REFERENCES

[1] A. Avizienis and J. C. Laprie, "Dependable computing: From concept to design diversity," *Proc. IEEE*, vol. 74, pp. 629-638, May 1986.

[2] K. G. Shin and Y. H. Lee, "Error detection process—Model, design, and impact on computer performance," *IEEE Trans. Comput.*, vol. C-33, pp. 529-540, June 1984.

[3] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE*, vol. 66, pp. 1221-1240, Oct. 1978.

[4] R. K. Iyer, S. E. Butner, and E. J. McCluskey, "A statistical failure/load relationship: Results of a multicomputer study," *IEEE Trans. Comput.*, vol. C-31, pp. 697-706, July 1982.

[5] R. K. Iyer and D. J. Rosetti, "Effect of system workload on operating system reliability: A study on IBM 3081," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1438-1448, Dec. 1985.

[6] X. Castillo and D. P. Siewiorek, "Workload, performance, and reliability of digital computing systems," in *Proc. 11th Annu. Int. Symp. Fault-Tolerant Computing*, 1981, pp. 84-89.

[7] J. G. McGough and F. L. Swern, "Measurement of fault latency in a digital avionic mini processor," Tech. Rep. 3651, NASA Contractor Rep., Jan. 1983.

[8] R. Chillarege and R. K. Iyer, "Fault latency in the memory—An experimental study on VAX 11/780," in *Proc. 16th Annu. Int. Symp. Fault-Tolerant Computing*, 1986, pp. 258-263.

[9] M. H. Woodbury and K. G. Shin, "Workload effects on fault latency for real-time computing systems," in *Proc. Real-Time Systems Symp.*, Dec. 1987, pp. 188-197.

[10] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer: Volume I FTMP principles of operation," NASA Contractor Rep., Tech. Rep. 166071, May 1983.

[11] J. H. Lala and T. B. Smith, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer: Volume II FTMP software," NASA Contractor Rep., Tech. Rep. 166072, May 1983.

[12] K. G. Shin and Y. H. Lee, "Measurement and application of fault latency," *IEEE Trans. Comput.*, vol. C-35, pp. 370-375, Apr. 1986.

[13] F. Feather, "Validation of a fault-tolerant multiprocessor: Baseline experiments and workload implementation," Master's thesis, Dep. ECE, Carnegie-Mellon Univ., Pittsburgh, PA, 1984.

[14] J. H. Lala and T. B. Smith, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer: Volume III FTMP test and evaluation," NASA Contractor Rep., Tech. Rep. 166073, May 1983.

[15] R. E. Barlow *et al.*, *Statistical Inference Under Order Restrictions*. New York: Wiley, 1972.

[16] D. A. Schoenfeld, "Confidence bounds for normal means under order restrictions, with application to dose-response curves, toxicology experiments, and low-dose extrapolation," *J. Amer. Stat. Assoc.*, vol. 81, pp. 186-195, Mar. 1986.

[17] E. L. Ellis and R. W. Butler, "Estimating the distribution of fault latency in a digital processor," NASA Tech. Memo., Tech. Rep. 100521, Nov. 1987.

Experimentally Characterizing the Behavior of Multiprocessor Memory Systems: A Case Study

K. GALLIVAN, D. GANNON, W. JALBY, A. MALONY, AND H. WIJSHOFF

***Abstract*—Although architectural improvements in memory organization of multiprocessor systems can increase effective data bandwidth, the actual performance achieved is highly dependent upon the characteristics of the memory address streams; e.g., the data access rate, and the temporal and spatial distributions. Accurately quantifying the performance behavior of a multiprocessor memory system across a broad range of algorithmic parameters is crucial if users (and restructuring compilers) are to achieve high-performance codes. In this paper, we demonstrate how the behavior of a cache-based multivector processor memory system can be systematically characterized and its performance experimentally correlated with key features of the address stream. The approach is based on the definition of a family of parameterized kernels used to explore specific aspects of the memory system's performance. The empirical results from this kernel suite provide the data from which architectural or algorithmic characteristics can be studied. The results of applying the approach to an Alliant FX/8 are presented.**

***Index Terms*—Characterization, memory systems, multiprocessor, performance.**

Manuscript received April 3, 1989; revised October 2, 1989. Recommended by R. K. Iyer. This work was supported by the National Science Foundation under Grant US NSF MIP-8410110, the Department of Energy under Grant US DOE-DE-FGO2-85ER25001, the Air Force Office of Scientific Research under Grants AFOSR-85-0211 and AFOSR 86-0147, and an IBM Donation.

The authors are with the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 8932234.

I. INTRODUCTION

For shared memory multiprocessors, access to the common memory is one of the key limiting factors in performance. One of the most attractive solutions to this problem is the use of a hierarchical memory system. This approach reduces the apparent memory latency as well as the memory contention. However, the performance is far from uniform and depends not only upon the characteristics of the memory hierarchy itself, but also on the characteristics of the address streams and the interaction between the two. This implies that the relationship of code characteristics to machine characteristics must be taken into account. For example, knowing the precise penalty in terms of number of cycles for a cache miss is not enough to understand the effectiveness of a given cache organization. We need to determine precisely, as a function of the temporal and spatial distribution of the requests, the data access rate and to try to correlate observed behavior with code characteristics. This requires a systematic investigation of the parameter space (code characteristics).

Classically, two main approaches are used for performance analysis: analytical or experimental (simulation or measurement). The first solution is extremely powerful in the sense that it allows the analytical correlation of the performance with organizational parameters. The drawback is that, in order to be tractable, they typically require a drastic simplification of the hardware model and of the memory request stream. For example, queueing theory-based models assume a randomly distributed (both in time and space) memory request stream. This is particularly disturbing when modeling scientific codes on vector machines because these codes tend to exhibit very regular data access patterns and the vector instructions used to implement the codes must exploit, and thereby emphasize, this regularity in the spatial and temporal distribution of the requests. Experimental performance analysis (simulation or measurement) provides more accurate information in the sense that it is possible to take into account more details of the hardware and code characteristics. The drawback of such a solution is its experimental nature which limits the number of codes analyzed and generally does not provide any methodology for extrapolating the performance of an arbitrary code from the performance of the benchmark codes. Furthermore, even when using very simple benchmarks, there is no general method for correlating code characteristics with the performance observed.

