1990

A Performance Predication Model for Parallel Compilers

Ko-Yang Wang

Report Number:
90-1041

A PERFORMANCE PREDICTION MODEL FOR PARALLEL COMPILERS

Ko-Yang Wang

CSD-TR-1041
November 1990

A Performance Prediction Model For Parallel Compilers †

Ko-Yang Wang

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

In this paper, we introduce a performance prediction framework for parallel compilers based on the *refined combined characterization model (CCM).* The CCM model characterizes the performance of a user program on a target machine by combining a set of characteristic functions called performance factors. Each performance factor represents a particular program aspect that relates program patterns to features of a target architecture and is quantified by an evaluation function. The evaluation functions are usually inexpensive to compute, so they can be evaluated repeatedly during the parallelism optimization process. The performance prediction framework is highly flexible and can be adjusted with a knowledge base to fit different needs at different stages of parallel compiling and to accommodate different classes of parallel architectures. The performance prediction model is utilized in an intelligent parallel compiler to guide the program optimization process. Its versatility allows the compiler to offer an adjustable optimization degree and can optimize code for a wide range of target machines.

November 30, 1990

† This work was supported in part by AFOSR 88-0243, ARO grant DDAG29-83-K-0026 and NSF grant CCF-8619817.

A Performance Prediction Model For Parallel Compilers †

Ko-Yang Wang

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

1. Introduction

Optimizing parallel compilers use heuristics and program transformation techniques to restructure program structures to improve the parallelism of the user programs. One key element in the decision making process of parallel compilers is to correctly assess effects of a transformation on a program or the performance of a program on a target machine. This process, call the *performance prediction* involves evaluating how the characteristics of a program match the constraints of an architecture and using this match to estimate the performance of user programs on a target architecture. The goal in building a performance prediction model is twofold:

1. To complete an analytical model for estimating the performance of different classes of parallel architectures with high accuracy and efficiency.
2. To define a template for implementing this analytical model and use it to estimate the performance of user programs on target architectures, and integrate this framework into the decision making process of the program optimization process.

The performance prediction module in an intelligent parallel compiler has to be inexpensive, accurate and flexible. It has to be inexpensive because the compiler uses it repeatedly to evaluate the merits of the applicable transformations during the program restructuring stage. It needs to be accurate because proper decisions of the compiler rely on a correct estimation of the performance. A mechanism for tuning the performance prediction model to suit different parallel architectures is needed so that the model can be applied to different classes of parallel architectures.

1.1. Performance Analysis and Performance Prediction

Performance analysis means understanding the behavior of a computer as a complete system and as a collection of subsystems. It involves both the theoretical models of machine behavior and the experimental analysis of the real hardware to examine whether the implementation of an

† This work was supported in part by AFOSR 88-0243, ARO grant DDAG29-83-K-0026 and NSF grant CCF-8619817.

architecture fulfills the goals of its original design. *Performance prediction* must be concerned with functions with real values in a formula to project to the real execution time of a program on the target machine instead of asymptotic complexity analysis used in the performance analysis. On the other hand, it needs to have analytical models of the hardware that are both accurate and easy to relate to algorithmic properties of the computation [BWJALG90]. Efficient performance analysis models can be utilized to provide such analytic models under the performance prediction framework.

2. Models of Performance Prediction

The execution time of a program on an architecture can be estimated based on the *simulation* or *characterization* model. The simulation model estimates the performance of the program by simulating or profiling the execution of the program or interpreting the execution-time on a computational model of the architecture. The accuracy of the prediction depends on how faithful the computational model is in simulating the actual hardware and how much program-dynamism the program has. The characterization model characterizes the performance of the machine by certain aspects of the machine.

In this section, we discuss a characterization model called *the combined-characterization model*. This model utilizes a set of easy-to-compute functions called *performance factors*. Each *performance factor* represents a particular aspect of the performance of a program on a target machine and is quantified by a corresponding function called the *evaluation function*. Variations of the evaluation function model that use certain formula to quantify the performance of programs have been applied to Parafrase and many other parallel compilers. However, most of these applications use fixed equations to compute the expected program performance. This has the following problems:

1. *Not suitable for a wide variety of architectures.* Most models don't take the architecture variations into consideration. Parafrase [Husm86] use several different parameters for computing program execution time for seven different virtual shared-memory machines, but the number of parameters that are considered are too few to be accurate for real parallel architectures.
2. *Can't deal with programs that contains variables in control flows.* Run-time tests are essential for programs that have dynamic behaviors. Existing performance prediction model does not have provision to help to decide what run-time tests are needed. This often leads to excessive run-time tests.
3. *Inflexible for different needs at different stages of parallel compiling.* More critical part of the program requires more accurate but expensive performance estimations. Existing performance prediction model are inflexible to accommodate to different objectives of the program

optimization.

The framework we proposed in section 4 solves these problems by providing highly flexible control. The framework, which is called *refined combined characteristic* model, is augmented with a knowledge base which can dynamically selects evaluation functions to apply based on features of the machine and objectives of the program optimization. Different weights can be given to the evaluation functions so that their effects can be adjusted dynamically. Inexpensive performance evaluation functions are combined into a single-value real function as an indication of the performance of the program on a parallel architecture and to compare representations of semantically equivalent programs.

2.1. Performance Issues of a Program

The execution time of a program on a processor can be broken down into four categories: computation cost, data synchronization cost, control synchronization cost, and control overheads. The computation cost is the time the processor spends in doing the computation. The data synchronization cost is the length of time that a processor is forced to idle while waiting for needed data to arrive. Two sources of the data synchronization are local data accesses that cannot be overlapped with the computation and the synchronization introduced by the data dependence between a pair of statements that are executed by different processors. The cost of a data reference can be broken down into the cost of accessing the data from or to the memory (memory access cost) and the cost of transmitting the data through the network or bus (data transition cost). The control synchronization is the time it takes to synchronize the control between two or more processors (semaphores, barrier synchronization, conditional or unconditional branch statements, etc.). The control synchronization is usually introduced by explicit control structures, while the data synchronization is normally defined implicitly by the data dependence relations. The control overhead is the overhead for concurrent execution, this includes vector startup time, process startup time, and dynamic task scheduling time, etc.

Based on the above discussion, the execution time, T, of a program structure can be defined as:

$$T = T^c + T^{ds} + T^{cs} + T^{co} = T^c + T^{da} - T^{do} + T^{cs} + T^{co.}$$

Where T^c is the computation cost, T^{ds} is the data synchronization cost (which is the difference of the data access cost (T^{da}) and the data overlapping time (T^{do})), T^{cs} is the control synchronization cost, and T^{co} is the control overhead.

