Behavioral Characterization of Multiprocessor Memory Systems: A Case Study

Kyle Gallivan Dennis Gannon William Jalby
Allen Malony Harry Wijshoff

Center for Supercomputing Research and Development
University of Illinois at Urbana Champaign, Urbana, Illinois 61801

Abstract

The speed and efficiency of the memory system is a key limiting factor in the performance of supercomputers. Consequently, one of the major concerns when developing a high-performance code, either manually or automatically, is determining and characterizing the influence of the memory system on performance in terms of algorithmic parameters. Unfortunately, the performance data available to an algorithm designer such as various benchmarks and, occasionally, manufacturer-supplied information, e.g. instruction timings and architecture component characteristics, are rarely sufficient for this task. In this paper, we discuss a systematic methodology for probing the performance characteristics of a memory system via a hierarchy of data-movement kernels. We present and analyze the results obtained by such a methodology on a cache-based multi-vector processor (Alliant FX/8). Finally, we indicate how these experimental results can be used for predicting the performance of simple Fortran codes by a combination of empirical observations, architectural models and analytical techniques.

1 Introduction

The speed and efficiency of the memory system is often the main limiting factor of supercomputer performance. Several architectural solutions are currently used to improve the data access rate: pipelining and parallelization of the memory system, distributed memory systems, and hierarchical memory systems. Although these solutions result in a substantial performance improvement, they make the task of code performance optimization more difficult. Due to their increased complexity, such memory organizations are hard to model, making performance prediction difficult. Furthermore, the performance of such systems is highly dependent upon the characteristics of address streams. For example, pipelining or parallelization works well if successive memory references are uniformly distributed among the memory banks; on the other hand, vector strides which are a multiple of the degree of interleaving or of the number of memory modules can result in dismal performance. Similarly, hierarchical memory systems will be efficient only if the amount of data locality in an algorithm is sufficient and exploitable.

Most studies of memory performance have been hardware oriented; the goal was to study the impact on performance of specific architecture characteristics. Two main approaches were used: queuing theory-based modeling which can produce results at the price of oversimplifying the code structure (especially the pattern of its memory pattern references); and simulation-based modeling which provides more accurate information using a set of benchmark codes [1]. Neither approach provides enough insight in the interaction between the code and the memory system in order to predict and tune performance. Recently, more attention has been paid to the study of characteristics of the code and their interaction with the memory: impact of registers [3], effects of strides [2,4,5] and hot spot contention [8]. Their interest is more in the analysis of worst case or pathological behavior and again the focus is too limited for predicting performance for more general codes.

Our primary goal in this paper is to develop a sys-

This work was supported by the National Science Foundation under grant US NSF MIP-8410110, the Department of Energy under grant US DOE-DE-FG02-85ER25001, the Air Force Office of Scientific Research under grants AFOSR-85-0211 and AFOSR 86-0147, and an IBM Donation.

tematic methodology for investigating the interaction between a cache based memory system and the various address streams generated in loop constructs typically found in scientific numerical computations. We first define a family of kernels parameterized for systematically exploring the behavior of the system (Section 1). The experimental set up used in our study is described in Section 2. In Section 3, performance data from the Alliant FX/8 are presented and analyzed. Finally, we indicate how the data produced by these kernels can be used for performance prediction and how these techniques can be integrated into a static or dynamic performance tuning system for cache based multiprocessor systems.

2 Target architecture and load/store kernels

In this section, we describe the global framework in which our performance study is integrated as well as the main guidelines for developing the family of kernels. The target architecture and the kernel set are described.

2.1 Motivations of load/store kernels

The goal of our work is to propose a methodology for characterizing and predicting the performance of basic Fortran multiply-nested loops. For simplicity, we assume that the loop body contains neither conditional statements nor call statements and that the arrays are referenced through linear subscript functions. The presence of conditional statements prevents accurate prediction; in such cases we can only predict a range of performance. The presence of calls can be easily overcome by in-line subroutine expansion. The choice of multiply-nested loops is motivated first by the fact that they account for a large part of numerical programs and second because due to the nature of subscripts, the sequence of memory locations accessed is extremely regular and can be analyzed by techniques similar to the ones used for automatic vectorization.

