from the analyzer, is used as a vehicle to measure the program locality of a set of large scientific applications. Therefore, we are not primarily interested in the overall cache performance which affects the throughput of its processors, but rather in addressing the following issues:

1. Program locality of large scientific applications, especially their vectorized execution
2. Differences in cache performance between a vectorized application and its original scalar version
3. Sensitivities of cache hit ratios of scientific (vectorized and scalar) applications with respect to different choices of cache parameters, where the *cache hit ratio* is the ratio of the number of memory references found in a cache to the total number of memory references during the execution of an application

The applications selected for the study differ from one another in the following three characteristics: (1) the extent (low to high) of vectorization, (2) the length (short to long) of data strides, and (3) the extent of using optimizations that take advantage of the Vector Facility and the cache in the IBM 3090.

Moreover, the problem sizes of these applications were chosen such that (1) an application uses as

Figure 1 An IBM 3090 processor unit



much memory as possible subject to a 16-megabyte limit on the total address space, and (2) the simulation time had to be tolerable. Therefore, these large applications generally require longer computation time, and their simulation is very time-consuming.

Following is a summary of our initial observations on this set of scientific applications:

1. A majority of data accesses in these applications have strong spatial locality. For example, most of the vector instructions in these applications are of stride 1. Therefore, their executions still can take advantage of cache memory.
2. The cache hit ratio of a vectorized application can be quite different from (but not significantly lower than) that of the scalar version of the same application.
3. Although all of these large applications favor bigger caches for achieving the same level of hit ratios of conventional applications, our results indicate that the cache structure of the IBM 3090 is also well suited for large scientific computations.
4. Of the applications studied, those which were developed to optimize the VF and the memory hierarchy in the IBM 3090 system usually can achieve a lower memory access rate than the others.

In this paper, a brief description of a processor unit in an IBM 3090 system is first presented. Next our simulator, its cache model, and the set of scientific applications used in this study are described. The 3090 cache model is then used to study the characteristics of memory references of these applications. The succeeding section further examines the program locality of these applications, where cache miss ratios are obtained by varying the parameters of the cache model. Finally, concluding remarks are presented and possible future work is described.

The IBM 3090 system with VF

In this section we describe briefly the main components in an IBM 3090 system which are relevant to our simulator, namely the Central Processor (CP), its cache, and the optional VF. A conceptual diagram of the structure of the 3090 system is shown in Figure 1. For the sake of clarity only one processor unit is depicted. The figure contains a CP executing instructions fetched from memory through the System Control Element (SCE) and a VF which can optionally be attached to the CP. The VF shares the CP access path

to the cache, the main memory, and the translate-lookaside-buffer (TLB), which translates the virtual address of a memory reference to a real address.

The IBM 3090 central processor and its cache. An IBM 3090 processor consists mainly of (1) an I-unit which decodes the instructions to be executed and initializes all necessary memory accesses, (2) an E-unit which executes the instructions and stores all of the changed data back to the memory, (3) a TLB, and (4) a cache. There are sixteen 32-bit general-purpose registers and four 64-bit floating-point (scalar) registers in the E-unit.

A cache miss occurs when the CP accesses a piece of data not currently in the cache.

Data from the memory are available to the CP only after they have been brought to the cache. A *cache miss* occurs when the CP accesses a piece of data that is currently not in the cache. For example, a floating-point instruction that references a single word may cause an entire cache line to be transferred from memory to the cache. However, if the next instruction references another word in the same cache line or any other line already in the cache, the data are immediately available and no memory access is needed. Although the operation of a cache miss is not completed until the whole missed line(s) containing the data is loaded into the cache, the CP is allowed to resume its execution as soon as the actual missed data are fetched to the cache.

The cache in an IBM 3090 has a capacity of 64K bytes (K = 1024) and a line size of 128 bytes. The cache is set-associative such that the 512 lines are distributed over 128 congruence classes and each congruence class contains four lines, where four is the *set associativity* (or *set size*) of the cache. The replacement algorithm at the congruence classes is a variation of the least-recently-used (LRU) algorithm, called the partitioned LRU algorithm.[14] The cache directory contains the real addresses of the lines currently in the cache. The store strategy of the cache is store-in (also known as write-back), which means that any data store from the CP updates only the cache line containing the data; the actual memory update takes place when the line is replaced from the cache.

The IBM 3090 VF. The IBM 3090 Vector Facility (VF) is an extension of the System/370 mainframe for vector processing.[15] Each processing unit in the 3090 contains a central processor and an optional VF. The VF, which has 16 vector registers of 128 words (or 64 double words), is implemented with a pipelined processor having a vector section size of 128; i.e., the pipeline has 128 stages.

For vector processing support, 171 new instructions operating on the vector, general, and floating-point registers are defined. Like the standard System/370 instructions, these vector instructions can access the memory and transfer data between the VF and the memory through the cache.

Unlike a scalar instruction whose operands are mostly in a few bytes or words, a vector instruction operates on data in streams of hundreds of bytes. If the utilization of a vector register file is not high enough, bursts of memory requests can create a bottleneck at the memory. But a vector operation in VF is actually executed in a pipelined way, i.e., one vector element per processor cycle. If all the operands needed by the instruction are already in the cache, they can be delivered to the VF at a rate of one operand every cycle to keep the VF busy during the execution of the instruction. In the case of a cache miss, the first operand takes whatever number of cycles needed to be fetched from the memory to the cache and the VF; subsequent operands in the missed line will need one or two cycles.

In this respect, the frequency of memory references and the cache miss ratio are important factors in the overall cache performance of an application. A programmer can optimize the cache performance of an application by the following general techniques: (1) rearranging program data and code so that frequently accessed data are stored in contiguous memory locations, (2) creating temporary data areas to buffer the frequently accessed elements in sparse matrices, and (3) maintaining the temporal locality of data usage, e.g., avoiding the use of several large matrices at a time.

The simulator and scientific applications

This section describes our performance analyzer (PA) and the set of applications used in the study. Our PA consists of a simulator of an IBM 3090 with VF and a cache model.