When the evaluation functions are applied to compound statements, they are usually applied to elements in the compound statement recursively and then the results are accumulated to obtain the final value for the compound statement. The execution time to execute a sequence of statement

blocks is the sum of the execution time of each of the statement blocks. A notable exception is the sequential execution time of a conditional statement which is defined to be the *expected value* of the execution time of the conditional branches. That is:

$$T(S) = p(S^T) * T(S^T) + p(S^F) * T(S^F).$$

Where S is the conditional statement, S^T and S^F are the true and false branches of S, and $p(S^T)$ and $p(S^F)$ are the probabilities for these branches to be taken. The execution time for a statically scheduled parallel-loop is defined to be the maximum of the execution time of each of the p instances of the loop. Similarly, the parallel execution time of a set of tasks spread over p processors is the maximum execution time over the p threads of tasks.

Typically, the definition of an evaluation function has two parts: evaluation of a non-compound statement and the accumulation of the results of applying the function on a compound statement. The basic framework for accumulating results of applying an evaluation function recursively on children of a compound statement is shown in procedure 1. By changing the definition of $F(S)$ in the line marked with (*) (the evaluation part) in the framework, different evaluation functions can be defined.

Procedure 1: Template for combining values of applying an evaluation function recursively to elements of compound statements:

$$f(S) = \begin{cases} \sum_{i=1}^{n} f(S_i) & \text{if } S = \{S_1, S_2, \cdots, S_n\} \\ \max_{i=1}^{n} f(S_i) & \text{if } S = \{S_1 \mid S_2 \mid \cdots \mid S_n\} \\ \sum_{i=l}^{u} f(S(i)) & \text{if } S = \text{for } (i = l \,..\, u)\ S(i)\ \textit{endfor} \\ \max_{i=l}^{u} f(S(i)) & \text{if } S = \textit{parfor } (i = 1 \,..\, u)\ S(i)\ \textit{endfor} \\ p(S_T)*f(S_T) + p(S_F)*f(S_F) & \text{if } S = \text{if } (\textit{cond})\ S_T\ \textit{else}\ S_F\ \textit{endif} \\ f(S_{fb}) & \text{if } S \textit{ is a function and } S_{fb} \textit{ is its body} \\ F(S) & \textit{otherwise} \quad \ldots\ldots\ldots (*) \end{cases}$$

Where $p(S)$ stands for the probability that the condition statement will branch to statement S and $S(i)$ stands for the $i-th$ instance of the loop statement S. Also, $\{S_1 \mid S_2 \mid \cdots \mid S_n\}$ denotes that the statements S_i are to be executed concurrently.

2.2. Examples of Performance Factors and Their Evaluation Functions

The evaluation functions defined by the performance factors map the analytical match of a program and a machine into numerical values. Below we list some sample performance factors and their corresponding evaluation functions. Each of these performance factors represents a particular aspect of the performance of the program. In the next section, we will discuss methodologies for integrating these factors into a framework for predicting program performance on target machines.

• *Statement count:*

The statement count characterizes the algorithm by counting the number of statements in each of the control threads. The execution time of the program may be estimated by multiplying by a constant called the *average_statement_cost* with the statement count. The statement count can be computed by replacing $F(S)$ in procedure 1 by

$F(S) = 1$ if S is not a compound statement

• *Operation count:*

A more accurate estimation, called operation counts, can be computed by counting the number of operations instead of just the statements in the control threads. An evaluation function for computing the operation count $f()$ for a program fragment S can be obtained by replacing the line marked with (*) in procedure 1 with the following definitions:

$$F(S) = \begin{cases} f(expr) + 1 & \text{if } S = operand\ op\ expr \\ \alpha^v + \beta^v * n & \text{if } S \text{ is a vector operation of length } n \\ 0 & otherwise \end{cases}$$

Where α^v and β^v are constants representing the vector startup time and unit time using the cost of a floating point operation as units.

The estimated program execution time can be obtained by multiplying the operation count with a constant called *average_operation_cost*. For RISC processors, this constant can be computed easily at the assembly instruction level since most manufacturers have studied and made assumptions about the distribution of operations in the target applications. On the other hand, for CISC processors or for operations at the language level, the variation in the execution time of operations can be quite large, as is the discrepancy between the estimated operation cost and the actual cost. An improvement to this approach is to classify the operations into groups of different costs and use different average-time estimations for each operation group. For example, integer operations often take less time than floating point operations, additions usually take less time than divisions, and array references usually take several integer operations to compute their addresses. A even more detailed estimation can be done by using the actual cost of each of the instructions.

For example, the machine knowledge base may record the costs of floating-point multiplying, integer addition, array address calculation, startup and element time for *vector add* operations, and startup time and element time for triadic operations, etc. Note that the costs of some operations (especially floating point operations) on certain machines take variable time depending on the operands, so estimations of the average costs of these operations are still needed.

• *Number of memory loads and the number of memory stores.*

The number of memory loads and stores are two special types of operation counts that we would like to discuss separately. The unit cost for memory load and store operations are usually constant (C^{load} and C^{store}) for a particular configuration of the machine. So the cost of memory access may be estimated by counting the number of memory load and store operations. For machines with a multiple level of memory hierarchy, the number of global memory loads, global memory stores, cluster memory loads, cluster memory stores, local memory loads, and local memory stores should be counted separately since their costs are usually different. Each number in the above categories can be computed based on the framework specified in equation (1).

For machines that support block-accesses, the memory load and store costs for referencing a data block of size n, $T^{load}{}_{block}$ and $T^{store}{}_{block}$, can be computed by:

$$T^{load^t}{}_{block}(n) = C^{load^t}{}_{start} + C^{load^t}{}_{unit} * n$$

$$T^{store^t}{}_{block}(n) = C^{store^t}{}_{start} + C^{store^t}{}_{unit} * n$$

Where the superscript t in above formula is either g, c, *or* l, which stands for global, cluster, and local memory, respectively. Also, $C^{load^t}{}_{start}$, $C^{load^t}{}_{unit}$, $C^{store^t}{}_{start}$, and, $C^{store^t}{}_{unit}$ are constants for the starting cost for block-load, unit cost for block-load, starting cost for block-store, and unit cost for each block-store, respectively.