Our primary goal in this paper is to present a systematic methodology for investigating and correlating the performance of a cache-based memory system (in our case, the Alliant FX/8) in terms of architectural parameters and code characteristics typical of scientific numerical computations. The resulting characterization can be used for performance prediction of scientific codes. Furthermore, the design of the empirical kernels upon which the meth-

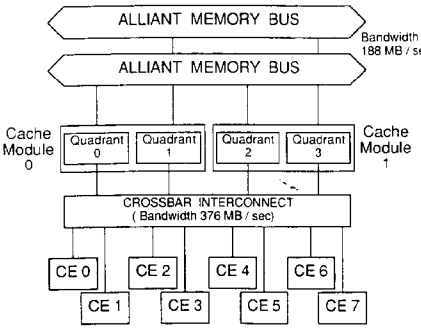


Fig. 1. Alliant FX/8 architecture.

odology is based allows an explanation of observed and predicted performance and is therefore suited to aid in performance tuning via an interactive restructuring compiler (See [9] for details of performance tuning usage on the Alliant FX/8.)

The remainder of this paper is organized in four sections. In Section II, the architecture of the Alliant FX/8 is described. Section III contains the motivation and description of the **LOAD/STORE** kernel hierarchy. Experimental results are presented and analyzed for an Alliant FX/8 in Section IV and conclusions are given in Section V.

II. The Target Architecture: Alliant FX/8

The Alliant FX/8 (see Fig. 1) machine consists of up to eight pipelined computational elements (CE's) connected by a concurrency control bus which is used as a fast synchronization facility. This mechanism enables the CE's to cooperate in performing the computations of a single program unit with small granularity, e.g., a Fortran loop. A special set of instructions support the use of the synchronization hardware. This enhances greatly the performance of the system in parallel mode and makes its behavior more predictable. When a portion of code requires the eight processors, e.g., a parallel loop, the iterations of the loop are directly allocated to the processors via an hardware self-scheduling mechanism without involving the operating system scheduler.

The memory system of the Alliant FX/8 combines parallel data access with a hierarchical memory structure. It is organized in three levels, a large main memory, a cache shared by the CE's, and scalar and vector registers private to each CE. The vector registers are 32 double precision (64-bit) words long and can be operated on via the vector processing capabilities of each CE. (Throughout the discussion below a word is taken as 64 bits.) The 16K-word write-back cache is organized into four banks and connected to the eight CE's via a crossbar switch. The cache can service up to eight simultaneous accesses per cycle (170 ns). The cache is direct-mapped, meaning each memory location can be cached in exactly one cache location, and uses a cache block (quantum of exchange between the memory and the cache) of 4 words. The cache and the four-way interleaved main memory are connected through the main memory bus which is able to deliver up to four words per cycle. Therefore, the peak bandwidth between main memory and CE's is 23.5 Mwords/s which is half of the 47 Mwords/s possible between the cache and the CE's.

The internal organization and behavior of the CE's is rather simple. Extensions, such as vector and concurrency instructions, have been made to the basic instruction set of the 68000. The vector instruction set contains compound instructions such as multiply-add (this corresponds to the chaining of a load from memory with a multiply followed by an addition). However, at most one operand can come from outside the CE, i.e., in cache or memory, due to the fact that there is only one port connecting the CE to the cache. All of the vector instructions of interest here involving one operand external to the CE, use the same cache request mechanism. In particular they request the cache or memory at the same rate. The only difference in their behavior is due to differing startup times—the time at which the first request is issued.

III. Load/Store Hierarchy

A. Motivation

In this section, we describe the types of code segments that will be used to generate the code characteristics of interest and the general principles of the characterization techniques used.

The techniques for analyzing the performance of a given memory organization depends upon the field of application. For example, in the area of benchmarks, the Lawrence Livermore Loops and LINPACK are used specifically to test the system performance for scientific computing. Similarly, we will focus on analyzing the performance of basic multiply-nested Fortran DO LOOPS. For the sake of simplicity, we will assume that the arrays in the loop body are referenced through linear subscripts. Indirect addressing can be handled via similar techniques [13]. Finally, we will assume that the innermost loop is vectorizable and therefore parallelizable. As a consequence of the last assumption, a CE will spend most of its time executing vector instructions.

The choice of such structures is motivated by the fact that they account for a large part of typical numerical programs execution time and because the hypothesis on the subscripts and conditional statements implies that the sequence of memory addresses accessed during program execution is extremely regular and can be analyzed statically at compile time by using techniques similar to the ones used for vectorization [7], [11].

The Alliant vector instructions can be grouped in 2 categories: *internal* (register–register) where all operands are contained in vector and scalar registers; and *external* (register–memory) where one operand comes from or goes to memory. Most of the vector instructions in each class have similar timing characteristics typically differing only in startup costs. Since the internal instructions do not depend on conditions external to the CE, their timings are essentially deterministic and their contribution to the total execution time can be derived in a straightforward manner from the hardware specifications. The case of the timings of the external instructions is much more complex. Theoretically, they could be determined from hardware specifications. In practice, such a technique is difficult to apply directly due to the fact these timings are very dependent upon runtime conditions such as contention (either due to the previous requests of the same processors or other processors) and the exact location of the operand (memory or cache). In such cases performance characterization can be divided into two subproblems: determining the runtime conditions; and determining the performance under such conditions.