A simulator for the IBM 3090 with VF. Our simulation model contains one IBM 3090 processing unit. It simulates all of the System/370 and System/370-XA (Extended Architecture) nonprivileged instructions[16] and all of the VF instructions. There is no address translation in the simulator, so the simulated instructions are operating on virtual addresses. The simulator accepts the module of a program ready to be executed on an IBM 3090 with VF, and it guarantees that the program result after the simulation is the same as that of the execution of the module on a real system. A simulation produces the following information:

1. An instruction trace which contains, for each instruction simulated, an instruction descriptor and the (virtual) addresses of any memory references generated by the instruction
2. The summary of types of instructions and memory references
3. The distribution of all of the instructions executed
4. The distribution of vector counts, i.e., the lengths of all of the vector instructions executed
5. The distribution of data strides of all of the vector instructions

As mentioned before, PA has been designed to simulate the execution of a user program which is also in the form of an executable module in a real system. This design can free the programmer from the burden of changing the source code in a user program to activate the simulator, as is done on other simulators. Therefore, the *same* copy of the program can run native or under PA. Another design feature of PA was to support VS FORTRAN Version 2[17] programs because the majority of scientific/engineering applications are written in FORTRAN.

The PA operates as follows: The first record in the file containing the user module is read. This record contains information such as the starting address and the program length of the module. After the PA allocates virtual storage for the program and a few buffers used at run time by the FORTRAN I/O routines, the program is transferred to memory and is ready for simulation.

The status of the simulation is kept by PA in several internal arrays. They are the general registers, the floating-point registers, the vector registers, the scalar variables that contain the simulated condition code, the program counter (PC), etc. These variables are initialized with the same values as in a real system; e.g., the variable PC is initialized with the starting address of the program. At simulation time, PA and the simulated program are residing in the same address space, but in order to simulate the execution

The simulator includes a cache model which can be activated as an option.

of the user module as if it were running in its own address space, every address referenced by the simulated program has to be corrected by first subtracting from it the loading address of the module and then adding to it the starting address of PA itself.

When the user module is loaded, the PA does not begin the simulation immediately; instead, it waits for user commands for cache setup and others. In particular, a simulation begins only when the commands START or STEP are given. The START command unconditionally begins the simulation, whereas the STEP command requests simulation of a predetermined number of instructions. After initializing the status of the simulation, the PA enters a loop that fetches simulated instructions from the memory, decodes them, and simulates their execution. A FORTRAN STOP statement in the user program terminates the simulation.

The following commands can help the programmer understand the behavior of the simulated program:

1. Begin and suspend the simulation
2. Trace instructions
3. Set traps
4. Pass data to the program from the terminal or from a disk
5. Display registers and memory

Table 2 Ratios (vector/scalar) of memory and cache accesses

Application	References	Misses
LINEQ	0.11	0.92
FFT1K	0.09	0.62
BOAST	0.12	0.22
SIMPLE	0.84	0.49
ARC3D	0.57	0.50

Table 3 Cache miss/memory reference ratios

Application	Scalar		Vector	
	(I & D)	(D only)	(I & D)	(D only)
LINEQ	0.000	0.000	0.002	0.002
FFT1K	0.005	0.009	0.032	0.034
BOAST	0.004	0.008	0.008	0.012
SIMPLE	0.062	0.102	0.036	0.056
ARC3D	0.049	0.086	0.043	0.048

I: instruction accesses, D: data accesses.

memory hierarchy of the IBM 3090 VF system, both LINEQ and FFT1K are highly vectorized and generate low memory-reference rates. SIMPLE is not a highly vectorized application, but its memory references are mostly over contiguous addresses. ARC3D was originally written for a Cray machine; it is highly vectorized but has long data strides. SIMPLE and ARC3D were simulated in their original versions; little effort was spent on rearranging the code for better performance. Since the scalar and vectorized versions of the BOAST application were both available for our study, we have been able to detect whether the program locality is significantly changed when it is rearranged for vector processing.

Characteristics of memory accesses

The basic cache model of the IBM 3090 is used in this section to study the locality of vector processing. It is also used to compare the memory access rates of the vector execution and the scalar execution of an application.

Total memory references. There are three types of memory references generated by a processor: *instruction fetch*, *data fetch*, and *data store*. The vector execution of an application should generate fewer memory references. First, there is a substantial reduction in instruction fetches because, as a result of the vectorization, many scalar instructions are packed into fewer vector instructions. Second, in addition to the scalar register file, there is a vector register file that works like another level of cache storage in front of the cache. In the IBM 3090 VF, the 16 vector registers, which have a total storage of 8K bytes, can reduce many data references for applications that can utilize the vector registers heavily. In Table 2, the reduction in memory references ranges from 16 percent to 91 percent, depending on the degree of vectorization and utilization of vector registers in each application. This reduction is an important factor in the speedup of a program.

Total cache misses. Table 2 indicates that, from scalar to vector execution, the number of cache misses was reduced by 8 percent to 78 percent. This reduction is partly because a vector execution has a comparable cache performance, but primarily it results from a substantial decrease in memory references.

Cache miss ratio. In going from the scalar execution to the vector execution of a highly vectorized application, the memory references and cache misses are substantially reduced. However, the vector execution may result in a higher miss ratio. The reason is that, from scalar execution to vector execution, the "reduced" memory references tend to cause cache hits in the scalar execution. For example, before vectorization, the instruction fetches corresponding to scalar instructions have a very high hit ratio because of their strong spatial locality. The LINEQ, FFT1K, and BOAST applications share this property. However, for an application that is not highly vectorized or is not effectively utilizing the vector registers, it may happen on the contrary that its vector execution has a lower miss ratio. For example, the scalar executions of SIMPLE and ARC3D have shown a higher miss ratio than the vector executions. Therefore, it is not true that the vector execution of an application always has a higher miss ratio than that of its scalar execution.[13] Table 3 shows that, due to a poorer locality of data references, the miss ratio of scalar execution is actually higher than that of the vector execution of the SIMPLE or ARC3D applications. However, because the program locality (especially instruction fetches) in these applications is still strong, their cache hit ratios are not significantly different.

