Performance Prediction of Loop Constructs on Multiprocessor Hierarchical-Memory Systems

Kyle Gallivan William Jalby Allen Malony
Harry Wijshoff

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

Abstract

In this paper we discuss the performance prediction of Fortran constructs commonly found in numerical scientific computing. Although the approach is applicable to multi-processors in general, within the scope of the paper we will concentrate on the Alliant FX/8 multiprocessor. The techniques proposed involve a combination of empirical observations, architectural models and analytical techniques, and exploits earlier work on data locality analysis and empirical characterization of the behavior of memory systems. The Lawrence Livermore Loops are used as a test-case to verify the approach.

1 Introduction

In recent years, all U.S. supercomputers manufactures, and dozens of mini-supercomputer vendors have entered the market with some form of general purpose parallel processing. It is expected that most computer companies will offer a scalable multiprocessor by the end of the decade. Unfortunately, very few of these systems provide a software environment for building parallel programs beyond a standard sequential language compiler and a micro-tasking library supporting parallel execution. None of the most widely used programming languages (C, Fortran, Lisp) have been officially extended to support concurrency and vendors generally will disagree on most of the unofficial extensions. Consequently, the task of porting ordinary programs to parallel hardware and developing new, large parallel applications codes has been moving at a slow pace.

Clearly, a set of tools to help the program designer transform existing code or develop new codes is highly desirable. These tools must provide the program designer with information such as: why a program does not exhibit the structure of concurrency needed to exploit a given machine; the influence of task granularity on the structure of the algorithm; the impact of memory hierarchy (or lack thereof) on the performance of the algorithm; and the ways in which the algorithm exploits multilevel concurrency. In short, these tools must provide expert analysis and advice on improving performance.

In this paper, we discuss performance prediction of loop constructs commonly found in scientific numerical computing, e.g. numerical linear algebra computations. In particular, we consider the consequences of one of the common observations from past performance studies on an Alliant FX/8: peak performance is often dominated by the data transfer/management capabilities of a particular architecture. The resulting prediction strategy involves the use of: data dependence analysis to estimate data locality [1,2]; a hierarchy of data motion kernels that characterize memory system behavior [3]; an assembler code analyzer to evaluate characteristics of the code generated by the compiler such as instruction counts and *ideal* cycle counts for certain segments of the code; and modeling techniques which use the resulting information to predict performance.

In section 3, 4 and 5 the necessary background material concerning the load/store primitive hierarchy and the data locality estimation techniques is presented. The use of the load/store hierarchy to pre-

This work was supported by the National Science Foundation under grants US NSF MIP-8410110 and US NSF MIP-88-07775, the Department of Energy under grant US DOE-DE-FG02-85ER25001, and an IBM Donation.

dict performance on the Alliant FX/8 is illustrated section 6 by considering the prediction some of the Lawrence Livermore Loops [4].

2 The Target Architecture: Alliant FX/8

The Alliant FX/8 machine consists of up to eight pipelined computational elements (CE's), each capable of delivering a peak rate of 11.75 Mflops for double precision calculations (two operations per cycle) implying a total peak rate of approximately 94 Mflops. The startup times for the vector instructions can reduce this rate significantly. For example, the vector triad instruction $v \leftarrow v + \alpha x$ (the preferred instruction for achieving high performance in many codes) has a maximum performance of 68 Mflops.

The CE's are 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.

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. 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. 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 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. Note that these are hardware specifications and not necessarily the achieved bandwidths that should be used when analyzing algorithm performance; in practice, stride-1 vector loads of data residing in main memory run at a rate of approximately 12 Mwords/s [3].

3 Load/Store Primitive Hierarchy

Often the architectural parameters supplied by the manufacturer bear little or no relation to the performance values important from the algorithmic point of view. In particular, in order to predict performance we must have a reasonably accurate characterization of the memory system's behavior. In this paper, we use a behavioral characterization of the memory We use a behavioral characterization of the memory system based on a hierarchy of *data-motion* primitives [3]. The hierarchy, informally called the LOAD/STORE model, consist mainly of vector moves (vmove) and null nop operation instructions. This section reviews the LOAD/STORE hierarchy and its relationship to the prediction of loop constructs.

3.1 Load/Store Model

The basic primitive in the hierarchy is 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. The structure of the basic primitive kernel is shown in table 1. 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, which are used to explore the behavior of a single vector read or write of length n, each iteration corresponds to the processing of a single block of length b. 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.

