Vector system performance of the IBM 3090

by R. S. Clark
T. L. Wilson

Performance of the Vector Facility of the IBM 3090 processor is discussed. The paper has two parts, the first presenting factors affecting performance measurement of the Vector Facility and the criteria for its design. In the second part, use of the 3090 storage hierarchy to support the vector processing implementation is the main aspect of the discussion.

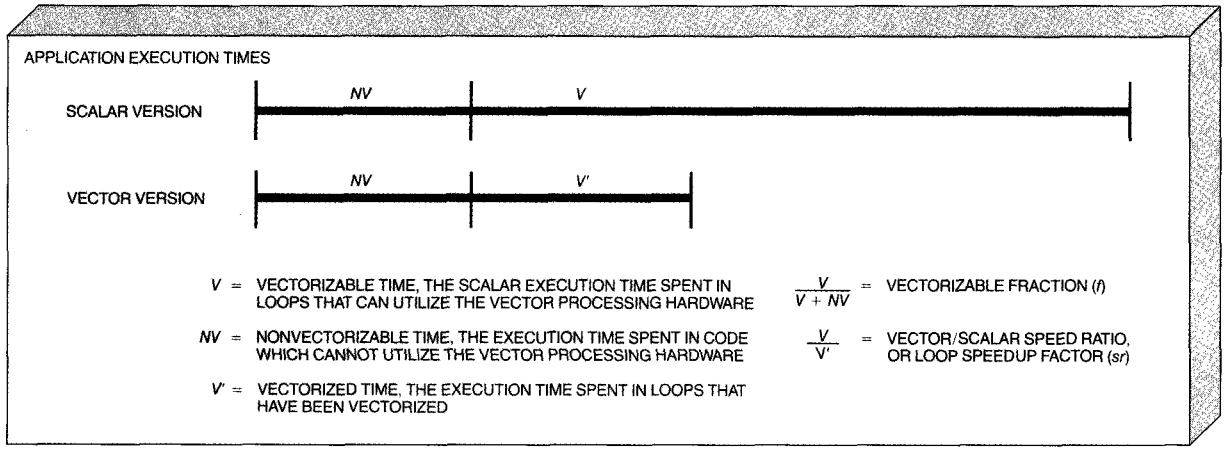
The introduction of the IBM 3090 Vector Facility system not only represents IBM's entry into large-scale vector processing, but also introduces a unique approach to the integrated structure of a vector processing system. Significant factors in the system design of the Vector Facility include the following:

- Use of an established high-performance system, the IBM 3090 computer, as a base upon which to build a vector processing system
- Use of the 3090 storage hierarchy to support vector processing operations on all models, particularly the advantage obtained from use of the cache, or high-speed buffer, of the 3090 storage hierarchy
- Support of tightly coupled multiprocessor vector processing, resulting from adding the Vector Facility as a feature to the base multiprocessor configurations of the 3090 Models 200 and 400 (The ability to support multiprocessor vector processing using the 3090 storage hierarchy is largely a result of the cache storage hierarchy of the 3090, with a shared main storage and a dedicated cache buffer storage for each Central Processor, or CP.)

This paper discusses the performance of the IBM 3090 processor with the optional 3090 Vector Facility. We focus the discussion on what we believe are two of the most significant aspects of the system design. The first is the use of application performance criteria as the basis for product design and evaluation. In Part I of the paper, the factors affecting job performance are explored, which leads to a discussion of the metrics of vector performance measurement, specifically the use of MFLOPS (millions of floating-point operations per second) measures. Finally, we describe the approach used on the 3090 Vector Facility for making product measurements, consistent with the evaluation framework of application performance. In this part of the paper, the intent is to provide insight into the 3090 Vector Facility product and a characterization of its performance.

The second key aspect of the system design, on which we focus in Part II of the paper, is the use of the 3090 storage hierarchy to support the vector

Figure 1 Vectorization terminology



processing implementation. This design is an element of the balance that follows from using application performance as the design and evaluation criteria. We discuss the factors which make it feasible and appropriate for the 3090 to have a cache-based vector implementation. We discuss the analysis of storage demand parameters for vector jobs, then discuss the analysis of vector loop performance on a cache system. Finally, we illustrate an analysis of vector loop performance with examples of matrix multiplication.

Part I — Application performance

From the viewpoint of system design implications, the most significant aspect of the 3090 Vector Facility was the objective of providing a balanced system implementation. What this means, very simply, is that the Vector Facility was not intended to be the fastest possible vector processing hardware. Rather, the objective was to complement the scalar performance of the 3090 system with a level of vector processing capability justified by application characteristics, without limiting multiprocessing configurations and opportunities for parallel processing that are inherent in the basic system design. This required that a context, or framework, of evaluation be established early in the design. It required that the analysis and evaluation of the product, as it progressed through implementation, be conducted within that established framework. For the 3090 Vector Facility system, this framework for evaluation was the performance obtained by computationally intensive *application programs.*

This context was established during the product definition, consistent with the balanced system approach, and was used in establishing product objectives. Such a framework for evaluation implied that the product was not to be designed for, nor evaluated by, peak performance specification or loop measurements. The performance of application programs is the performance of *complete* application codes, such as structural analysis, fluid dynamics applications, or seismic processing, and solving problems of significant size, including not only the computations needed to solve the problem, but also the I/O, data manipulation, and other noncomputational data processing typically found in large production applications.

With application performance established as the criterion for product evaluation, it was necessary to build a set of sample engineering/scientific application programs for use in product measurement. Measurements on the finished product provide a means of evaluating the product against its objectives, but just as importantly, they also serve as a means for setting expectations for potential users, who may observe a confusing terminology and overwhelming variation of numbers applied to the performance of high-speed engineering/scientific computer systems.

Application performance factors

The computationally intense nature of engineering and scientific application programs results from iterative mathematical computations on sets of data. These applications also contain code to manage and manipulate data, perform I/O operations, initialize data values, and report or summarize the solution results. The portions of an application that may be able to make use of vector hardware are the iterative computations on array data which are typically found in the loops coded as FORTRAN DO statements. The performance improvement of an application on a vector processor derives from the improved speed of performing loop computations using the vector hardware. The proportion of scalar execution time spent in such loops, compared to the full scalar execution time of the application, is called the vectorizable fraction for the application problem.

The improved execution time for the application problem results from the collapsing of the execution time of vectorizable loops. Figure 1 illustrates this simple relationship and defines the terminology of vectorization used in this paper. The factor by which the execution time of a vectorizable loop improves when the vector instructions and hardware are used is called the loop speedup ratio, or the vector/scalar ratio.

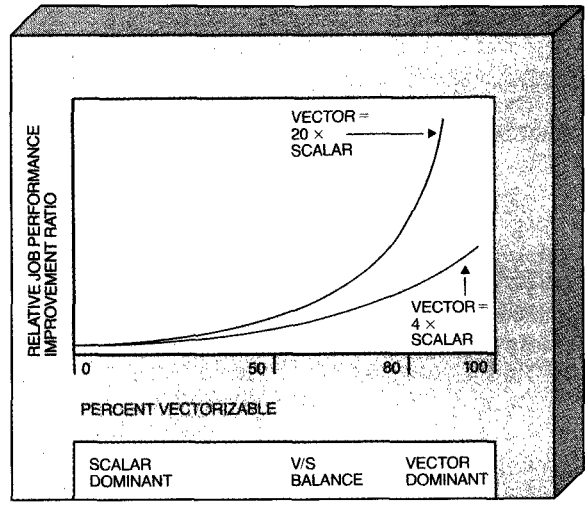
The relationship between an improvement in loop execution time and the overall improvement in application problem execution time is shown in Figure 2. It plots vector application performance (compared to the performance of a scalar version) as a function of vectorization percentage, for several vector/scalar ratios.

Another view of the same relationship may be seen in Figure 2 of the paper in this issue by Gibson et al.[1] These curves illustrate the relationship known as Amdahl's Law,[2] which states that the overall speedup factor S for an application is determined by the relationship

$$S = \frac{1}{(1-f) + f/sr}$$

where f is the vectorizable fraction for the application and sr is the loop speedup factor obtained by moving computation to the vector hardware.

Figure 2 Vector processing job speed improvement



This fundamental relationship is the key to all aspects of vector system performance analysis, from setting design goals, to projecting the performance of individual benchmarks, to analyzing the results of specific benchmark measurements.