Bursts of cache misses. Since a vector instruction is an ensemble of scalar instructions, the number of memory references for a vector instruction is in general much higher than that for a scalar instruc-

tion. For example, a vector instruction may need hundreds of words instead of one or two words. However, as with most of the vector processors, the VF is implemented by pipelining; the cache is able to cope with a higher access rate from VF as long as it is a one-cycle cache and an executing application does not cause too many cache misses.

Cache misses usually occur in bursts in conventional computer systems.

It is known that cache misses usually occur in bursts in conventional computer systems. This can be explained in the context of program locality. When the CPU has in its cache most of the data needed to execute a program, it causes no or few cache misses, but when it changes its locality because of moving to a new step in the program, or simply when it executes a new program, bursts of cache misses are needed to build up the locality for the new step or program. When the miss distance is measured by the number of instructions, this phenomenon is even more striking for highly vectorized applications.

Figures 2A through 2D show that the distributions of miss distances for the scalar and vector executions of each application are quite different, and cache misses are concentrated on far fewer instructions in a vector execution. Particularly when a highly vectorizable application is written to optimize the use of vector registers and the cache, its misses can occur only in a few instructions. In contrast, by comparing the difference between the two distributions in each of the applications, we see that they are more similar when an application is less vectorized.

Sensitivity to cache parameters

In this section, the locality of vector executions is examined more closely with respect to the changes in three important parameters of a cache model, namely *line size*, *cache size*, and *set associativity*. The vector section size is fixed at 128 in this study, but the results should be applicable to other section sizes as well.

Line size. Line size is the most important cache parameter affecting cache performance (access time and hit ratio) of an application.[25] For a fixed-size cache, the line size is also the most appropriate parameter for examining the spatial locality of an application. In the caches used in this experiment, the set size and the cache size are fixed at 4 and 64K bytes respectively, but the line size varies between 8 and 512 bytes. Two measures of miss ratios from each simulation are presented in Figures 3A through 3D: miss ratio of only data references, and miss ratio of all (instruction and data) memory references.

In these applications, the instruction fetches maintain a very low cache miss ratio even for small caches; therefore, as shown in the figures, the shape of a curve of miss ratios is mostly determined by that of the data misses. Also, since the applications favor a larger line size, the spatial locality of their data references is quite strong.

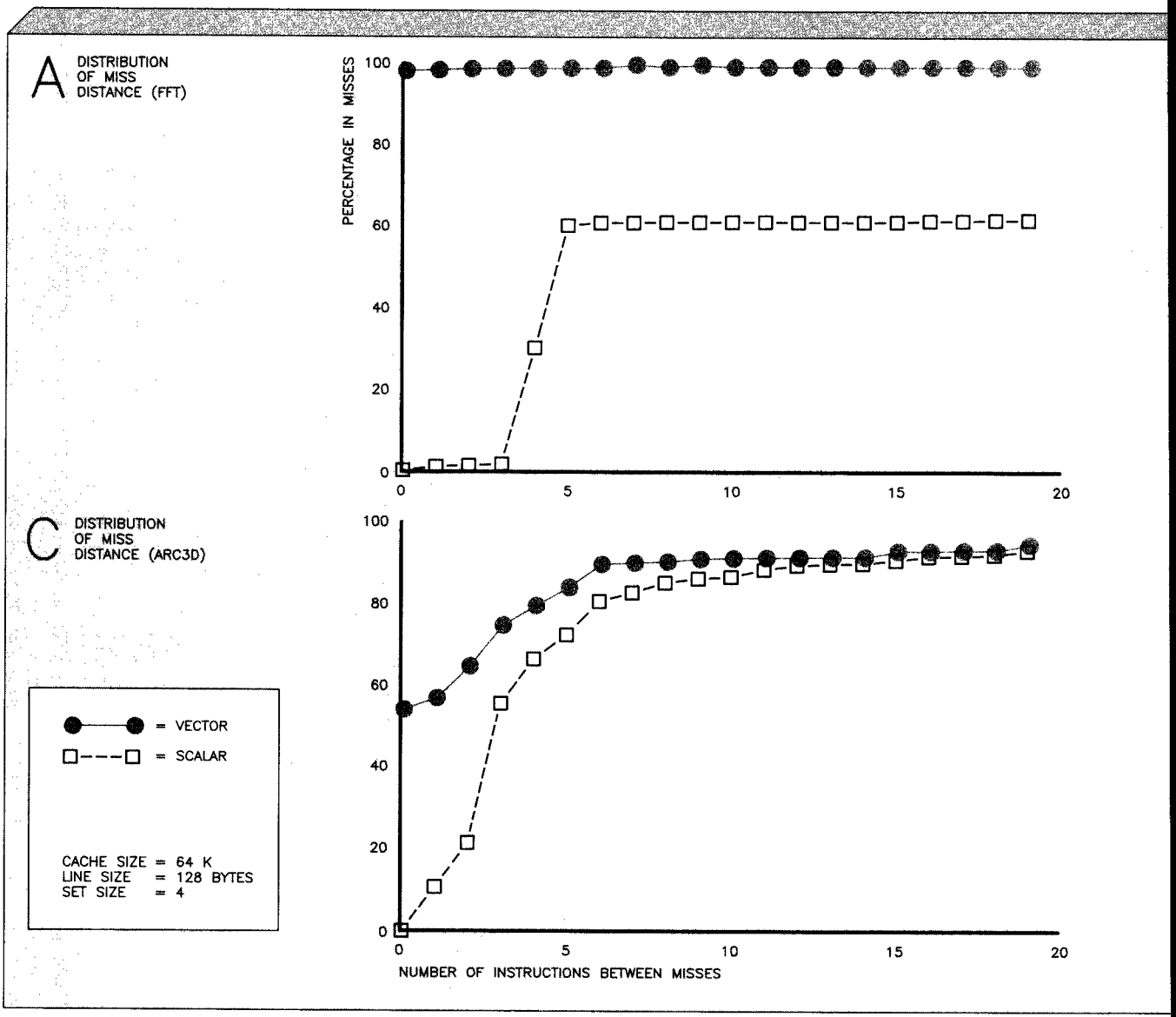
Figures 3A through 3D indicate that the line size has a point of diminishing return at about 128 bytes. Beyond this, the utilization of cache lines is so low that the miss ratio either remains constant or increases. The figures also reveal that the cache performance of vector execution is more sensitive to line sizes than in the scalar case. A careful choice of line size is therefore very important for vector processors.

