Precise Compile-Time Performance Prediction for Superscalar-Based Computers

Ko-Yang Wang*
IBM T. J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, USA

Abstract

Optimizing compilers (particularly parallel compilers) are constrained by their ability to predict performance consequences of the transformations they apply. Many factors, such as unknowns in control structures, dynamic behavior of programs, and complexity of the underlying hardware, make it very difficult for compilers to estimate the performance of the transformations accurately and efficiently. In this paper, we present a performance prediction framework that combines several innovative approaches to solve this problem. First, the framework employs a detailed, architecture-specific, but portable, cost model that can be used to estimate the cost of straight line code efficiently. Second, aggregated costs of loops and conditional statements are computed and represented symbolically. This avoids unnecessary, premature guesses and preserves the precision of the prediction. Third, symbolic comparison allows compilers to choose the best transformation dynamically and systematically. Some methodologies for applying the framework to optimizing parallel compilers to support automatic, performance-guided program restructuring are discussed.

1 Introduction

Compiler optimization is ad hoc. Optimizing parallel compilers improve the performance of user programs by applying a sequence of restructuring transformations to uncover parallelism provided by the underlying architecture. Many restructuring techniques may significantly change the program performance. Performance trade-offs among applicable restructuring transformations have to be evaluated carefully. Unfortunately, most optimizing parallel compilers do not have the performance estimation capability to match their sophisticated program restructuring capability. As a result, it is not uncommon for an optimizing compiler to apply a sequence of complicated program restructuring transformations and yield only modest performance gains or even performance losses.

1.1 Difficulties of Static Performance Prediction

The problem of predicting program performance at compile time is inherently difficult. First, some critical information needed by the compiler may not be available at compile time. Often, such information depends on the input data of the program. Second, detailed knowledge of architecture features need to be incorporated into the cost model to estimate the performance of programs on the target machine. This often leads to a non-portable system that only works for a particular type of machines. Multiprocessor machines add another dimension of complexity to the performance prediction. Third, the optimization unit of the compiler has to take into consideration the low-level optimizations done by the compiler back-end. This is particularly true for superscalar architectures since they rely heavily on low-level compiler optimizations to achieve their potential performance. Fourth, due to the large number of decision points in a compiler, estimations of program performance have to be done repeatedly in the decision making process. Therefore, the compiler is severely constrained as to how much computing resources it can spend on performance estimation. Finally, effects of compounding estimations may magnify errors significantly when estimations for multiple program pieces are combined.

Prior researchers [1, 3, 5, 6, 7, 8, 9, 10, 11, 13, 15, 16, 17, 18, 19] attempted to solve the above problems by using heuristics, profiling, run-time tests, querying

*For correspondence: kyw@watson.ibm.com

SIGPLAN 94-6/94 Orlando, Florida USA
© 1994 ACM 0-89791-662-x/94/0006..$3.50

users, analytical models, or combinations of the above. So far, the results have been mixed. The fundamental problems of efficiency, accuracy and portability of performance estimation are still the weak points of state-of-the-art optimizing compilers.

1.2 Low Level Parallelism in Superscalar Architectures

A recent trend in parallel computer architecture is to use superscalar processors to exploit the low level parallelism (such as the IBM SP1s that use RS 6000 chips or Cray T3Ds that use Alpha Chips). Superscalar architectures have multiple instruction execution units that can operate concurrently. Other features such as instruction pipelines, operation overlapping, cache prefetching, and powerful instructions (such as multiply-and-adds) are also incorporated into the architecture. This can significantly decrease the program execution time but also greatly increases the complexity for the compiler to estimate the performance of the program. If not applied carefully, a conventional cost estimation model may be off by a factor of ten or more!

Estimating the execution time of assembly code on superscalar architectures is already complicated. Estimating the performance of high level language programs, such as Fortran and its variants, is even more challenging. The back-end code generator may perform a sequence of low-level optimizations that are aimed at increasing low-level parallelism or locality. If the cost estimate fails to take these factors into consideration, the resulting estimate may be seriously distorted. We do not know any compiler that currently utilizes compile-time performance predictions of superscalar architectures in program restructuring.

1.3 Requirements

A good performance prediction framework for parallel and sequential compilers has to meet the following requirements:

Precision: The prediction has to be accurate for the compiler to make correct decisions.

Efficiency: The performance prediction needs to be very efficient to make repeated calls practical during the program optimization process.

Robustness: The framework should be able to handle programs with unknowns in control structures, unknown branching probabilities, etc.

Portability: Given the complexity and cost of parallel compilers and shortened hardware product cycle time, portability across multiple hardware platforms is ever more important for parallel compilers.

1.4 Problem Statement

The major focus of this paper is a framework that can estimate performance of user programs efficiently and accurately across different architecture platforms. Algorithms and methodologies for estimating performance of programs on superscalar-based architectures efficiently will be presented.

1.5 Our Approach

We designed a performance prediction framework that attempts to fulfill the requirements discussed in Section refrequire by integrating several approaches into a unified framework. The key ideas behind our design are: 1. Estimate the cost of straight line code (code without branches and iterations) as accurately and efficiently as possible. 2. When aggregating performance data, estimates of values of unknowns in control structures are delayed as long as possible. This is accomplished by representing the performance data as a symbolic expression of polynomials whose variables are unknown values in program constructs. 3. Recognize situations where the estimates of values of unknowns can be eliminated. 4. Methodologies for comparing performance symbolically are introduced.