• *Cache hit ratio.*

The cache hit ratio of an array is the ratio of the references to the elements of the array that are already in the cache. It represents the degree of the locality of the array in the program and can be used not only in deciding cache allocation problems for architectures that have cache, but also in utilizing block-access operations in machines that support them. To compute the cache hit ratio, we define the image in the computation of an array x of dimension d, $Im(x)$ as:

$$Im(x) = \left\{ w \in Z^d \mid x[w] \text{ is referenced in the computation} \right\}$$

Where Z^d is the bounded integer interval space of the indices of x. Assuming that R is the total number of references to array x in the program, the cache hit ratio is defined as [GaJaGa87]:

$$hit(x) = \frac{R - |Im(x)|}{R}$$

For example, in the following program fragment

```
for i := 1 .. N loop
    for j := 1 .. M loop
        for k := 1 .. L loop
            a[i,j] := b[i,k] * c[k,j];
        end for
    end for
endfor
```

the number R, which is the total number of references to array c, is equal to $N * M * L$. Furthermore, $Im(c) = \{(k,j): 1 \le j \le M,\ 1 \le k \le L\}$ and $|Im(c)| = M * L$. So the cache hit ratio for the array c in the program fragment is:

$$hit^{L_1}(c) = \frac{N * M * L - M * L}{N * M * L} = \frac{N-1}{N}$$

On the other hand, if we focus on the loop j, then $R = M * L$ and $Im(c)$ is the same, so the cache hit ratio of array c in loop j becomes 0.

• *Data transmitting cost.*

The time it takes to transmit a set of data of size n over the network or bus can be computed by the following formula:

$$T^{dt}{}_{block}(n) = \alpha + \beta * n$$

Where α and β are the constants for start up cost (for hand-shaking, setting up routing connections, etc) and unit transmitting cost. For distributed computers such as hypercube where data may need to be transmitted by several hops through several processors, these two constants α and β can be computed by the following formula:

$$\alpha(hops) = \alpha^0 + \alpha^1 * hops$$

$$\beta(hops) = \beta^0 + \beta^1 * hops$$

Where *hops* is the number of hops along the path. More precisely, the constant α^0 for a distributed machine actually includes the time for a zero-length send and a zero length receive; the constant β^0 is the time for the sending task to access the data and the time for the receiving task to store the data. As an example, these constants for NCUBE/1 and NCUBE/2 are shown in the following table. Therefore sending a one word message to a neighbor takes 461.75 microseconds on NCUBE/1 and 160.2 microseconds on NCUBE/2. And sending a 1000-word message across a 128 node cube takes 37362.0 microseconds on NCUBE/1 but 12971.4 microseconds on NCUBE/2.

Time (in microseconds)						
MACHINE	α^0	α^1	β^0	β^1	mach. cycle	fl. add
NCUBE/1	261	193	4.25	4.5	0.137	2.47-3.84
NCUBE/2	156	2.2	0.2	1.8	0.05	0.35

Figure 1. Constants for computing message transmitting cost on NCUBEs.

• *Voided data-prefetch ratio.*

For architecture that supports data-prefetch, most data accesses can be overlapped with the computation if they are not void by conditional or unconditional branch statements. Therefore, the cost of data synchronization is a function of the ratio of prefetched data that are voided by the control statements. A simple heuristic to estimate this ratio is to count the number of statements, N^{stmt}, in the statement block following a branch, and assume that all but one of them are benefited by the data prefetching. This gives us a ratio which is $\frac{1}{N^{stmt}}$. And the data synchronization cost can be estimated as:

$$C^{ds}(S) = C^{da}(S) * p^{vdp}(S) = C^{da}(S) * \frac{1}{N^{stmt}}$$

As the formula shows, the longer a non-interrupted statement block is, the more data access can be overlapped with the computation.

• *Ratio of data access overhead.*

This function characterizes the ratio of the non-overlapped cost of data access over the total execution time. The difference between this ratio and void data-prefetch ratio is that the first ratio is the data access overhead over the total execution time of the statement, whereas the second ratio is over the total data access time of the data in the statement. This means that this factor is used in case the computation of data access time is to be avoided. Of course, the non-overlapped cost of data access depends on the architecture (has prefetching mechanism, has data cache, etc.) and the program (density of the local/external memory references, distribution of the external references, etc) and is hard to be estimated without computing the data access cost. This evaluation function is under the assumption that that the cost of data access is proportional to the cost of computation which is not always true though. We may estimate this cost by analyzing a set of patterns of the program structure and find the ratio of data access that can be overlapped with the computation, then use these patterns as the basis for interpolating non-overlapping data access cost for general programs. With this ratio, the data synchronization cost, C^{ds} over program region S, becomes:

$$C^{ds}(S) = C(S) * p^{ds}(S).$$

where $C(S)$ is the cost for computing in S, and $p^{ds}(S)$ is the ratio of the data access overhead in S.

• *Do-across delay.*

A good measure for data synchronization cost is the do-across delays. Do-across delay is the artificial idle-time in the shared-memory environment that the compiler inserts into a do-across loop to minimize the load (the hot-spot effect) that a busy-waiting processor induced on the memory, bus, or network in checking the global synchronization data. This delay represents the expected data synchronization cost of the loop instance and is the minimum delay that satisfies all control (condition and unconditional branching statements) and data dependence in the parallel loops. The do-across delay of a parallel loop is computed by finding the maximum static time interval between the pairs of statements involved in lexically backward data dependences in the loop and then spreading the delays into loop instances [Cytron84]. Given the do-across delay d_{L_i} for loop L_i, assuming that the loop has N loop instances, the estimated execution time for distributing the do-across loop over p processors is given in the following formula:

$$T = \begin{cases} (N-1)*d_{L_i} + T_{L_i} & \text{if } T_{L_i} \le p * d_{L_i} \\ \left\lfloor \frac{(N-1)}{p} + 1 \right\rfloor * T_{L_i} + mod(N-1, p) * d_{L_i} & otherwise \end{cases}$$

• *Do-across parallelism degree.*

Do-across parallelism degree [Cytron84] is the reciprocal of the ratio of the do-across delay over the actual execution time of the loop body. Assume that T_{L_i} is the execution time of the body of loop L_i and d_{L_i} is the do-across delay for loop L_i, then the do-across parallelism degree of the loop can be computed as:

$$Para^{doacross} = 1 - \frac{d_{L_i}}{T_{L_i}}$$

It follows that given the parallelism percentage, the do-across delay can be computed by the following formula:

$$d_{L_i} = (1 - Para^{doacross}) * T_{L_i}$$

Doacross parallelism degree is an indication of the parallelism presented in the do-across loop. The better the computed execution time approximates the real execution time, the better the do-across parallelism degree is as an index of the parallelism.