Cache size. In this simulation, the set size and line size of a cache are fixed at 4 and 128 bytes respectively, but the cache size is varied between 16K bytes and 2 megabytes. These conditions allow us to find out how strong the temporal locality is in the applications.

Figures 4A through 4D show that the miss ratios of these applications can be higher than those of commercial applications, e.g., databases and operating systems, running on mainframe computers with about the same memory size. As an example, in our experience a cache size of 64K bytes usually can maintain a hit ratio between 96 and 98 percent.[14]

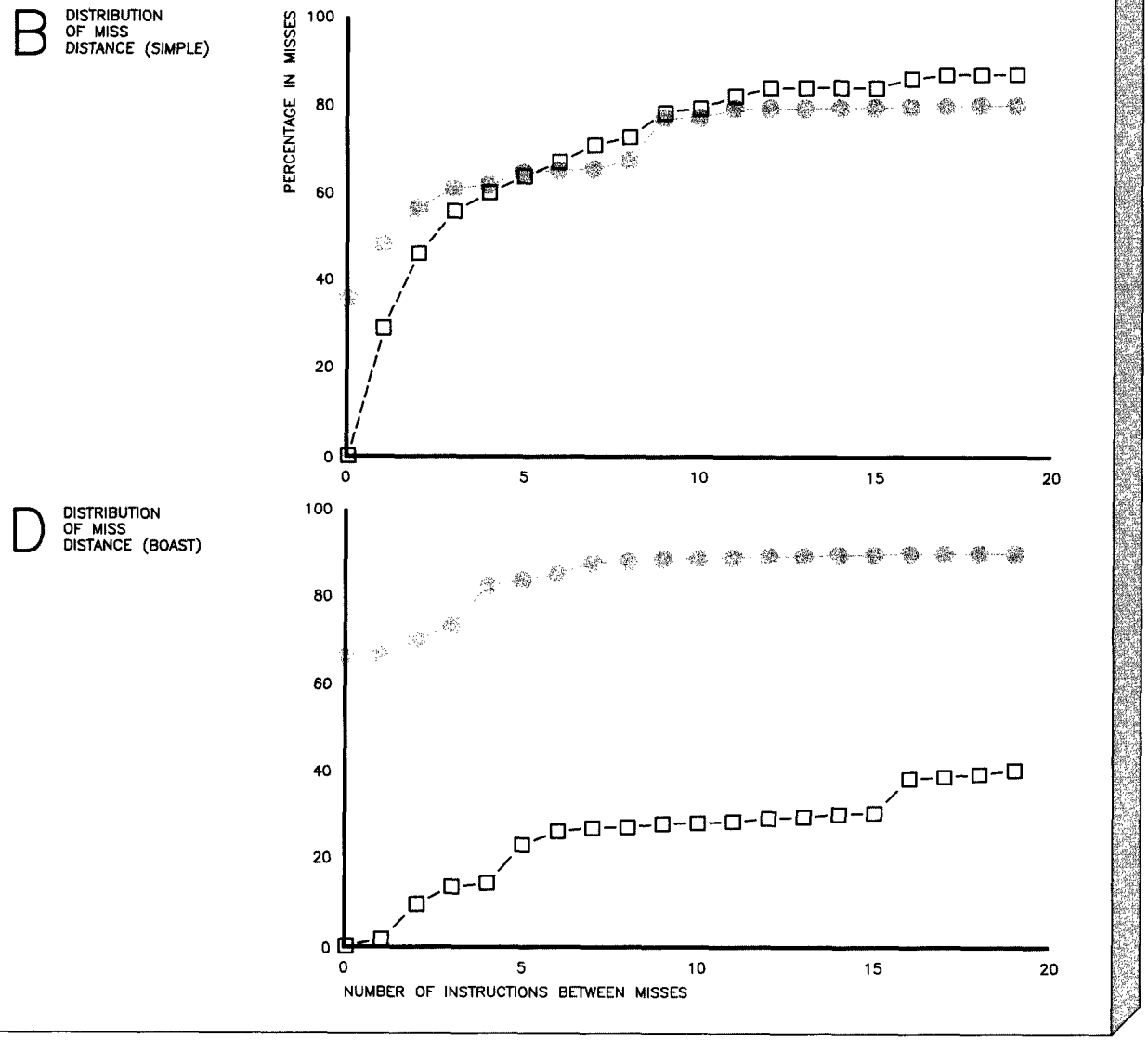
Therefore, in order to maintain a comparable level of hit ratios, a large cache is desirable for supercomputer applications. Not only can a larger cache hold a bigger working set, but it can also cover up long data strides. For example, the cache hit ratio of the

Figure 2 (A, B, C, D) Distribution of miss distance



ARC3D application, which has long strides, is very poor when small caches are used, but it becomes significantly higher when the cache size is 128K bytes or larger.

Set associativity. The set associativity (or set size) of a cache can be viewed as an allowable degree of collisions when many memory references are accessing a congruence class in a short period of time. If the capacity and line size of a cache are fixed, a fully associative cache with the replacement of lines controlled under a single LRU stack can attain the maximum allowable degree of collisions; a direct-map (or set size of one) cache on the other hand has the minimum. Unless the memory references of an application are highly sequential, a larger set size (though more expensive to implement) can generally achieve a higher hit ratio and is therefore preferable.



