TECHNICAL REPORT NO. 277

Building Analytical Models into an Interactive Performance Prediction Tool

by

Daya Atapattu and Dennis Gannon

May 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

Title: Building Analytical Models into an Interactive Performance Prediction Tool

Authors: Daya Atapattu
Computer Science Department
Indiana University
Bloomington
Indiana 47405
(812)855-5184
daya@iuvax.cs.indiana

Dennis Gannon
Computer Science Department
Indiana University
Bloomington
Indiana 47405
(812)855-5184
gannon@iuvax.cs.indiana

Corresponding Author: Daya Atapattu
Presenting Author: Dennis Gannon

Abstract:
In this paper we describe an interactive tool designed for performance prediction of parallel programs. Static performance prediction, in general, is a very difficult task. In order to avoid some inherent problems, we concentrate on reasonably structured scientific programs. Our prediction system, which is built as a sub-system of a larger interactive environment, uses a parser, dependence analyzer, database and an X-window based front end in analyzing programs. The system provides the user with execution times of different sections of programs. When there are unknowns involved, such as number of processors or unknown loop bounds, the output is an algebraic expression in terms of these variables. We propose a simple analytical model as an attempt to predict performance degradation due to data references in hierarchical memory systems. The predicted execution times of some Lawrence Livermore loop kernels are given together with the experimental values obtained by executing the loops on Alliant FX/8.

Keywords: parallel processing
performance prediction
analytical model

Research reported herein was supported, in part, by the National Science Foundation under grant number CCR-8803432.

Building Analytical Models into an Interactive Performance Prediction Tool

Daya Atapattu, Dennis Gannon
Department of Computer Science, Indiana University
Bloomington, Indiana

May 1, 1989

1 Introduction

As the field of parallel computation has evolved from a theoretical subdicipline of computer architecture and algorithm design to an active branch of experimental computer science it has become increasingly dependent upon performance analysis as the principle tool for explaining the behavior of multiprocessor systems. In particular, researchers working on parallelizing large applications are increasingly dependent on their techniques. While automatic parallelizers for FORTRAN and C are becoming more powerful, they still frequently fail to recognize potential concurrency (especially if it involves interprocedural analysis). When they do fail, it is up to the user to determine where parallelization and vectorization are best applied in the given application.

In this paper we consider the design of an interactive performance predictor that allows users to consult an analytical model of the performance of a machine and the output of the compiler in order to help them better explain the performance of their applications. To see where a tool like the one we will describe fits into the activities of a user, it is best to briefly consider an anatomy of a performance evaluation session for a user with a large application which, for some unknown reason, is failing to get the desired performance on a vector multiprocessor supercomputer. We view the process as a sequence of five levels of analysis.

- *Global View.* At this level the user would like to understand the performance of the application in the large. To understand why the program might be running slowly, a global view of the

code is needed. The appropriate tool here is an accurate parallel profiler that shows the percent of time spent in each routine and speedup statistics on a routine by routine basis. Tools of this type are widely available, but vary in accuracy. GPROF is the most well know example and some vendors provide parallel versions of this tool.

- *Procedural View.* If the programmer desires more information about the performance details of a given procedure or function invocation, a new view is needed that depicts the loop and "if" control structure of the program. Statistics must be gathered to show the percent of time spent in each loop and subroutine call. For some runtime system the overhead for executing concurrent loops is large in relation to typical loop bodies. In loops with heavy synchronization there may be a large cost associated with this overhead. If that is the case one needs an event based trace and a summary to see if the granularity is too small or if too much time is spent in critical sections or waiting for semaphores. Tools that provide this information are now becoming available for parallel systems. Gist for the BBN butterfly [GIST] is a good example, Leech for CSRD Cedar is another.

- *Code Generation View.* It may be the case that bad performance was due to a failure of the automatic parallelization process to do its job. Users need to see where the compiler failed and what sort of code was generated when the compiler managed to do something. In particular, if a loop is 100 or more lines long, it may not be easy to see why the automatic tool failed. Segments of code this long often contain too many complex scalar expressions and subroutine calls to be sorted out by contemporary analyzers. The user is then required to put in directives to command the compiler to parallelize a section of code. One of the main jobs of our system, known as Sigma, is to help the user sort out serious data dependences that prohibit parallelization from those that might confuse the compiler but are not serious. In particular, global flow analysis is essential here.

- *Model Prediction View.* If the speedup for a given program segment was less than was expected, the programmer should be able to ask the system tools to make a prediction of performance for that segment of code. The prediction should be based on a theoretical model

of machine performance applied to the code generated by the compiler. If the predicted behavior agrees with the actual behavior, the user can study the dominant terms in the predicted performance formula to understand why the desired speed-up was not achieved. If the predicted behavior does not agree with the actual behavior, the programmer is then made aware that another factor outside the loop in question (or not covered by the theoretical model) must be influencing behavior.