One of the key design considerations for the IBM 3090 Vector Facility was the goal for a vector/scalar ratio, and one of the main factors in determining the goal was the range of application vectorizability projected for anticipated 3090 Vector Facility applications. On the basis of application studies and analysis of the role of vector speed in application performance, the design goal for the 3090 Vector Facility was established as a vector/scalar ratio in the vicinity of four, optimizing the design for the midrange of application vectorizability.[1] It is important to note, from a performance perspective, that while the design goal was oriented to the midrange of vectorizability, the resulting product has good price/performance for applications well into the high vectorizability range. One of the reasons for this is that the design goal was compatible with the use of a cache storage hierarchy for vector operand delivery, making it feasible to build a vector system based on the 3090 processor and its storage hierarchy. We examine this aspect of the system design and look at some of the sensitivities it produces in Part II of this paper.

Product measurement

3090 Vector Facility performance was measured in order, first, to provide validation that the system performs as it was designed to perform, and second, to set general expectation levels for potential users of the system through publication of performance data at the time of product announcement. The methodology was to measure application programs similar to, or sampled from, the ones customers would actually run on the system. While measurement of short loops is useful in the detail of processor design, it is our view that measurement of real application codes is the way to judge a system, and that expectations for potential customers should be stated in terms that can be related to the applications they might run.

Two approaches may be taken to stating measured performance: (1) measuring results of standard application problems relative to the same problems running on known, predecessor systems, or (2) measuring results in some absolute terms of work per unit of time. IBM processor performance for commercial workloads, such as batch processing, time-sharing, or transaction processing, is published today in terms of relative performance using standard measurement workloads. The metrics include Internal Throughput Rate (ITR) ratio, a measure of work accomplished per unit of processor time, and External Throughput Rate (ETR) ratio, a measure of work per unit of elapsed time.[3,4]

The use of instruction execution rate, or MIPS (millions of instructions per second), has proven to be unsatisfactory for describing processor performance, simply because an instruction is not a unit of work that can be correlated to the external function of a transaction or application. The IBM 370/XA vector architecture offers an excellent example; while the speed of solving the application problem improves, the number of instructions needed to perform a computation is significantly lower using vector instructions (because one instruction performs many operations).

MFLOPS as a metric. For scientific and engineering applications, there exists a usable, though by no means perfect, measure for computational work. This measure is the count of computational floating-point operations which the application performs, i.e., the Floating-Point Operation (FLOP) count. Calculating the rate of floating-point operation execution yields MFLOPS, or millions of floating-point operations per second. The problem with MFLOPS as a metric, however, is that it can be, and has been, applied indistinguishably in at least three different contexts:

- A full application job (JOB_MFLOPS)
- The loop instructions or computational kernels within an application (LOOP_MFLOPS)
- The hardware design of the vector instruction execution element (PEAK_MFLOPS)

In addition to being used to describe the execution speeds of jobs, loops, or instructions, MFLOPS is also used as a system rating or as a general yardstick of system capacity, in statements of system requirements from customers. MFLOPS has generally been applied as a processor rating to a single processor; when it is used to describe the capacity of multiprocessor systems, additional qualification is needed to distinguish throughput capacity (multiple jobs running concurrently) from single-job parallel execution capability, where the measure of concern is the elapsed time of a single job. The precision of meaning that is lacking in the use of MFLOPS terminology is one of the main advantages of the ITR and ETR metrics.

JOB_MFLOPS. The number of computations needed to solve a problem may, within limits, be viewed as an indication of the computational size of the problem, or as a simple "weight" for the problem. In the context of a problem plus the mathematical approach and solution method(s) used to solve it, the essence of arriving at a solution is to perform the computations; the rate at which computations are performed in solving the particular problem can provide an expression of speed of solving the problem. It is the speed at which an engineering/scientific problem can be solved that is ultimately of concern to the owner of the problem.

JOB_MFLOPS, however, is a time-dependent mixture of vector loop speeds, scalar loop speeds, and scalar serial speed, meaningful only in the context of the particular job under study. It is not a constant across all jobs, as illustrated by the JOB_MFLOPS data which are presented in Table 1. The variability of job content is demonstrated by two jobs with approximately the same level of vectorization (Seismic Analysis and Black-Oil Res-

Table 1 SEAP application benchmark measurements

3090 Single-Job Measurements			
SEAP Application	**Percent Vectorizable[1]**	**ITR Ratio Vector:Scalar[2]**	**Vector JOB_MFLOPS[3]**
Static Structural Analysis	82	2.03	14.7
Black-Oil Reservoir Simulation	80	2.35	8.4
Seismic Analysis	82	2.65	16.9
Buckling Analysis	60	1.57	8.5

[1] Calculated using processor times of both vector and scalar versions of each job, plus Vector Facility usage time, reported by SMF system accounting:

$$\%V = \frac{T_s - (T_v - T_u)}{T_s} \times 100$$

where T_s = scalar version total processor time, T_v = vector version total processor time, and T_u = vector usage processor time of vector version.

[2] Ratio of application times, 3090 running scalar version compared to 3090 VF running vector version.

[3] For vector version, job floating-point operation count (FLOPs), in millions, divided by processor time.

ervoir Simulation) having significantly different JOB_MFLOPS, whereas two jobs with dissimilar vectorization (Buckling Analysis and Black-Oil Reservoir Simulation) have comparable JOB_MFLOPS values. In the context of a particular job, however, the measure does represent one perspective of what is sometimes called sustained, or effective, computational speed. JOB_MFLOPS can be calculated from the job FLOP count and the processor execution time for the job, as illustrated in the measurement data in Table 1. While the figure is calculable, if the FLOP content of the job is known, we have not found it to be useful. It may be observed from Table 1 that the job performance improvement, that is, the ITR ratio in going from scalar to vector, does not correlate to the JOB_MFLOPS for the applications we have examined.

One of the drawbacks of using JOB_MFLOPS as a measure of job performance comes from the fact that the floating-point operations (FLOPs) performed in a job are in general difficult or impossible to count. For a simple loop, the number of operations can be counted by inspection or calculation, but for a large, long-running application, the count is often unobtainable in environments other than the laboratory. Job FLOP counts were obtained in the lab for the jobs in our benchmark measurement set, and JOB_MFLOPS for some of these jobs are shown later in this paper.

Sustained JOB_MFLOPS is a function not only of processor speeds but of individual application content and structure, and shows wide variability as a result. It is therefore impossible to assign a single MFLOPS rating to a system, even sustained JOB_MFLOPS, as a basis for comparing the performance of systems.

LOOP_MFLOPS. LOOP_MFLOPS is a measure of execution speed for a subroutine loop or code kernel. Understanding the performance of the code utilizing the vector hardware, that is, the vectorized loops, is crucial to understanding what the performance of an application will be. Measurements or estimates of loop performance are useful in projecting the performance of full application programs, if something is known about their vectorization potential. The projection is not a simple linear relationship, however, as illustrated by the curves showing Amdahl's Law in Figure 2.

Analysis and measurement of loop speed are also useful in analyzing the sensitivities of vector system performance, as long as it is qualified that the observed loop speeds will always be dampened in application measurements by the nonvectorized portions of the application. An analysis of matrix multiplication loop speeds is presented in the second part of this paper, for example, to illustrate the effect of cache usage and vector register reuse on loop performance. The data are summarized

in LOOP_MFLOPS terms, although even here the use of the MFLOPS metric is not crucial to the analysis of the sensitivities.

Loop measurement is particularly difficult on the 3090 Vector Facility, since loop measurement techniques have often removed the full data-model context in which the computational logic of a loop would operate in an application. Loop measurements, therefore, are subject to the cache behavior the loops create on small arrays of data and usually do not reflect accurately the effect of the cache storage hierarchy on full application performance.

Peak execution rate is more a description of the product design than a definition of performance.

Measures of loop performance which are taken out of context—out of the context of the application in which the loop is found, its vectorization percentage, and its cache behavior on large data arrays—can be misleading. Measures of loop performance, or similar measures of functional kernels, do not provide a valid basis for comparison of vector processing systems. Much measurement of loop performance has unfortunately lost the context in which it started, becoming data without a frame of reference. Published reports of measured loop performance, often without a frame of reference, may carry the implication that users' job performance expectations can legitimately be based on the relationships seen in loop results. Amdahl's Law clearly illustrates the inaccuracy of such an implication.