For the purpose of investigating memory behavior, the key parameter to be varied is the global memory request stream of all the processors. Unfortunately, the domain of this parameter is prohibitively large. We need to a systematic technique of exploring this space. In particular, we need to provide a parameterized mechanism by which the memory system can be probed to determine potential bottlenecks and, conversely, situations where high data transfers can be achieved and maintained. Additionally, it is advantageous to have the characterization facilitate the prediction of performance of a given loop.

The approach taken here is based on the definition of a family of parameterized kernels (these kernels, in fact, correspond to the choice of a set of address request streams). The choice of these kernels was guided by three major constraints. First, the kernels must be able to mimic the access patterns of the loop structures of interest using different parameter combinations. Second, the kernels should be elementary enough to allow the study of the impact of only one characteristic of the request stream at a time, e.g., varying the hit ratio but keeping the temporal distribution of the request constant. Third, we must be able to *decompose* a given loop in terms of these elementary kernels and then reconstruct the per-

formance of the loop using the performance data obtained for the kernels. The remainder of this section describes the kernel hierarchy in more detail and discusses the way the above requirements are satisfied. Since, in this paper, we are demonstrating the techniques by applying them to an Alliant FX/8, the discussion of the use of the kernel hierarchy is based on the characteristics of this machine.

B. Load/Store Kernels

The basic kernels in the hierarchy are a simple vector load or store. On the Alliant FX/8, these simple operations use both the concurrent and vector processing capabilities of the machine. Their basic function is loading (storing) a single vector of consecutive elements from (to) the memory system. The intent of these kernels is to determine the bandwidths of reads and writes that each component of the memory system is able to sustain and the conditions influencing these bandwidths. They are parameterized by the length of the vector (n), the location of the vector (cache or memory), the number of processors (p), and the partitioning and scheduling of the operations across the processors.

The structure of the basic kernel is shown in Table I. The kernel has the form of a concurrent loop construct with the body of the loop iteration being a set of vector move instructions.

At the top of the loop, the processors enter a concurrent processing mode (this has little overhead on the Alliant FX/8 due to its hardware concurrency support). The preamble code is executed once per processor and consists mostly of address computations for the load and store streams; in general, it can perform any initializing computation. The remainder of the kernel consists of code which is performed for each of the iterations of the concurrent loop. For the simplest load and store primitives, each iteration corresponds to the processing of a single block of length b of the n elements of the vector. The iteration code consists of address computations followed by a loop of **nop** instructions and a sequence of **vmove** instructions. The vector loop is required since b may be larger than the vector register size.

Several scheduling strategies are possible; in this paper we will restrict ourselves to the most interesting: self-scheduling with contiguous blocks (for a detailed analysis of the different variants of scheduling see [8]). In self-scheduling, the vector of length n is broken into blocks of b contiguous elements. The original vector loop is decomposed into two loops. The outermost is performed in parallel across the CE's while the innermost (operation on a block of b elements) is executed in vector mode within each CE. On the Alliant FX/8, the dispatching of the blocks to the processors is done by an hardware self-scheduling mechanism. That is, the blocks are logically arranged in a queue and as soon as a processor has finished operating on a block, it accesses the queue to get another block or goes idle if the blocks are exhausted. By changing the block size, the effect of synchronization can be analyzed as well as load balancing issues.

This basic family is extended by including the variation of three other code characteristics: the temporal distribution of requests, the distribution of requests in the hierarchy (hit ratio), and the spatial distribution of the requests. The use and purpose of these parameters are as follows:

Temporal Distribution: The main purpose of this parameter is to study the interaction between a burst of requests, which corresponds to an external vector instruction, and a subsequent interval of several cycles without memory requests, which corresponds to the execution of internal vector instructions, address computation, or the different start-up times of the various vector instruction that could generate the memory request. The variation of the density of memory requests made by each processor is accomplished by altering the number of null instructions (informally called *NOPS* below) making up the **nop** sequence in the kernel.

Hit Ratio: The purpose of this parameter is to study the effect of the distribution of requests between the two memory levels. The hit ratio can be experimentally varied by manipulating the **vmove** sequence in each iteration of the kernel. The manipulation consists of inserting, after the single **vmove** instruction which loads (stores) a portion of vector, a variable number k of **vmove** instructions referencing exactly the same locations. The first vector reference generates a miss in the cache (this can be controlled) while the subsequent k references cause hits. Notice that this parameter describes the behavior of the memory system under variations of temporal locality with respect to the memory hierarchy while suppressing spatial locality variations of the requests (since the same locations just brought in to cache are repeatedly accessed). Furthermore, insight is gained into the behavior of the cache/main memory combination when simultaneously addressed.

TABLE I
THE BASIC LOAD/STORE PRIMITIVE TEMPLATE

```
start concurrent execution
    preamble code executed once per processor
loop_body:
    initial computations of iteration body
vector_loop:
    nop sequence
    vmove sequence
    jump to vector_loop if work left in block
    get next concurrent iteration index and jump to loop_body
resume sequential operation after iterations are exhausted
```

Spatial Distribution: Manipulating the stride of the vector access in the basic kernel can be used to characterize the effect on performance of the mapping strategy used to assign elements of an array to the banks in the two levels of the hierarchy. By manipulating the vector length all references can be kept in cache and the effectiveness of the mapping of elements to cache banks and bank conflict resolution strategy can be characterized. Similarly, by working with vector lengths large enough to flush cache on each pass through the vector, the effectiveness of the interleaving of the main memory system is probed. The stride can also be varied to manipulate the cache banks servicing the misses if knowledge of the address mapping is exploited. Finally, the stride can be used to identify the effect of the main-memory-to-cache mapping on performance. Characterizing this effect can be particularly important when implementing high-performance kernels such as the BLAS3 on machines, like the Alliant FX/8, which use a direct mapping [10].