In this paper we look at the last two items in this list with special emphasis on the process of integrating analytical models of performance into a working software tool. In section 2 of this paper we will describe the design of an object code analyzer and its integration into a tool with a source code interface. In section 3 we will illustrate its behavior by examining the well known Livermore Loops kernels [McMa]. Section 4 gives an outline how an analytical model of bus memory traffic can be incorporated into the predictor and Section 5 will describe extensions for other classes of parallel systems.

2 Object Code Analyzer

The problem of extracting a static performance estimate from a segment of code can be very problematic. For example, if the code contains calls to an unknown subroutine there is nothing we can do. Also if there are data dependent branches we may have no idea of the branch frequency. In our experiment, we have focused on the problem of giving a static analysis of simple loop structures and we have focused our efforts on understanding the behavior of the concurrent/vector execution of these loops. Even here we have a hard problem. The behavior of loop are a function of loop bounds and strides as well as vector lengths and the number of processors. Consequently, any static estimate must be expressed as a mathematical function (or algebraic expression) of these quantities.

We are currently in the process of designing a performance predictor as a part of the Sigma system. Sigma can be viewed as a collection of tools to help programmers with the task of parallelizing FORTRAN and C programs. It consists of

1. Parsers and dependence analyizers for FORTRAN and C.

2. A large data base to store data dependence and other information about a program.

3. A library of program transformations to help in the restructuring of programs.

4. An integrated, interactive front end that allows programmers to access the data base and transformation system via the original source code of the application.

More details can be found in [GGGJ], [GSLA]. In this section we describe the design of the performance estimation tool that is the part of Sigma that is the focus of this paper.

The current implementation of the predictor analyzes fortran programs targeted to the Alliant FX/8. The extension of this tool to analyze the programs targeted to other MIMD architectures is discussed in the section 5.

The Alliant FX/8 is a shared memory vector multiprocessor with from 1 to 8 processors. The processors each have a vector instruction set and a large set of vector registers. In addition there is a powerful concurrency control instruction set that allows very efficient parallel execution of fortran DO loops. The system has a 512KB cache that is shared by all processors. The bandwidth of the cache is well matched to the rate at which processors can access data. The cache is connected to a bus and main memory with a bandwidth equal to about half that of the channel from cache to the processors. The software of the machine consists of an automatic parallelizing and vectorizing fortran compiler. This has the advantage that the programmer does not have to spend time in optimizing the program by inserting compiler directives or library calls as in a system where automatic parallelization is unavailable. The programmer writes the program in standard fortran and compiles with necessary compiler switches. The compiler analyzes data dependences and uses any parallelizing and vectorizing constructs to optimize the program where possible.

However, the compiler optimizations are limited due to several reasons.

- The compiler is restricted by the data structures and algorithms used in the program.

- The compiler has to assume the worst cases on unknown input data or hard to determined data at compile time.

- Automatic parallelizing compilers usually fail to recognize many aspects of the code that influence performance.

The above restrictions imply that the programmer can almost always improve the performance if the necessary information is available. But the automatic parallelization takes the control away form the programmer. Even though one can influence compilation by compiler directives, the very limited knowledge about the final object code usually prevents the programmer from doing so. This is where the assembly code produced by the compiler becomes a good source of information. Assembly code represents the exact sequence of instructions which runs on the target architecture. Some of the critical information can be only obtained at this level. However, the instruction sets of modern parallel processing systems have become rather complicated. The vectorizing/parallelizing techniques are so complex that it will be hard for a programmer to extract any useful information. Even if the programmer does understand the basic optimizations and transformations done by the compiler, performance estimates usually involve more complex analytical analysis of this knowledge.

At the initialization stage, the object code analyzer scans the assembly code and records all the important events that would affect performance. These events include:

- The start and end of sequential loops.
- The start and end of concurrent loops.
- The start and end of vector loops.
- The location of vector instructions.
- Instruction cycle counts.
- Locations and targets of branch instructions.

Cycle counting is more complicated than it appears. In the Alliant FX/8 instructions are pipelined. Certain instruction sequencing can cause the pipeline to stall. These stalls have to be accounted for in computations. The actual timing of a memory reference depends on the location of data in the hierarchical memory. We will only be able to make a probabilistic assumption of

cache hits and misses in this case. As we discuss section 4, determination of the location of data and the estimation of delays due to data movements are complicated factors that we will attempt to resolve in our system.

In our system this collected information is stored in the central data base of Sigma where the assembly language instructions are associated with the program statements of the source. This association is necessary in order to compute certain information about the performance of the program. For example, this data structure makes it is possible for us to compute cpu cycles within loops and display it together with the symbolic loop bound variables used in the source program. This data structure is also very useful in predicting hierarchical memory factors. The memory estimation requires a more global analysis of the program which can be done more easily with the help of the source program and data dependence information.