1.6 Organization of the Paper

The remaining sections of the paper are organized as follows. In Section 2, we introduce a performance prediction framework and present its components, data structures and algorithms for estimating program performance efficiently and accurately. In Section 3, program optimization using performance prediction is discussed. We discuss methodologies for comparing effects of different transformations using the symbolic performance expressions. Situations when performance prediction can be avoided or simplified are also discussed. In Section 4, we summarize our work and present future research directions for optimizing parallel compilers.

2 A Precise Performance Prediction Framework

In this section, we present a performance prediction framework that can be used in optimizing compilers.

The structure of the framework is shown in Figure 1. Expressions from a sequence of statements are sent to the *instruction translation module* which translates operations of the high level language into low level instructions. The generated instruction stream is then fed into the *instruction, memory, and communication cost models* The instruction cost model estimates the cost of executing the instructions by taking the low

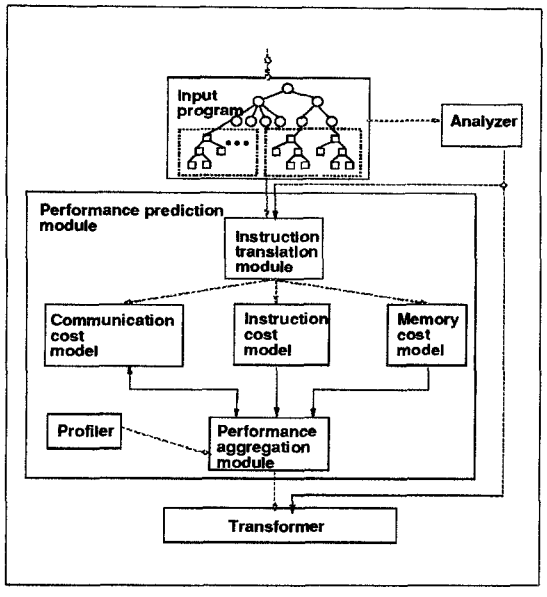

Figure 1: Framework of the Performance Prediction System

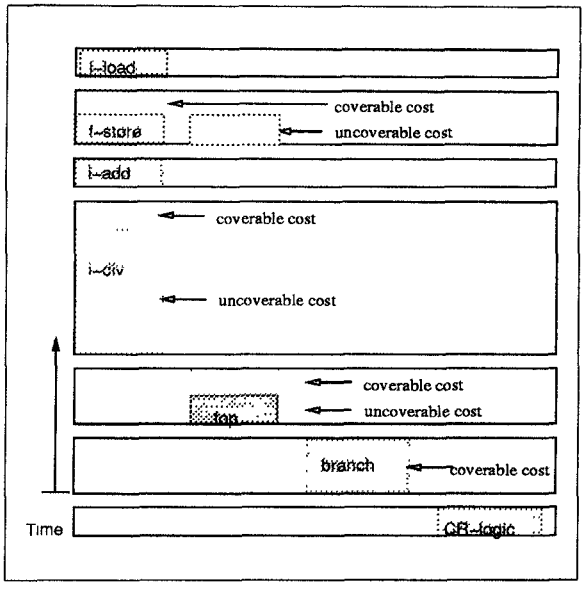

Figure 2: Example of instruction costs.

level parallelism of the architecture and data dependencies of the program into account. The memory cost model estimates the cost related to cache accesses and page faults. For distributed memory machines, message passing instructions are sent along with the sequential cost estimation to the communication cost module to get cost of moving data among processors. The cost estimations of the program fragments are then combined through the *performance aggregation model.* Unknowns in control statements and array subscripts are treated as variables in the performance expressions. The performance of a compound statement is computed symbolically to avoid error magnification when estimations are combined. Symbolic comparison of the performance expressions is supported. It can be applied to guide program restructuring or choose run-time tests based on sensitivity analysis (see Section 3.4).

2.1 A Cost Model for Modeling Parallelism in Superscalar Architectures

To account for the low level parallelism present in modern CPUs, a cost model for straight line code based on detailed architecture features is defined. The cost model that we designed can be applied to both traditional processors and modern superscalar architectures. For the latter case, it captures the overlapping effects and honors data dependencies.

Cost of operations is assigned based on operation units that we called *atomic operations.* Atomic operations are specific low level instructions supported by the processor architecture. Each atomic operation has a cost associated with it. Unlike previous cost models, the cost of operations is divided into two components:

- *noncoverable cost:* The time that a functional unit actually dedicates to the operation.
- *coverable cost:* The time when the next operation that does not depend on the result of the current operation can be started. Instructions that use the result of the operation have to wait until the current operation is completed.

For example, with the IBM Power architecture, each floating-point *add* operation has one cycle of noncoverable cost and one cycle of coverable cost on the floating point unit. If the compiler can schedule another operation on the same functional unit, the cost of the operation can be thought of as one cycle, but if no other executable operations can be found to fill the coverable cycle, then the operation will cost two cycles.

An operation can have costs on multiple functional units. For example a *floating point store* operation will occupy one floating point unit for two cycles with one cycle being coverable and will occupy one integer unit for one cycle on the IBM Power architecture.