• *Number of data synchronization points.*

Synchronization adds overhead and may cause processors to idle but is needed to enforce data dependences that cross the task boundaries (to be more precise, for those dependence that cross processor boundaries). To estimate the effect of synchronization on the proposed task scheduling, the number of cross processor data dependences can be used. This number can be collected by simply counting the cross-task data dependences and eliminating those among tasks that are assigned to the same processor.

• *Uniformness of the execution time.*

Let $\{P_i \mid i = 1 \,..\, n\}$ be a set of n tasks, $T(P_i)$ the estimated execution time of the tasks, T^{max} be the maximum of the estimated execution time, and T^{mean} be the mean of the estimated execution time. The uniformness of the execution time is defined to be:

$$Unif = 1 - \frac{\sum_{i=1}^{n} |T(P_i) - T^{mean}|}{n * T^{\max}}$$

For tasks that have more uniform execution time, load balance is easier to achieve and static scheduling can be used. On the other hand, for tasks whose execution times vary, dynamic scheduling may be better.

• *Hot-spot percentage.*

A hot-spot is a module in the multi-stage blocking network that has sufficient concentration of the traffic. The non-uniform network traffic of hot-spots can produce effects (called tree saturation) that severely degrade *all* network traffic [PfNo85]. Although message combining is an effective technique to solve this problem when the sources of hot-spots are global shared locks, the hardware needed for supporting the combining is quite expensive (Pfister and Norton estimated that the combined network increases the size and cost of the switch in a factor of 6 to 32).†

Let the number of processors to be p, and the number of network packets emitted per processor per switch cycle to be r ($0 \leq 1$). The *hot-spot percentage*, ρ, is defined to be the fraction of the data references directed at the hot-spot (i.e. each processor emits packets directed to the hot-spot at a total rate of $r*h$). Then the effective data rate into the hot module is $r(1-h) + rhp$. And the asymptotic limit of the total communication bandwidth available is:

$$B = \frac{p}{1 + h(p-1)}$$

This function gives a limit on the available speedup for a given number of processors. The function was derived in [PfNo85] for shared memory-modules on multi-stage networks, but can be

† The combine-network on the IBM RP3 was dropped because of the cost and the technical complications encountered.

generalized to more general distributed systems. This limit imposed by the hot-spot degradation is very significant. For example, for a 1000-processor system, a hot-spot percentage of 0.125% can limit the potential speedup to 50%! The hot-spot percentage can be estimated by analyzing patterns of the data dependence graph. The hot-spot percentage can be used to generate network traffic in simulations, but is very difficult to be computed for real programs except in synchronized parallel loops.

3. A Framework for Performance Prediction

Based on the characterization model, a framework for predicting performance of programs on different parallel compilers can be defined. This framework consists of four phases: the prediction-setup phase, prediction-construction phase, prediction-update phase, and prediction-refining phase.

Procedure 2. A framework for predicting performances of programs
for different parallel computers.

1. The prediction-setup phase.

- Select evaluation functions.
- Set constants for the selected evaluation functions based on machine features.
- Choose the method for combining the evaluation functions.
- Set weights for the selected evaluation functions based on objectives of the optimization.

2. The prediction-construction phase.

- Invoke the prediction-setup phase to setup the constants.
- Apply evaluation function recursively on elements of compound statements.
- Use prediction combining function to integrate the evaluation functions.

3. The prediction-update phase.

- Invoke prediction-setup phase based on the perspective transformations to select and setup the evaluation functions.
- Use the evaluation functions to compute the change that the transformation has on the execution time.

4. The prediction-refining phase.

- Invoke prediction-setup phase based on the objectives of the transformation to select and setup the evaluation functions.
- Invoke either prediction-construction phase or prediction-update phase to compute the finer estimation.

The prediction-setup phase is invoked by other phases to prepare for the prediction estimation. The selected evaluation functions are combined by a weighted linear combination of values of the evaluation functions:

$$T(S,\ Machine) = \sum_{i=1}^{m} Weight^{E_m} * E_m(S,Machine).$$

Therefore, for moderate optimization, a default prediction function is defined to be

$$T = N^{op} * C^{op} + N^{load} * C^{load} * p^{ds} + N^{store} * C^{store} + T^{cs} + T^{co} \quad (2)$$

Where N^{op} is the operation count, C^{op} is the unit cost of an operation, N^{load} is the number of load operations, C^{load} is the unit cost of a load operation, N^{store} is the number of store operations, C^{store} is the unit cost of a store operation, P^{ds} is the percentage of prefetch-breaker (percentage of data access that cannot be overlapped with the computation), T^{cs} is the cost of the control synchronization, and T^{co} is the control overhead.

More accurate prediction can be achieved by modifying this default definition. In particular, the set of evaluation functions, their weights, architectural-dependent constants, and the method for integrating them are determined by a set of heuristics, machine features and the current objectives of the compiler.

The prediction-construction phase estimates the performance of the program by applying the set of evaluation functions to each node in the program dependence graph according to a depth first search order based on the control dependence (with back edges being ignored). For each compound statement, the evaluation functions are applied on all its children and the results are then combined for it. The predictions for subtrees of a node are combined by a prediction combining function. The prediction combining function determines how and when to integrate values computed by each of the evaluation functions. For example, it can use the linear combination of the weighted values or other functions to combine the results of different evaluation functions. Depending on the algorithm, the values can be combined while they are computed at each node or they can be computed individually and combined at the task level. The results of applying an evaluation function on children of a compound statement are accumulated by an procedure which is based on the template defined in procedure 1 and the performance evaluation procedure.

```
Procedure 3. Performance construction procedure.
PerformanceEvaluation(Node, PDG, ListOfEvalFunctions, Results)
Begin
   if (Node is a compound statement) then
      for each Child of Node in PDG do
         PerformanceEvaluation(Child, PDG, ListOfEvalFunctions, Result0);
      endfor
      Accumulate(ListOfEvalFunctions, Result0, Result1);
   else
      Evaluation(Node, ListOfEvalFunctions, Result1);
   endif
   if (combine_at_node or Node is a task) then
      Combine(ListOfEvalFunctions, Result1, Result);
   else
      Result = Result1;
   endif
end
```

The prediction-update phase is applied during the course of the program optimization process. The evaluation functions involved in this process estimate the *changes* that a transformation might have on the performance of the program and adjust the prediction accordingly. Only the performance estimation for the affected program region needs to be updated. This information can be used to estimate the effects of a transformation on the performance of the program. On the other hand, the overall performance prediction can be updated incrementally.