As an example let us take the Lawrence Livermore loop kernel in Figure 1. Here the do loop is converted to a vector parallel loop by the compiler. This means the loop is split into two nested loops, so that the outer loop is concurrentized over the processors and the inner loop is vectorized within the processors. The Alliant compiler blocks the loop into p different parts (where p is the number of processors) for this purpose. This way the range of each vector loop becomes $(na/p) * pn + 1 : (na/p) * (pn + 1)$, where pn is the processor number. Figure 2 shows the inner vector loop made by the Alliant compiler. This code segment provides most of the information that we need to analyze the loop. Knowing the number of cpu cycles for each instruction and the cycles introduced due to pipeline stalls it is possible to compute the total number of cycles in the loop. Here we obtain the loop bounds by associating the assembly code with the source program. Note that for the purposes of cycle counting it is the length of each vector, na/p, not the actual bounds of the loop, that is important to us. The vector loop executes $(na/p)/32$ times as the Alliant vector instructions handle the vectors of 32 elements. This program structure is reflected in our analyzer output shown in Figure 3. The Figure 3 also shows that the cycles for inner vector loop is multiplied by $(na/\#p)/32$ to obtain the total cycles taken for the loop as we discussed earlier.

To use the performance predictor interactively, the user loads the data base for the application (this requires a pass through the data dependence analyzers and the Alliant compiler to produce an

```
      subroutine llll(na)
      real*8 x,y,z,u,r,t
      common x(0 + 1000),y(0 + 1000),
     +     z(0 + 1000),u(0 + 1000 + 6),r,t
      do 7 k= 1,na
         x(k)=     u(k  ) + r*( z(k  ) + r*y(k  )) +
     +        t*( u(k+3) + r*( u(k+2) + r*u(k+1)) +
     +        t*( u(k+6) + r*( u(k+5) + r*u(k+4))))
    7 continue
      return
      end
```

Figure 1: Source Program

```
llll.label_LE:
       movl d2,d1
       addl a5@(-780),d1
       movl d1,d7
       movl d3,d5
       fmoved fp7,fp0
       vmoved __BLNK__+24000+32:l[d7:l:d],v0
       vmuadd fp0,v0,__BLNK__+24000+40:l[d7:l:d],v0
       vmuadd fp0,v0,__BLNK__+24000+48:l[d7:l:d],v0
       fmoved fp6,fp0
       vmuld v0,fp0,v0
       vmoved __BLNK__+24000+8:l[d7:l:d],v3
       fmoved fp7,fp0
       vmuadd fp0,v3,__BLNK__+24000+16:l[d7:l:d],v3
       vmuadd fp0,v3,__BLNK__+24000+24:l[d7:l:d],v3
       vaddd v0,v3,v0
       fmoved fp6,fp0
       vmuld v0,fp0,v0
       vmoved __BLNK__+8000:l[d7:l:d],v6
       fmoved fp7,fp0
       vmuadd fp0,v6,__BLNK__+16000:l[d7:l:d],v6
       vmuadd fp0,v6,__BLNK__+24000:l[d7:l:d],v6
       vaddd v0,v6,v0
       vmoved v0,__BLNK__+0:l[d7:l:d]
       addl a5@(-796),d2
       vcnt32  llll.label_LE
```

Figure 2: Assembly Code for Inner Loop

```
S: 5  [ cycles = 6 ]
Concurrent Prolog Start
   S: 8  [ cycles = 14 ]
   Concurrent Vector Loop Start k = 1  : na
      S: 13  [ cycles = 24 ]
      Vector Loop Start
         S: 11, V: 14 [ cycles = 636 ]
      Vector Loop End
      S: 1  [ cycles = 3 ]
Concurrent End
S: 3  [ cycles = 7 ]

No of cycles = 54+na/#p/32*636
```