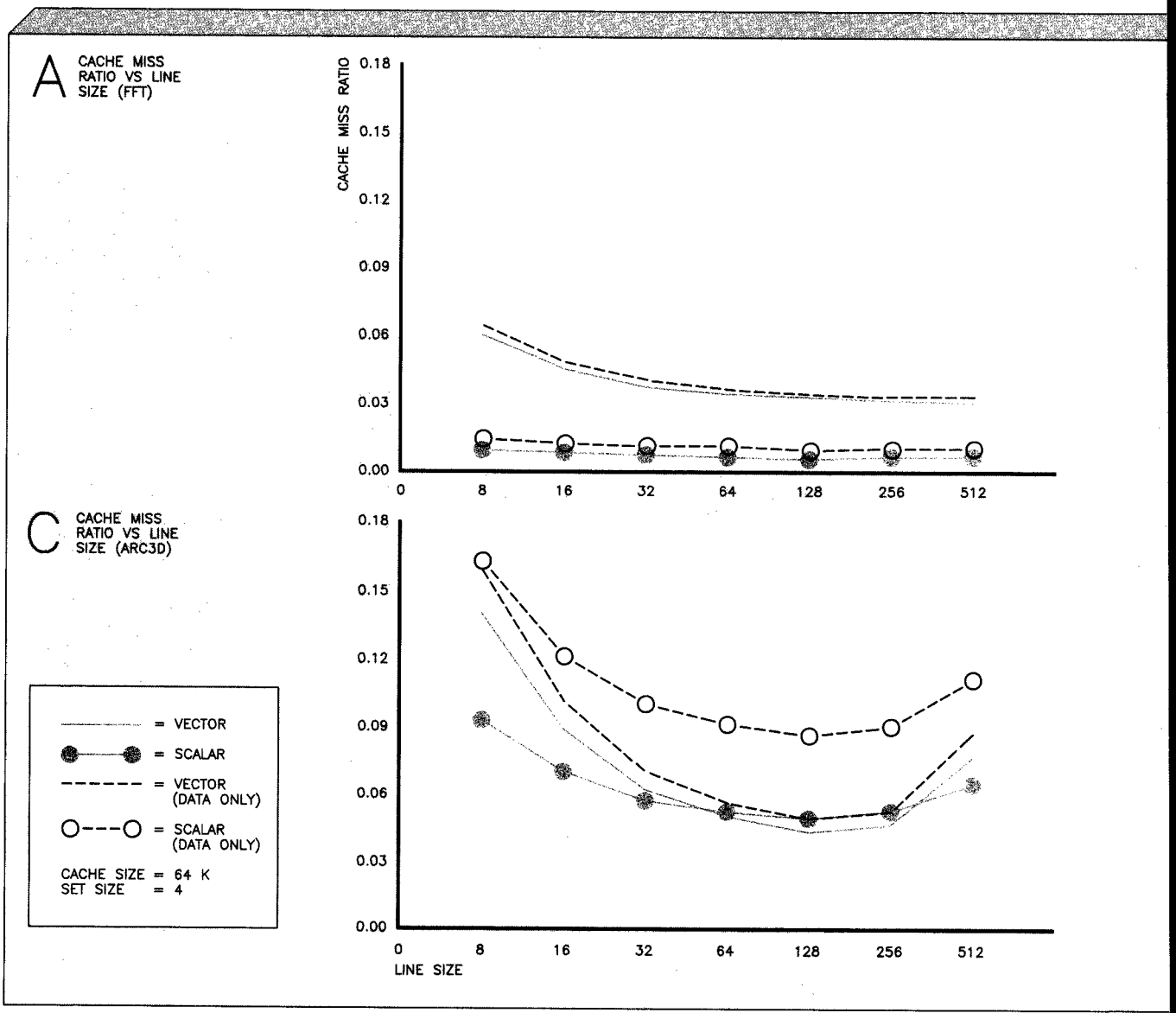
In this simulation, the cache size and line size are fixed respectively at 64K and 128 bytes, but the set size varies over a practical range of 1 to 4. This allows us to understand the direct impact of data strides on the cache performance.

Table 4 indicates that a larger set size is a necessity for scientific applications. The direct-map scheme is

Table 4 Set associativity vs cache miss ratios

Application	Set size		
	1	2	4
FFT1K	0.036	0.033	0.032
BOAST	0.010	0.008	0.008
SIMPLE	0.059	0.044	0.036
ARC3D	0.063	0.045	0.043

Figure 3 (A, B, C, D) Measures of cache miss ratios with line size

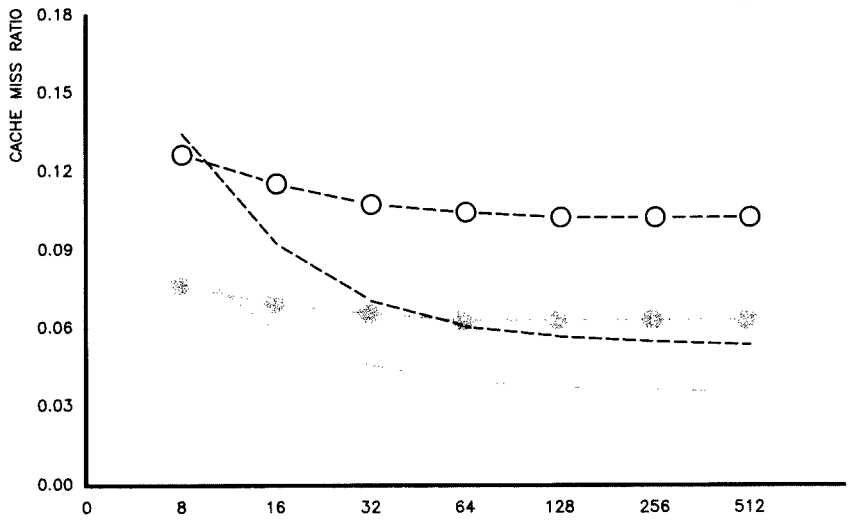


simply not enough to manage the varieties of data accesses in large applications. For example, in SIMPLE and ARC3D, a direct-map cache can generate 50 percent more cache misses than a four-way set-associative cache.