The prediction-refining phase can be engaged optionally during the program optimization process to refine the estimation by using a set of more detailed evaluation functions. It can either compute the estimation from scratch using a set of more sophisticated evaluation functions or it can refine the original estimate by updating certain aspects of the estimation.

Estimating Program Execution Time Using the Framework

The execution time of the program can be estimated to different degrees of accuracy with different sets of evaluation functions. The selection of the evaluation functions and the method for integrating the functions depend on how much computing resources can be allocated to the estimation and selected by a set of rules. We will use an example to demonstrate how to use the above framework to predicte the program execution time for parallel computers. The sample program that we use is the matrix-vector multiply program. We assume that the program has been decomposed

to be executed on P processor and that array a and y have been previously distributed in blocks into the processors, a local copy of x is available on all processors, and the result y is to be collected at processor 0. This example is chosen because it is simple but presents some communication imbalance for distributed-memory computers due to the processor 0 being a hot-spot.

```
forall pid := 1 .. P do          ---   tl := ⌈n/P⌉.
    local a : array [1..tl, 1..m] of real;
          x : array [1..m] of real;
          y : array [1..tl] of real;
    for i := 1 .. tl loop
          for j := 1 .. m loop
                y[i] := y[i] + a[i,j] * x[j];
          end for
    end for
    if (pid == 0) then
          received = 0;
          for i := 1 .. P-1 loop
                wait_for_message(source, type);
                recv(source, mbuf[source*tl], mbufln, flags);
          end for
    else          --   send vector y to processor 0;
          send(0, y, tl, flags);
    end if
end forall
```

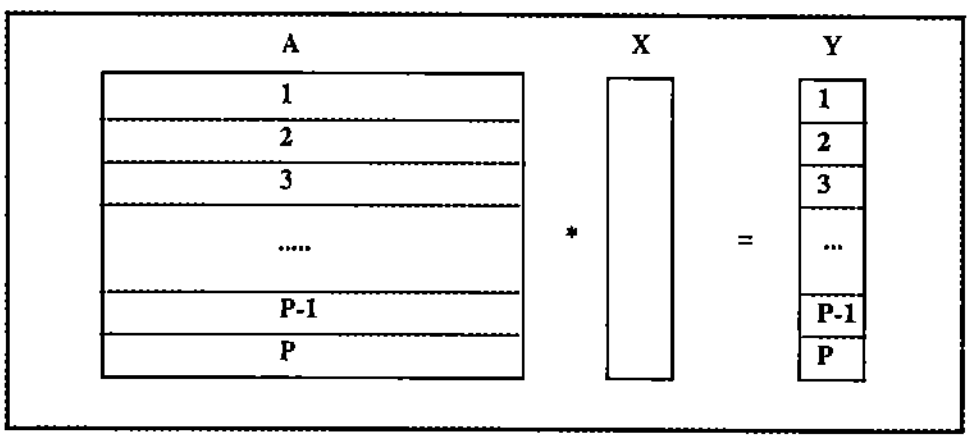

Figure 2. Sample program and the array distribution for the arrays in the matrix vector multiply example.

We chose to use the NCUBE/2 to demonstrate how the estimation is established because the NCUBE/2 has some interesting characteristics that present a challenge to the performance

prediction. For example, integer addition and subtraction takes one machine cycle but an operand decoding and fetching takes up to 3 cycles; array address calculation is expensive (the code generated by NCUBE/2 compiler takes about 11 cycles for one dimensional arrays and about 31 cycles for 2 dimensional arrays), integer multiply (7 cycles) is slower than floating point multiply (6 cycles); integer division (38 cycles) is much slower than floating point division (7-18 cycles), communication between processors is very expensive (156 cycles startup time per message for one hop), etc. These factors imply that data prefetch has dominant effects on the performance of the program; simply classifying operations into floating point and integer operations may not be fine enough, and interprocessor communication needs to be overlapped with computation as much as possible. All these factors plus the fact that the compiler on the NCUBE/2 does not generate optimal code complicate the estimation of the performance on NCUBE/2. Nevertheless, this presents a good opportunity for examining our performance prediction model.

During the performance-setup phase, the following performance factors are selected based on the architectural features of NCUBE/2: counts for floating point operations, integer operations, memory loads, and memory stores, array references, data synchronization, data transmitting cost, and control synchronization cost. All of the above factors except the data transmitting cost are in the list of default evaluation functions. The data transmission cost over the network is needed because the NCUBE/2 is a distributed machine and the balance between the communication and computation is very important. The results of the evaluation functions will be combined at the task level. The constants that are used by the evaluation functions are provided by the machine knowledge manipulation mechanism and are listed in the following table:

Evaluation constants for NCUBE/2 (in microseconds)			
performance factor	time	performance factor	time
floating point op, C^{fop}	0.33	integer op C^{iop}	0.35
local memory load, C^{ld}	0.1	local memory store C^{ls}	0.1
message startup cost, α^0	156	memory startup (per hop), α^1	2.2
data trans.(per word), β^0	0.2	data trans.(per word & hop), β^1	1.8
loop overhead, C^{lo}	0.6	0 bytes read	80

Figure 3. Constants for evaluation functions for NCUBE/2.