A database of empirical results is produced by varying several aspects of this basic template. The two simplest parameters to vary are the number of processors, p, and the length of the vector loop, n. A more complex variation investigates different scheduling/partitioning choices. Two scheduling strategies have been chosen for our database on the Alliant: self-scheduling with contiguous blocks, and static scheduling with both contiguous blocks and interleaved blocks. In self-scheduling, the vector is divided into blocks of size b and the vector operation is transformed into a concurrent outer loop over the blocks. The blocks are allocated to the processors at run-time via the hardware self-scheduling mechanism of the FX/8. With static-scheduling, the vec-

```
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
        jump to loop_body
resume sequential operation
```

Table 1: The basic load/store primitive template.

tor operation is either divided into p blocks, each of which consists of either contiguous elements of the vector operands (contiguous blocks), or in an interleaved fashion where each element of a set of p contiguous elements is placed in a different block (interleaved blocks). Vector partitioning choices (contiguous versus interleaved) additionally serve to help characterize synchronization costs, the effects of load balancing, and memory conflict arbitration.

The variation of the number of nop instructions making up the nop sequence in the kernel is used to evaluate the effect of changing the density of memory requests made by each processor. This parameter is also used in the prediction methodology described below to take into account the differences in the start-up times of the various vector instructions as well as model any register-register operations and address computations which take place between vector instructions in the iteration body code.

The stride of the vector is varied in the basic primitive hierarchy in order to characterize the effect on performance of the mapping strategy used to assign elements of an array to cache banks. This parameters can also be used to probe the effects of bursts of successive misses. The stride can also be varied to manipulate the cache banks servicing the misses if knowledge of the address mapping is exploited.

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 hit ratio. 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 primitives found in vector computations are: load-load, load-store, and load-load-store. The hit ratio is manipulated by choosing the vector sequence to be a vmove instruction to or from memory followed by a sequence of k vmoves to the same location. The first vmove causes a cache miss with the subsequent k accesses being cache hits. The hit ratio and multiple address stream variations can also be combined.

3.2 Experimental Results

Many experiments with the LOAD/STORE kernels were performed on the Alliant FX/8 using different parameter combinations. Here we present results from this empirical study as background to the issues faced in kernel performance prediction (see also [3]). The major conclusion from the study is that the LOAD/STORE kernels accurately matched empirical observation of vector codes with similar form. Also, the characteristic curves obtained from the results showed smooth behavior relative to the varying parameters allowing the qualitative prediction of performance trends. From a quantitative point of view, however, explaining kernel performance behavior is more complex.

Figures 1 and 2 show the characteristic curves for five basic kernels for one and eight processors, respectively. The key observations are that performance is at its peak within cache, performance falls off as more cache misses occur, and performance reaches a minimum as references are made entirely from memory. The significant difference between the one and eight processor curves is that one processor performance shows no memory saturation effects and its peak-to-minimum performance reflects the 2-to-1 bandwidth performance difference between cache and memory. On the other hand, whereas the eight processor in-cache performance shows no cache stressing effects (the peak performance is roughly eight times the peak performance of one processor), the falloff is significantly greater showing a 3-to-1 performance difference (it is worse in the cases of store operations because of the write-back cache). Clearly, the memory bandwidth is being saturated at this point.

The effects of varying NOPs are shown in figure 3. Overhead processing, represented by the NOPs, can significantly reduce performance within cache because of their relatively larger percentage of total execution time. From memory, the effect of this overhead is less severe because of the predominance of data movement time. One also observes the influence of NOPs on memory bandwidth saturation. The performance difference becomes less as the overhead increases because the *density* of memory operations reduces. It is conjectured that the effect is a spreading out of memory references to a point that the instantaneous memory bandwidth request does not saturate the available bandwidth.

The effects of varying cache hit ratio are shown in figure 4. As expected, performance vastly improves for long vector lengths as the cache hit ratio increases. However, the rate of improvement decreases because the time to service one cache miss per k hits still represents a greater portion of total execution time posing a performance barrier. For vector references within cache, the vector hits simulate the effects of loop unrolling. The performance improvement for greater vector hits corresponds to the reduction in the significance of the kernel's fixed loop overhead. Interestingly, there is a limiting point where the overhead is irrelevant and the kernel is performing at the limit of the cache bandwidth.

The hierarchy formed by the different combinations of the LOAD/STORE kernels together with variations of the initial nop sequences and cache hits forms the basis of the strategy for predicting the behavior of concurrent-vector operations on the Alliant FX/8.

4 A Prediction Database

The prediction of parallel numerical DO-loop constructs are obtained by interpolating/extrapolating from data in an empirical database. Although the behavior of parallel numerical DO-loop structures can be perfectly modeled with the help of the LOAD/STORE kernels, because the number of parameters involved in the LOAD/STORE kernels and the number of basic templates can be large, providing all the necessary database entries can be an overwhelming task. Therefore some abstraction of the parameter space and the basic templates must be done in order to make the construction of such a database feasible.