The above variations of the primitive kernels form the basis of a set of experiments used to characterize the behavior of the Alliant FX/8's memory system. From this base, other levels of the **LOAD/STORE** hierarchy are built by manipulating two additional aspects of the primitive: the number of vector address streams and the dimension of the structured data element accessed. The former is accomplished by manipulating the **vmove** sequence within the vector loop in the primitive template. This sequence is modified to be a series of **vmove** instructions to and from memory for several address streams. The three most basic multiple address stream kernels found in vector computations are: **load-load**, **load-store**, and **load-load-store**. The variation of the dimension of the structured variable accessed corresponds to determining the performance of the memory system when access pieces of structures more complex than simple vectors. For example, one of the most important class of algorithms in scientific computations is numerical linear algebra which must access efficiently submatrices as well as vectors. Accesses of such objects can be modeled by a simple modification to the basic kernel which uses a hierarchy of strides to access a various portions of a contiguous section of memory.

The family of kernels is built in such a way that any parameter can be varied while the others remain constant. If all the points in the parameter space were to be tested, this would result in an overwhelming number of experiments. In practice, we proceed in a hierarchical manner. The parameter space is first explored in a coarse way, by making large steps in parameter variations. After coarse

performance is observed, refinement techniques are used by stepping with smaller increments. The number of levels produced in the hierarchy and the subset of kernels used is determined by a knowledge of the architecture, the intended use of the information and the consistency (or lack thereof) found in the experiments as levels are added (see the comments on commutativity below).

IV. Experimental Results

In this section, after a brief description of the experimental setup, we summarize the major findings of the load-store experiments (for a more detailed study see [8]). We implemented a set of load/store kernels in assembly language parameterized by the parameters described in the previous section. All the loads and stores instructions operated on double precision data (64 bits). The alignment did not significantly influence performance due to the fact that the address streams were fairly long. Each experimental value was obtained by running each kernel five times, eliminating the best and the worst experimental values, and taking the arithmetic average of the three remaining values as a final number. Confidence intervals were computed for each set of five values and found to be satisfactory. All experimental values showed less than 5 percent variation between the extremes.

The resolution of the timer used was 10 ms. In order to increase timing accuracy, each code was enclosed in a repetition loop such that the interval between two consecutive calls to the timer was at least 0.1 s. However, on a cache based system, such a technique has the following drawback. If the total working set of the loop fits in the cache (i.e., the working set is smaller than the cache size), the first timing iteration loads the cache and the subsequent iterations will operate from cache. This explains the general shape of our experimental curves. They have three distinct regions which depend on the relationship of the vector length n and the size of the cache, 16K double precision words. These regions are as follows.

Cache Region: $0 \leq n \leq 16K$ for the load and store kernels (respectively $0 \leq n \leq 8K$ for the load-load and load-store kernels and $0 \leq n \leq 5.3K$ for the load-load store kernels). In this region, all the operands are in cache. It should be noted that this region is large enough so that we reach a speed very close to the asymptotic rate.

Fall Off Region: $16K \leq n \leq 32K$ for the load and store kernels (respectively, $8K \leq n \leq 16K$ for the load-load and load-store kernels and $5.3K \leq n \leq 10.6K$ for the load-load-store kernels). In this region we observe a very strange phenomenon due to the direct mapped cache. Portions of the vector overlap in the cache forcing the elements to be fetched from memory from one iteration to the next. The other parts of the vector remain in the cache. In this region the hit ratio is decreasing from 1 to 0.25.

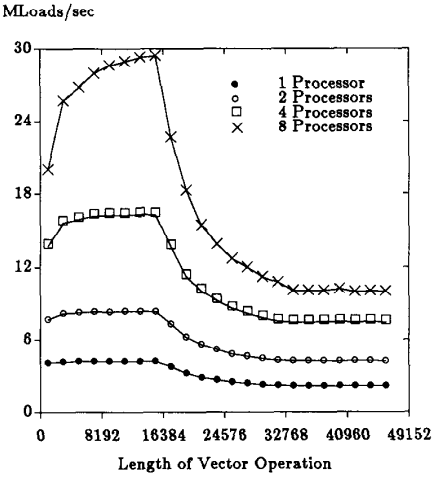
Memory Region: $32K \leq n$ for the load and store kernels (respectively, $16K \leq n$ for the load-load and load-store kernels and $10.6K \leq n$ load-load-store kernels). In this case the cache is flushed each iteration and the operands come from memory.

A. Load Kernel

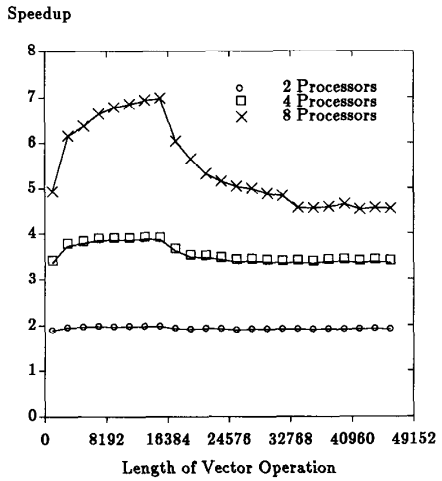
In these experiments, a simple kernel reading a vector of length n was timed [Fig. 2(a),(b)].

On one processor, the performance is very regular. The memory region presents small drop in performance not due to the saturation of the memory bandwidth but rather due to the limit in the pipelined request-issue of the CE. Delays occur because the processor can only have two misses outstanding at any time. Notice the two processor case is perfectly scaled from the one processor (speedup of 2) performance. For four processors, where the contention at the cache level is negligible (speedup of 4), the memory contention becomes more important (speedup of 3.3).

For eight processors, cache contention begins to affect performance. In the cache region, 30 Megaloads/s may seem disappointing compared to the potential peak of 43.5 Mwords/s. However, there are two reasons for such a situation. One reason is that the



(a)



(b)

Fig. 2. (a) Load performance. (b) Load speedup. (Block size 128.)

overhead associated with each block (synchronization, address computations) is nonnegligible. This decreases the intensity of requests from each processor. By using larger blocks (which decreases the relative importance of the overhead compared to the sequence of accesses) or by unrolling the loop (same effect), speeds up to 38 Megaload/s can be achieved. Still, in all cases the speedup is around 7.3.