During the performance-prediction construction phase, first the performance evaluation procedure defined in procedure 3 is applied to compute the selected evaluation functions, then the evaluation accumulation procedure as shown in procedure 1 is used to accumulate the values of the evaluation functions. Consequently, we find that there are 2 floating point operations, 5 array address calculations, 3 load operations and 1 store operation in the loop instance (k,j). And there

are $2*m*tl$ floating-point operations, $5*m*tl$ array address calculations, $3*m*tl$ load operations, and $m*tl$ store operations for loop k. Also, the loop overhead for each processor is $m*tl$ times the unit cost for each loop instance. The execution time of the computation of the vector y in each processor can be summarized in the following formula:

$$T^{computation} = (N^{fop} * C^{fop} + N^{iop} * C^{iop} + N^{ar} * C^{ar} + N^{load} * C^{load} * p^{ds} + N^{store} * C^{store} + N^{lo} * C^{lo}) * m * tl^{i}$$

Where N^{fop}, N^{iop}, N^{ar}, N^{load}, N^{store}, and N^{lo} are the counts for floating point operations, integer operations, array references, memory loads, memory stores, and loop overheads, respectively, in the loop instance (k,l). Also C^{fop}, C^{iop}, C^{ar}, C^{load}, C^{store}, and C^{lo} are the unit cost for a floating point operation, integer operation, array reference memory load, memory stores, and loop overhead, respectively. For processor P, tl^{P} is $n - (P-1) * \left\lceil \frac{n}{P} \right\rceil$, and for all other loop instances, tl^{i} is $\left\lceil \frac{n}{P} \right\rceil$. For the program shown in figure 2, there is a very limited degree of data pre-fetching since there is only one statement inside the loop branches and the expressions in the statement are not very long. So the data synchronization overhead is set to be 100% of the memory load cost. Suppose N is 6400, M is 100, and P is 64. By substituting the constants in figure 3 into the formula, the estimated computation time for the loop k is 4600 microseconds. As a comparison, the measured computation time for loop k is 4823.6 microseconds.

The cross-task data dependence is caused by the I/O statements corresponding to the collection of the array y, so the data-synchronization is computed based on these I/O statements. The control synchronization is hard to predicte on NCUBE/2 because there is a non-trivial cost in loading the node programs so most programs do not synchronize the program unless it is necessary. This means that tasks on the nodes start at different time which depend on how the programs are loaded (broadcasted or not), the size of the node program, and the position of the node in the hypercube. The control synchronization is needed only when values of y are needed later (which is out of the scope of the program that we discuss here). Lacking the control synchronization at the beginning of the program makes the estimation of the data synchronization cost difficult since tasks are not started at the same time.

To estimate the cost of sending messages, we need to estimate the cost for read/write and data transmission. Since message sending on NCUBE/2 is non-blocking, on tasks that send the data only the time to copy the data into system buffer needs to be counted. On the other hand, on the receiving side, the data synchronization cost includes the time for the sender to send out the data, the transmitting time, and the time for the reader to receive and store the data. This timing information is a function of the message length and can be computed by the equation (1) below.

Since all the data are sent to processor 0, the processor 0 becomes a hot spot and the computation time of processor 0 dominates the computation time of the program. For processor i, the distance to processor 0 is the number of 1's in its binary representation of the processor identified, represented as $dist^0(i)$. Therefore, the cost of transmitting the vector y computed in processor i to processor 0 is:

$$T^{dt}(i) = \alpha^0 + \alpha^1 * dist^0(i) + tl * (\beta^0 + \beta^1 * dist^0(i)) \quad (1)$$

$$= 156 + 2.2 * dist^0(i) + \left\lceil \frac{N}{P} \right\rceil * (0.2 + 1.8 * dist^0(i))$$

Although the received statements in processor 0 are executed sequentially, the data synchronization cost is not the sum of costs of transmitting vector y from all other processors because the data are transmitted simultaneously. If all processor are synchronized then the data synchronization cost would be:

$$T^{dt} = \max\left\{ \min\left\{ T^{dt}(i): i \in [1..P] \right\} + \sum_{i=1}^{P} T^{read}(i) , \right.$$

$$\left. \max\left\{ T^{dt}(i) + T^{read}(i): i \in [1..P] \right\} \right\}$$

Where $T^{read}(i)$ is the cost of moving the message from the system buffer into the data structure of the user program.

For our sample program, $\min\left\{ T^{dt}(i): i \in [1..P] \right\} = 158.2 + 2.0 * \left\lceil \frac{N}{P} \right\rceil$ $\max\left\{ T^{dt}(i): i \in [1..P] \right\} = 156 + dim * 2.2 + (1.8 + dim * 0.2) * \left\lceil \frac{N}{P} \right\rceil$ and $T^{read}(i) = 80 + 0.1 * \left\lceil \frac{N}{P} \right\rceil$.

By substituting different values of n, m, and P, we obtain the estimations as shown in figure 4. The measured execution times for these parameters are also shown for comparison.

			Predicted performance			measured performance		
n	m	p	Computation	Communication	Total	Computation	Communication	Total
400	200	2	120400.00	904.20	121304.20	123632.00	730.00	124362.00
400	200	4	60200.00	990.20	61190.20	61806.00	889.00	62695.00
400	200	8	30100.00	1390.20	31490.20	30909.00	1247.00	32156.00
400	200	16	15050.00	2268.20	17318.20	15475.00	2261.00	17736.00
400	200	32	7525.00	4027.20	11552.20	9391.00	3004.00	12395.00
400	200	64	3762.50	7510.70	11273.20	6977.00	6789.00	13766.00
1600	100	4	120400.00	1770.20	122170.20	124623.00	2008.00	126631.00
1600	100	8	60200.00	1870.20	62070.20	62179.00	2038.00	64217.00
1600	100	16	30100.00	2598.20	32698.20	31082.00	2990.00	34072.00
1600	100	32	15050.00	4282.20	19332.20	15564.00	5022.00	20586.00
1600	100	64	7525.00	7728.20	15253.20	7792.00	9266.00	17058.00
6400	20	16	24080.00	3918.20	27998.20	26010.00	6514.00	32524.00
6400	20	32	12040.00	5302.20	17342.20	13008.00	7632.00	20640.00
6400	20	64	6020.00	8598.20	14618.20	6516.00	36930.00	43446.00

Figure 4. The predicted and measured performance of the matrix-vector multiply program.

The performance prediction established in the performance-construction phase provides a foundation for estimation of effects of program transformations on the program. During the program optimization process, we can use this estimation and apply the performance-update phase to adapt the estimation based on the program transformations. For example, in the above program, y[i] has an output dependence and a flow dependence on itself in loop j. This means that the result of y[i] is loaded and updated right after it is stored in the previous loop iteration. So if we allocate y[i] into the register during the execution of loop j, then only one load and one store of y[i] is needed. This is a reduction of $m-1$ loads and $m-1$ stores for each y[i]. Therefore, by allocating y[i]'s into registers, the load and store counts are each reduced by $(m-1)*tl$.