Figure 3: Summary Made by the Analyzer

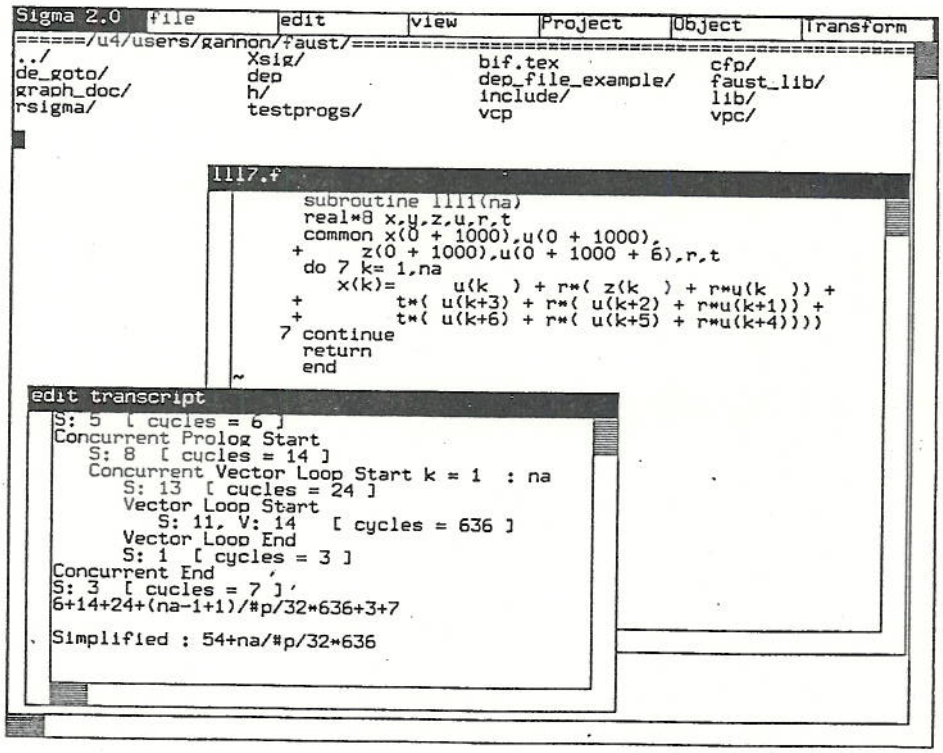

Figure 4: The Analyzer Output in Sigma

assembly listing.) The system gives a list of all the routines in the program and the user then picks the subroutine of interest with the mouse. This generates a view of the text in an editor window. The process of getting an estimate only requires that the user select a block of code with the mouse and make the appropriate menu selection. As shown in Figure 4 a new window is created that displays the summary of the generated code and cycle time estimate as an algebraic expression of the routine parameters (such as loop bounds). As can be seen, the system also goes to some length to simplify the expressions.

3 Experimental Results

In order to investigate the accuracy of the system, we experimented with some of the Lawrence Livermore loop kernels. In section 2 we illustrated the fortran program (Figure 1), assembly language code for the inner vector loop (Figure 2) and the analyzer output (Figure 3) for the loop 1 used in the experiment. Six such loops were used in this this experiment which was done on the 4-processor Alliant FX/8 at the Indiana University. The fortran subroutines used in the experiment are listed in the appendix at the end of the paper.

The following procedure was used to obtain the formulae of the cpu cycles for the loops. The kernels were first compiled using the Alliant parallelizing/vectorizing fortran compiler. We used the compiler switches $-$O (to optimize), $-$AS (to allow the compiler to use associativity) and $-$S (to force the compiler to produce the assembly code). Separately, the fortran sources were parsed by our special fortran parser. Then the assembly code and the dependence graph were fed into the analyzer and the formulae for the whole program was obtained. These formulae are shown in the table 1. Note that "P" indicates the the number of processors in the target machine. The divisions in the formulae should be rounded up to obtain the correct results. Once the number of cpu cycles are obtained, it was multiplied by the clock period, $170ns$ in this case, to obtain the actual time.

These formulae were compared with the actual performance figures of the loops. A driver was written to call the loop as a subroutine. For the purpose of these experiments we selected the arrays small enough so that the program data will completely fit in the cache. The subroutine containing the loop was called in a loop of several hundred repetitions. These repetitions were done to nullify

Loop	*Formula*
1	54+na/P/32*636
2	329+3678*((−1+na)/P/32)
3	79+na/P/32*555
4	145+na/P/32*1131
5	52+na/P/32*120
6	90+(−1+kn)/P*(82+(−1+jn)/32*3076)

Table 1: Formulae for Cycles

the effects of the the $10\mu s$ resolution of the clock used for timing.

The timing function was called immediately before and after the repetitive loop. Another dummy subroutine with no executable statements was called the same number of repetitions to obtain the function calling overhead which is then subtracted. The Alliant command "execute" was used to run the program on one, two and four processors. Actually each execution time was obtained after six runs, discarding the highest and the lowest and averaging the other four. Finally the time was divided by the number of repetitions to obtain the time taken for a loop. The experimental values (in μs) are shown together with the predicted values in the table 2.

4 Incorporating an Analytical Model of Bus Behavior

The experiment in the previous section predicted performance for executions out of the cache. However in Alliant FX/8 the bus is clearly the limiting factor in the performance for that system when executing parallel-vector loops (this fact is well documented in a number of reports [GJMW], [GGJMW]). Consequently any methodology for predicting the performance for the Alliant hardware must include a model of the memory architecture.

In this section we illustrate how a model of memory contention can be incorporated into the prediction software. Our objective is to show that the object and source code information discussed in the previous sections of this paper together with known parameters of the hardware contain enough information to give a first order prediction of memory system performance problems.