We propose the following strategy. On a particular architecture, the load kernel is considered first. It is probed on a subset of the parameter space which characterizes the major bottlenecks and performance gradients in the memory system. The initial guess at such a set is based on some knowledge of the general architectural structure of the machine as well as specific detailed knowledge about a particular machine. We will call such a set the *characterizing subset* of the parameter space. Next the hierarchy of sequences is built up, store, load-store, load-load-store, load-load-load-store, and probed for the same subset of the parameter space, until we reached the point that for some $k - load$ the sequence load$^{k_{load}}$-store achieves essentially the same performance as the load$^{k_{load}+1}$. The same procedure is performed with store-oriented sequences, e.g. load-store-store, load-store-store-store, to identify k_{store}. The two sequence of LOAD/STORE kernels obtained form the *fundamental set of kernels* for the particular architecture. For instance, on the Alliant FX/8, $k = 4$ determines the fundamental set, see figures 5 and 6. Specifically, the fundamental LOAD/STORE set for the Alliant FX/8 comprises the kernels: load, store, load-store, load-load-store, load-load-load-store, load-store-store, and load-store-store-store.

The second phase in the strategy considers the determination of properties of the fundamental set which allow the transformation of an instruction sequence found in a code segment or a sequence of NOP and vmove instructions into an element or combination of elements in the fundamental set. (Notice that the determination of the parameters k_{load} and k_{store} also yields such information.) The most fundamental property used in this transformation is commutativity. In particular, two forms of commutativity are considered:

- Structural Commutativity
- Spatial Commutativity.

The first form of commutativity asserts that the order of the loads and stores in each fundamental kernel is independent of the performance of the kernel. So, for instance, for the load-load-load-store kernel it is investigated whether the performance of this kernel matches the performance of the store-load-load-load, the load-store-load-load and the load-load-store-load kernel on the characterizing subset of the parameter space. Such structural commutativity holds for the fundamental set of kernels on the Alliant FX/8.

Spatial commutativity pertains to the relationship of performance and the location of the NOP instructions in the vmove sequence in the loop body of the kernels. It asserts that the NOP instructions distributed over the loop body of the fundamental kernels can be replaced by a single group of NOPs at the beginning of the loop body. This is easily determined empirically and is clearly required if the fundamental kernels are to have the form described earlier. In figures 7 and 8 the performance of the load kernel is depicted for different locations of NOPs insertion. Again on the Alliant FX/8, the fundamental kernels are spatially commutative.

After the commutativity properties are determined, the fundamental set is extended to include the structural and spatial variations which were determined not to be equivalent via commutativity to the original elements. For the Alliant FX/8 such an extension is not required. In fact, the size of the extended fundamental set of kernels can be viewed as a measure of the complexity of run-time behavior of a particular machine. Clearly, if the set grows to large it is an indication that the behavioral characterization of the memory system proposed in [3] and the prediction strategy discussed here are probably not applicable.

After the fundamental set of kernels and its transformation properties are determined, the parameter space is probed to minimize the amount of information required in the database (the number of sample performance points which must be evaluated) to achieve a reasonable accuracy in predicting performance. This is accomplished by sweeping the parameter space iteratively with grids of increasing refinement to identify and probe regions where the performance gradient is large. (The refinement is, of course, carried out only in those areas where large gradients are observed.)

5 Predicting the performance of DO-loop structures

Based on the empirical database described in the previous section, a method has to be defined which allows the extraction from the test code (to be analyzed) the value of the parameters. This method should allow us to establish a correspondence between experimental points and the test code. In other words, we need to determine from the source code: hit ratios (i.e., location of the operands), temporal distribution, patterns of accesses, and strides. Most of these parameters (except the location of the operands) can be determined by inspection of the assembly code generated from the loop studied. The general strategy to establish the correspondence between the test code and the empirical database is as follows:

- From the assembly code generated from the test code, a template of the load/store pattern of the code is constructed and a NOP value is determined which accounts for the differences in the startup times of the arithmetic vector instructions and the vmove instructions that have replaced them as well as for any register-register operations.
- The amount of non-data-movement activity that has not been accounted for above is determined and converted into a second NOP value for the template. (This non-data-movement includes scalar operations that involve operands which may be in memory or cache and must therefore be treated slightly differently than the deterministic NOP count generated above.)
- The basic template is reduced to one of the fundamental kernels.
- The relative number of floating point operations per data-movement operation (load or store) is determined.
- The location of the operands in the memory hierarchy is analyzed to produce an average hit/miss ratio.