As noted above, it has been observed that the memory system is often the architectural feature which determines performance of a code. As a result, our approach was to develop a set of elementary parameterized kernels. 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 provide the ability of studying the impact of only one code characteristic at a time. Third, we must be able to "decompose' a given loop under study in terms of these elementary kernels and then reconstruct the performance of the loop using the performance data obtained for the kernels. The following describes the **Load/Store** method and discuss its ability to satisfy these requirements. Since, in this paper, we are restricting the application of the methodology to an Alliant FX/8, the discussion of the motivation of the use of the load/store model is based on the characteristics of this machine. The generalization to other machines is considered in turn.

2.2 Target architecture

The memory system of the Alliant FX/8 combines parallel data access with a hierarchical memory structure (see Figure 1). It is organized in three levels, a large main memory built of modules delivering data by blocks (32 bytes), a cache organized as four independent modules (32 Kbytes each) simultaneously accessible by eight computational elements (CE's) through a crossbar, and scalar and vector registers private to each CE. The cache is direct mapped and the main memory is updated by a write back policy. The vector registers are 32 double precision words long and can be operated on via the vector processing capabilities of each CE. Since we are interested in multiply-nested Fortran loops which exploit both the concurrent and vector processing capabilities of the Alliant, we will concentrate on exploring the memory system's influence on code for which the execution time of the iteration sent to each processor is dominated by vector instructions.

The Alliant vector instructions can be grouped into two categories:

- Internal (register-register): All operands for these instructions are contained in vector and scalar registers.
- External: For these instructions 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 deterministic and can be derived from hardware specifications. The timings of external vector instructions can also be determined from hardware specifications. The problem in this case is that these timings depend heavily on runtime conditions such as the location of the operands (cache or memory) and degree of contention. Due to the complexity of the dependence on conditions external to the CE, carefully

designed empirical probing of the memory system is necessary for generating the behavioral characteristics of these instructions (although in some cases cycle count predictions are possible).

Since all the external vector instructions issue memory requests at the same rate (one operand per cycle) we can probe the performance characteristics by considering only two basic instructions: vector load and vector store. (The handling of the difference in startup is discussed below.)

2.3 Load/store kernels

We first focus on the **1-D** kernels. They model a single parallel DO-LOOP (i.e. all iterations can be executed concurrently). These kernels generate synthetic address streams whose characteristics can be varied by adjusting certain parameters; see below. Similar kernels were also used in [7] for studying contention at the cache level, however their main goal was to study contention under various workloads and not to predict performance.

The parameterized family of load and store kernels form the basic building blocks of the benchmark set. Their basic operation 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. This simple family is parameterized by the length of the vector, the location of the vector (cache or memory), the number of processors, and the splitting of the operations between the processors.

For spreading the iterations across the processors, we used the following strategy:

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. The dispatching of the blocks to the processors is done by self-scheduling. 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 two more parameters: Temporal Distribution and Hit Ratio. The use and purpose of these parameters are as follows:

Temporal Distribution:The goal here is to study the effect of the variation of the temporal distribution of requests. We want to study the interaction between a burst of requests (corresponding to an external vector instruction) followed by several cycles without memory requests (corresponding to the execution of an internal vector instruction or some address computation). In order to parameterize this factor, we insert between each vector instruction a variable number of NO-OP instructions (informally called *NOPS* below).

Hit Ratio: The goal of this parameter is to study the distribution of requests between the two memory levels. The hit ratio can be manipulated experimentally by inserting after a vector load (store) a variable number, k, of vector operations 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. The goal here is to analyze the behavior of the memory system under variations of temporal locality with spatial locality variations suppressed. Furthermore, insight is gained into the behavior of the cache/main memory combination when simultaneously addressed.

Based on the simple load/store family, more complex effects of the memory system are probed by building a hierarchy of load/store combinations parameterized by reference pattern. These load/store combinations are also used to model the behavior of more complex loop bodies. The hierarchy is built by considering more complex addressing stream patterns together with varying access strides.

The effects of mixing several address streams can be studied by replacing the basic vector loop used in the load/store kernel by a vector loop which accesses more than one data stream with combinations of loads and stores. The three multiple address stream kernels which typify many basic vector computations are: *load-load*, *load-store*, and *load-load-store*.

By including stride as a parameter in the simple load and store kernel family, the interleaving of the memory system can be probed as well as the effects of bursts of consecutive misses and spatial locality.

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.

Clearly, this approach generalizes easily to multivector architectures with characteristics similar to the

Alliant; e.g one memory port per processor and sufficient memory bandwidth at the fastest level. Generalization is also possible to architectures which differ more significantly. Consider, for example, the CDC Cyber 205 or the Cray X-MP with their multiple memory ports. In this case, the building block level of what were termed external vector instructions above becomes the parameterized load-load-store family. The lower levels of the hierarchy are still needed but they now correspond to a type of instruction between the internal and external types which requires slightly more careful handling.