The model we use is very simple, and is chosen more for the sake of illustration than as a final

Loop No	*Parameters*	*Procs*	*Estimated(μs)*	*Actual(μs)*	*% error*
1	na=512	1	1739.1	1732.1	0.417
		2	874.14	880.93	0.770
		4	441.66	447.34	1.269
2	na=512	1	10040	10290	2.426
		2	5048.2	5176.8	2.484
		4	2552.1	2641.3	3.378
3	na=512	1	1523.0	1542.1	1.237
		2	768.23	781.46	1.687
		4	390.83	400.79	2.485
4	na=512	1	3101.0	3295.6	5.905
		2	1562.8	1659.8	5.844
		4	793.73	855.71	7.243
5	na=512	1	335.24	336.61	0.406
		2	172.04	176.27	2.399
		4	90.440	94.852	4.652
6	kn=512 jn=5	1	33472	34072	1.761
		2	16743	16871	0.753
		4	8379.6	8549.4	1.986

Table 2: Expeimental results

solution. (We are indebted to William Jalby for suggesting this model as a good test case.) A more interesting, but more complex model can be based on the work of any of the large number of research results in this area (for example [BYA], [KrSn], [SeD], [YPD])

Consider a simple example of the form

```
do i = 1,n
        x(i,1:k) = y(i,1:k)
enddo
```

The Alliant compiler will parallelize the loop and use vector instructions on each processor to do the computation in the loop body. Assuming no references to memory the computation will take

$$\hat{T}_P = C_{ser} + \lceil n/P \rceil (\alpha + \beta \lceil k/32 \rceil)$$

cycles of time where P is the number of processors, C_{ser} is the serial prolog and epilog time and α and β are respectively the amount of scalar and vector code in the loop body. Assume that the cache miss rate of one execution of the loop body is χ and that the total number of references to memory or cache in one pass of the loop is W words. If we make the (very rough) assumption that the effect of cache misses is to make the memory requests to the processor "appear" as a random exponentially distributed process with request rate

$$M_r = \chi \frac{W}{\alpha + \beta \lceil k/32 \rceil}$$

Assume that the bus-memory system is a exponentially distributed random process with a service rate of Ms words per second. In other words, the memory system is either busy or available for a request. (In fact, processors in the FX/8 can each have two outstanding memory requests, but for the sake of this simple discussion, that is not important.)

Let our P processors be considered as a requesting system with request rate PM_r. View the memory system as a server with a queue of length L where L is the number of outstanding requests that can be pending at any time. A classical result for Poisson queues and state dependent arrival and service times is given as follows. Let p_s be the probability that the queue contains s elements,

then

$$p_s = \frac{\rho^s(1-\rho)}{1-\rho^{L+1}}$$

for $s = 1, 2, ...L$ where $\rho = \frac{PM_r}{M_s}$. This result is subject to the following key assumption. If a request is made to the system when the queue is full, the request is dropped. In other words the requesting processor is blocked until the queue has free space and then it continues generating requests as if it had not been blocked. (For a proof see Coffman and Denning [CoDe]). This assumption is inaccurate for our situation, because a blocked processor will issue a new request as soon as possible after there is room in the queue rather than resuming the previous request rate. On the other hand, for the purposes of our interactive system approximations and for the purposes of this discussion it will be sufficient.

Consider the simplest model derivable from these equations, i.e. assume that the queue size L is one so that the memory system is either busy servicing a request or is free and that the processors are only working during the free periods. (In other words, when one processor is stalled, they are all stalled.) In this case, we have the probability that the memory system is busy serving a request is

$$p_1 = \frac{\rho}{1+\rho}$$

and the probability that it is free is

$$p_0 = \frac{1}{1+\rho}$$

The assumption that the processors are only working when the memory system is in the free state implies that, if the parallel section of code requires $TPar_P$ cycles of work to complete, the total number of cycles to complete the execution will be

$$Ttotal_p = TStall_P + TPar_P$$

where $TStall_P$ is the time spent in the stalled state waiting for the memory system. From the probabilities above we can expect $TStall_P$ to be $p_1 Ttotal_p$ and $TPar_P$ to be $p_0 Ttotal_P$. But

$TPar_P$ is just

$$\lceil n/P \rceil (\alpha + \beta \lceil k/32 \rceil)$$

If we add in the execution time for the serial section of code then execution time in cycles T_p to complete the job satisfies

$$T_P = C_{ser} + (1+\rho)(\lceil n/P \rceil (\alpha + \beta \lceil k/32 \rceil))$$

or

$$T_P = C_{ser} + (1 + \frac{PM_r}{M_s}) \lceil n/P \rceil (\alpha + \beta \lceil k/32 \rceil)$$

Another way to say this is that the speedup of the parallel section is bounded by

$$speedup = \frac{P(1+\frac{M_r}{M_s})}{1+\frac{PM_r}{M_s}}$$

Clearly a more accurate model can be built by considering the case where the queue size reflects the number of memory request the system may have pending and where the processors are only stalled when a request is in the queue, but the important point is that we have built an estimate of multiprocessor performance that is a function of the following terms

- C_{ser}, α and β which are the coefficients derived from object code analysis.
- M_s which is the memory service rate
- P which is the number of processors, n the outer loop bound and k the the vector length.
- M_r which is a function of the above items and the cache miss ratio χ.

Of course, the key missing link to generating performance estimates for different values of n, k and P is the computation of χ. For the example at the beginning of this section, it is easy to see that χ is 1 because each element is referenced only once.