The construction of the basic templates can be easily obtained by analyzing the assembler code. A choice must be made, however, concerning the range of a basic template. By this we mean that one could view a whole program as being represented by the basic template, or only part of it. It is important to note that there is trade-off between predictability of the test code and reducibility of the basic templates: the larger the range of the basic template, the better the performance prediction – this also implies an attendant increase in the complexity of the reduction process which maps the template to the fundamental set of kernels. On the Alliant FX/8 we have chosen the range of a basic template to be the range of a concurrent or vector-concurrent loop. The template may in turn have a number of component loadstore patterns which correspond to a sequence of distinct vector loops with in the body of each iteration of the loop under consideration (see the template for LLL8 below).

The reduction of each basic template to one of the fundamental kernels has to be handled with some care. According to the commutativity rules the loads and the stores of the basic template are grouped together. Note that the commutativity rules were derived from the fundamental kernels and do not necessarily have to apply to the basic templates in general. Our experience has shown, however, that in most cases the error introduced by applying the commutativity rules to the basic templates is negligible. On the Alliant FX/8 where all the fundamental kernels are structurally symmetric the grouping of the load's and store's will always yield a template in the form: load^{k}store^m. This template is now reduced by normalizing the number of store's if $k \geq m$ or $k = 0$, or the number of load's if $k < m$ or $m = 0$. After this normalization the basic template has the form: load^{q}store (loadstore^q) with $q = k/m$ ($q = m/k$). Whenever q is an integer the corresponding fundamental kernel depends upon the relationship of q and k_{load} or k_{store} depending on which are dominant in the template the loads or the stores. If $q \leq k_{load}$ ($q \leq k_{store}$) then the template has been reduced to an element of the fundamental set. If k is not an integer then the basic template will be bounded by the corresponding fundamental kernel of load$^{\lfloor q \rfloor}$store (loadstore$^{\lfloor q \rfloor}$) and load$^{\lfloor q \rfloor+1}$store (loadstore$^{\lfloor q \rfloor+1}$). The fact that the number of data-movements in both of the enclosing fundamental kernels are not related to the original basic template by an integer scaling function, which was the case if q is an integer must be taken into account when determining the relative number of floating point operations per data-movement as well as when determining the values for the other parameters needed to extract information from the empirical database.

Since the set of all basic templates can be embedded in a complete binary tree and the set of fundamental kernels is a distinct subset of this tree, the reduction process of a basic template to an element of the fundamental set is a mapping of a given node in a binary tree onto one of the nodes representing a fundamental kernel. This process is depicted in figure 9 where the set of fundamental kernels is that of the Alliant FX/8. So the set of fundamental kernels has the property that it spawns a complete binary tree. This concept is related to codes as defined in the theory of codes, see [6,7].

The relative number of floating point operations per data-movement is obtained by determining for each data-movement operation (load or store) in the basic template whether or not it corresponds to an arithmetic computation or to a simple load/store into/from a vector register. Note that this number must be scaled appropriately in the reduction and, in particular, when the template is not exactly reducible to a fundamental kernel.

The average hit/miss ratio is determined by the location of the operands of the data movements in the basic template. Predicting the location of an operand is more complex than determining the values of the other parameters of the database. 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. We can refine our approach by assuming an infinite cache and assuming that at the beginning of the loop all the operands are initially coming from memory. In this case the problem of determining if, at a given point of the program, data is in cache is equivalent to the problem of determining if a previous instruction has already accessed the data. This, of course, is the classical data dependence problem encountered by any restructuring compiler. 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 [5]. 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.

The amount of non-data-movement operations can be obtained by a simple instruction count in between each pair of load/store instructions or controlling instructions (e.g. the bounding instructions of the concurrent and vector iterations) in the basic templates. A simple estimate of the average number of cycles involved for each of these instructions can be used to get the average number of corresponding NOPs in between each two load/store instruction. (This assumes that the register-register vector instructions have already been converted to NOP instructions.) The spatial commutativity properties of the fundamental kernels are then used to group the NOPs together as much as possible. In the case of the Alliant FX/8 where all the fundamental kernels are spatially commutative the NOPs can all be shifted to a fixed place in the basic templates. Thereafter, the number of NOPs are scaled according to the reduction of the basic template to the corresponding fundamental kernel as are the other parameters.

After this procedure the performance prediction of the test code is interpolated from the entries in the experimentation database which are closest in the parameter space. Note that, because of the construction of this database (see the previous section), the error introduced by this interpolation is small even when the distance in the parameter space between the test code and the database entry is large. In the next section we show the feasibility of the performance prediction strategy by applying this strategy to the Lawrence Livermore Loops.

6 Predicting the Performance of the Lawrence Livermore Loops

In this section we consider the application of the performance prediction strategy outlined in section 5 to the Lawrence Livermore Loops [4]. The LLL kernel set comprises 24 samples of Fortran source codes, which are meant to expose many specific inefficiencies in the formulation of Fortran code, in the quality of the compiler, and in the capability of the architecture. Because of these properties we have chosen the first twelve of these kernels as a testbed for our prediction strategy. For the purpose of presentation we will handle only kernel 2 (LLL2) in detail. The performance predictions of the other kernels will be summarized.