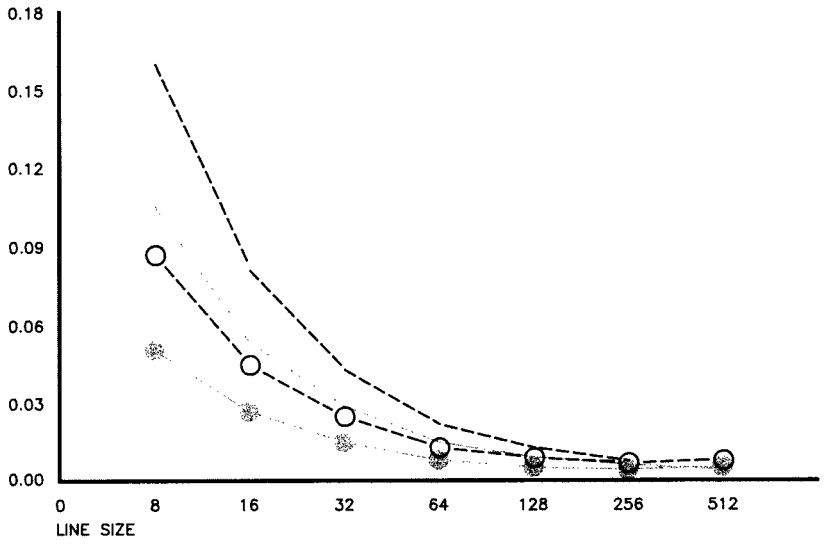
Concluding remarks

An IBM 3090 with VF performance analyzer and the results of using it in a study of the program locality of several large scientific applications have been presented. A cache is used in this study to measure the locality of these applications. By varying the cache parameters, it was found that the spatial and temporal locality of these applications (in scalar and vector executions) is strong enough to show a sufficiently high hit ratio on conventional cache structures. In particular, the cache structure of the IBM 3090 was found to be very effective in coping with the locality of these applications.

B CACHE MISS RATIO VS LINE SIZE (SIMPLE)



D CACHE MISS RATIO VS LINE SIZE (BOAST)

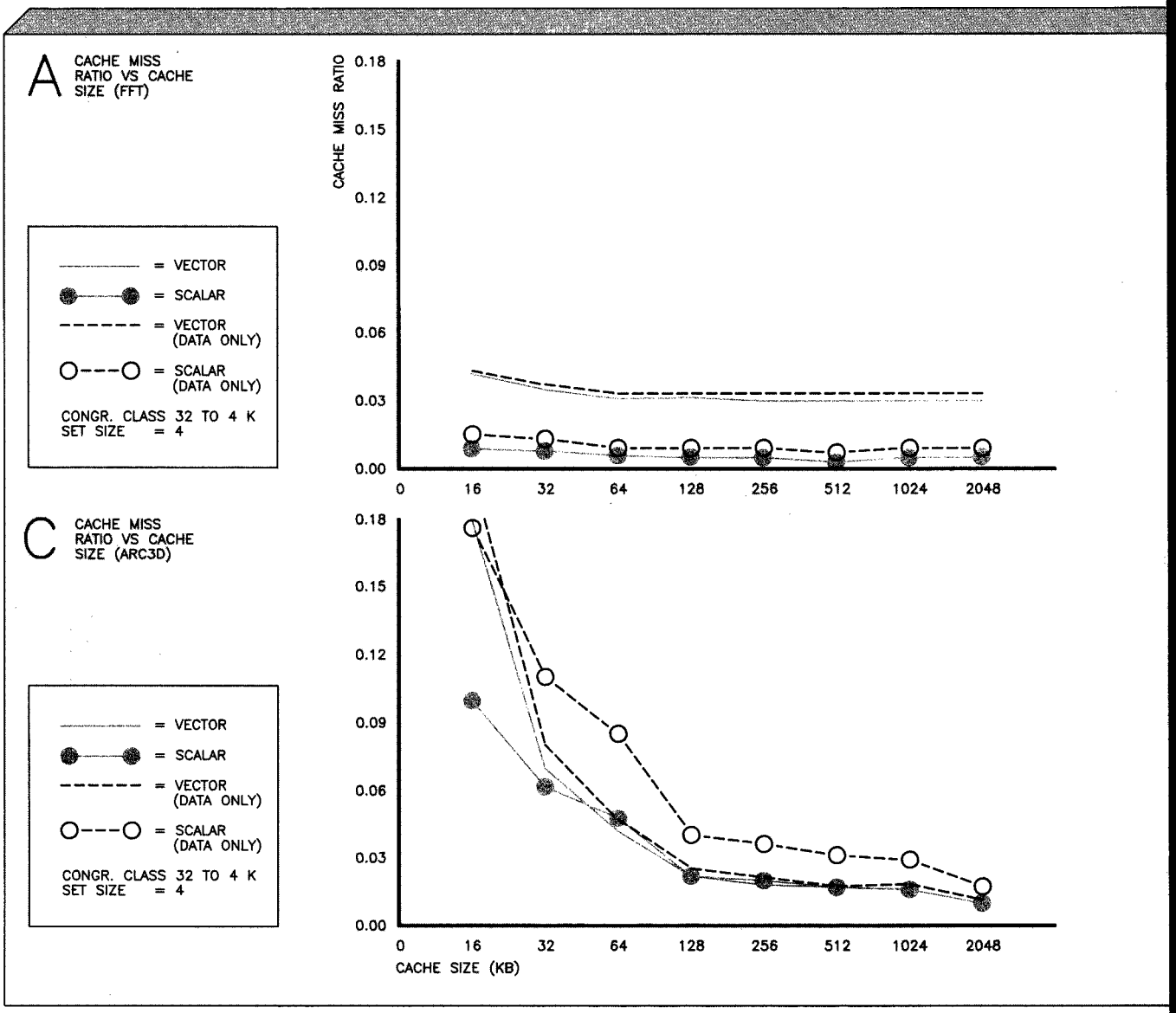


Although the program locality of these applications is in general very strong, in order to optimize the performance of an application in a vector processor, a mere dependence on the vectorizing compiler may not be enough. A substantial improvement of performance is achievable if an application is modified with the memory hierarchy of the system in mind. For example, the memory-access rate and cache miss ratio of the LINEQ and FFT1K applications are much lower than those of the SIMPLE and ARC3D applications in our study. We think that the ESSL library with optimizing routines sets the right direction.

Future use of the simulator can take the following directions:

1. Enhance the Performance Analyzer to simulate the IBM System/370 privileged instructions; this allows us to study the impact of the operating system on the cache performance of an executing program.