Secondly, due to the cyclic nature of bank referencing, we would expect to have first an initial transient phase with some bank conflicts followed by a steady state where processors are synchronized with each other without any additional conflicts. Such a phenomenon would occur if each processor had an infinitely long sequence of requests regularly spaced in time. In reality, due to the splitting in blocks and vector operations, the sequence of requests are not necessarily regularly spaced in time. Some operations between vector requests could encounter conflicts. This acts to disrupt the synchronized conflict-free referencing between the processors. By looking at a trace of processor activity generated with a logical analyzer, we noticed that the processors indeed entered a phase-sync with respect to their referencing behavior after a short transient "conflict" phase at the beginning of a vector instruction. However, we also observed that random access could easily disrupt

the phase synchronization. Thus, instead of observing only a single transient phase followed by a long steady state, we saw a succession of transient phases (with conflicts) followed by short steady state periods.

At the memory level, the contention is far more severe (speed of 10 Megaloads/s and speedup around 5 while the peak speed of the system is supposedly 23.5 MWords/s). This is mainly caused by the fact that the memory bus cannot satisfy the requested bandwidth from the processors. The contention of the memory system combined with the request mechanism is responsible for most of the performance loss.

B. Store Kernel

The kernel used in these experiments is the same as the load kernel except that a vector write is performed instead of a vector read. A close look at the results [Fig. 3(a),(b)] indicates that the behavior of the store is very similar to the load. Only two main differences are worth noticing. First, the performance in cache is slightly lower (around 5 percent) due to the fact that the startup of a vector store is slightly higher. Second, the performance from memory is similar to the load for one and two processors, but four and eight processors show significantly less performance: 6.5 Megastores per second versus 10.5 Megaloads per second in the eight processor case. This drop is due to the limitations in bandwidth at the memory level (bandwidth for sequential writes is 80 percent of the bandwidth for sequential reads) and to the combined effect of miss-on-write and write-back mechanisms; although the write-back mechanism defers the penalty for write. When the cache is full and new blocks are to be loaded, part of the memory bus bandwidth will be consumed by these delayed writes. From the memory point of view, each block is accessed two times: first when it is written into the cache and then later when it is written back to memory.

The speedup curves show very clearly this phenomenon. The contention at the memory with four processors is already severe (speedup less than 2.9), because each write requires two transactions on the bus. So, four processors writing are almost equivalent to eight processors reading.

C. Temporal Distribution

For these experiments, we modified the basic load kernel by inserting a fixed number of NOPS after each vector instruction. We varied the number of NOPS between 0 and 80 by increment of 8.

Introducing the NOPS has two effects: first, the time for an iteration is lengthened, and second, the memory reference rate is decreased thereby decreasing memory contention. For the one processor case [Fig. 4(a)], the first effect is predominant because there is no contention. Accessing 32 elements from cache costs approximately 35 cycles. Adding 80 cycles (80 NOPS) after each vector instruction approximately triples the cost and we observe a corresponding decrease in performance of around three. From memory, accessing 32 elements cost around 80 cycles, so adding 80 NOPS should roughly divide the performance by two, which correlates exactly with the experimental results.

The eight processor case is more complex to study due to the variations in contention [Fig. 4(b)]. From cache, the contention was not very important (speedup around 7) so increasing NOPS does not have a significant impact on the speedup. The main effect is seen in lengthening the time and we observe a corresponding drop in performance.

From memory the situation is more complex. Notice with 0 NOPS the memory bandwidth is saturated: the total aggregate bandwidth requested by the processors exceeds the bandwidth of the memory system. By adding NOPS, we are effectively decreasing the bandwidth requested by the processors. Therefore, we observe the performance remaining constant while the bandwidth requested by the processors exceeds the capacity of the memory system until some point where the number of NOPS introduced results in the requested bandwidth exactly matching that of the memory. From that point the bandwidth requested becomes less than the capacity of the memory system, and we observe a decrease in performance.

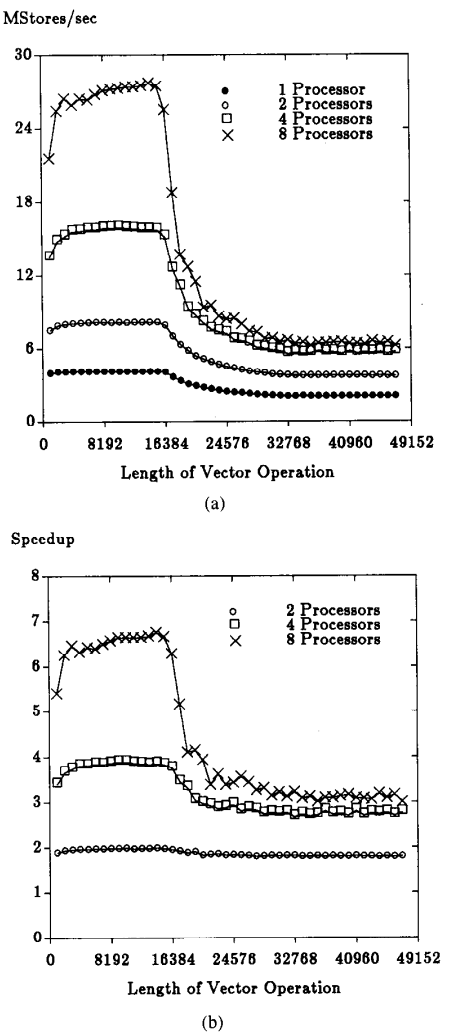


Fig. 3. (a) Store performance. (b) Store speedup. (Block size 128.)

D. Vector Hits

For these experiments, each vector instruction was followed by a fixed number of vector instructions (called vector hits) with the same starting address (i.e., each vector instruction accessed the exact same set of addresses). For example, the 0 vector hit kernel corresponds to the standard load. The k vector hit kernel corresponds to a sequence of $(k + 1)$ vector instructions referencing exactly the same set of addresses.