A conceptual view of our cost model of superscalar architecture is a two dimensional unit with multiple functional bins in one dimension and time slots in another dimension (see Figure 3). Operations are represented as a two dimension objects which have costs on each functional unit being chained together (Figure 2). Noncoverable costs of an operation are represented as solid objects that cannot coexist with other objects in

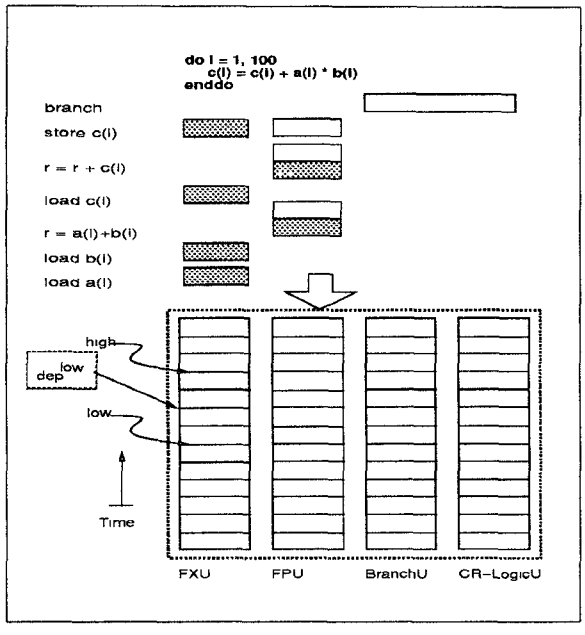

Figure 3: An example of dropping operations into time slots of the bins.

the same time slot, and coverable costs are represented as transparent objects which can share the time slots with a noncoverable object of another operation. Furthermore, the top of an operation object can be viewed as a filter which can blocks operations that depend on the result of the operation. Operations that do not use the result of another operation can pass through its filter if there are open time slots below it that the operations can fit in. All costs of an operation have to fit in all functional units at the same time for it to occupy the time slots. For architectures with multiple operation pipes, more bins can be added. The bins are flushed before being used for another block of statements.

Estimating the cost of executing a sequence of operations can be viewed as finding a way to drop all operation objects into the virtual architecture bin with the goal of minimizing the unfilled slots (see the example in Figure 3).[1] The total cost of the operations is the time difference between the highest time slot and the lowest time slot occupied by the operations.

Under our cost model, we assume that operations can be reordered based on mathematical rules and dependence relations, and the compiler is intelligent enough to order instructions so that full overlapping is possible. Since the cost model is supposed to model the combination of the compiler and the architecture, it should try to imitate the compiler, not to outperform it.

By taking multiple functional units and dependency into account, our approach is much more precise than operation-count based cost models. On the other hand, the key factor in deciding whether this approach is useful or not lies in the efficiency of the implementation.

Algorithm For Estimating Operation Cost

The problem of scheduling a list of operations with constraints is an NP complete problem. Translating the problem from finding the schedule and cost of the operations into a "block matching problem" of minimizing the time-span in completing the operations led us to efficient algorithms of finding approximated solutions.

Below we describe an efficient way of estimating the cost of operations. Our approximate solution for the scheduling problem is to place the cost object of each operation into the lowest time slots that all cost components of the operation can fit simultaneously. [2] We derived a linear time algorithm to solve this modified problem. Before we describe the algorithm, we will first discuss some observations that can greatly simplify the solution.

First, only the overall cost of the operations in the basic block is of interest; there is no need to keep track of the actual order or schedule of the operations. Second, since the mission of the algorithm is to find the lowest time slots for an operation in the functional units, the data structure that represents the time slots needs to allow the cost model to search through or update the time slots quickly. Third, only a certain number of slots (called *focus span*) under the highest occupied time slot need to be considered. This can greatly reduce the complexity of the problem when reordering of operations is considered. It also minimizes the space requirement for storing information about time slots. Furthermore, the *focus span* is an adjustable parameter, thus allowing more flexible allocation of computing resources based on accuracy and efficiency considerations.

Based on the above observations, the time slots of instruction execution units are decomposed into lists of alternating filled and empty blocks that are represented by a two-dimensional array. The first and last slots of a block are used to record the size of the block. If the block is empty, we record the negative value of the block size (see Figure 4). The array representation has the advantages of double linked lists since reaching the adjacent blocks is only one operation. It also allows corresponding time slots in other bins be found quickly. By looking at blocks instead of individual array elements, simultaneously searching for empty spaces in multiple bins can be done much more efficiently with our data structure than regular array or list representations.

[1] I hope this reminds you the computer game Tetris.

[2] This cost model can be used as a base for a full feature scheduling algorithm by allowing reordering of operations that are already scheduled.

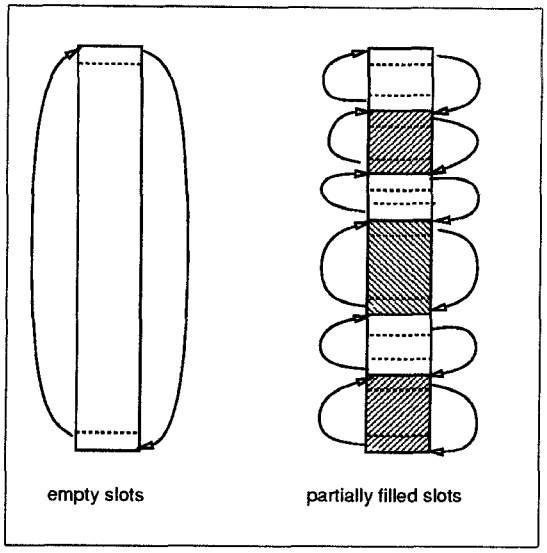

Figure 4: Data structure used for representing a list of time slots for a functional unit.