Peak MFLOPS. The final step in removing the application context from the discussion of performance is to define the instantaneous maximum speed at which the hardware could possibly operate, based on its design. This is called peak MFLOPS. It assumes a time during which operations match perfectly with the capability of the system to overlap multiple functional elements of the vector hardware and operand delivery mechanisms. Peak MFLOPS is basically a value calculated using the system cycle time and a definition of the vector hardware design. Peak MFLOPS bears little relation to sustained application performance, except that the former will be significantly higher than the latter. In the 3090 system with Vector Facility, the vector element of each processor can perform two computations per system cycle when executing a compound vector instruction (several are defined by the architecture), which leads to the conclusion that each processor has a maximum instantaneous execution rate of 108 MFLOPS.[5] However, the peak execution rate is more a description of the product design than a definition of performance.

Benchmark selection and measurement. On the basis of the disadvantages of using MFLOPS terminology and the advantages of using Internal Throughput Rate (ITR) ratios, as discussed above, the latter method was chosen for describing the performance of the 3090 Vector Facility product.

In order to perform product measurements, a set of application programs was collected and analyzed for suitability as laboratory benchmark applications. These application programs were assembled from a variety of sources, including

- Public domain codes
- Customer benchmark jobs
- Customer production or research applications obtained under special agreements
- Licensed application software packages, together with customer problem data or vendor-supplied benchmark data

A large number of applications were collected, from which a benchmark set was established representing the types of workloads expected to be run on the 3090 Vector Facility system. The selected applications became the Scientific/Engineering Application Program (SEAP) set.

Measurements of the jobs in the SEAP application set were performed in a variety of environments, reflecting the diversity implied by a need to measure the system on the basis of application performance. Measurements of scalar versions of the programs

were made on both the IBM 3081 and 3090 computer systems to provide bases of comparison for the vector versions. These also serve as a base for comparing the parallel versions of a subset of the jobs. Both the parallel scalar and serial vector versions of the jobs serve in turn as the bases of comparison for parallel vector measurements. In all cases, measurement was conducted for application programs, consistent with the overall product evaluation framework. Data for the full set of benchmarks were published at the time of product announcement.[6] Additional measurement data are normally published by the IBM marketing divisions as the data become available.

Individual job measurements were made to establish specific application performance levels for each job. Nonparallel jobs made use of only one processor in the Model 200 configuration for the initial runs; they were later run on the Model 180 uniprocessor. Parallel jobs were run using all the processors in the configuration. Table 1 presents a subset of the single-job measurement data as published by the time of writing (December 1985). On the basis of the published parameters of FLOP counts and processor execution time, JOB_MFLOPS can be calculated as job FLOP content divided by processor time.

For capacity measurements, use of multiple processors in the Model 200 and 400 configurations was accomplished by running multiple copies of the jobs to create multijob environments. Multiple processor measurement data yielded low degrees of elongation experienced by vector applications in environments with multiple central processors, thus confirming the value of the 3090 storage hierarchy for multiple-processor vector processing.

Implications of application criteria

In this part of the paper, we have examined the performance of the 3090 Vector Facility product, with the intent of providing insight into how the objective of a system designed for application performance was carried into the design, implementation, and evaluation of the product. The significance of using application performance criteria for design and evaluation is reflected in many of the characteristics of the product and the data with which it was introduced. These implications include

- Departure from the usage of MFLOPS terminology to describe vector processor system performance
- Publication of application performance data with the product announcement
- Vector unit speed balanced for midrange vectorization
- Use of a cache-based storage hierarchy to support vector processing
- Construction of a benchmark measurement set of application programs
- Emphasis on total system throughput capacity as well as on individual job performance
- Measurement of multiple versions of many applications, covering scalar, vector, serial, and parallel

Part II — Vector performance with the 3090 cache storage hierarchy

One of the most significant aspects of the 3090 Vector Facility system design is the use of the 3090 storage hierarchy to supply operands to the vector processing execution element in the processor. The use of the cache to supply operands from storage to the processor may seem obvious to one familiar with IBM large-system processors, for the use of a storage hierarchy with a cache, or high-speed buffer, has been well-established since the late 1960s.[7] Vector processing systems, however, have not been cache-based for vector operand data. Vector operands are transferred directly between the vector execution element and main storage in many other vector processing systems, even those which employ a cache for scalar operands and/or instructions.

The use of a high-speed cache buffer is a proven technique for achieving high performance in large computers. The cache is a small high-speed memory which services most storage requests from the processor, in combination with a large main memory which takes longer to access, but which is needed for only a small percentage of the processor storage requests.[8]

The use of the cache principle for the 3090 Vector Facility is based on several factors:

- The ability of the cache to supply operands to the vector processing execution element at an adequate rate
- The operand addressing patterns of engineering/scientific codes, which produce observed operand reuse and adjacent-data line reuse for operands in the cache
- The advantages that accrue from the use of a common storage hierarchy with the base 3090 system, which include a storage hierarchy designed to support not only uniprocessor configurations, but also two-way and four-way multiprocessing, as well as benefits of configuration and migration flexibility

In this portion of the paper, we explore the performance characteristics and sensitivities of the IBM 3090 Vector Facility, with emphasis on those related to its cache-based structure. In laboratory analysis of application performance, it has been beneficial to decompose a complete application into vectorizable and nonvectorizable portions, and examine the characteristics of each separately. This examination can be approached both analytically and with measurement. Insight into the performance characteristics and sensitivities of the 3090 Vector Facility hardware can be obtained, using simple models, by analyzing those portions of the applications in which vector instructions are executed. The performance of loops from sample applications has been analyzed in order to understand performance sensitivities of coding variations. The vector and scalar execution times for loops of varying characteristics have been analyzed in order to project the overall application speedups which result from improved performance in loops.

Use of the cache storage hierarchy

The use of the 3090 cache for vector operands is *feasible* based on the performance design point of the 3090 Vector Facility. To achieve a vector/scalar loop speedup goal in the vicinity of 4,[1] compared to a 3090 scalar processor that can perform a floating-point operation in three or more cycles,[9] a vector execution element capable of performing one to two floating-point operations per cycle is required. In the 370 Vector Architecture, a vector-register architecture, operations that involve storage require a storage delivery rate of one vector operand per cycle. For data contained in the cache of a processor, the cache has the ability to supply one 64-bit operand per cycle to the vector execution element. Data may be stored at the same rate from the vector element into the cache. This is true for each processor in the configuration in the case of a multiple processor complex—transfers between processor execution units and processor cache can go on simultaneously at the rate of an operand per cycle for each processor in the configuration. The delivery rate of operands coming from main storage as a result of a cache miss

The delivery rate of operands coming from main storage is a key factor in vector performance.

is also a key factor in vector performance. The rate of data transfer is again one double-word operand per cycle during data movement, but main storage access times can lower the effective transfer rate for data not in the cache. To counterbalance the effect of such delays, there are, in the 3090 Vector Facility system, optimizations of main storage operand delivery, which can improve the effective rate at which vector operands move from main storage to the central processor vector-processing execution element.

Optimization of storage operand delivery. For System/370 scalar code, instructions used in floating-point or binary computational code reference at most one operand from storage, and it is generally a double word (eight bytes) or a word (four bytes) in length. In the case of vector instructions, a much larger amount of data is referenced by each instruction. The storage references implicit in a single vector instruction provide information that can be used for optimization of the system design, just as the knowledge that multiple computations must be performed for a vector instruction allows efficient implementation of pipelined vector arithmetic components. If required operands are in the cache, they can be supplied to the vector execution element as fast as they are needed. When required operands are not in the cache, optimization of

delivery may occur based on the knowledge that many operands are needed, a form of pipelining of main storage requests. In the 3090 Vector Facility system, this optimization of storage operand delivery has two modes of operation. They apply to vector operations at Strides 1 and 2 when operands are not in the cache.[10] It is at these strides that the most benefit is obtained, since data are transferred between main storage and the processor in groups of contiguous operands called cache lines.

The optimization of main storage operand delivery has two components. The first is prefetching, effectively an anticipation of the need for operands before the vector element actually needs them. Both optimization modes employ prefetching. The second component of optimized delivery is direct delivery of operands from main storage to the vector element as well as to cache. This component is used when the rate of operand delivery from main storage matches the one per cycle needed by the vector element. The effect of prefetching plus direct operand delivery is to give more stable vector loop performance over variations of data-in-cache and data-not-in-cache situations. For Stride 1 or 2 vector operations which miss the cache, and for which the rate of operand delivery from main storage does not match the rate of processing in the vector element, an optimization mode that uses prefetching only is employed. In these cases, prefetching is initiated, but the data are placed in the cache, and the processor, on behalf of the vector element, obtains the operands from the cache. The paper in this issue by Tucker[9] provides more description of these optimization modes.