For the cache region, such experiments allow us to study the effect of unrolling; increasing k is going to decrease the impact of the loop overhead on the performance. For the memory region, they give information about the interaction between the references to the two memory levels and the relationship between miss-ratio and performance: for the k vector hits kernel, the miss-ratio is $8/(k + 1) * 32$.

In cache, the effect of unrolling is very clear, it allows to reach 38 Mwords/second [Fig. 5(a),(b)]. From memory, we observe a phenomenon similar to the one observed with the NOPS; increas-

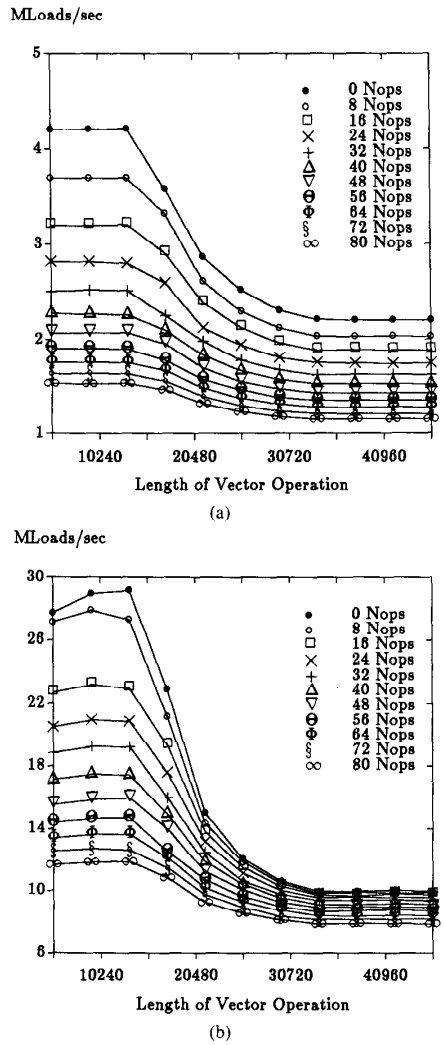


Fig. 4. (a) Effects of NOP's on one processor. (b) Effect of NOP's on eight processors. (Load kernel, block size 128.)

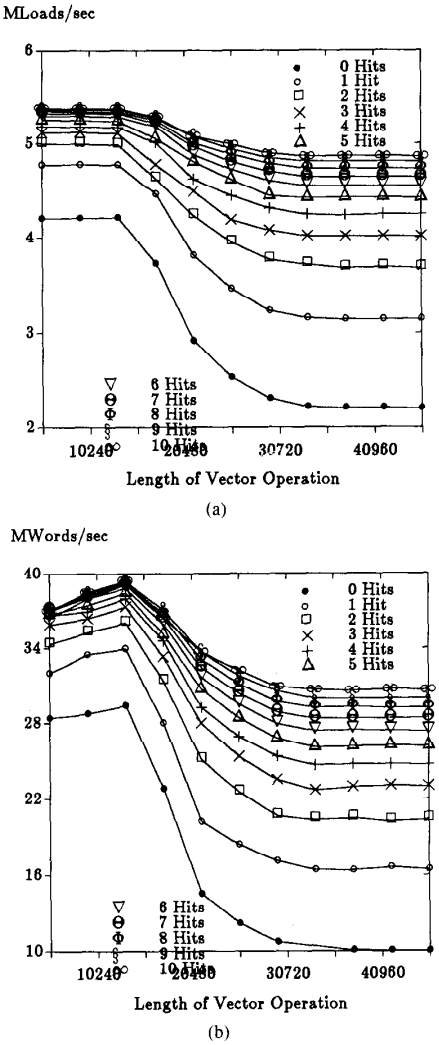


Fig. 5. (a) Effect of cache miss ratio on one processor. (b) Effect of cache miss ratio on eight processors. (Load kernel, block size 128.)

ing the number of hits decreases the intensity of memory requests alleviating the contention problem at the memory level. Correspondingly, the speedup increases regularly from 5 to 6.3. The difference compared to the NOPS case is that hits and misses are contending with each other at the cache level. It is worthwhile to note that the difference in performance between 0 vector hits and 10 vector hits reaches a factor of three while the advertised peak bandwidth between the two levels of memory differ only by a factor of two.

E. Strides

The kernel used corresponds to reading a vector of n elements and varying the stride.

In cache, the main effect of strides is to partition the requests among the four banks. Stride 1 and stride 5 sweep all four banks and the performance is linear in the number of processors [Fig. 6(a)]. Due to an Alliant-specific data skewing scheme, stride 2 still goes across all four banks and the performance is very similar to the stride 1 case. However, stride 4 concentrates the request on two banks effectively halving the potential performance. Four processors can saturate the bandwidth in this case. Stride 8 concentrates all requests on one bank and bandwidth saturation can be reached with two processors.

From memory, the effect of strides is complicated by the cache line size [Fig. 6(b)]. Any stride greater than four will imply a miss for each access. Except for strides which are multiples of eight (missing occurs on a single bank), all strides greater than four will achieve bandwidth saturation with only four processors. Surprisingly, the performance of the stride-4 load and the stride-5 load are about the same. This indicates that missing on two banks gives the same performance as missing on all four cache banks. To be precise, in the case of stride 4, we are missing on two cache banks located on different cache boards. Recall that the four cache banks of the Alliant are arranged on two cache boards, with each cache board having one port to the bus. More detailed experiments, where missing occurred on only two cache banks located on the same board, indicated that the speed obtained was about the same as the speeds stated above, implying that the memory bus cannot fully support two cache boards requesting at their maximum rate.