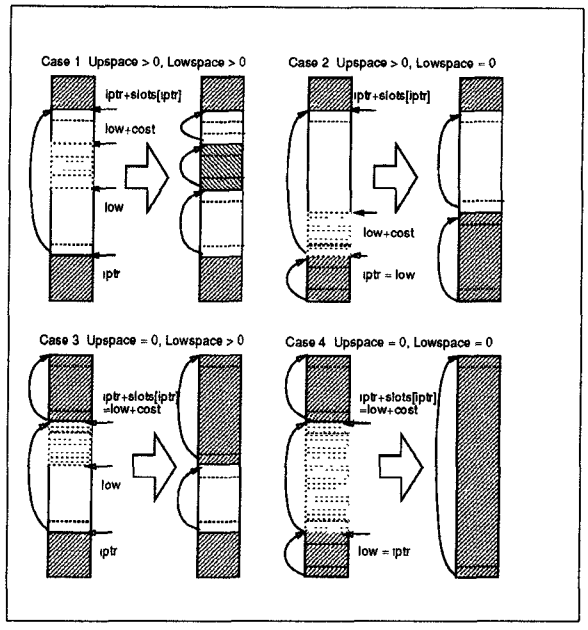

Figure 5: Adding an atomic operation to time slots of a functional unit.

2.2 The Instruction Translation Module

The instruction translation module has two responsibilities. First, it converts expressions in a high level language (such as HPF) into machine level instructions. Second, it imitates the compiler back-end to perform common optimizations so that the cost estimate will match the cost of code that will eventually be generated.

It is possible that one can use an existing module in the compiler – the back-end code generator – to generate low level code for performance estimation purposes. In fact, the IBM *xlf* and *xlc* compilers provide such a facility for estimating the cost of assembly instructions [3]. However, since the compiler needs the cost estimate during the program restructuring process (before back-end is invoked); it is impractical at this stage to do code generation for every intermediate step. Therefore, an efficient substitute is needed. The instruction translation module utilizes information made available by the program analysis module and a detailed knowledge about language and machines to generate a stream of operations based on the input program.

2.2.1 Adapting to Languages and Target Machines

The instruction translation module uses four tables to perform the instruction translation: the *high-level operation table* which represents operations in the high level languages; the *basic operation table*, which is a table of predefined operations that are type-specific, is capable of representing operations in different languages; the *atomic operation table* which contains predefined atomic operations that represent low level machine operations; and the *atomic operation cost table*, which stores the costs of each atomic operation.

The actual operation translation is done in a two level translation process (as shown in Figure 6). In the first level, the *operation specialization mapping* translates language specific expressions into language independent basic operations such as integer-add operation, floating-point multiply-add operation, etc.

The second level translation, called *atomic operation mapping*, translates the basic operations into a list of atomic operations. The cost model then uses the *atomic operation cost table* to access costs of each operation (different atomic operations may have the same costs).

Operation specialization mapping is language dependent but architecture independent, while *atomic operation mapping* is architecture dependent but language independent. This simple arrangement has many advantages. First, different *operation specialization mappings* can be defined for different high level languages; and different *atomic operation mappings* and *atomic operation cost tables* can be defined for different architectures. This allows the cost model be used for multiple languages and architectures. Since the cost model operates at the machine level, the atomic operation cost table can be easily set up based on manufacturer's specifications. When low level cost information is not available, a training-set like approach can be used

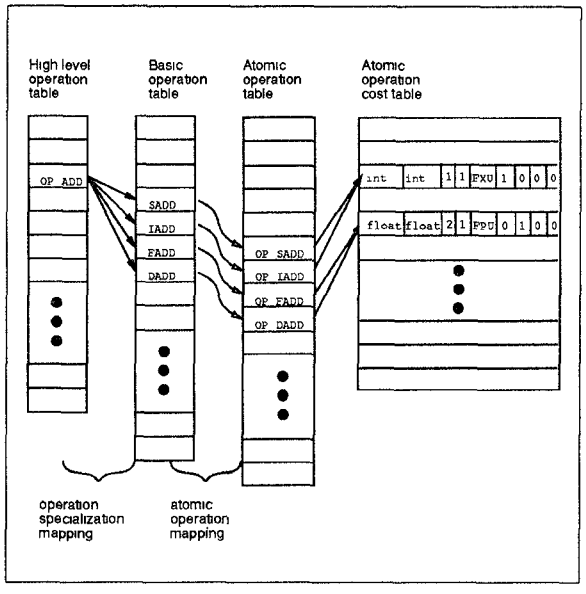

Figure 6: Tables used for translating high level operations into atomic operations.

with some loss of precision. Adding a new architecture to the cost model is a matter of defining the *atomic operation mapping* and the *atomic operation cost table.*

Some architectures have operations that take variable time to execute. For example, on IBM RS 6000, the integer multiply takes three cycles when the multiplier has a value between -128 and 127, but takes five cycles for general values. For this case, multiple basic operations are used to represent the integer-multiply operation, and the *operation specialization mapping* can map different cases to different basic operations.

Architecture specific operations such as the multiply-and-add operations are recognized by the compiler and represented in the *high level operation table.* They are mapped to low level atomic operations if the architecture supports them. The effect of the limited number of registers on performance is simulated by using a heuristic that forces a store after certain number of loads.