LLL2 represents an excerpt of an Incomplete Cholesky-Conjugate Gradient code. The loop body has the form:

```
          IL = na
       IPNTP = 0
  222   IPNT = IPNTP
       IPNTP = IPNTP+IL
          IL = IL/2
           i = IPNTP
CVD$ NODEPCHK
CVD$ NOSYNC

      DO 2 k = IPNT+2,IPNTP,2
           i = i+1
    2   X(i) = X(k) - V(k)*X(k-1)
              - V(k+1)*X(k+1)
         IF ( IL.GT.1) GO TO 222
```

NODEPCHK and NOSYNC are FX/Fortran compiler directives. After compilation by the FX/Fortran (version 4) compiler the assembler code has essentially the following structure:

```
57:   cvector
1112.label_LC
1112.label_LF
104:  vmoved   __BL__+8000+16:l[d7:l:d],v3
106:  vmoved   __BL__+0+8:l[d7:l:d]    ,v2
108:  vmoved   __BL__+8000+8:l[d7:l:d] ,v0
110:  vmuld    __BL__+0:l[d7:l:d]    ,v0,v0
112:  vmuld    __BL__+0+16:l[d7:l:d],v3,v3
116:  vrsubd   v2,v0,v2
117:  vrsubd   v2,v3,v2
121:  vmoved   v2,__BL__+0:l[d7:l:d]
125:  vcnt32   1112.label_LF
128:  crepeat  1112.label_LC
```

The numbers in front of each line denote the actual line number in the assembly code, and the __BL__ strings indicate memory addresses. So, LLL2 yields [LLLLLS] as a basic template (L and S denoting load and store respectively). Note, that this basic template is already of the form load^{q}store, so we can directly map it to a fundamental kernel. The translation of the non-data-movement instructions into corresponding NOPs is a little bit more tedious. As noted earlier we distinguish between two different sources of NOPs: fixed and variable. The fixed number of NOPs is the sum of those counted for each concurrent start and branch instructions, register-register vector instructions and the difference in startup costs between a vmove instruction and the arithmetic instruction it replaces. The variable number of NOPs provides an estimate of other scalar instructions whose costs may vary, e.g. scalar instructions with operands which may be in memory or in cache. An estimate is computed by counting the number of assembler lines in between each two vector instructions and multiplying this number with scalar α which is the average number of cycles required for a non-vector instruction. For example, on the Alliant FX/8 $\alpha = 3$ when operands are assumed in cache.

The resulting basic template for LLL2 is then $9+14\alpha+[LLLLLS(112, 42\alpha)]$. The numbers in parentheses are the values of the fixed and variable NOPs per iteration of the vector loop. The initial terms in the template are the NOPs which are associated with the concurrent loop level. Note that the extraction of this template and the calculation of the various values of NOPs is easily automated. Such a tool is currently under development at the University of Indiana by D. Gannon and D. Atapattu.

Because the length of the basic template is greater than $k_{load}(4)$, see Section 5, this basic template reduces into the the fundamental kernel: $9 + 14\alpha + [L(19, 7\alpha)]$. So, in fact kernel 2 can be modeled by an simple load kernel. The average number of floating point operations per data-movement is 4/6, and the average vector hit/miss ratio of this kernel is 3/3. Note however that some of the accesses are stride 1 and others have stride 2. In order to predict the behavior of LLL2 we interpolate the performance of the mixed stride load by weighting appropriately the performance figures of the fundamental kernel $9 + 14\alpha + [L(19, 7\alpha)]$ with vector hit/miss ratio of 1 for stride 1 and stride 2. This results in a bandwidth of approximately 9 Mword/s. The final predicted performance value of LLL2 is then computed by applying the operations-to-data-movement ratio yielding $0.66 * 9.15$ Mflops $= 6.1$ Mflops. The actual performance of LLL2 is 5.4 Mflops. In table 2 the basic templates of each kernel are presented. table 3 contains the equivalent fundamental kernels and the vector hit/miss ratio, the stride values, and the predicted and actual performance of LLL1 through LLL12 for vector lengths long enough so that the data starts out in main memory, i.e. the execution of LLL2 on each iteration of the timing loop is guaranteed to flush the cache of all of the operands needed by the next iteration.