3 Experimental results

We implemented a set of load/store kernels in assembly language parametrized by the parameters described in the previous section. All the loads and stores instructions operated on double precision data (64 bits). We varied the starting address in order to detect the sensitivity due to alignment within a cache line; due to the fact that we were streaming regularly long vectors, the alignment did not significantly influence performance. 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% variation between the extremes.

The resolution of the timer used was 10 microsec. 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 sec. However, on a cache based system, such a technique has the following drawback. If the total working set of the loop fits in 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:

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.

3.1 Load/Store Kernels Performance

3.1.1 Load Kernel

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

On one processor, the performance is very regular. There is a small loss in performance when operands come from memory: this is not due to the saturation of the memory bandwidth but rather 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 overhead associated with each block (synchronization, address computations) is non-negligible. 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.

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 Megaload/s and speedup around of 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.

3.1.2 Store Kernel

The kernel used in these experiments is in all points the same as the load kernel except that a vector write is performed instead of a vector read. A close look at the results (Figures 3a, 3b) 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%) 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% 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), since each write requires two transactions on the bus. So, four processors writing are almost equivalent to eight processors reading.

3.1.3 More complex patterns

The kernels used in these experiments were analogous to the basic load kernel, the main difference is that instead of accessing one vector, several vectors are accessed simultaneously. The purpose of these kernels is to study the impact of a mix of several address streams on the performance.

We measured the performance of these kernels for for two different block-sizes: 64, for which the overhead associated with each iteration is relatively important, and 512, for which the overhead is much smaller and relatively negligible (Figures 4a, 4b). The main conclusion is that the interference between several address streams is relatively small. Basically, the speed of Megaload-load/s is roughly half the speed of Megaloads/s independent of the location of the operands (cache or memory) and the number of processors. This equality holds within 5% when operands come from cache. For operands coming from memory rather than from cache the equality may differ by as much as 10%; this is mainly due to the fact that the density of cache requests (average number of cache request per cycle) is slightly higher: three consecutive vector access as inside a loop for load-load-store versus one vector access inside a loop for vector load. The quick succession of three vector instructions amplifies the performance loss due to the succession of transient states. Similar simple relationships also hold for load-store and load-load-store kernels: Megaload-store/s is equal to 1/4 (Megaload/s + Megastore/s) and Megaload-load-store is equal to 1/9 (2 Megaload/s + Megastore/s). We tried more complex kernels such as load-store-store, load-load-load-store, and similar relations were verified. Such relations express the conservation of the aggregate memory system bandwidth.

3.1.4 Temporal Distribution

For these experiments, we inserted 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 (Figure 5a), the first effect is predominant because there is no contention. Accessing 32 elements from cache costs approximatively 35 cycles. Adding 80 cycles (80 NOPS) after each vector instruction approximately triples the cost and we observe a correspond-

ing 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 (Figure 5b). 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.

3.1.5 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/sec (Figures 6a, 6b). From memory, we observe a phenomenon similar to the one observed with the NOPS; increasing 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 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.

3.1.6 Blocking Strategies

In these experiments, a simple load kernel was used and we varied the size of consecutive iterations allocated as a block.

According to the experimental results (Figures 7a, 7b), changing block sizes does not seem to affect the load balancing between the processors. The major effect on performance is the decreased importance of the overhead associated with each block. This effect is much more sensitive in cache because vector access are much shorter.

3.1.7 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 (Figure 8a). 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 (Figure 8b). 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.

4 Conclusions

As noted earlier, there are two basic goals of the proposed approach. The first goal is obtaining a detailed characterization of the behavior of the memory system and the parameters through which it can influence algorithm performance. The progress towards this goal was discussed in the previous section. The second goal is the development of a strategy, based on the load/store model for predicting the performance of application codes. In this section we will first review some of the key conclusions obtained from our experimental data set and then we will indicate how such experimental data may be used for predicting performance of simple DO-LOOPS.

4.1 Experimental Conclusions

The first major 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, one and two processor experiments), we were able to predict the performance of the load/store kernels to within 10% 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 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 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.

4.2 Performance prediction

The performance prediction proceeds in three basic steps.