In a more interesting scenario, some items will be fetched from cache and some from memory. For example, in the matrix vector product

```
do i = 1, n
     do j = 1, m
          y(j) = y(j) + a(i,j)*x(i)
     enddo
enddo
```

In this case, we are reading the vectors x, y and the array a and writing the array y. The total amount of data moving from the cache to the processors is then $4mn$.

To compute the value of χ accurately we need to know the cache replacement policy of the machine. In the case of the Alliant, the cache is a direct mapped cache controlled by a write back scheme on dirty words. In other words, a word is loaded into cache position equal to it address modulo the cache size. Consequently, it is possible that $y(1)$, $a(1,1)$ and $x(1)$ might be all mapped to the same position in the cache and almost all references will be cache misses. On the other hand, if we assume that this anomaly is rare, we can build a simple model of the behavior. In fact, from the perspective of building a performance predictor we would like an optimistic model if an accurate one is not available. The reason for this claim is that we assume that this tool is being used to try to explain bad performance to an application programmer. If a performance predictor yields a pessimistic result based on an optimistic analysis, we can feel safe that the prediction is guiding the programmer to the correct conclusions about his code. If, on the other hand, the prediction claims that performance should be good when, in fact, it is poor, the programmer well advised to turn to more accurate runtime performance analysis tools. With this (admittedly arguable) justification, let us make the optimistic assumption that the cache is smart in the following sense.

Define the *reference window* for an array at a given instance in time to be the set of elements that will be accessed prior to a repeated access to the element of the array. In other words, the reference window is like a *working set* for the array. In the example above, the reference window for x at time (i,j) is $x(i)$. Consequently, the window is a single element and is stationary until i changes. In the case of y we see the reference window is $y(1:m)$ and is fixed. Elements of a are accessed only once and never reused in the loop, so the reference window for a is empty. Let us assume that the cache is smart enough to keep the reference window for each array, and that it

adds new elements and drops old ones whenever the window shifts position in the iteration space. If the total of the sizes of the reference windows exceeds the cache size, we assume that the least recently used policy is employed to attempt to fit the elements of the reference windows. Hence, if the total is less than the cache size, any reference to an element of the current window for an array can be considered as a cache hit and any reference to another element is a miss. The total of the sizes of the reference windows in our example is $1 + m$. Let CS be the size of the cache. If we ignore the first few outer iterations, the number of cache misses in the case that $1 + m < CS$ is just mn corresponding to the references to a. If $1 + m > CS$ then there is not enough room in the cache. According to our least recently used assumption, the element of x will remain in cache (in fact, a good compiler will keep it in register), but only $CS - 1$ of the elements of a will be in cache and $m - CS - 1$ elements will have to be loaded for each outer iteration. Putting all this together with the fact that the total number of references is equal to $4mn$ we have

$$\chi = \begin{cases} \frac{1}{4} & \text{if } 1 + m < CS \\ \frac{2m - CS - 1}{4m} & \text{otherwise} \end{cases}$$

In summary, we have reduced the estimation of the memory system performance to the estimation of the cache miss ratio which we have reduced to the symbolic identification of the reference windows. This last task is something we have already studied extensively and it is described in [GaJG87] and [GGJ88]. One further point must be made. An important factor of cache behavior not considered here is the effect of the cache line size on the estimate. This is particularly important when memory references are not sequential and a much more detailed analysis is needed in this case.

5 Extending to Other Architectures

Our experiments, and the tools discussed in this paper concentrated entirely on the architecture and the compiler of the Alliant FX/8. However, the general concepts discussed in this paper are applicable to many shared memory machines. The assembly language analyzing techniques that we used are very general. For each different machine a set of regular expressions would have to be

written to capture the different types of instructions and compute the cycles. In fact in our software we were careful to keep this regular expressions as a separate module and create a general data structure containing important events for analysis. This basically allows us to proceed in a machine independent way once the events are recorded. The overall structure of the object code will be less useful when the program is explicitly parallelized and vectorized by the user. In this case, the user already knows the overall structure. However, the analyzer can point out the deficiencies of the programmer's decisions. For example, the programmer may not have an accurate knowledge of the overheads involved in parallelization and be lead into making a poor decision. The delays due to synchronizations, pipeline and other hardware facilities are always complex they are best handled by programs like ours sometimes taking probabilistic approaches.

Extensions of our theories and concepts of the memory system timing will be admittingly less straight forward. By studying and developing tools to analyze the memory system in the Alliant we cover most of the hierarchical cache based system. Note that our analysis in section 4 has a very general view of the overall memory systems. It will be necessary that some of the architecture specific parameters such as cache line size to be included in the implementation. However, as long as the memory system of target machine fits our model the concepts should be valid and easy to implement. Some of the other machines, however, would have different memory systems which cannot be modeled by ours. In the BBN Butterfly for example, the processors are connected to the memory by a network. Each processor also has a part of the memory which is "local" to it. The processors can access local memory without network delays. What is kept in the local memory is decided by the programmer. Our experience with the butterfly shows that the decision of "what to keep in local memory" is not a trivial one for the programmer. The "hot spot contention" [PHNO] is another factor which degrades the performance in such a network connected system. Even though, there is no obvious extension to our memory system studies to cover such architectures, it is evident that our basic prediction system can easily accommodate a different model in this case. Then the predictor can be used as a guide by the programmer in localizing variables and avoiding the hot spots.