Analysis of job performance parameters

The operand delivery ability of the 3090 cache makes it feasible to use the cache to supply the vector execution element with operands, and prefetching reduces the delays associated with missing the cache. However, the cache storage hierarchy concept really works only if the cache can service a significant portion of the operand requirements for the processor, shielding main storage and thereby keeping the demand on main storage within the limits for which it was designed. The ability of the system to achieve multiprocessor vector performance also results from the cache serving as a dedicated high-speed memory for much of the data needs of each processor, allowing shared main storage to serve the multiple processors in the configuration.

The support of the argument that the cache really works and is an effective design for 3090 vector processing lies in the cache performance parameters of the vector applications, specifically in the cache hit ratios achieved by the vector benchmark jobs. Before the data are examined, it is worthwhile first to take a closer look at the concepts of locality of reference and data reuse in the context of vector operations.

Locality of reference is the phenomenon upon which caches work. When a request is made from storage for operand data or instruction data, the probability is good that the same data or closely adjacent data will be referenced in the near future. When the processor needs data that are not in the cache, those data and an adjacent set of data are placed in the cache on the assumption that some of the data will be used in a short time. Reuse means that a data request can be satisfied from the cache because the data were recently used. If the data in question are instruction data, locality of reference and reuse are common because of the iterative loop nature of the code.

For operand data in vector operations, locality of reference is often more than a probability—it may be a certainty. In some cases, the use of adjacent operands may take place within the operation of a single instruction. Operations at Stride 1, for example, use contiguous storage operands. The cache does not provide any "locality of reference" advantage for a Stride 1 instruction where operands must be brought from main storage. In fact, in the previously discussed optimization of delivery, operands are supplied to the vector execution element as they are brought from main memory. The operands are *also* put into the cache to take advantage of the reuse that may occur from references to the same operands in subsequent instructions.

Subsequent instruction locality of reference with reuse of operands in the cache is the type where the cache provides benefit and the type also seen in computational code, and thereby in vectorized loops. It may result from the load-update-store logic of computation upon an array, or from the repeated use of one vector from storage in a series

of computations, while another vector is stepped through an array. Examples of such locality of reference and reuse of operands from the cache are exhibited by the matrix multiplication examples given later in this paper.

For vector operations at higher strides, a set of adjacent data is placed in the cache (if it is not already there) for each operand requested by the vector execution element. In Stride N vector operations resulting from operating on a matrix row, locality of reference often comes from subsequent operation on adjacent rows. This type of reference pattern is exhibited by the Stride N matrix multiplication example later in this paper.

Measured cache hit ratios have shown a decrease when running vectorized applications.

Cache hit ratios. Quantification of the degree of cache usage is often made in terms of cache hit or miss ratios. It is desirable to quantify cache usage for vector applications also, but there are some terminology considerations that should be addressed. The term "cache hit" has a slightly broader meaning than a strict interpretation as an operand that is fetched from the cache. In addition, expectations concerning cache hit ratios need some adjustment based on differences between vector code and scalar code.

Counting the number of operands furnished by the cache is complicated by the nature of vector instructions and the locality of reference inherent in Stride 1 or 2 vector operations. The prior discussion of main storage operand delivery optimizations implies that Stride 1 or 2 operands not in the cache at the beginning of an instruction are in fact supplied to the vector element directly from main storage, and not from the cache. For each cache line of data brought from main storage, there is only one operand "cache miss." For consistency, we assume that every operand should be counted as a storage reference and consequently as a hit or a miss; therefore, the rest of the operands in the storage line must be counted as "hits," even though strictly speaking the operands are not supplied by the cache. However, the rest of the operands in the cache line are available to the vector execution element as a result of the cache hierarchy storage organization of the system, so it is not entirely inaccurate to label them as "cache hits."

The "miss" count is an accurate reflection of the demand placed on main memory for vector operands. The number of cache misses that occur is a parameter that can be modeled, can be measured with laboratory monitors, and can provide a basis for comparison of main storage demand rates between applications. Miss counts may be turned into miss ratios or converted to cache hit ratios as follows:

$$\text{Cache Miss Ratio} = \frac{\text{Miss Count}}{\text{Reference Count}}$$

$$\text{Cache Hit Ratio} = 1 - \text{Cache Miss Ratio}$$

Cache hit ratios may be measured by hardware instrumentation devices connected to a system external interface.[11] Cache hit ratios of 95 to 99 percent plus are commonly measured for workloads on IBM systems. Rules of thumb for cache hit ratios generally hold that a decreasing hit ratio reflects a degradation of performance, since more time is implied for transferring data from main storage to the cache.

Cache hit ratios of vector applications. With the IBM 3090 Vector Facility, measured cache hit ratios on internal test systems have shown a decrease when vectorized applications are running, as compared to the scalar versions of the same programs, although the application performance improved substantially. A key contributor to this change in a measured parameter is the significant decrease in the number of instructions executed by the vector application. Requests from the processor for instructions are valid storage references and are counted as such. Instructions are fetched from the cache to be kept in the processor for decoding and execution. With substantial portions of the scalar

Table 2 Laboratory measurement data

SEAP Application	Scalar Version			Vector Version		
	Miss count ($\times 10^6$)	Reference count ($\times 10^6$)	Hit ratio	Miss count ($\times 10^6$)	Reference count ($\times 10^6$)	Hit ratio
Job 1	508.2	60580	0.9916	492.5	27260	0.9819
Job 2	325.3	38500	0.9915	344.4	24360	0.9858
Job 3	33.83	10130	0.9967	32.28	3368	0.9904

application execution time being spent in loops, the cache hit ratio for instruction requests is high; stated another way, loop instruction references contribute little to a count of cache misses, but account for many references. With a single vector instruction potentially replacing many scalar instructions, the number of instruction references is sharply reduced for the vectorized portions of an application. With the assumption that the number of cache misses for operands does not change significantly, the new hit ratio is calculated on the basis of a lower total number of references. Examples are shown in Table 2, using measured data from three of the laboratory benchmark jobs used for measurement of performance in the October 1985 announcement of the 3090 Vector Facility. For each of the examples in the table, the number of misses stayed relatively constant, whereas the number of references to cache decreased dramatically, mainly because of the decreasing number of instructions.

Although not illustrated by these examples, other changes that affect the number of cache references and/or the number of misses may also occur when code is vectorized. The number of operand references and the order of the operand reference are determined for FORTRAN scalar code mainly by the structure of the FORTRAN statements—loops are not logically switched around by the scalar FORTRAN compiler. For vector code, however, a further divergence relative to scalar parameters may occur when code is optimized to take advantage of holding operands in vector registers rather than storing and loading intermediate results. The number of storage references is reduced, so the miss ratio increases, and the hit ratio again decreases.

The opposite effect may occur when code is optimized, either by the compiler or within a subroutine library, to achieve high cache reuse. In this case, the number of cache misses decreases, while the number of operand references may stay constant, causing the miss ratio to decrease and the hit ratio to improve.

These latter effects are examples of how the order of the operand reference may significantly change for vector operations, relative to scalar code compiled from the same FORTRAN code. Of course, operand reference order must inherently change as code is vectorized. A sequence of vector instructions usually references up to Z elements of one array, then up to Z elements of another array, whereas scalar code typically alternates referencing of operands between arrays. (Z is the vector register section size of the system, 128 in the case of the 3090 Vector Facility.) If the *inner* loop of a nested set of FORTRAN DO-loops is vectorized, the order of reference for operands changes only modulo Z; the same operands are used by both scalar and vector code within the scope of each Z iterations of the scalar loop. For a nested set of DO-loops, however, vectorization need not occur on the innermost loop; when an intermediate or outer loop is vectorized, the order of the operand reference can change significantly from the scalar to the vector code. An example of such vectorization is shown later in the matrix multiplication discussion. When this occurs, the number of cache misses may also decrease, thereby further altering the hit and miss ratio parameters.

Main storage demand of engineering/scientific applications. After much preparatory discussion, we come back to the fundamental question of whether computationally intensive engineering/scientific application programs do exhibit enough data usage characteristics to make a cache system structure advantageous.

Although it is always possible to construct counter-examples, laboratory analysis has shown that many computational programs do exhibit cache usage characteristics, as exemplified by the buffer hit rate data shown in Table 2. Some of the reasons for this locality of reference and operand reuse are illustrated in the following sections of the paper through the examination of matrix multiplication codes.

Individual computational algorithms for particular solution methods, combined with coding style variations, will obviously cause cache usage characteristics to vary. The overall demand on main storage seen in the laboratory benchmark set of programs, however, leads to the conclusion that use of cache is a common factor across many applications.