7 Conclusions

Since the performance of parallel numerical algorithms on super/parallel computers is largely dependent on the transfer rates of data loads and stores, the performance prediction strategy proposed in this paper is based on the load/store behavior of a particular architecture. The characterization of this behavior is accomplished via a strategy proposed and evaluated for the Alliant FX/8 [3]. The strategy relies on a systematically constructed empirical database of a hierarchy of data-motion primitives In order to minimize the data storage required for the database, its structure is kept orthogonal. This is achieved by the identification of a set of fundamental kernels which, in some sense, spawn the entire space of possible load/store templates. Further, for each of these fundamental kernels the performance data is gathered for a set of points in the parameter space which are chosen, based on the value of the variation of the performance gradient, to keep the density of database entry points reasonable.

The prediction of the performance of a particular test code is obtained by first extracting a basic load/store template of the test code then reducing this template to a fundamental kernel via transformations which have been determined during the construction of the database. The values of the parameters for the fundamental kernel which index the database are then determined via an automatable procedure followed by an interpolation based empirical performance results in the database predicts performance. We conclude that for the LLL kernels on an Alliant FX/8 this strategy yields a reasonably accurate prediction of the observed performance.

Our present and future work on this topic includes: the automation of the prediction process, the verification of the approach on more complex loop structures and other architectures such the CEDAR system and the Cray 2 and Cray X-MP architectures.

References

[1] Gannon, D., Jalby, W., and Gallivan, K., *Strategies for Cache and Local Memory Management by Global Program Transformation*, Proc. 1987 Int. Conf. on Supercomputing, 1987, pp. 229-254.

[2] Gallivan, K., Gannon, D. and Jalby, W., *On the problem of optimizing data transfers for complex memory systems*, Proc. 1988 Int. Conf. on Supercomputing, 1988, pp. 238-253.

[3] Gallivan, K., Gannon, D., Jalby, W., Malony, A., and Wijshoff, H., *Behavioral Characterization of Multiprocessor Memory Systems: A Case Study*, CSRD Report 808, University of Illinois, 1988.

[4] McMahon, F., *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Report UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.

[5] 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.

[6] Berstel, J. and Perrin, D., *Theory of Codes*, Vol 117, Pure and Applied Mathematics, Academic Press, New York, 1985.

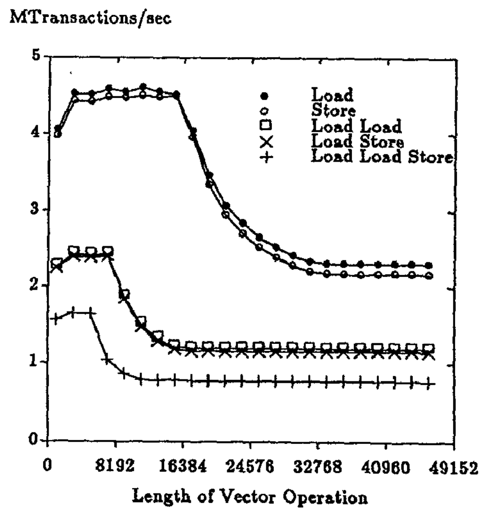
[7] Wijshoff, H., *Data Organization in Parallel Computers*, Kluwer Academic Publishers, Boston, 1988.

Kernel	Basic Template
LLL1	$9+10\alpha+[\mathrm{LLLS}(41,23\alpha)]$
LLL2	$9+14\alpha+[\mathrm{LLLLLS}(112,42\alpha)]$
LLL3	$34+40\alpha+[\mathrm{LL}(17,7\alpha)]$
LLL4	$30+47\alpha+[\mathrm{LL}(17,10\alpha)]$
LLL5	$20+15\alpha+[\mathrm{LLLS}(14,10\alpha)]$
LLL6	$30+61\alpha+[\mathrm{LL}(17,11\alpha)]$
LLL7	$20+21\alpha+[\mathrm{L}^9\mathrm{S}(224,20\alpha)]$
LLL8	$23+119\alpha+[(\mathrm{LLS})^3(26,43\alpha)],[(\mathrm{L}^7\mathrm{S})^3(389,83\alpha)]$ †
LLL9	$20+40\alpha+[\mathrm{L}^{10}\mathrm{S}(131,27\alpha)]$
LLL10	$20+34\alpha+[\mathrm{LS}(\mathrm{LSS})^9(76,86\alpha)]$
LLL11	
LLL12	$20+16\alpha+[\mathrm{LLS}(10,8\alpha)]$

†The basic template is a composition of two different templates.