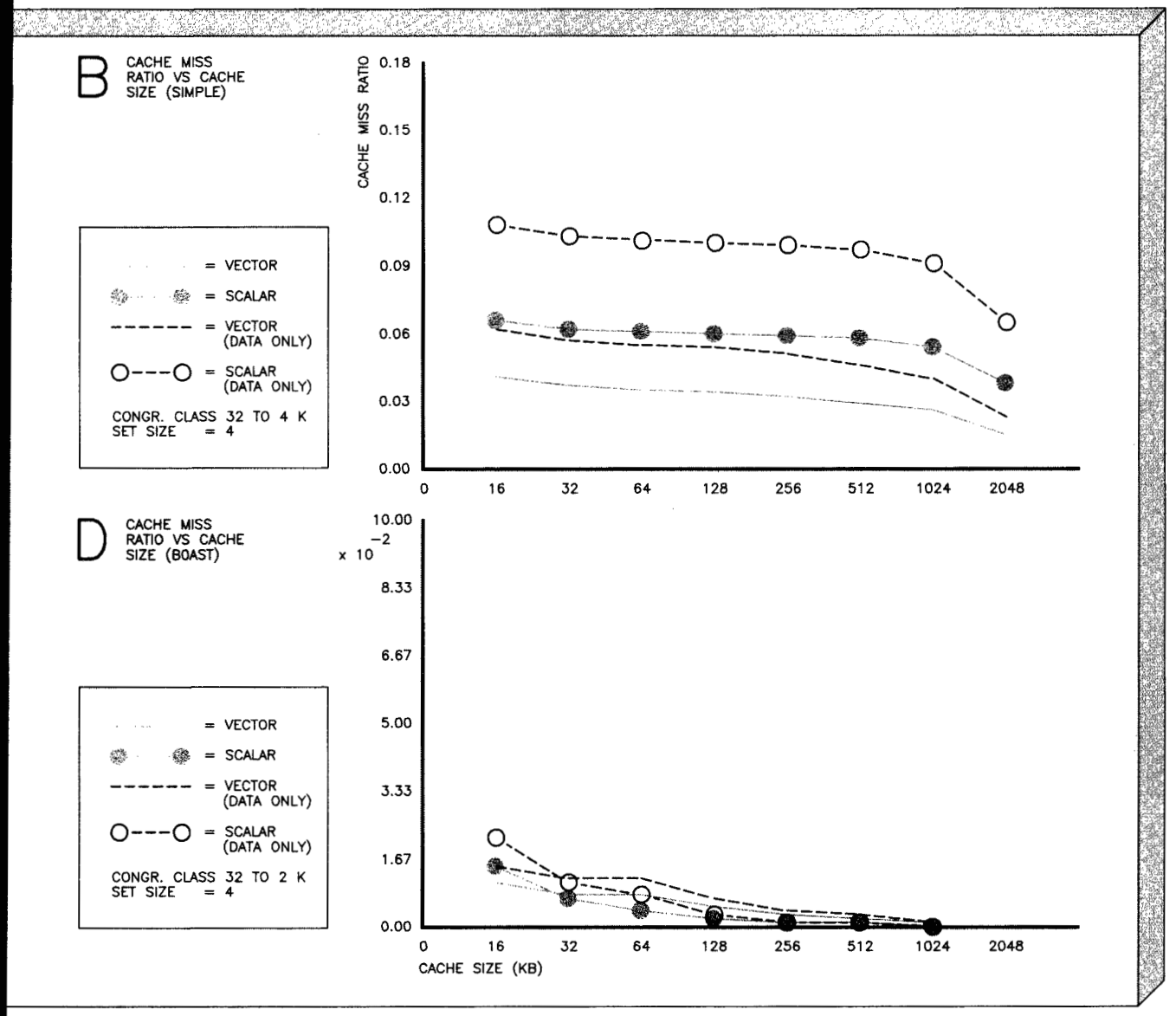
Figure 4 (A, B, C, D) Measures of cache miss ratios with cache size



2. Study more large applications. In particular, it is desirable to characterize applications that have long data strides or poor locality in using vector registers or cache storages.
3. Use the Performance Analyzer to help a programmer improve the performance of an individual application on the IBM 3090 with VF. The simulator can point to problems and can suggest ways to improve the performance of an application. It also allows a programmer to try different programming practices for improving the utilization of register and cache spaces in the system.

Acknowledgments

We would like to thank K. Natarajan, H. Stone, and the referees for helping us to improve the presentation of the paper. We are also indebted to B. Bartley for providing the BOAST application for this study.



Appendix: A sample of Performance Analyzer output

The following output is produced by PA in the simulation of the Installation Verification Program (IVP) distributed with the ESSL package. This IVP aims to determine whether the ESSL installation is correct. We have been using it as a verification program for the PA itself. It calls all of the vector ESSL routines and tests the correct simulation of 119 vector instructions out of a total of 171.

The top of the file contains a header given by the PA programmer and the cache dimensions. The total number of instructions, the cache misses, and the vector instructions follow. The memory reference summary gives details on the number of instructions and data fetches generated by scalar and vector instructions. The same is provided for data store.

Next item is the cache miss summary, a distribution of cache misses over instruction fetches, data fetches,

and data stores. The first row applies to scalar instructions and the second to vector instructions.

The distribution of vector counts gives the programmer an idea of how often the program is using the pipeline, while the stride distribution can help the programmer in detecting pathological situations where long strides could impair performance.

The distribution of distance between cache misses helps the programmer to determine how the cache misses are occurring.