F. Commutativity

Remarkable observations can be made on the Alliant FX/8 with respect to the interchange of different load/store kernels, and/or

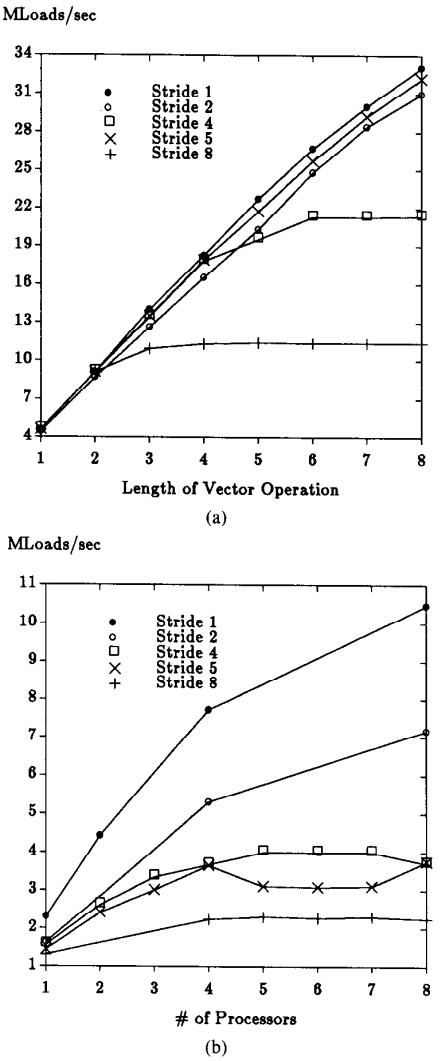


Fig. 6. (a) Effect of strides in cache. (b) Effect of stride in memory. (Load kernel.)

different insertions of **NOP's**. By this we mean that the Alliant FX/8 satisfies structural and temporal commutativity. Structural commutativity asserts that the order of the loads and stores in a kernel composed of multiple loads and stores is independent of the kernel's performance; e.g., load-load-store shows similar performance to store-load-load-load, load-store-load-load, and load-load-store-load. Structural commutativity was observed on the Alliant FX/8 for all complex kernels discussed above. Spatial commutativity relates to the effect that nonmemory reference operations have on kernel performance with respect to their location relative to the memory reference operations.

Not only is the characterization simplified when these commutativity properties hold, it is also of crucial importance when predicting codes. In the latter case, the database of empirical results can be considerably compressed to contain only data for representatives of kernel equivalence classes. For more detailed discussion of the commutativity properties and their implication on predicting the performance of DO-loop structures see [9].

V. Conclusions

The first conclusion is that the system has smooth behavior relative to varying the different parameters and the trends can be easily predicted qualitatively. From a quantitative point of view, the situation is more complex: we must distinguish two cases depending on whether the system has one of its components at a saturation point or not.

In the latter case (for example, the one and two processor experiments) we were able to predict the performance of the load/store kernels to within 10 percent error using simple cycle counting techniques, i.e., inspecting the assembly code and computing the total execution by summing up the timings of elementary instruction, augmented by certain empirically determined quantities. Such a simple technique proved to be very powerful and accurate even in the fall-off region. For this region, the situation was a bit more complex due to the fact that there is a mix of hits and misses. We used a straightforward model of the direct-mapped cache in order to determine how accesses will be directed to cache versus memory. The resulting prediction gave good results.

In the former case (saturation of one component, four and eight processor cases) the situation is different. First, the numbers provided by the manufacturer (peak bandwidth) were inadequate. Furthermore, knowledge of the protocol used in handling exchanges between the different levels of the hierarchy did not help to determine quantitatively the loss in performance due to saturation. This was due to the complexity of memory transactions from the presence of multiple address streams.

In such cases, controlled experimentation of the load/store type is crucial. This is particularly true when the experimental results can be deduced from more elementary data via relatively simple combinations. For example, the behavior of complex load patterns could be accurately approximated by averaging corresponding component load kernel results. We observed similar properties for the more complex parameter variations such as the hit ratio series of experiments.

The experimental characterization as described in this paper is also used successfully for predicting performance for very simple kernels such as the Lawrence Livermore Loops [9]. We stress the point that even without being integrated into a performance prediction tool, the results of building an experimental database can be extremely useful for guiding the choice of a restructurer between several possible code optimization. For example determining the sensitivity of the system to the hit ratio, indicates quantitatively how much effort has to be devoted for increasing data locality.

The basic conclusion is that for the Alliant FX/8, and most likely for similar architectures, careful combination of local analytical models and empirical observation can characterize the performance of the memory system.

References

[1] W. Abu-Sufah and A. Kwok, "Performance prediction tools for Cedar: A multiprocessor supercomputer," in *Proc. 12th Int. Symp. Computer Architecture*, 1985, pp. 406–413.

[2] J. Andrews, D. Lavery, and R. Iyer, "A measurement based study of cache contention in a shared memory multiprocessor," Univ. Illinois, CSL Rep., 1987.

[3] D. Bailey, "Vector computer memory bank contention," *IEEE Trans. Comput.*, vol. C-36, no. 3, pp. 293–298, Mar. 1987.

[4] I. Bucher and M. Simmons, "A close look at vector performance of register-to-register vector computers and a new model," in *Proc. 1987 ACM SIGMETRICS*, 1987, pp. 39–45.

[5] D. Calahan, "Performance evaluation of static and dynamic memory systems on the Cray-2," in *Proc. 1988 Int. Conf. Supercomputing*, ACM Press, 1988, pp. 519–524.

[6] T. Cheung and J. Smith, "An analysis of the Cray X-MP memory system," in *Proc. Int. Conf. Parallel Processing*, Aug. 1984, pp. 494–505.

[7] K. Gallivan, D. Gannon, and W. Jalby, "On the problem of optimizing data transfers for complex memory systems," in *Proc. 1988 Int. Conf. Supercomputing*, ACM Press, 1988, pp. 238–253.