Analysis of loop performance

Although the interpretation of loop performance measurements is difficult enough that it ought to discourage reliance on such measures, measurements are inevitably made, especially for vector processing systems. Though the use of the cache by the 3090 Vector Facility provides substantial performance benefits, it also introduces further difficulties into the interpretation of loop performance measurements.

The problem arises from the fact that loops, or code kernels, generally represent the logic of a sample computation from an application, but seldom represent the full scope of the data that are computed. Most kernels exercise their logic against small data arrays. Typical measurement techniques repeat the execution of a loop many times in order to average out small variations that may occur in timings. On a cache machine, this method can result in all the operands being in the cache for all but the first pass through the loop. This result produces a best-case measurement, one very close to what we call "primed cache," a condition where all operands required by a set of code are forced into the cache prior to execution.

To overcome the advantage the cache provides, measurements can be made for a single iteration of the loop, starting with none of the required operands in the cache. This method is a worst-case measure, which we refer to as "empty cache." No single interpolation between these two points is an adequate figure of merit, for it would assume a specific proportion of data residing in the cache. Rather, the two measures must be treated as definitions of the performance extremes of the loop logic for a cache-based vector system. Projection of the actual performance of a loop within an application program context must be based on analysis of the cache usage characteristics of the loop operating on the full data arrays.

We have used the approach of bounding best-case and worst-case loop performance in analyzing loop performance on a cache-based system. The matrix multiplication performance is presented in this manner in the following sections. Although this model provides insight into the effects of the cache and the sensitivities of loop performance to cache usage, the cache is only one of several key factors that affect vector loop performance and therefore the vector/scalar ratio for a given loop. Other factors include vector lengths, the number of storage references required in the vector code (whether results can be held in vector registers), and the number of vector instructions needed to perform the computation (whether compound operations can be utilized, for example).

In examining the extremes of best-case and worst-case loop performance *vis-à-vis* the cache, the results are valid only within the context of a particular loop logic. Only the cache hit/miss characteristics are assumed to vary. If another variable changes, for example, by using vector registers to hold intermediate results of computations, loop performance may reach beyond the limit of previous best-case code, although exactly the same computation is being performed. An example is shown in the last matrix multiplication example in the following section of the paper.

Examples using matrix multiplication

Several examples of cache usage characteristics are illustrated here using matrix multiplication algorithms. There are six basic arrangements of the nesting of three FORTRAN DO-loops needed to perform a matrix multiplication.[12] We concentrate mainly on two of these to illustrate the effects of data reference patterns and cache data reuse. All the examples that follow were coded in FORTRAN and compiled with the VS FORTRAN compiler, Version 2, which is the IBM vectorizing compiler. In some cases, the compiler had to be forced to

vectorize less optimally than it would naturally have done. In most of the following examples, we wanted vectorization to occur on the innermost loop to force known vector operand reference patterns that were similar to the order of scalar operand references. The purpose of these examples is not to provide coding guidelines for writing good vectorizable FORTRAN programs; that type of advice may be found in Dubrulle et al.[13] and in the VS FORTRAN Version 2 product publications.

Matrix multiplication with Stride *N* method. As a starting point for examining the performance characteristics of matrix multiplication, we looked at a simple vector inner-product computation, which is at the heart of textbook matrix multiplication for the problem $\mathbf{A} = \mathbf{X} \times \mathbf{Y}$. One element of a result matrix **A** is computed at a time by forming the inner product of a row vector from matrix **X** and a column vector from matrix **Y**. We picked dimensions for the matrices that were not particularly optimal for vector register length, initially letting **X** be of dimension 300 by 200, and **Y** be 200 by 100. Later the inner-product dimension was varied. In order to understand the bounds of performance for such an operation, we looked first at the inner loop, the inner-product computation, represented in FORTRAN as

```
   DO 10 K = 1,200
10 SUM = SUM + X(I,K) * Y(K,J)
```

Figure 3 shows the performance points of this loop for a single (*I*,*J*) combination, in both scalar and vector form, using assumptions of primed cache and empty cache. Primed cache represents minimal (zero) delays due to cache misses, and empty cache represents maximal delays for this particular loop. The intermediate points are interpolated linearly versus time, for it is the amount of delay time that is ultimately important, not the absolute number of cache misses. In fact, in this example, cache miss delay time between the two points is not linearly proportional to the number of cache misses, for there are both Stride 1 and Stride *N* ($N = I$) misses in the cache empty measurement. Miss delays are not constant for all types of cache misses, as discussed previously in the section on optimization of operand delivery. Speed (LOOP_MFLOPS) is a computed parameter, inversely proportional to time, and therefore not linear between the cache-empty and the cache-primed points on the scale of proportional delay time.

These sets of curves form the range of performance for the following matrix multiplication code, which uses inner-product, or Stride *N* method, calculations.

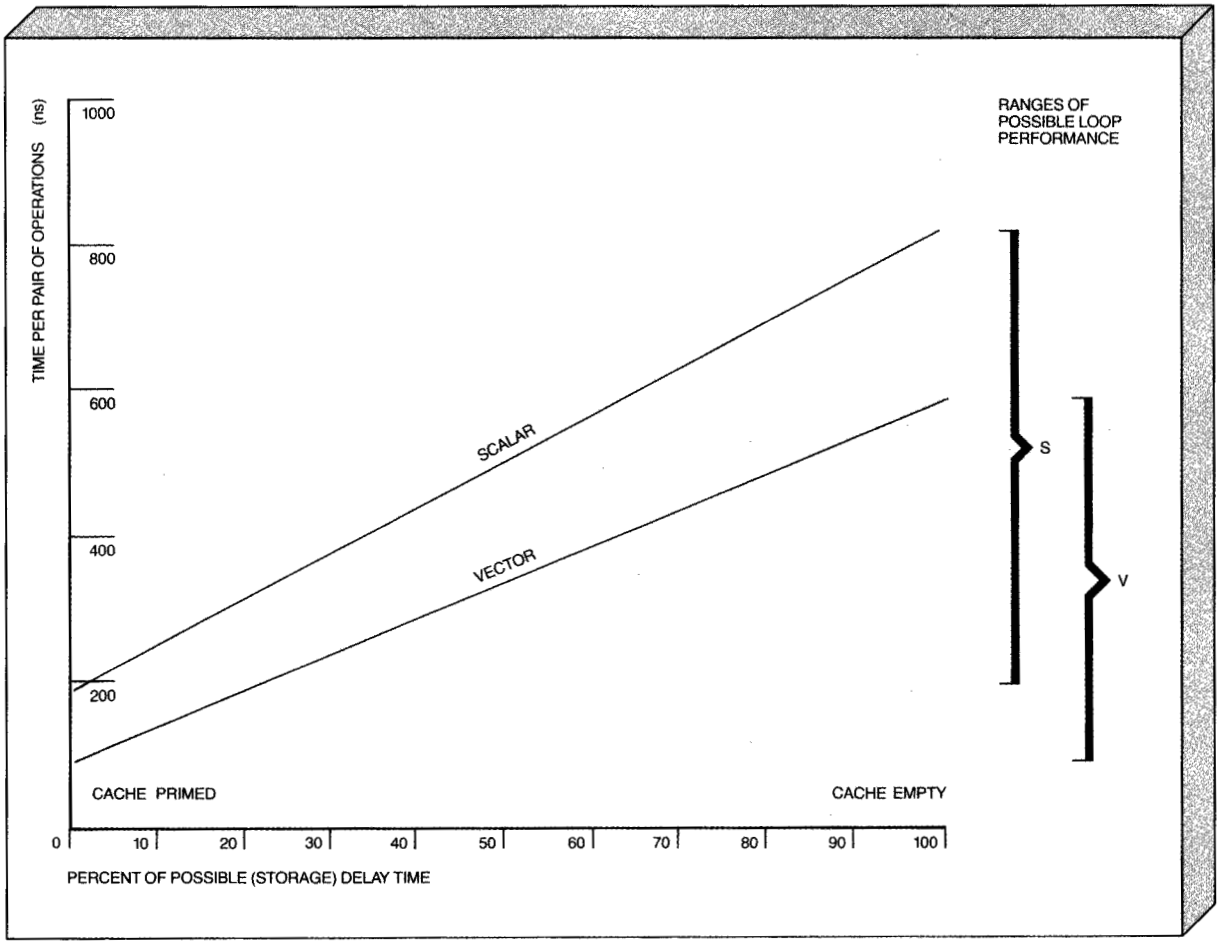
To compute the matrix product $\mathbf{A} = \mathbf{X} \times \mathbf{Y}$, the inner-product logic is used to compute each element of the product matrix **A**. The FORTRAN code is shown in Example 1 in the Appendix. Vectorization was forced to the inner loop, the *K* variable, so that the inner product is computed with vector instructions for the vector test case. (Vectorization on the *I* induction variable would occur without this forcing. See final FORTRAN example.)