1. Building a database of experimental points in the parameter space. This experimentation can be conducted fairly systematically by for example sweeping first the parameter in a very coarse way, then by refining further the investigation (increasing the number of experimental points) in regions where the performance gradient is large.

2. Extracting from the test code (to be analyzed) the value of the parameters which allow us to establish a correspondence between experimental points and the test code.

3. Using simple analytical models, interpolate the predicted performance from the experiments which are the closest in the parameter space.

Let us focus on the second problem. In our case, following the choice of parameters, we need to determine from the source code hit ratios (i.e location of the operands), temporal distribution, patterns of accesses, strides. In fact most of these parameters (except the location of the operands) can be determined by inspection of the assembly code generated from the loop studied. This analysis at present is done manually, but many of the software analysis tools to calculate the required parameters exist and will be integrated into a performance prediction tool in the near future.

Predicting the location of an operand is more complex. The simplest solution is to consider the two extreme cases: either all the operands coming from cache (upper bound on the performance) or all the operands coming from memory (lower bound). Unfortunately, these approximations can give a very large

range of potential performance. In the case of linearly indexed array, which is a very common case in numerical computations, the problem can be solved using recently proposed data dependence analysis techniques [6]. Moreover, these techniques also allow the computation of the total number of distinct references to an array and a static estimate of the cache hit ratio.

Our present and future work on this topic include: the verification of the approach on more complex loop structures; the refinement of multiprocessor prediction; further integration of the data locality estimates into the prediction process; and the extension of the approach to more complicated architectures such as Cedar. In all cases, careful attention is given to determine the possibility of automation of the process.

References

[1] Abu-Sufah, W. and Kwok, A. *Performance prediction tools for Cedar: A multiprocessor supercomputer.* 12th Int. Symp. on Comp. Arch., 1985, pp. 406-413.

[2] Bailey, D. *Vector computer memory bank contention.* IEEE TC, C-36,3, 293 - 298, March, 1987.

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

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

[5] Cheung, T. and Smith, J. *An analysis of the Cray X-MP memory system,* Proc. Int. Conf. on Parallel Processing, August 1984, pp. 494-505.

[6] Gannon, D., Jalby, W., and Gallivan, K. *Strategies for Cache and Local Memory Management by Global Program Transformation,* Jour. Par. and Distr. Computing, Oct. 1987, pp. 587-616.

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

[8] Phister, G. and Norton, A. *Hot spot contention and combining in multistage interconnection networks.* Proc. Int. Conf. on Parallel Processing, 1985, pp. 790-797.

Figure 1
Alliant FX/8 Architecture

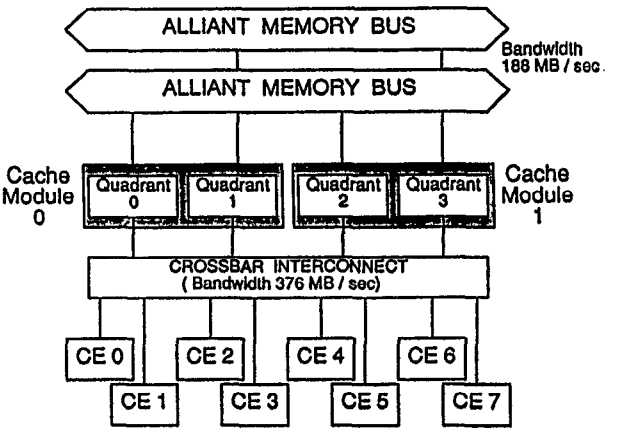

Figure 2a
Load Performance
(Block Size 128)

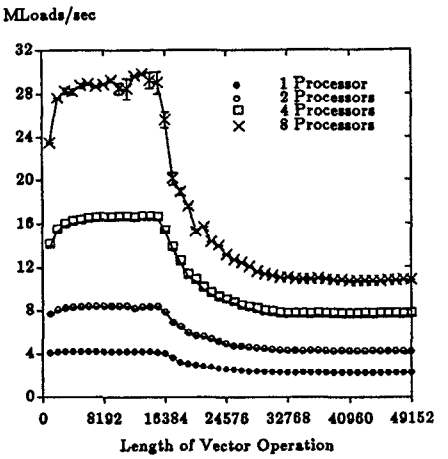

Figure 2b
Load Speedup
(Block Size 128)