6 Conclusions

As we pointed out earlier, a performance prediction tool will greatly enhance the user's ability to tune programs to exploit the architectures of today's high speed supercomputers. We have described the design of such a system which combines the object code and memory system analysis. The results of our object code analyzer are presented in section 3. The estimated times are within 8% of the values obtained in all the cases. Actually, our estimate was always optimistic suggesting that the incorporation of the proposed memory modeling could even correct the discrepancies.

To include the delays due to hierarchical memory and the memory bus saturation, we proposed a probabilistic model. How well the model fits into the system depends on the assumptions made in the derivations. This model will be incorporated to our system shortly, and the accuracy will be verified experimentally. We will also be including the effects due to the cache line.

We will also be concentrating on extending our results to different architectures. It is important that our basic concepts are general enough to facilitate these extensions. We believe that the techniques used in our object code analysis and memory modeling are general enough to apply to different architectures with minimal effort.

References

[BYA] Bhuyan, L, Yang, Q., Argawal, D, *Performance of Multiprocessor Interconnection Networks* IEEE Computer, Feb. 1988, pp.25-37.

[CoDe] Coffman, E., Denning, P, *Operating Systems Theory*, Prentice-Hall, 1973

[GGGJ] Gannon, D, Guarna, V., Gaur, Y., Jablonowski, J., *FAUST: An Environment for Programming Parallel Scientific Applications* Proceedings Supercomputing 88, Nov. 14, Orlando, Florida, 1988, pp. 3-11. IEEE, ACM SIGARCH.

[GSLA] Gannon, D., Shei, B, Lee, M., Attapatu, D, *A Software Tool for Programming Parallel Systems* Parallel Processing for Scientific Computing, Rodrigue ed., SIAM, 1987.

[GaJG87] 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.

[GGJ88] 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.

[GGJMW] 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.

[GIST] *Gist - a performance monitor* in "Software tools for Mach100", BBN ACI, 1987.

[KrSn] Kruskal, C and Snir, M., *The Performance of Multistage Interconnection Networks for Multiprocessors* IEEE Trans. Computers, Vol. C-32, Dec 1983, pp. 1091-1098.

[GJMW] Gallivan, K., Jalby, W., Malony, A., Wijshoff, H., *Performance Prediction on the Alliant FX/8* Technical Report, Center for Supercomputing Research and Development University of Illinois at Urbana Champaign Urbana, Illinois 61801

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

[SeD] Sethi, A., and Deo, N., "Interference in Multiprocessor Systems with Localized Memory Access Probabilities," IEEE Trans. on Comp., vol. C-28, no. 2, Feb 1979.

[YPD] Yen, D., Patel, J., Davidson, E., *Memory Interference in Synchronous Multiprocessor Systems* IEEE Trans. Computers, Vol. C-31, Nov. 1982, pp. 1116-1121.

[PHNO] G. Phister and A. Norton. Hot spot contention and combining in multistage interconnection networks. In *Proceedings of International Conference on parallel Processing*, pages 790–797, 1985.

APPENDIX

```
      subroutine llll(na)
      real*8 x,y,z,u,r,t
      common x(0 + 1000),y(0 + 1000),
     +     z(0 + 1000),u(0 + 1000 + 6),r,t
      do 7 k= 1,na
         x(k)=     u(k  ) + r*( z(k  ) + r*y(k  )) +
     +        t*( u(k+3) + r*( u(k+2) + r*u(k+1)) +
     +        t*( u(k+6) + r*( u(k+5) + r*u(k+4))))
    7 continue
      return
      end
```