PA also provides a summary of all the branch instructions executed. Finally, there is a table of all instructions executed in the simulated program.

```
VECTOR ESSL INSTALLATION VERIFICATION PROGRAM
CACHE SIZE     64K BYTES
CACHE LINESIZE    128     CONGRUENCE CLASSES     128     SET ASSOCIATIVITY     4
END OF SIMULATION     NISTR =     542518     CACHE MISSES =     8658
VECTOR INSTRUCTIONS =     9145
```

MEMORY REFERENCE SUMMARY

	I FETCH	D FETCH	D STORE	DF+DS	IF+DF+DS
SCALAR	298053	231176	152724	383900	681953
VECTOR		17122	6217	23339	23339
TOTAL		248298	158941	407239	705292

CACHE MISS SUMMARY

	I FETCH	R	D FETCH	R	D STORE	R	DF+DS	R	IF+DF+DS	R
SCALAR	4886	0.016	1387	0.006	2071	0.014	3458	0.009	8344	0.012
VECTOR			199	0.012	115	0.018	314	0.013	314	0.013
TOTAL	4886	0.016	1586	0.006	2186	0.014	3772	0.009	8658	0.012

VECTOR COUNT DISTRIBUTION

(0,10)	6508	(11,20)	358	(21,50)	240
(51,128)	105	(129,256)	0	(257,512)	0
(513,1024)	0	>1024	0		

STRIDE DISTRIBUTION

(4 BYTES)	
(0)	0
(1)	2147
(2)	366
(3)	27
(4,9)	206
.....	
<-1024	0
(8 BYTES)	
(0)	0
(1)	1332
(2)	344
(3)	13
(4,9)	120
.....	
<-1024	0

DISTRIBUTION OF DISTANCE BETWEEN CACHE MISSES

DISTANCE	#	%
0	240	2.77
1	866	10.00
.....		
19	99	1.14
>=20	3110	35.92

BRANCH INSTRUCTIONS
NUMBER 105576 BRANCHES TAKEN 74943 % TAKEN 70.98

DISTRIBUTION OF INSTRUCTIONS EXECUTED

I MNEMONICS	#	%
VAE	101	0.02
VSE	113	0.02
VME	117	0.02
VDE	3	0.00
VSTVP	134	0.02
VLI	70	0.01
VSTI	9	0.00
VLID	162	0.03
VSTID	67	0.01
VSRL	44	0.01
VSLL	2	0.00
.....		
SPM	3	0.00
BALR	6335	1.17
BCTR	1247	0.23
BCR	10511	1.94
SVC	22	0.00
MVCL	309	0.06
PACK	2	0.00
UNPK	4	0.00

Cited references and notes

1. E. W. Kozdrowicki and D. J. Theis, "Second generation of vector supercomputers," *Computer* **13**, 71–83 (November 1980).
2. R. M. Russell, "The Cray-1 Computer System," *Communications of the ACM* **21**, No. 1, 63–72 (January 1978).
3. S. Chen, "Large scale and high-speed multiprocessor system for scientific applications," *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, J. S. Kawalik (Editor), Springer-Verlag, New York (1983).
4. *IBM System/370 Vector Operations*, SA22-7125, IBM Corporation; available through IBM branch offices.
5. J. S. Liptay, "Structural aspects of the System/360 Model 85: II. The cache," *IBM Systems Journal* **7**, No. 1, 15–21 (1968).
6. A. J. Smith, "Cache memories," *ACM Surveys* **14**, No. 3, 473–530 (September 1982).
7. E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1973).
8. H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Co., Inc., Reading, MA (1987), Chapter 5.
9. P. Budnik and D. Kuck, "The organization and use of parallel memories," *IEEE Transactions on Computers* **C-20**, No. 12, 1566–1569 (1971).
10. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, Inc., New York (1981).
11. K. E. Jordan, "Performance comparison of large-scale scientific computers: Scalar mainframes, mainframes with integrated vector facilities, and supercomputers," *Computer* **20**, No. 3, 10–23 (March 1987).
12. G. Paul, *Studies in Vector Architecture*, Research Report RC-9234, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.
13. R. S. Clark and T. L. Wilson, "Vector system performance of the IBM 3090," *IBM Systems Journal* **25**, No. 1, 63–82 (1986).
14. K. So and R. Rechtschaffen, "Cache operations by MRU-Change," *IEEE Transactions on Computers* **37**, No. 6, 700–709 (1988).
15. W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Systems Journal* **25**, No. 1, 51–62 (1986).
16. *IBM System/370 Extended Architecture—Principles of Operation*, SA22-7085, IBM Corporation; available through IBM branch offices.
17. *IBM VS FORTRAN Version 2—Language and Library Reference*, SC26-4221, IBM Corporation; available through IBM branch offices.
18. V. Zecca, *A Performance Analyzer for the IBM 3090/VF with Cache*, IBM ECSEC Technical Report, IBM European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy (1988).
19. R. G. Scarborough and H. G. Kolsky, "A vectorizing Fortran compiler," *IBM Journal of Research and Development* **30**, No. 2, 163–171 (1986).
20. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, SC23-0184, IBM Corporation; available through IBM branch offices.
21. R. C. Agarwal and J. W. Cooley, "Fourier transform and convolution subroutines for the IBM 3090 Vector Facility," *IBM Journal of Research and Development* **30**, No. 2, 145–162 (1986).
22. The SIMPLE code was developed by W. P. Crowley, C. P. Hendrickson, and T. E. Rudy at the Lawrence Livermore Laboratory, Livermore, CA, 1978. Our version is from Axelrod et al.[23]
23. T. S. Axelrod, P. F. Dubois, and P. G. Eltgroth, "A simulation for MIMD performance prediction—Application to the S-1 MkIIa Multiprocessor," *Proceedings of the 1983 International Conference on Parallel Processing* (August 1983), pp. 350–358.
24. The version for fluid dynamics was written in September 1981 by Benek and revised by T. H. Pulliam at NASA Ames Research Center, Cleveland, OH, June 1983.
25. A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Transactions on Computers* **C-36**, No. 9, 1063–1075 (1987).

Kimming So *IBM Research Division, T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. So received his Ph.D. in computer science in 1980 from the University of California at Santa Barbara. He has been a research staff member at the Research Center since 1980, working on the design and performance evaluation of multiprocessor organizations and trace-driven simulation of processor and cache operations. Dr. So's current interests also include parallel processing and programming.

Vittorio Zecca *IBM European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy.* Dr. Zecca obtained a degree in electronical engineering from Rome University in 1981. From 1982 to 1985 he worked in the aerospace field having responsibility for data management for the San Marco project. In 1985, he joined the IBM Rome Scientific Center. He later was one of the initiators of ECSEC. He received an IBM Outstanding Technical Achievement Award for his contribution to the area of parallel processing. Dr. Zecca's current interest is tools for efficient vector/parallel processing.

Reprint Order No. G321-5337.