2.2.2 Specialization for Compilers

There many some optimizations that are routinely applied by compilers. These transformations include operation overlapping (see Section 2.1), code motion to move loop invariant or inductive expressions out of loops, common sub-expression optimization, dead code elimination, branch prediction, speculative scheduling, etc. Most of these optimizations are done by the compiler back end [4] (or are more convenient to be left to the back-end to do). However, the performance estimation is used by program restructurer which is several phases before the code generation phase. This implies that the cost model needs to imitate these optimizations to get accurate estimates. The following is a list of low-level compiler optimizations that our cost model can estimate their effects:[3]

- *Speculative scheduling and code motion.* Speculative scheduling and code motion are handled naturally by the base model.

- *Branch optimizations.* IBM xlf and xlc compilers are capable of branch optimizations, such as code replication, gluing, branch swapping, etc. to minimize the cost of branches [3]. The cost model handles the branch optimization by matching shapes of the cost blocks to decide whether the branching cost needs to be included.

- *Loop unrolling.* If the compiler does not unroll the loop in the transformation phase, it might unroll the loop in the code generation phase. For a loop with a small basic block, unrolling the loop a few times is usually enough to enlarge the innermost simple basic block so that sufficient overlapping is possible. Our model provides two ways for estimating cost saving of unrolling a loop: examining the shape of the cost block or dropping the innermost basic block into the functional bins multiple times.

- *Sum-reduction operations, etc.* The cost model can use pattern matching techniques to recognize some commonly used operations such as sum-reductions for which all but one store instruction can be eliminated by using registers. The same technique can be applied to other operations such as inner products, array-constant multiply, or array multiplications, etc.

- *Common sub-expressions, loop invariant or inductive expressions.* Evaluating common sub-expression only once and moving loop invariant or loop inductive expressions outside loops are done in the operation specialization phase. Two functional bins are used to count the one-time and iterative costs separately.

Porting the cost model to a new compiler is more involved because the cost model needs to be tuned based on the low level optimization capability of the compiler. To ease this process, flags representing the optimization capabilities of the back-end are defined and used for tuning the cost model.

[3] the cost model does not need to do most of the analysis needed for these tasks since program analyzer can provide these information.

Time in cycles	F1	F2	F3	F4	F5	F6	F7	Matmul	Jacobi	RB
IBM xlf	6	10	16	19	34	60	279	46	11	10
Our model	6	10	19	17	31	63	234	50	10	7

Figure 7: Results of straight line code examples.

2.2.3 Preliminary Results

Figure 7 shows some preliminary data of our system. Our result is obtained by applying the ptran2 compiler on some small F90 programs. It is compared against the estimation provided by IBM xlf compiler. [4] Since the cycle counts provided by xlf does not include function call and memory costs, they are excluded from our results for comparison purpose.

For the table in Figure 7, F1-F7 are innermost basic blocks taken from Purdue benchmarks in the HPF Benchmark suite. Matmul is the innermost basic block of a matrix-multiply loop which is blocked and unrolled 4 times in both dimensions (a total of 16 FMA operations in the basic block). Jacobi is the innermost basic block of Jacobi loops. And RB is the innermost basic block of the red-black loops. This comparison is preliminary and only covers the straight-line cost estimation. A more thorough performance comparison needs to be done to access the overall performance of the prediction. We plan to compare the estimated performance of the HPF benchmark programs to the actual run-time. This will be done when the communication cost and symbolic manipulation modules are completed.

2.3 The Memory Access Cost

The memory access cost (cache misses, TLB misses and page faults) is computed independent from the straight line code estimation because the former is a more global matter. Many methodologies for estimating cache cost were proposed [8, 14, 10]. We adopt an algorithm that was introduced in [8]. The total number of cache line accesses is counted and the cost of filling these cache lines is used to approximate the memory cost.

2.4 Cost Aggregation of Compound Statements

We will first discuss a cost aggregation model for general architectures and then present a cost aggregation model for superscalar architectures using cost blocks.

[4] The IBM xlf compiler prints out a listing of assembly code with a cycle count for each assembly instruction when the flags *-qdebug=cycles -qlisting* are specified. We identified the basic blocks and added up the cycle counts by hand to get the reference data.

2.4.1 Symbolic Cost Aggregation

At the simple basic block level, the counters for each basic operation contain integer values. At the compound statement level, we use symbolic expressions that we call *performance expressions* to represent the estimated performance.

The cost of executing a sequential loop is the cost of running all iterations plus the cost of computing upper and lower bounds:

$$C(do\ i_k = lb_k,\ ub_k,\ step\ \{B\}) = C(lb_k) + C(ub_k) + C(step) + \sum_{k \in Iter} C(B,k)$$

For conditional statements, the cost of the conditional statement is the sum of the follows: the cost of the conditional expression, the cost of the true branch ($C(B_t)$) times the branching probability of the true branch ($p_t(cond)$), the cost of the false branch ($C(B_f)$) times the branching probability of the false branch ($p_f(cond)$), and the estimated branching cost c_{br} (which may be zero, see Section 2.2):

$$C(if\ (cond)\ B_t\ else\ B_f\ endif) = \mathrm{C(cond)} + p_t(cond) * C(B_t) + p_f(cond) * C(B_f) + c_{br}$$

The major difference between our cost aggregation model and previous work is that we compute and represent performance expressions symbolically when control structures contain unknowns. This preserve the precision of the estimates and has a profound affect on the way the estimates are used in optimization.