Note that the performance of the matrix-vector multiply program is far short of the advertised peak performance of the NCUBE/2. This is because there is only one statement in the loop body and the loop branches prevent the prefetching mechanism of the NCUBE/2 to pre-load the operands. Consequently, the computation of the operands dominates the computation time. By applying the transformation *loop unrolling* we may increase the size of the code between the conditional branch of the loop, thus allowing the data loading to be overlapped with the computation. For example, if we unroll the loop m 5 times, the estimated data fetching time becomes one-fifth of

the original cost since p^{ds} is 1/5. The loop overhead is also decreased by a factor of 5. This implies that by applying loop unrolling the cost for loading can be decreased significantly. This is confirmed by the following table:

The innermost loop is unrolled 5 times								
			Predicted performance			measured performance		
n	m	p	Computation	Communication	Total	Computation	Communication	Total
400	200	2	72400.00	904.20	73304.20	69332.00	729.00	70061.00
400	200	4	36200.00	990.20	37190.20	34691.00	776.00	35467.00
400	200	8	18100.00	1390.20	19490.20	17357.00	1249.00	18606.00
400	200	16	9050.00	2268.20	11318.20	8689.00	2270.00	10959.00
400	200	32	4525.00	4027.20	8552.20	5881.00	3004.00	8885.00
400	200	64	2262.50	7510.70	9773.20	5127.00	5899.00	11026.00
1600	100	4	72400.00	1770.20	74170.20	69898.00	1972.00	71870.00
1600	100	8	36200.00	1870.20	38070.20	34977.00	2056.00	37033.00
1600	100	16	18100.00	2598.20	20698.20	17499.00	2950.00	20449.00
1600	100	32	9050.00	4282.20	13332.20	8758.00	5065.00	13823.00
1600	100	64	4525.00	7728.20	12253.20	4406.00	9266.00	13672.00
6400	20	16	14480.00	3918.20	18398.20	14913.00	6478.00	21391.00
6400	20	32	7240.00	5302.20	12542.20	7468.00	7703.00	15171.00
6400	20	64	3620.00	8598.20	12218.20	3761.00	37045.00	40806.00

Figure 5. The predicted and measured performance of the matrix-vector multiply program with innermost loop unrolled 5 times.

Note that in figures 4 and 5, the estimation of the computation can be fairly close to the actual cost of the computation. Although there are instances that the simple-minded estimation for the effects of the data prefetching is not accurate enough, this model can be utilized by the program optimization process to estimate the effects of program transformations on the performance. More detailed estimation of the data prefetching effects is needed only when this factor is important, for which case, the performance-refine phase can be used to improve the accuracy of the estimation. On the other hand, the performance estimation for the communication is good for some cases but far from accurate in some other cases. This is because we did not consider the hot-spot effects of the message passing in the example. Since all nodes are sending messages to node 0, node 0 becomes the hot-spot, and when the message is long and the size of the cube is large, performance

degradation occurs. Especially for the case where (n, m, P) = (6400, 20, 64), the cost of sending 64 messages of 100 words each to node 0 is more than five times that of sending 32 messages of 200 words each to node 0! Since NCUBE/2 use the *worm–hole routing* mechanism where a channel is reserved for a message and the data are pipelined to the destination, messages that need to use the busy links will be blocked until the previous message is finished with the channel. The longer the messages, the higher chances that a message will be blocked at the sender side.

When a more accurate performance prediction is required, the performance-refine phase can be invoked. For example, one way to refine the estimation is to use a finer classification of the operations. This will not help much for our sample program here since it does not contain any expensive operations such as divisions. For distributed architecture like NCUBE/2, one way to better estimate the data synchronization cost is to analyze the performance degradation due to the communication hot-spot. The performance degradation of the network caused by the hot-spot saturation can be estimated by the hot-spot percentage. However, the computation of hot-spot percentage is very expensive and difficult to be accurate due to the dynamic behavior of the program. A more practical approach is to model the performance degradation with a set of selected patterns of different degrees of congestion (as described by the data dependence graph). For data dependence that does not match any of the pre-selected patterns, an interpolation to the nearest pattern is done to find an estimation of the degradation. The more patterns we use, the better the estimation will be. However, matching the patterns is an expensive operation, so this method can be used only when the user can afford the needed computing resources.

4. Dynamic Performance Prediction and Run-time Tests

There are programs for which the static analysis of the compiler fails to predict the control flow and thus the performance of the program. These cases normally have variables or indirect references in the control structures (such as loop bounds or conditions), or conditional statements for which probabilities of the branches are unknown. The performance prediction model we described here can be applied to these cases by utilizing the inference capability of the system. For each performance evaluation function there is a list of parameters that are used to compute the evaluation. The inference engine evaluates the evaluation function by unifying the variables with their values first. If there are variables remain undefined after the unification stage (for example, if there are variables in loop bounds), the evaluation function will be evaluated as a function of the undefined variables by a common compiler optimization technique called *constant folding [AlSeUl86]*.

Operations (such as adds, multiplies, comparisons) can be performed on results of evaluation functions (including those with non-instantiated variables) to merge different evaluations or use them to compute other evaluation functions. When dynamic decisions are needed, the evaluations

with uninstantiated are used to find and generate the minimum run-time tests to decide the control flow at run-time.

5. Applying Performance Prediction to the Intelligent Decision Making

The intelligent program optimization needs to be based on accurate and efficient performance prediction to make sound decisions. The performance prediction model can be applied to the intelligent parallel compilers in the following areas:

1. Compare and choose the most promising transformation to apply among a set of applicable program transformations.
2. Provide a measure for selecting pre-optimized algorithm when several algorithms are available under the algorithm substitution approach.
3. Decide when to stop the optimization model.
4. Help to generate minimal run-time tests for situations where the compiler can't make decisions by static analysis.

The performance prediction model we proposed is designed to be integrated into the decision-making process for optimizing program parallelism. In particular, it was used in an intelligent parallel compiler [WaGa89, Wang90] to guide the decision making process to improve the program parallelism. When applied to systematic decision-tree traversing algorithms such as the A^* algorithm (discussed in [Wang90]), automatic parallel program optimization is achieved by using the evaluation functions as the heuristic functions. The flexibility of our model makes it easy to refine the heuristic functions at different parts of the decision tree, which makes the framework very dynamic. It can also be combined with rule-based systems to improve the quality of the optimization. An evaluation function or the combination of several evaluation functions can be used to decide the merits of program transformation techniques. Heuristic driven rules can use the prediction to select appropriate program transformations or pre-optimized algorithms. In an interactive program optimization session, it provides users with performance information for them to make optimization decisions.