The performance of this loop, in both scalar and vector form, is shown in Table 3. The loop times are shown, as well as the normalized figures for the processor time per basic pair of operations—a floating-point multiplication and addition. Placement of these points on the lines in Figure 3 indicates that the loop experiences a small amount of delay due to cache misses, or in other words, use of data from the cache is high.

The next step was to generalize the array sizes and measure across a range of dimensions. We wanted to change only the vector length of the inner-product operation, so that more data would be pulled into the cache by each inner-product computation. We let the **X** array be of dimension 300 by *NDIM*, and the **Y** array be of dimension *NDIM* by 100. One complication is that total loop execution time changes not only from cache effects, but because of more computations taking place for larger arrays. To normalize the data across varying array dimensions, time is expressed as nanoseconds (ns) per multiplication/addition pair in the following examples.

When we allow *NDIM* to vary between 50 and 600, we see a definite effect from the cache. The timings that result for both scalar and vector code are shown in Figure 4, curves 1S and 1V, respectively. Each inner-product computation references *NDIM* operands in the **X** array, at Stride 300, so *NDIM* cache lines are needed for the **X** array operands. When the amount of data in *NDIM* cache lines approaches or exceeds the size of the cache, the

Figure 3 Timing of inner-product computation (200-element) Stride 300 operation



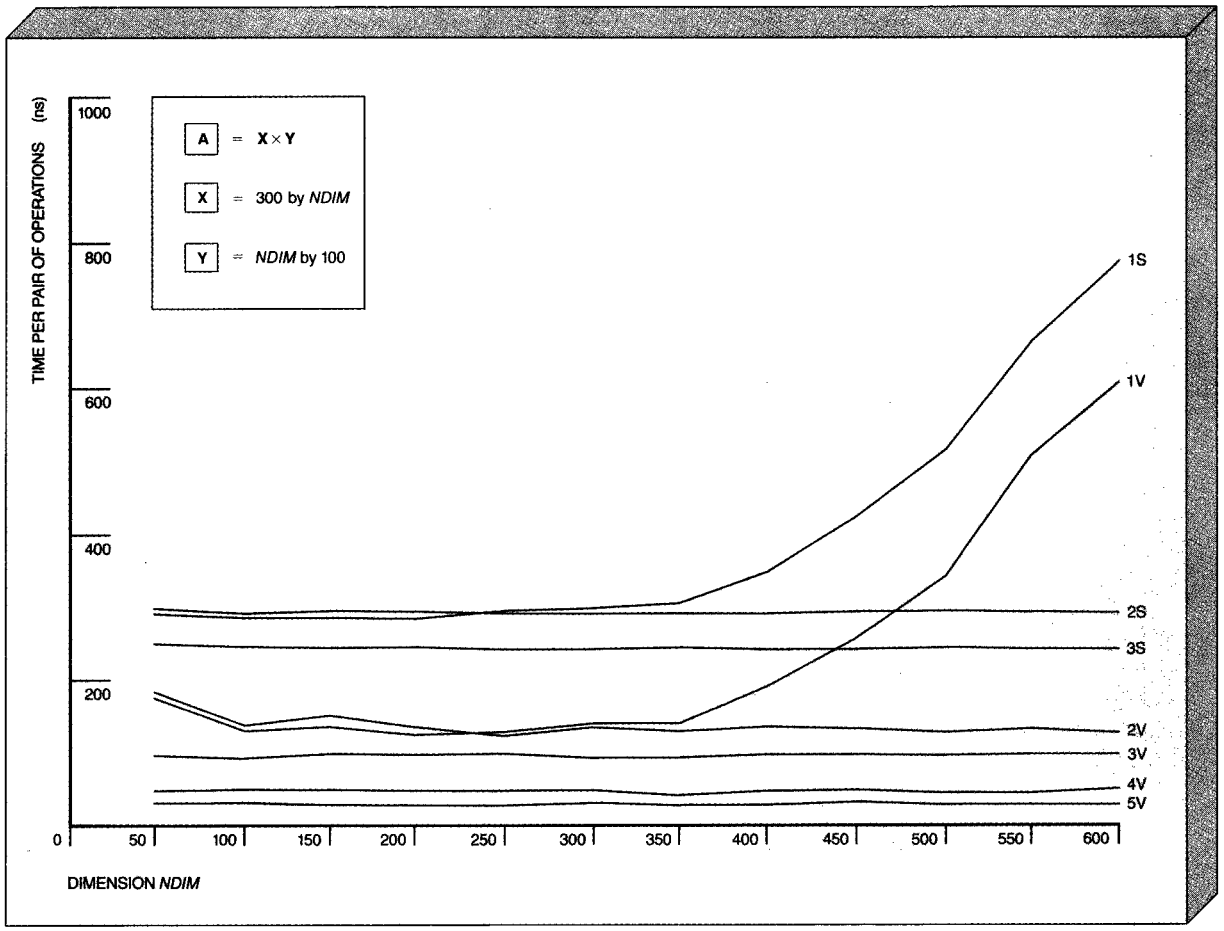
cache begins to "roll" on every inner-product computation. Cache lines fetched for the first operands in the **X** row vector are displaced by later operand references for the same vector, so the next loop iteration that uses the same or an adjacent **X** row vector finds none of the required operands in the cache. This is the reason for the drop-off of performance beginning above *NDIM* = 350 in Figure 4. Data usage from the cache, i.e., "hits," are low, as indicated by the placement of the speed results for *NDIM* ≥ 400 on the curves of Figure 3. Note that both the scalar and vector codes suffer the same degradation, since their operand reference patterns are essentially equivalent. Use of Figure 3 indicates that the cache use of the scalar and vector code is basically equal for each value of *NDIM* tested.

Table 3 Performance of FORTRAN inner product matrix multiplication (300,200) × (200,100)

	Subroutine Time (s)	Time per Mult/Add (ns)
Scalar	1.71	285
Vector	0.70	117

Sectioning the Stride N method. Sectioning is a term used in vector architectures that are register-based. It refers to the implementation of performing a long vector operation using registers of a

Figure 4 Loop timing for matrix multiplication (A = X × Y)