2.4.2 Cost Blocks and Their Uses in Cost Aggregation

The first and last occupied time slots in functional units define the actual cost of a basic block and the area they enclosed is called the cost block (as shown in Figure 8) of the basic block. Using our data structure, the shape of the cost block is defined by the first and last rows and the height of the block. The shape of the cost block reveals many useful information that can be used to combine costs of adjacent basic blocks or aggregate costs of compound statements. For example, overlapping between basic blocks or iterations of a loop can be estimated by matching the top and bottom of the geometry shape of the cost block (see example in Figure 9). By checking the ratio of the occupied and empty slots in the critical functional bin(s), the compiler can decide whether statement reordering and loop unrolling are beneficial. The shapes of the cost blocks can be used to decide the order of statement blocks or the rough estimation of the loop unrolling factor.

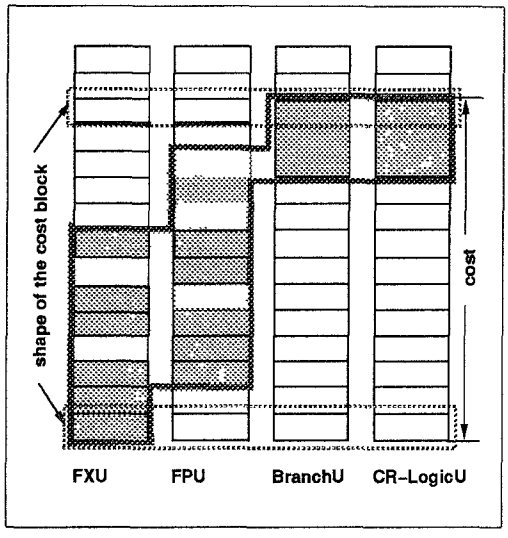

Figure 8: A cost block.

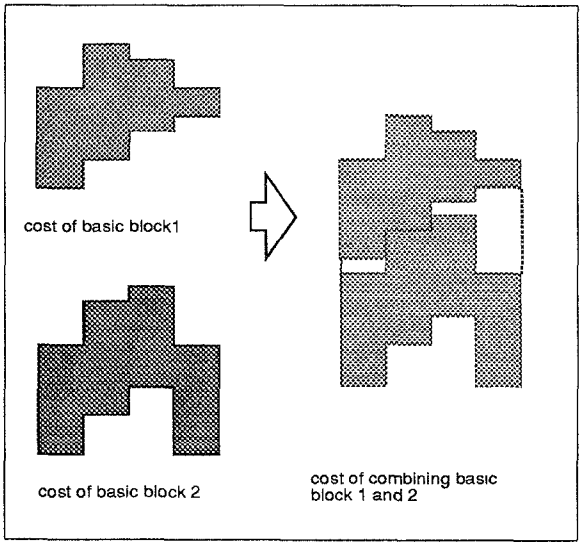

Figure 9: An example of overlapping between basic blocks.

The cost of branch operations can be estimated by checking the number of load instructions before operations in other units started (this can be approximated as the difference between the bottom of FXU and other units).

3 Performance-Guided Program Optimization

The compiler is interested in both comparing the effects of different transformations and in choosing the best parameters in a selected transformation. Basing these decisions on performance estimation allows the compiler to make better decisions. In traditional compilers, when there are unknowns in the control structures, the compilers guess the values of the unknowns (or the reaching probabilities). Although this makes the performance comparison simple (comparing two numbers), the results are highly unreliable. Our idea is to delay the guesses by incorporating these unknowns into the performance expressions. There are many situations where it is possible to determine whether the expression is positive or negative based on bounds on the variables. In these cases, the compiler may not have to guess values of the unknowns.

3.1 Symbolic Comparison of Performance Expressions

The solution of the problem of symbolic comparison of performance expressions is the subject of another paper, but we discuss a few examples here.

For example, if the difference in the performance expressions is a polynomial of one variable, [5] then it is simple to find the roots of the equations for polynomials of up to degree of 4. And since polynomials are continuous, it is usually straight forward to determine the range(s) where the expression is positive. Assuming performance changes of transformations f and g are $C(f)$ and $C(g)$, and let $P = C(f) - C(g)$, and P^+ and P^- be the positive and negative functions of P then f is better than g on regions where $P^-! = 0$. The function P can be used to find regions where f performance better than g. Either the value of the function, size of the area where $P+$ and P^- are nonzero, or integral values of $P+$ and P^- can be used to compare the transformations f and g. Figure 10 shows regions of a general cubical function for which the value of the function is negative.

For cases where the bounds on the related variables are not enough to decide whether the value of the expression is positive, the compiler can compute the condition when the value is positive (this can be used in

[5] Since loop transformations modify only one structure at a time, this is likely to happen.

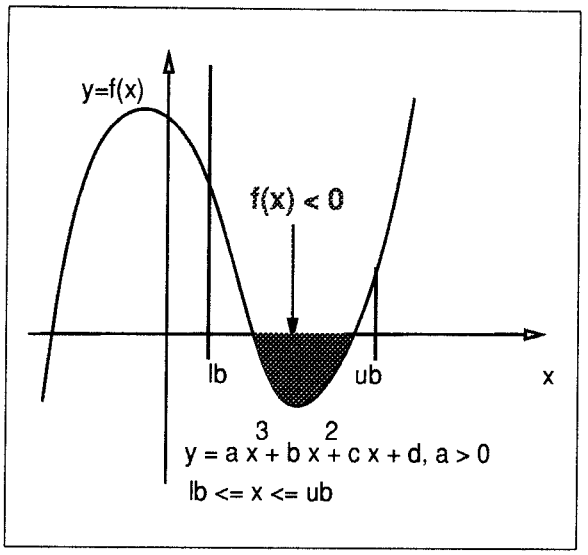

Figure 10: A cubical polynomial and its values

generating run-time tests), or guess the values of the unknowns that are involved.