For pre-optimized algorithm substitution approach, there are cases where difficult decisions need to be made. For example, there may be more than one algorithms that are equivalent to the program under consideration; or the same algorithm may be optimized for several different parallel architectures but not for the current target machine. For the formal case, the performance prediction may help the system to decide which algorithm is most efficient for the problem. For the latter case, the performance prediction model can be integrated with the machine knowledge manipulation system to find the architecture that fits the target machine most.

We mentioned in [Wang90] that the optimal solution for the program optimization is usually not known until the entire decision tree is traversed. This makes deciding the termination condition difficult. Since the performance estimation mechanism can be used to find the lower bound of the execution time, this number can be used to decide the termination condition of the optimization. First, a tolerance to the lower bound can be selected; the higher the optimization degree is, the smaller the tolerance will be. When the estimated performance falls into the tolerance range of the "optimal execution time" the optimization process can be terminated.

Although any performance prediction model can be applied to these tasks, our model is more flexible and efficient and is suitable to be integrated with the decision making process in intelligent parallel compilers.

6. Related Works

There are many existing performance tools for parallel architectures. For examples, Fause [GGJMG89] and IPS [MiYa87] allow program behavior be described at many levels of detail and abstraction including the program, process, procedure and instruction levels. PIE, developed at CMU [SeRu85], uses a metalanguage to provide support for an efficient manipulation of parallel modules and programming for observability. These systems are designed to be used as a semi-automatic performance evaluation tool for the user. The Parafrase [AbKw85] provides a performance prediction module based on a similar program hierarchy but is designed to be used in the compiler and is inexpensive to compute. The "load/store" modeling method is used in [GJMW89, BWJALG90] to characterize the performance of shared memory architecture by a set of template sequences of vector load, store and "nop" instructions. There are many other environments that evaluate the performance of a particular type of architecture or characterize the potential parallelism of programs on the architecture. However, none of the existing system can be used to predicte program performance for a wide range of architectures accurately and inexpensively. Also, none of the system provide a systematic framework that is flexible enough to be utilized in an intelligent parallel compiler. Our framework fills this gap by providing a flexible mechanism for the knowledge base system to adjust the prediction model dynamically to suit different optimization objectives and different architectures.

7. Conclusion

To summarize, our framework for the performance prediction of parallel computers has the following advantages:

1. Different classes of parallel computers can be handled under the same framework. The performance prediction model can be adjusted to suit different architectures.

2. The estimation can be tuned to suit different objectives by adjusting the performance combining function and the weights associated with the evaluation functions.
3. Different amounts of resources can be committed at different stages of the compiling process by using evaluation functions of different complexities.
4. The selection of evaluation functions and weights offers a good opportunity for the compiler to learn to improve itself. This point will be fully investigated in [Wang91].
5. The prediction update process is inexpensive; only the effects of the related transformations are computed. It can be applied repeatedly during the program optimization process.

Intelligent parallel compilers need an accurate, inexpensive, and flexible performance prediction model to make critical decisions. Our framework is simple and yet flexible enough to be integrated into intelligent parallel compilers with multiple target machines. It can be used by systematic state-space search algorithms such as A^* and other best-first search algorithms to find optimal program transformation sequences.

8. REFERENCES

[AbKw85] W. Abu-sufah, and A. Kwok, "Performance Prediction Tools for Cedar: A Multiprocessor Supercomputer," Proc. of the 12th Int'l Symp. on Computer Architecture, 1985, 406-413.

[AhSeUl86] A. Aho, R. Sethi, and J. Ullman, "Compilers: Principles, Techniques, and Tools," Addison Wesley, 1986.

[BWJALG90] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D Gannon, "Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000," *Proceedings of the 1990 International Conference on Supercomputing,* 1990, 401-413.

[Cytron84] R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, Report No. UIUCDCS-R-84-1177, University of Illinois, Urbana-Champaign Aug., 1984.

[Dong87] Dongarra, J.J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," in W. J. Karplus (ed.) *Multiprocessors and Array Processors,* Simulation Conciles Inc. San Diego, CA, pp. 15-33, Jan. 1987.

[Feng72] T. Y. Feng, "Some Characteristics of Associative/Parallel Processing," Proc. 1972 Sagamore Comp. Conf., Syracuse Univ. 1972, 5-16.

[GJMW89] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff, "Performance Prediction of Loop Constructs on Multiprocessor Hierarchical-Memory Systems," Tech. Report, CSRD Rpt No. 853, Center for Supercomputer Research and Development,

University of Illinois, 1989.

[GaJaGa87] D. Gannon, W. Jalby, K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation," *Proc. of 1987 Int'l Conf. on Supercomputing, 1987*, 229-254.

[GGJMG89] V. Guarna Jr., D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur, "Faust: an Integrated Environment for the development of Parallel Programs," *IEEE software*, July 1989.

[Händler77] W. Händler, "The Impact of Classification Schemes on Computer Architecture," Proc. 1977 Int'l. Conf. on Parallel Processing, 1977, 7-15.

[Husm86] "Compiler Memory Management And Compound Function Deifintion For Multiprocessors," Ph.D. thesis, CSRD Rpt No. 575, Center for Supercomputing Research and Development, University of Illinois, 1986.

[MiYa87] B. Miller and C. Yang, "IPS: an Interactive and Automatic Performance Tool for Parallel and Distributed Programs," *Proceedings of the 7'th Int'l Conference on Distributed Computing Systems*, 1987, 482-489.

[PfNo85] G. Pfister, V.A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proc. of the 1985 International Conference on Parallel Processing*, 1985, 790-797.

[SeRu85] Z. Segall and L. Rudolph, "PIE: a Programming and Instrumentation Environment for Parallel Processing," IEEE Software, 2(6), November, 1985.

[Wang85] K. Wang, "An Experiment in Parallel Programming Environment: The Expert Systems Approach," in K. S. Fu, editor, *Some Prototype Examples for Expert Systems*, TR-EE 85-1, School of Electronic Engineering, Purdue University, Mar. 1985, 591-624.

[Wang88] K. Wang, "Machine Knowledge Representation and Manipulation For Parallel Compilers," Technical Report CSD-TR-843, CER-90-51, Department of Computer Sciences, Purdue University, Dec. 1988.

[WaGa89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimizations For Parallel Computers," in K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, MCGraw-Hill, 1989, pp. 441-485.

[Wang90] K. Wang "A Framework For Intelligent Parallel Compilers," Tech. Report, CSD-TR-1044, CER-90-52, Department of Computer Science, Purdue University, Nov. 1990.

[Wang91] K. Wang "A Parallel Compiler That Learn," paper in preparation.