Table 2: The Basic Templates of LLL1 through LLL12

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



Kernel	Fund. Kernel	Fl./Tr.	V. H/M	Stride	Pr. Perf.	Act. Perf.
LLL1	9+10α+[LLLS(41,23α)]	5/4	1/3	1	14.1	13.5
LLL2	9+14α+[L(19,7α)]	4/6	3/3	1,2	6.1	5.4
LLL3	34+40α+[L(9,4α)]	2/2	0/2	1	10.5	9.8
LLL4	30+47α+[L(9,5α)]	2/2	0/2	1,5	4.3	3.4
LLL5	20+15α+[LLLS(14,10α)]	2/4	1/3	1	5.7	6.0
LLL6	30+61α+[L(8,6α)]	2/2	0/2	1,dim	3.6 ††	2.1
LLL7	20+21α+[L(22,2α)]	16/10	6/4	1	21.0	21.0
LLL8	23+119α+[LLS(3,14α)],[L(16,3α)]	36/33	21/12	1,5	9.1	7.1
LLL9	20+40α+[L(12,2α)]	17/11	7/4	dim	6.4	5.4
LLL10	20+34α+[LS$^{1.9}$(8,9α)] ‡	9/29	20/9	1,dim	2.1	2.0
LLL11 ‡‡						2.0
LLL12	20+16α+[LLS(10,8α)]	1/3	1/2	1	4.4	4.1

††This kernel actually represents a triangular solve, so the vector lengths of the instructions range from 2 up to dim. This explains the larger deviation from the actual performance.

‡The performance of this fundamental kernel is approximated by the corresponding LS template and LSS template.

‡‡Due to the fact that the code generator produced a call to an intrinsic of which we did not have the source code, the performance could not be estimated.

Table 3: The Fundamental Kernels, Predicted Performance and Actual Performance of LLL1 through LLL12

Figure 2
Kernel Performance on 8 Processors
(Block Size 512)

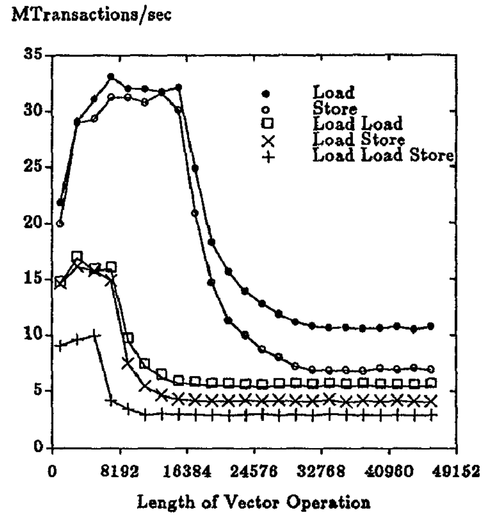


Figure 3
Effect of NOPs on 8 Processors
(Load Kernel, Block Size 128)

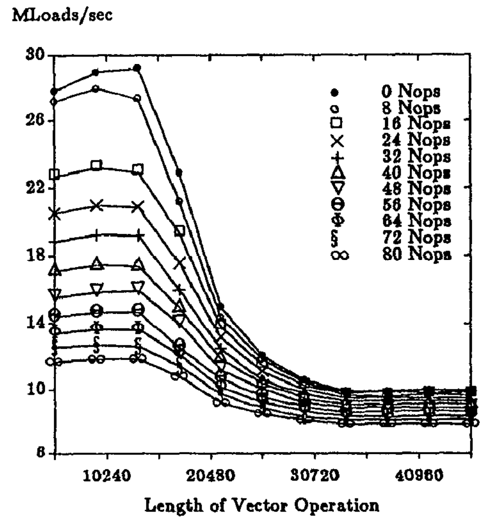


Figure 4
Effect of Cache Miss Ratio on 8 Processors
(Load kernel, Block Size 128)

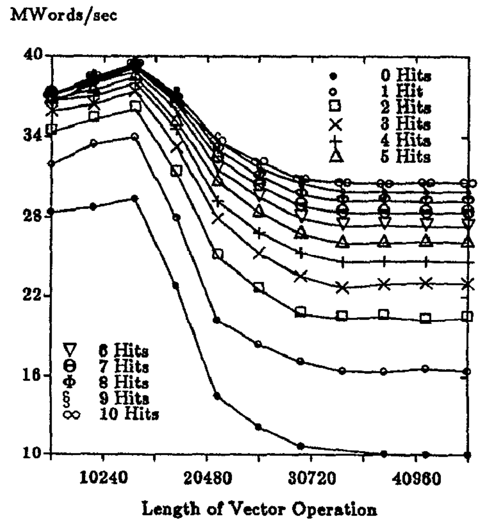


Figure 5
Kernel Performance
(8 Processors)
(Block Size 256)
(0 Nops)