Loop No: 1

```
      subroutine llll(na)
      real*8 du1,du2,du3,u1,u2,u3,sig,a11,a12,a13,
     +     a21,a22,a23,a31,a32,a33
      common du1(0 + 1000),du2(0 + 1000),
     +     du3(0 + 1000),
     +     u1(5,1000 + 1,2), u2(5,1000 + 1,2), u3(5,1000 + 1,2),
     +     sig,a11,a12,a13,
     +     a21,a22,a23,a31,a32,a33
      nl1 = 1
      nl2 = 2
      do  8     kx = 2,3
cvd$  nodepchk
         do  8     ky = 2,na
            du1(ky)=u1(kx,ky+1,nl1)  -  u1(kx,ky-1,nl1)
            du2(ky)=u2(kx,ky+1,nl1)  -  u2(kx,ky-1,nl1)
            du3(ky)=u3(kx,ky+1,nl1)  -  u3(kx,ky-1,nl1)
            u1(kx,ky,nl2)=u1(kx,ky,nl1) +a11*du1(ky) +a12*du2(ky)
     +           + a13*du3(ky) + sig*(u1(kx+1,ky,nl1) -2.*u1(kx,ky,nl1)
     +           + u1(kx-1,ky,nl1))
            u2(kx,ky,nl2)=u2(kx,ky,nl1) +a21*du1(ky) +a22*du2(ky)
     +           + a23*du3(ky) + sig*(u2(kx+1,ky,nl1)
     +           - 2.*u2(kx,ky,nl1) +u2(kx-1,ky,nl1))
            u3(kx,ky,nl2)=u3(kx,ky,nl1) +a31*du1(ky) +a32*du2(ky)
     +           + a33*du3(ky) + sig*(u3(kx+1,ky,nl1)
     +           - 2.*u3(kx,ky,nl1) +u3(kx-1,ky,nl1))
 8    continue
      return
      end
```

Loop No: 2

```
      subroutine llll(na)
      real*8 px,co,dm22,dm23,dm24,dm25,dm26,dm27,dm28
      common px(0+ 25, 1000),co,dm22,dm23,dm24,dm25,dm26,dm27,dm28
      do 9  i = 1,na
         px( 1,i)= dm28*px(13,i) + dm27*px(12,i) + dm26*px(11,i) +
     +        dm25*px(10,i) + dm24*px( 9,i) + dm23*px( 8,i) +
     +        dm22*px( 7,i) +  c0*(px( 5,i) +      px( 6,i))+ px( 3,i)
    9 continue
      return
      end
```

Loop No: 3

```
      subroutine llll(na)
      real*8 cx,px,ar,br,cr
      common cx(0+25, 1000),px(0+25, 1000),ar,br,cr
      do 10  i= 1,na
         ar      =      cx(5,i)
         br      = ar - px(5,i)
         px(5,i) = ar
         cr      = br - px(6,i)
         px(6,i) = br
         ar      = cr - px(7,i)
         px(7,i) = cr
         br      = ar - px(8,i)
         px(8,i) = ar
         cr      = br - px(9,i)
         px(9,i) = br
         ar      = cr - px(10,i)
         px(10,i)= cr
         br      = ar - px(11,i)
         px(11,i)= ar
         cr      = br - px(12,i)
         px(12,i)= br
         px(14,i)= cr - px(13,i)
         px(13,i)= cr
   10 continue
      return
      end
```

Loop No: 4

```
      subroutine llll(na)
      real*8 x,y
      common x(0 + 1000),y(0 + 1000)
      x(1)= y(1)
      do 11 k = 2,na
 11      x(k)= x(k-1) + y(k)
      return
      end
```

Loop No: 5

```
      subroutine llll(na)
      real*8 za,zb,zp,zq,zr,zu,zv,zz,zm,t,s
      common za(0 + 1000,7),zb(0 + 1000,7), zp(0 + 1000,7),
     +     zq(0 + 1000,7), zr(0 + 1000,7),zu(0 + 1000,7),
     +     zv(0 + 1000,7),zz(0 + 1000,7), zm(0 + 1000,7),s,t
      t= 0.0037
      s= 0.0041
      kn= 6
      jn= na
      do 70  k= 2,kn
         do 70  j= 2,jn
            za(j,k)= (zp(j-1,k+1)+zq(j-1,k+1)-zp(j-1,k)-zq(j-1,k))
     +           *(zr(j,k)+zr(j-1,k))/(zm(j-1,k)+zm(j-1,k+1))
            zb(j,k)= (zp(j-1,k)+zq(j-1,k)-zp(j,k)-zq(j,k))
     +           *(zr(j,k)+zr(j,k-1))/(zm(j,k)+zm(j-1,k))
 70   continue
c
      do 72  k= 2,kn
         do 72  j= 2,jn
            zu(j,k)= zu(j,k)+s*(za(j,k)*(zz(j,k)-zz(j+1,k))
     +           -za(j-1,k) *(zz(j,k)-zz(j-1,k))
     +           -zb(j,k)   *(zz(j,k)-zz(j,k-1))
     +           +zb(j,k+1) *(zz(j,k)-zz(j,k+1)))
            zv(j,k)= zv(j,k)+s*(za(j,k)*(zr(j,k)-zr(j+1,k))
     +           -za(j-1,k) *(zr(j,k)-zr(j-1,k))
     +           -zb(j,k)   *(zr(j,k)-zr(j,k-1))
     +           +zb(j,k+1) *(zr(j,k)-zr(j,k+1)))
 72   continue
c
      do 75  k= 2,kn
         do 75  j= 2,jn
            zr(j,k)= zr(j,k)+t*zu(j,k)
            zz(j,k)= zz(j,k)+t*zv(j,k)
 75   continue
      return
      end
```

Loop No: 6