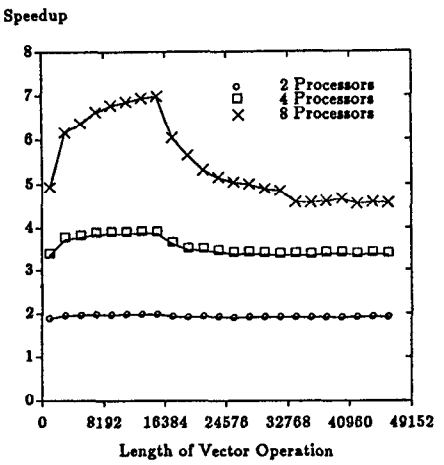

Figure 3a
Store Performance
(Block Size 128)

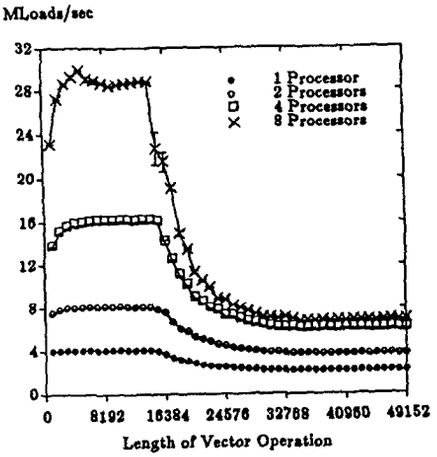

Figure 3b
Store Speedup
(Block Size 128)

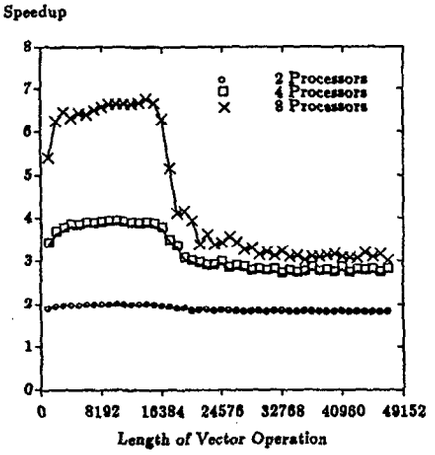

Figure 4a
Kernel Performance on 1 Processor
(Block Size 512)

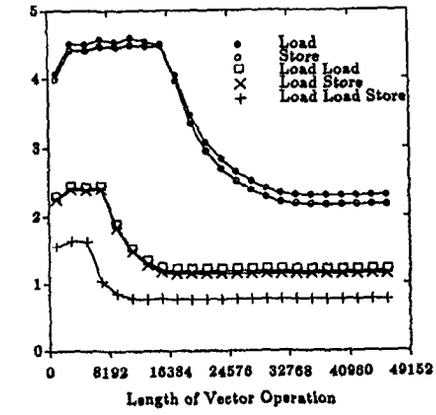

Figure 4b
Kernel Performance on 8 Processors
(Block Size 512)

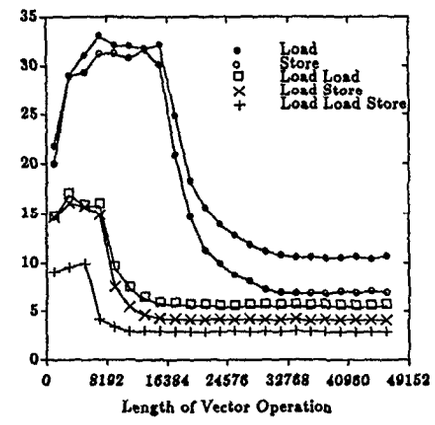

Figure 5a
Effects of NOPs on 1 Processor
(Load kernel, Block Size 128)

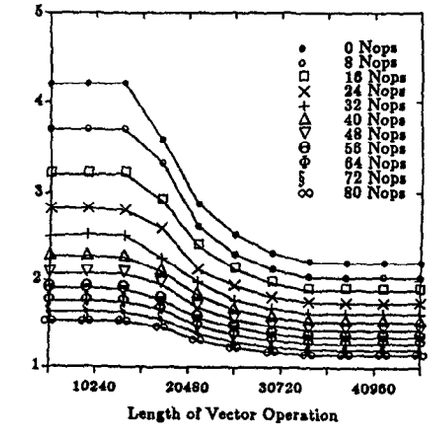

Figure 5b
Effect of NOPs on 8 Processors
(Load Kernel, Block Size 128)