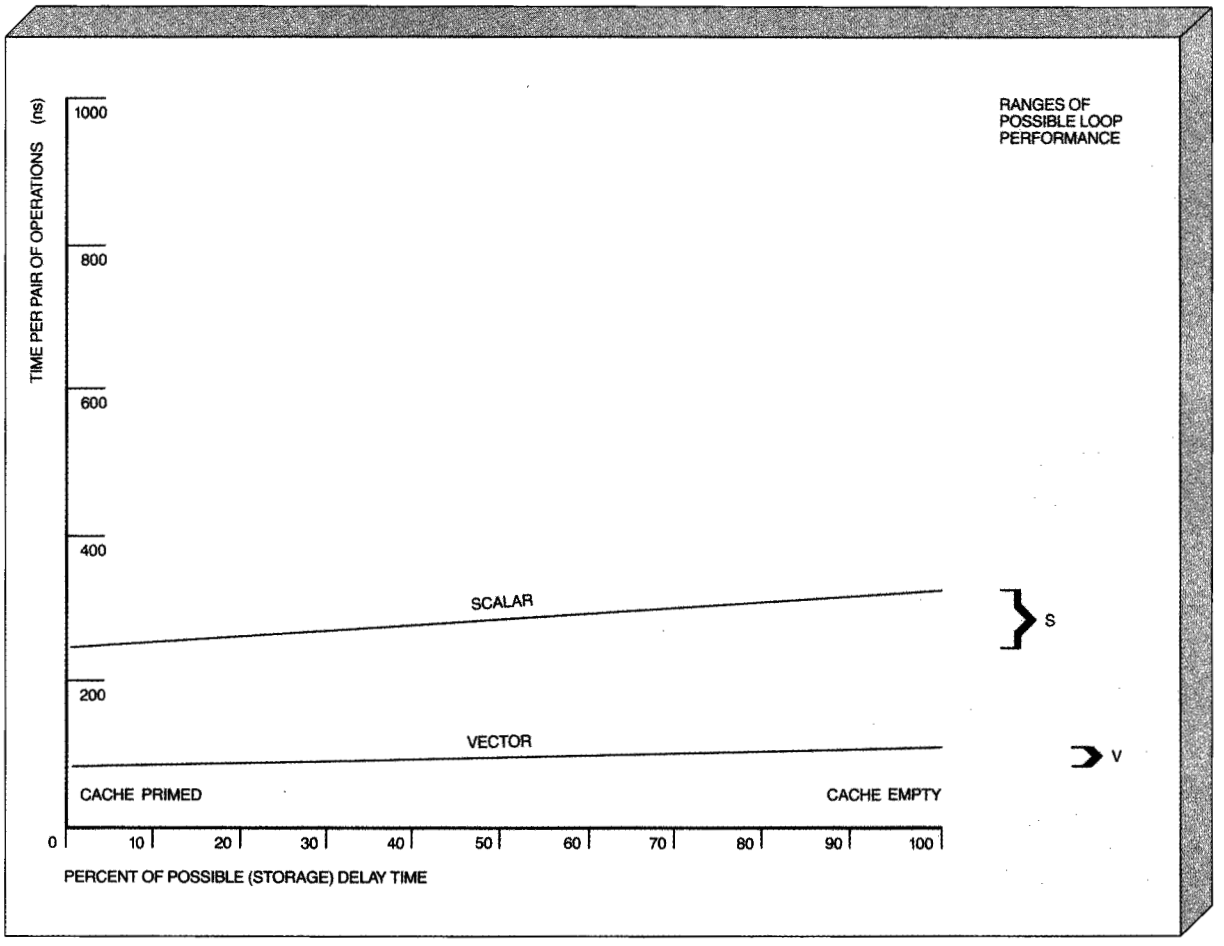


specific length Z; in the case of the 3090 Vector Facility, Z is 128, the number of operands a vector register can contain. A long vector operation is performed Z operands at a time, with a loop that repeats the Z-operand operation on successive "sections" of the long vector, until it is complete. Such loop control is referred to as *sectioning logic*. The length of the vector register is referred to as the *section size*. The vectorizing compiler automatically produces all the code required for sectioning of vectors longer than Z or of length unknown at compilation.

The experiment we performed next on the Stride N matrix multiplication was to externally impose a form of sectioning on the vectors used in the inner-product calculation. The objective was to preserve the cache usage that occurs at lower values of *NDIM*. This preservation was done in the FORTRAN code, so that both scalar and vector codes would again follow similar operand reference patterns. The FORTRAN code, with its artificial sectioning constructs, is shown in Example 2 in the Appendix. Some additional overhead is introduced by the addition of another loop in the nesting, but the operand reference pattern can be controlled. Vectorization again is forced to the innermost loop that performs the inner-product calculation.

The resultant code computes partial results of each element **A**(I,J), but effectively partitions the large **X** and **Y** arrays in a manner that results in cache usage similar to what is seen on the smaller array

Figure 5 Timing of vector * scalar computation (300-element) Stride 1 operation



sizes. The performance results, shown in Figure 4 as curves 2S and 2V, indicate performance that does not degrade. Comparison to Figure 3 indicates that cache usage is high even for large values of *NDIM*.

It should be observed that the effects described here were produced by forcing vectorization on a certain loop induction variable. Scalar loops coded as illustrated would perform as in these experiments. Vectorization of these loops would not normally result in the performance we observed, because vectorization would not take place on the innermost loop. Without the INTEGER * 2 induction variables that force vectorization, the loop used in this example would be vectorized on the loop having the *I* induction variable, resulting in a Stride 1 vector loop with stable performance across values of *NDIM*.

The sensitivity of loop performance to the pattern of operand reference in both scalar and vector code on a cache-based system is clearly illustrated. Vectorization algorithms in the compiler may reorder the reference pattern implied by the FORTRAN code. In cases where this is not possible, and the FORTRAN and scalar code exhibit poor operand reference patterns, the performance of the vector code will similarly be suboptimal.

Matrix multiplication with Stride 1 method. An alternative method for performing the matrix mul-

tiplication function is to access elements in the arrays in a contiguous pattern, that is, columnwise in FORTRAN. The philosophy of coding for contiguous access patterns has been advocated for many years, since it tends to produce stable performance characteristics over large array sizes in virtual storage systems and systems with cache storage hierarchies.[14]

The basic operation is a vector times scalar multiplication, which produces partial results of the product matrix column vectors. In FORTRAN code, the inner loop is

```
   DO 10 I = 1,300
10 A(I,J) = A(I,J) + X(I,K) * Y(K,J)
```

The *K*th column vector of matrix **X** is multiplied by an element from the **Y** matrix, **Y**(*K*,*J*), to produce a partial sum of the *J*th column vector of matrix **A**. The code for a matrix multiplication coded for a contiguous data access pattern is shown in Example 3 in the Appendix.

Figure 5 shows the bounds of a single iteration of the inner-loop operation, which is now a vector times scalar multiplication rather than the inner product seen in the Stride *N* method. Vectorization occurs on the innermost loop without any unnatural forcing, since contiguous operand access is preferred by the vectorization analysis of the compiler. The vector length is 300 and does not change as the *NDIM* dimension of the **X** and **Y** arrays changes. Empty-cache and primed-cache measurements form the range of performance for the following examples. An immediately noticeable difference between Figure 5 and Figure 3 is that the bounds of possible performance for the Stride 1 inner loop are much narrower than for the Stride *N* inner loop. The performance of the Stride 1 method should be more consistent regardless of cache miss characteristics.

Performing the matrix multiplication experiment for values of NDIM ranging from 50 to 600 yielded the results in Figure 4, curves 3S and 3V. As expected from the use of this coding style, the results are stable across the varying array sizes.

Sectioning the Stride 1 method. Although the performance of the Stride 1 method is stable and shows good improvement over the scalar operation, further improvement may still be obtained by altering the sectioning logic, as was done for the Stride *N* inner-product method. It might appear that little advantage is to be gained since the bounds of performance are rather tight, as shown in Figure 5. This is the basic reason this approach is preferred. However, when the sectioning logic is altered for the Stride 1 method, the partial results that are calculated are not scalar partial inner products but sections of the product matrix columns which are summed to form the result matrix column. When this change is made for the vector code, the partial results can be kept in a vector register, and loading and storing of intermediate results to the result matrix can be eliminated. The FORTRAN code is shown in Example 4 in the Appendix.[15] The inner loop in the compiled code has no vector LOADs or STOREs to the **A** matrix and has just one compound vector instruction performing the multiply/add operation of a vector times scalar product.

Measurement results for this vector code are shown in Figure 4, curve 4V. A scalar counterpart is not applicable, since the result was obtained by exploiting the vector register to hold a temporary vector variable. The results of this measurement do not fall in the range of the loop performance of the original loop as shown in Figure 5 because the inner-loop logic has changed. Though exactly the same computation has been performed, the ratio of storage references per computation has been altered by the change to the sectioning logic.

Matrix multiplication summary. The progression of examples in this portion of the paper has illustrated the role the cache plays in the overall performance profile of the 3090 Vector Facility system. The cache storage may present some pitfalls, but they have not been introduced by the addition of vector processing. It also presents opportunities for optimization. In the Engineering/Scientific Subroutine Library (ESSL) software product, coding of matrix functions has taken advantage of the types of optimizations discussed in this paper and further combined them to achieve optimal vector length, vector register usage, and cache data reuse. The performance of the ESSL vector matrix multiplication function illustrates the point, as shown in Figure 4, curve 5V. In this measurement, ESSL was called to perform the same set of matrix multiplications used in the previous examples.

Table 4 Performance of vector matrix multiplication (300,500) × (500,100)

	LOOP_MFLOPS
Vectorized FORTRAN	21
Vectorized FORTRAN	46
ESSL	68

Table 4 presents the data measured during these investigations in the form of LOOP_MFLOPS speeds for the vectorized forms of matrix multiplication on a pair of matrices of size (300,500) and (500,100). The data shown are for the two cases in which the VS FORTRAN Compiler was allowed to choose the method of vectorization, plus the ESSL measurement. In each of these situations, the previous results show that performance is essentially stable over a range of dimension represented by the *NDIM* variable.

These loop speeds represent vector/scalar ratios of approximately 2.5 to 8 times. The improvement an application would experience is also a function of the vectorizability of a particular application, as discussed in Part I of this paper.

Summary and conclusions

In the last section, we have examined the performance of the 3090 from the perspective of its use of the cache storage hierarchy for vector operand delivery. We have discussed how the cache storage hierarchy supports the delivery of operands to the vector processing execution element in each processor, at the rate required for vector pipeline execution. In the most commonly observed reference patterns, operands that are not in the cache are delivered in an optimized mode to minimize the delay effects of main storage access. It has been shown that the introduction of vector instructions produces results in traditional "cache hit ratio" terms that run contrary to rule-of-thumb expectations. Care must be taken in applying metrics to an analysis of the effectiveness of the cache hierarchy.