It is also possible for the compiler to change expressions to simpler expressions by dropping some terms. For example, if the range of x is $[3, 100]$, then the equation $4x^4 + 2x^3 - 4x + 1/x^3$ can be changed into $4x^4 + 2x^3 - 4x$.

3.2 Automatic Program Transformation Based on Performance Estimation

One importance feature of the proposed framework is that it supports automatic program optimization. Based on the symbolic performance comparison, the compiler can utilize graph search algorithms, such as the A^* algorithm, to choose program transformation sequence systematically.

3.3 Minimizing Cost in Estimating Performance

In addition to the use of efficient algorithms in computing the cost estimations, the cost of performance prediction can be reduced by updating the prediction incrementally and avoiding unnecessary computations whenever possible.

3.3.1 Incremental Update of Predictions

The performance prediction framework needs to support incremental update so that cost of maintaining up-to-date performance during the program optimization process is as small as possible.

To avoid unnecessary recomputing, each transformation defines an *affected region* of performance based on the structure it changes. The affected region is defined for each category of the estimation. For example, when a loop is blocked, the execution time for the straight line code inside the loop is not changed, but the expression that computes the number of iterations should be changed. The cache access cost for the loop is also changed and we can either substitute the bounds of the new inner loop into the expression (if they are represented symbolically – another advantage for symbolic processing) or recompute the cache cost.

When choosing among two transformations, only the changes that the transformations have on the performance expressions need to be computed. This usually allows cheaper evaluation before the transformations are actually carried out.

3.3.2 Avoid Unnecessary Performance Computations

There are many cases when detailed performance data is not needed, or the performance computation or representation can be simplified to improve efficiency. The following heuristics can be used to minimize the cost of static performance prediction.

- If the two branches of a conditional statement have performance estimations that are very close, the reaching probability of the two branches can be ignored and the performance of the conditional statement can be simplified.
- Some simple conditional expressions whose reaching probabilities can be guessed should be recognized whenever possible. For example, when a variable in the conditional expression is a loop index, we may assume equal probability for each iteration of the loop (i.e. for a loop with loop bounds in $[lb, ub]$ and step *step*, the probability that the index has a particular value is $step/(ub - lb)$.
- If the cost of one branch of a conditional statement is small and the iteration set that falls into that branch is also small compared to the other branch, then the cost of that branch may be ignored.

For example, consider the loop statement that has a nested conditional statement (where k is unknown):

```
do i = 1, n, 1
   if (i .le. k) then B_t
   else B_f
   endif
enddo
```

the cost of the conditional statement inside loop is:

$$C(L) = k * C(B_t) + (n - k) * C(B_f)$$

and if $C(B_t) \simeq C(B_f)$ then the cost expression can be further simplified to be:

$$C(L) = n * C(B_t)$$

3.4 Profiling and Run-Time Tests

Profiling [15] can be used to eliminate some variables that result from unknown values in the control structures (such as the branching probabilities of conditional statements). This is useful when the program behavior is relatively independent of the input data.

Multiple branches of instructions guided by well-chosen run-time tests can be effective for programs whose performances depend on input data. However, deciding when and how to generate effective run-time tests is an open problem. Usually only a few run-time tests can be afforded to maximize the benefit of multiple branches of control. Excessive run-time tests may lead to negative effects on performance. Our use of performance expressions provides a foundation for solving this problem.

After the performance expression is found for a program fragment, sensitivity analysis can be applied to find the top few variables that produce the most perturbations to the performance. (Sensitivity analysis varies the values of the variables for small amounts and measures the resulting perturbations to the values of the function). Run-time tests can be formulated based on the most sensitive variables. Furthermore, the conditions on the performance expressions can be used to formulate the run-times tests.

3.5 Procedure and Library Routine Interface.

Table look-up of the performance expression can be used to find the cost of external function calls or library routines. If the source code of a library is not available, approaches such as the training-sets can be used to get run-time measurements to build the performance table. However, this usually results in estimates that are only good for certain function arguments. If source code is available, the performance expressions of the external library routines can be computed and stored in an *external library cost table*. The performance expressions are parameterized with the formal parameters. Actual parameters are substituted at the call site to get more specific performance expressions.

4 Related Work

The "load/store" modeling method used in [9, 5] characterizes the performance of shared memory architectures by a set of templates of vector load, store, and "nop" instructions. This works at the assembly level and only reflects the cost of memory hierarchy. D. Atapattu and D. Gannon [AtGa89] built an interactive tool that used a similar analytical machine model to predict performance for Alliant FX/8.

V. Sarkar [15] computes at compile time a set of performance parameters and estimates execution time based on profiling data for single assignment languages. The symbolic performance analysis we introduced here allows the compiler to rely less on profiling and, at the mean time, preserves the accuracy of the estimation.