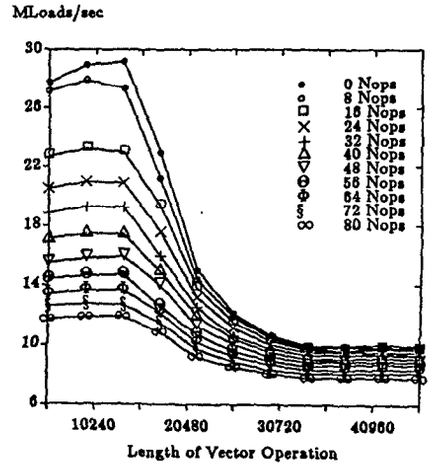

Figure 6a
Effect of Cache Miss Ratio on 1 Processor
(Load kernel, Block Size 128)

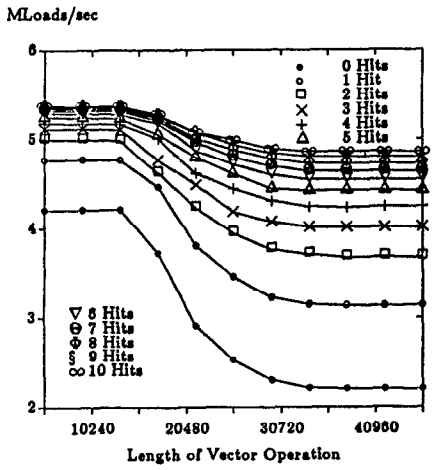

Figure 6b
Effect of Cache Miss Ratio on 8 Processors
(Load kernel, Block Size 128)

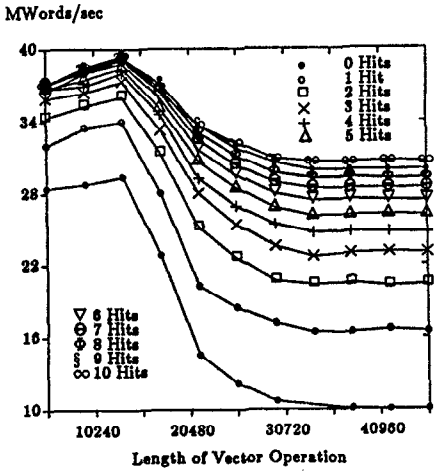

Figure 7a
Effect of Block Size on 1 Processor
(Load kernel)

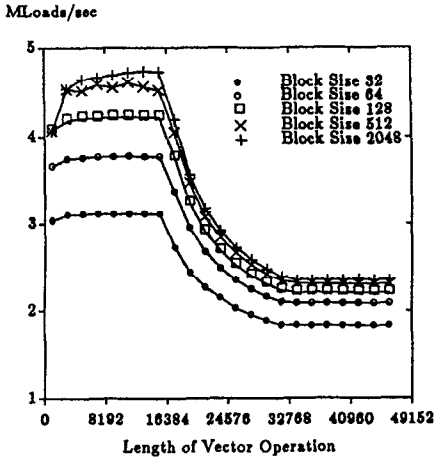

Figure 7b
Effect of Block Size on 8 Processors
(Load kernel)

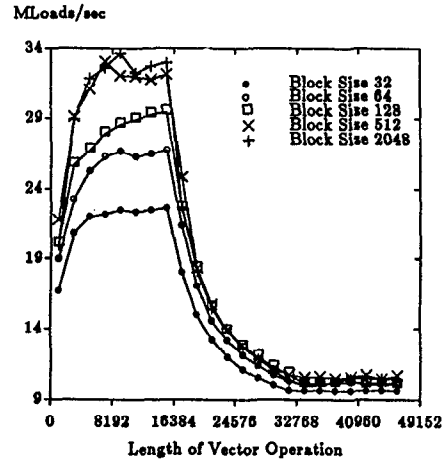

Figure 8a
Effect of Strides in Cache
(Load kernel)

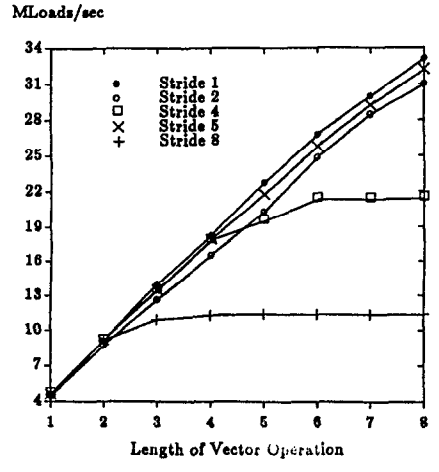

Figure 8b
Effect of Stride in Memory
(Load kernel)