[8] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Behavoral characterization of multiprocessor memory systems: A case study," Univ. Illinois at Urbana-Champaign, CSRD Rep. 808, Oct. 1988.

[9] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff, "Performance prediction of loop constructs on multiprocessor hierarchical memory systems," in *Proc. 1989 Int. Conf. Supercomputing*, ACM Press, 1989, pp. 433-442.
[10] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "The impact of hierarchical memory systems on linear algebra algorithm design," *Int. J. Supercomput. Applicat.*, vol. 2, no. 1, pp. 12-48, Spring 1988.
[11] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel Distributed Computing*, pp. 587-616, Oct. 1988.
[12] G. Phister and A. Norton, "Hot spot contention and combining in multistage interconnection networks," in *Proc. Int. Conf. Parallel Processing*, 1985, pp. 790-797.
[13] Y. Saad and H. Wijshoff, "A benchmark package for sparse matrix computations," in *Proc. Int. Conf. Supercomputing*, 1988, pp. 500-509.

Debugging Effort Estimation Using Software Metrics

NARASIMHAIAH GORLA, ALAN C. BENANDER, AND BARBARA A. BENANDER

Abstract—**Measurements of 23 style characteristics, and the program metrics LOC, $V(g)$, VARS, and PARS were collected from student Cobol programs by a program analyzer. These measurements, together with debugging time (syntax and logic) data, were analyzed using several statistical procedures of SAS, including linear, quadratic, and multiple regressions. Some of the characteristics shown to significantly correlate with debug time are GOTO usage, structuring of the IF-ELSE construct, level 88 item usage, paragraph invocation pattern, and data name length. Among the observed characteristic measures which are associated with lowest debug times are: 17 percent blank lines in the Data Division, 12 percent blank lines in the Procedure Division, and 13 character long data items. A *d*ebugging effort *es*timator, DEST, was developed to estimate debug times. This estimator, a quadratic function of nine characteristics, has a coefficient of multiple determination (R^2) of 0.7551 with the total debug time (significance level 0.0001). None of the software metrics LOC, $V(g)$, VARS, and PARS has r^2 values greater than 0.3 when regressed with total debug time. The variables of DEST, when regressed with debug times from various subsets of the programs stratified by LOC, $V(g)$, and student GPA, had high R^2 values.**

Index Terms—**Cobol, debugging, regression analysis, software metrics, statistical analysis, style analyzers.**

I. Introduction

It has been estimated that in a typical programming project, debugging accounts for over 50 percent of the time spent on the project [10]. Many factors influence debugging time, including programming style, program length, program complexity, program volume, and programmer competence. Among these factors, however, a programmer has most control over programming style. This experiment studies the relationship between programming style and debugging time, while taking the other factors into consideration.

Research has been conducted investigating the components of good programming style and their effect on various programming tasks [2], [4], [6]-[9], [11]. Program style analyzers [2], [8] have been written for the purpose of assigning a numerical value to the style of a program, based on some set of program style characteristics. However, little experimental research using software metrics for Cobol has been done, despite its prominence as the most widely used language in industry [3].

In the experiment described in this paper, raw data measurements of 23 style characteristics were collected. Also data on program correctness and program debugging time (syntax and logic) were obtained. Linear, quadratic, and multiple regression analyses were performed using SAS (statistical analysis system) procedures to obtain relationships between style characteristics and debugging times. Using the results of these analyses, desirable ranges (those associated with lowest debugging times) for some of the style characteristics are suggested, and a debugging effort estimator, DEST, based on selected style characteristics is developed. Other well known software metrics, as well as student GPA are shown to have lower correlations with debugging time than DEST.

Manuscript received September 6, 1989; revised October 2, 1989. Recommended by R. K. Iyer.

The authors are with the Department of Computer Science, Cleveland State University, Cleveland, OH 44115.

IEEE Log Number 8932235.

II. Experimental Methodology

A total of 311 student Cobol programs from 5 intermediate Cobol classes were saved on secondary storage following the quarter in which the classes were conducted. All of the classes dealt with files using Cobol. All students were computer and information science (CIS) majors. Most of the students were in their second year (a few of them were in their third year) of the CIS program. The students had very similar programming backgrounds, having had an introductory Pascal course and a data structures course in Pascal prior to this files course in Cobol.

In each class, about five programs were collected. The types of programs were similar, involving creation and updating of (in the following order) sequential files, relative files, and indexed sequential files. The level of difficulty, as judged from student feedback, was neither increasing nor decreasing. All students were encouraged to use structured programming techniques.

In order to maintain consistency in grading of programs among instructors, it was made clear to each participating instructor that a program should be regarded as "correct" (for purposes of this experiment) if and only if the output was complete and accurate. These were the two criteria for program correctness used by all instructors in this experiment.

Also, for each lab assignment, each student was asked to complete a form indicating their syntax and logic debugging times. Syntax debugging time was defined to be time spent in correcting errors before a successful compilation. Logic debugging time was defined to be time spent in correcting errors after successful compilation. To encourage honesty in their responses, students used code names on their forms and were assured that in no way would their responses affect their grade. All programs were run on an IBM 3081 under OS/VS1 in a batch environment.

A PL/I program was written to obtain raw data measurements of 23 style characteristics found in Cobol programs (see Fig. 1). In addition, the analyzer produced measures for LOC (lines of code), $V(g)$ (complexity), VARS (number of data items), and PARS (number of paragraphs).

The style characteristic measurements from correct programs which had corresponding debugging data information (a total of 57) were used as input to SAS procedures for statistical analyses. Linear and quadratic regression analyses were performed through the GLM Procedure of SAS to correlate the style characteristic measurements and the logic, syntax, and total debugging times for these programs.

Also, to identify the most significant style characteristics which were used in forming the debugging effort estimator, DEST, the stepwise regression procedure with quadratic terms was used. The stepwise procedure was also used to obtain the relative contributions toward debugging time of identification and environment divisions, data divisions, and procedure divisions.