V. Balasundaram et al. [2] presented a performance estimator for evaluating the relative efficiency of data partitioning schemes by computing cost of message passing statically. They assumed constant loop bounds and guessed for values of unknowns in programs.

T. Fahringer and H. Zima [7] discussed a static performance prediction tool which uses a combination of a parameter-based performance tool and a profiler. Their system attempts to correlate statically computed parameters and the actual measurements, while our model actually derives a precise mathematical expression based on a set of implicit parameters to represent the run-time cost of the program. A. Gemund [17] defines a modeling language to model the serialization effects of parallel computer systems.

Our code-model for message-passing is based on [19] which is a parameterized, static, performance prediction tool that supports different types of architectures. This tools characterized program performance into a set of cost categories that includes instructions, cache, message passing, synchronization, and hot spot contentions, etc.

To summarize, the work reported in this paper distinct from previous work in the following areas.

1. The framework we presented in this paper is comprehensive. Different categories of program costs are unified into a single, comparable performance expression.

2. Our cost model for superscalar architecture is the first cost model that can estimate performance of high level programs accurately. The two-level translation approach makes it portable across different architecture platforms.

3. Low level compiler optimizations that are usually done at compiler back-end are imitated in the operation translation process. This allows the cost model to accurately model the program performance at the source language level.

4. Through the symbolic manipulation of performance data, the compiler delays or avoids having to guess values of unknowns in control structures. This not only preserves the accuracy of the performance prediction, but also enables the compiler to

use symbolic comparison to make optimization decisions that are not otherwise possible.

5 Conclusion

In this paper, we have described a framework for performance prediction for superscalar-based parallel computers. The framework is implemented and is an integrated module of the PTRAN II compiler which is a prototype HPF compiler [12]. Preliminary results show that the predictions are fairly accurate for straight-line code on the RS6000 based machines. Work is underway on communication cost for distributed memory computers.

The framework combines four innovative ideas: an efficient but detailed cost model for modeling superscalar architectures, the idea of delaying or avoiding compiler-time guesses to preserve precisions by symbolic manipulation, the two level operation translation mapping that makes performance prediction at the source level possible and maintains portability, and the use of symbolic comparison to select transformations. This research can be applied to other optimizing compilers and may enhance their optimization capability significantly. Many techniques, such as evaluating the effects of transformations symbolically, efficient computation and manipulation of performance data, and systematic graphic search algorithms for guiding program transformation based on performance prediction need to be further developed to make automatic program optimization feasible. Nevertheless, the work reported in this paper paves a foundation for a new generation of parallel compilers that employ performance prediction to aggressively optimize program automatically.

ACKNOWLEDGMENTS

I would like to acknowledge the valuable comments and insight of Fran Allen, Mike Burke, W.M. Ching, Jeanne Ferrante, Manish Gupta, Roy Ju, K.G. Kumar, Sam Midkiff, Emily Plachy, Vivek Sarkar, Edith Schonberg, and Peter Sweeney.

References

[1] J. Andrews and C. D. Polychronopoulos. *An Analytical Approach to Performance/Cost Modeling of Parallel Computers.* PhD thesis, University of Illinois at Urbana-Champaign, Ctr. Supercomputing Res. & Dev., April 1991. CSRD Report No. 1110.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceeding of the Third ACM Sigplan Symposium on Principles and practice of parallel programming (PPOPP)*, April 1991.

[3] D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish. Performance evaluation of instruction scheduling on the ibm risc system/6000. In *Proceedings of MICRO-25*, pages 226–235, 1992.

[4] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, Ontario, Canada, June 1991.

[5] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D Gannon. Performance evaluation and prediction for parallel algorithms on the bbn gp1000. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 401–413, August 1990.

[6] T. Fahringer, R. Blasko, and H. P. Zima. Automatic performance prediction to support parallelization of fortran programs for massively parallel systems. In *Proc. 6th ACM International Conference on Supercomputing*, pages 347–356, Washington D.C., July 1992.

[7] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 7th International Conference on Supercomputing*, pages 207–219, Tokyo, Japan, July 1993.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, Santa Clara, California, USA, August 1991.

[9] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. Performance prediction of loop constructs on multiprocessor hierarchical-memory systems. In *Proceedings of the ACM International Conference on Supercomputing*, 1989.

[10] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. In *Proceedings of the 1987 International Conference on Supercomputing*, pages 229–254, 1987.

[11] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. 6th International Parallel Processing Symposium*, Beverly Hills, California, March 1992.

[12] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K.Y. Wang, and M. Burke. Ptran ii - a compiler for high performance fortran. In *Proceedings of 4th Workshop on Compilers for Parallel Computers*, Dec 1993.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 1–14, July 1992.

[14] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operation Systems*, Santa Clara, CA, April 1991.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London, 1989.

[16] B. Stramm and F. Berman. Predicting the performance of large programs on scalable multicomputers. In *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[17] A. J. C. van Gemund. Performance prediction of parallel processing systems: the pamela methodology. In *Proceedings of the 7th International Conference on Supercomputing*, pages 318–327, Tokyo, Japan, July 1993.

[18] K. Wang and D. Gannon. Applying ai techniques to program optimization for parallel computers. In *Parallel Processing for Supercomputers and Artificial Intelligence*, pages 441–486. McGraw-Hill, New York, New York, 1989.

[19] K. Wang and E. Houstis. A performance prediction model for parallel compilers. Technical Report CSD-TR-1041, Department of Computer Sciences, Purdue University, November 1990.