Given these qualifications, however, both the analysis of cache miss ratios and resultant main storage demand, as well as the empirical data of application benchmark measurements, support the conviction that the cache hierarchy does indeed provide the data needed by the processor to achieve the vector/scalar speedup goals of the 3090 Vector Facility system.

Finally, we have examined several matrix multiplication codes in detail to illustrate the sensitivities that exist in a cache-based design. These also serve to illustrate the opportunities that exist in such a system structure for effective software/hardware optimization, since neither the unconstrained FORTRAN vectorization nor the ESSL loop function exhibits the sensitivity illustrated.

In its innovative approach to vector processing, building on the strengths associated with IBM's state-of-the-art large-systems hardware and software products, the 3090 represents a major step in the maturation of systems designs in support of engineering/scientific data processing requirements.

Acknowledgments

The departments and individuals who planned, designed, developed, and tested the IBM 3090 Vector Facility system, both hardware and software, are too numerous to acknowledge individually, but without them we would not have products to analyze and measure. We particularly recognize the work of the applications analysis and performance measurement departments, which have provided the foundation of our understanding of the performance of the system. The applications analysis groups, directed by A. L. Lim and B. D. Rudin, analyzed, selected, and migrated the vector benchmark applications used in characterizing the performance of the system. Product measurement has provided us not only with the application performance data, but also the cache characteristics and analysis of cache usage sensitivities using matrix multiplication codes.

Appendix: FORTRAN code segments used in matrix multiplication examples

Example 1. This example shows matrix multiplication using inner-product logic (fixed array dimensions).

```
      DIMENSION A(300,100), X(300,200), Y(200,100)
      INTEGER*2 I,J
      DOUBLE PRECISION A,X,Y,SUM
      DO 20 J = 1,100
      DO 20 I = 1,300
      SUM = 0.0D0
      DO 10 K = 1,200
   10 SUM = SUM + X(I,K) * Y(K,J)
   20 A(I,J) = SUM
```

Example 2. This example depicts matrix multiplication using inner-product logic and variable array dimension (externally imposed sectioning logic).

```
      DIMENSION A(300,100), X(300,NDIM), Y(NDIM,100)
      INTEGER*2 I,J
      DOUBLE PRECISION A,X,Y,SUM
      MSECSZ = 128
      DO 20 J = 1,100
C     PRODUCT COLUMN SECTION MUST BE
C     INITIALIZED TO 0
      DO 5 I = 1,300
    5 A(I,J) = 0.0D0
C     VECTOR LENGTH IS NDIM, SECTIONING ON I
      K = NDIM
      DO 20 KSECT = 0, (K - 1)/MSECSZ
C     DO LOOP INDEXED BY I INSIDE SECTIONING
C     LOOP
      KK1 = KSECT*MSECSZ + 1
      KK2 = MIN(K, ((KSECT + 1)*MSECSZ))
      DO 20 I = 1,300
      SUM = 0.0D0
      DO 10 KK = KK1,KK2
   10 SUM = SUM + X(I,KK) * Y(KK,J)
   20 A(I,J) = A(I,J) + SUM
```

Example 3. In this example, matrix multiplication uses vector * scalar logic (variable array dimension).

```
      DIMENSION A(300,100), X(300,NDIM), Y(NDIM,100)
      DOUBLE PRECISION A,X,Y
      DO 10 J = 1,100
      DO 20 I = 1,300
   20 A(I,J) = 0.0D0
      DO 10 K = 1,NDIM
      DO 10 I = 1,300
   10 A(I,J) = A(I,J) + X(I,K) * Y(K,J)
```

Example 4. This example illustrates matrix multiplication with optimum vectorized performance.

```
      DIMENSION A(300,100), X(300,NDIM), Y(NDIM,100)
      DOUBLE PRECISION A,X,Y,SUM
      DO 20 J = 1,100
      DO 20 I = 1,300
      SUM = 0.0D0
      DO 10 K = 1,NDIM
   10 SUM = SUM + Y(K,J) * X(I,K)
   20 A(I,J) = SUM
```

Cited references and notes

1. D. H. Gibson, D. W. Rain, and H. F. Walsh, "Engineering and scientific processing on the IBM 3090," *IBM Systems Journal* **25**, No. 1, 36–50 (1986, this issue).
2. G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceedings* **30** (1967).
3. K. Radecki, *Introduction to Processor Performance Evaluation*, IBM Washington Systems Center Technical Bulletin, GG66-0232, IBM Corporation (February 1986); available through IBM branch offices.
4. Y. Singh, G. M. King, and J. W. Anderson, "IBM 3090 performance: A balanced system approach," *IBM Systems Journal* **25**, No. 1, 20–35 (1986, this issue).
5. Calculation of Peak MFLOPS is

   Peak_MFLOPS = (number of operations per cycle)/(machine cycle time)

   for the 3090 Vector Facility = 2/(18.5 ns) = 108 Peak_MFLOPS (one processor).
6. *IBM 3090 System Summary—Engineering/Scientific*, IBM Information Systems Group Product Announcement 182-120 (October 1, 1985); available through IBM branch offices.
7. C. J. Conti, D. H. Gibson, and S. H. Pitkowski, "Structural aspects of the System/360 Model 85; Part I, General organization," *IBM Systems Journal* **7**, No. 1, 2–14 (1968).
8. C. J. Conti, "Concepts of buffer storage," *Computer Group News*, 2 (March 1969).
9. S. G. Tucker, "The IBM 3090 system: An overview," *IBM Systems Journal* **25**, No. 1, 4–19 (1986, this issue).
10. *Stride* refers to the distance in memory between operands referenced in an array. Contiguous data are referred to as Stride 1. Reference to the row of an $N \times M$ array in FORTRAN uses operands in memory that are N elements apart, and is termed Stride N.
11. Hardware monitors may be attached to IBM 3090 and IBM 3081 processors via an external interface available by a customized order. Signals are provided at the interface that can allow an attached device to monitor events such as cache misses.
12. J. Dongarra, F. Gustavson, and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Review* **26**, No. 1 (January 1984).
13. A. A. Dubrulle, R. G. Scarborough, and H. G. Kolsky, *How to Write Good Vectorizable FORTRAN*, IBM Palo Alto Scientific Center Technical Report, G320-3478, IBM Corporation (September 1985); available through IBM branch offices.

14. A. A. Dubrulle, *The Design of Matrix Algorithms for FORTRAN and Virtual Storage*, IBM Palo Alto Scientific Center Technical Report, G320-3396, IBM Corporation (November 1979); available through IBM branch offices.
15. In FORTRAN, this is accomplished in a somewhat oblique manner by returning to the coding of the inner-product loop nesting and letting the compiler choose how to vectorize. Vectorization occurs on the *I* induction variable (the Stride 1 direction), and a vector temporary is introduced because of the use of the temporary variable SUM in the FORTRAN code. This vector temporary variable is kept in a vector register as an accumulator of a section of the A column vector, resulting in an inner loop without vector LOAD and STORE operations.

Ronald S. Clark *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Mr. Clark is currently manager of System Performance in the Scientific/Engineering Processor Products development function. He was involved in the design and system testing of the initial releases of MVS, after which he spent two years on assignment to the International Systems Center in Hursley, England, providing large-system support to MVS customers in IBM's World Trade Europe/Middle East/Africa Corporation. Following that assignment, he worked in the Washington Systems Center large-system product support group. His next assignment was with the National Program Office in Norwalk, CT, providing national account marketing support to the General Electric Company. Upon returning to the development lab, he held several positions in software development for scientific/engineering products before assuming his current position. Mr. Clark joined IBM in 1968. He received a Sc.B. in applied mathematics from Brown University in 1968 and an M.S. in computer science from Rensselaer Polytechnic Institute in 1972.

Troy L. Wilson *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Mr. Wilson is currently manager of System Management and Performance. In this position, he is responsible for providing the analysis necessary to understand the processing behavior of scientific applications on IBM's large systems, for supporting the engineering projects to achieve high performance, and for generating guidance in applications requirements. His prior work in IBM included numerous management positions in software development, in group staff activities, and in performance technologies. He founded the corporate Internal Technical Liaison Committee on Performance, established and directed the Tokyo System Evaluation Laboratory, and served as director of performance and of reliability and serviceability for the Information Systems and Technology Group staff. Mr. Wilson is a graduate of the University of Arkansas.

Reprint Order No. G321-5262.