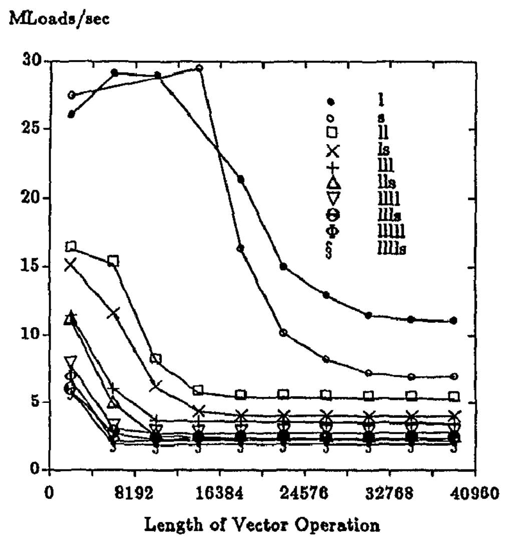


Figure 6
Kernel Performance
(8 Processors)
(Block Size 256)
(64 Nops)

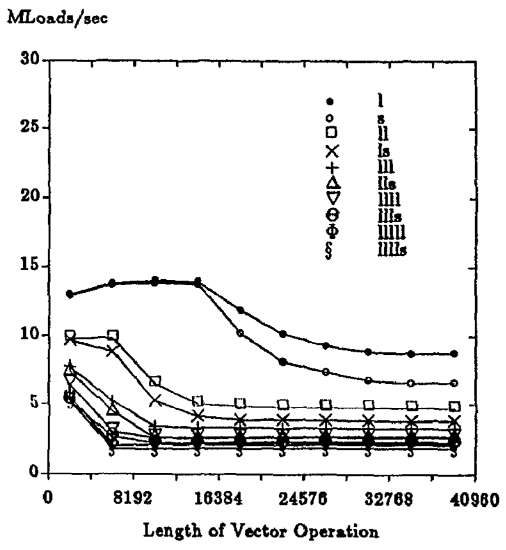


Figure 7
Load Performance
(Block Size 8192)
(Nopsb=4096, Nopsc=512, Nopsd=16)

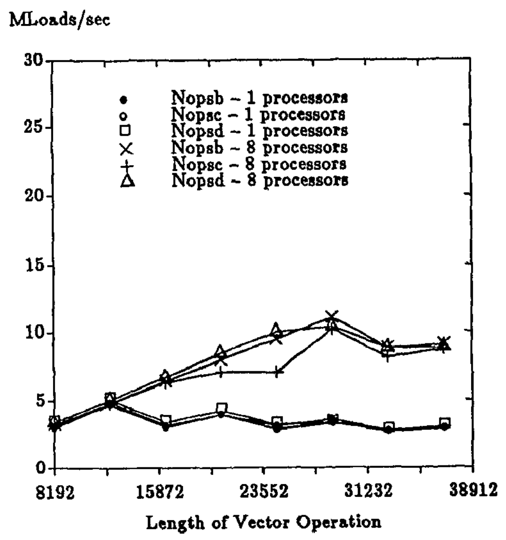


Figure 8
Load Performance
(Block Size 1024)
(Nopsb=1792, Nopsc=224, Nopsd=56)

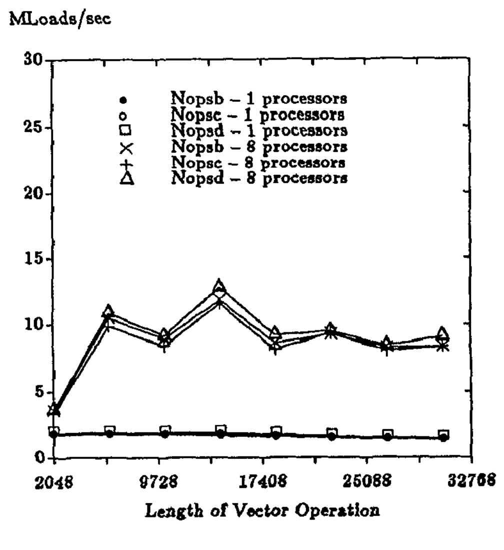
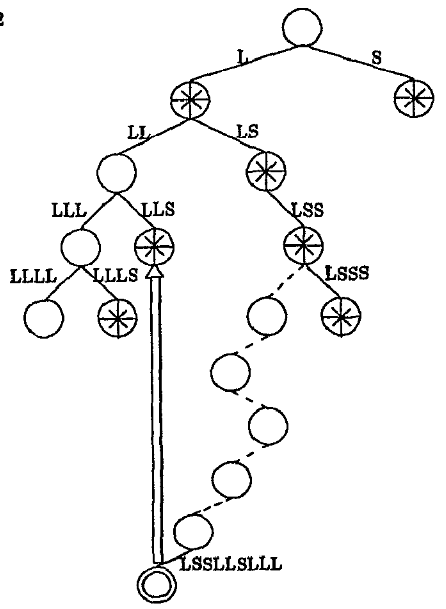


Figure 9
Reduction to a Fundamental Kernel


L
S
LL
LS
LLL
LLS
LSS
LLLL
LLLS
LSSS
LSSLLSLLL