Static Performance Estimation in a Parallelizing Computer

Ken Kennedy
Nathaniel McIntosh
Kathryn McKinley

CRPC-TR92204-S
April 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

First revision: May, 1992.
Second revision: October, 1993.
Formerly entitled CRPC-TR92204: "Static Performance Estimation."

Interprocedural Static Performance Estimation

Ken Kennedy Nathaniel McIntosh* Kathryn S. McKinley

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251-1892

Abstract

Static performance estimation seeks to determine at compile time how long a given program construct, such as a loop or a subroutine call, will take to execute. Performance estimation provides useful information to parallelizing compilers, particularly compilers which aggressively use code transformations to improve parallelism. In an interactive parallel programming tool, performance estimation can direct users to the most important and computation-intensive portions of their programs. This paper describes the design and implementation of a performance estimator developed to assist in the parallelization and optimization of scientific Fortran programs for shared-memory multiprocessors in the ParaScope programming environment.

1 Introduction

When compiling scientific Fortran programs for high-performance computer architectures, a compiler must effectively exploit the parallelism in the program and make effective use of the target machine's memory hierarchy. Because such programs spend most of their time executing loops [Knu71], much research has been devoted to techniques for determining which loops in the program may be performed in parallel and which loop transformations may be carried out safely [AK87, Ban88, BC86, Wol89]. After analysis identifies all of the potentially parallel loops and legal transformations, however, a parallelizing compiler must still discover

*Corresponding author. Email: mcintosh@cs.rice.edu. Phone: 713-527-6077. FAX: 713-285-5136.

how to transform the program to best make use of its parallelism and data locality on a specific architecture. Some decisions in this process are very easy to make; parallelizing a large loop is nearly always beneficial, for example. In other cases, however, transformation decisions may not be as clear-cut, and some additional form of analysis is needed.

Performance estimation uses static analysis to annotate interesting program constructs (statements, loops, subroutines) with estimates of their execution time. This information can be useful to a compiler to decide how to use a particular transformation. In contrast to other forms of analysis such as live variable analysis or dependence analysis, where imprecision can result in an incorrect transformation, performance estimation is generally used in a manner which doesn't require absolute accuracy to maintain program correctness. If the estimate generated is a little too large or too small, the program will still have the correct semantics, but the improvements from the optimization will be reduced.

This paper describes the design and implementation of a static performance estimator, along with some experiments designed to measure its usefulness. A major strength of our design is its use of interprocedural analysis, which allows us to deal with modular programs that make good use of subroutine calls, and which adds to the ways in which the information can be used. Also important is the use of training sets, which isolate machine-dependence details as much as possible.

1.1 Example: Latency-hiding transformations

A significant amount of recent research in compilers has focused on restructuring programs to reduce the time spent waiting due to memory latency. These transformations are particularly important for machines with deep memory hierarchies and for distributed-memory multiprocessors, where memory and communications latencies can be very high. Even in situations where it is impossible to eliminate memory latency, it is often feasible to hide it by initiating a read or load and then

performing some other computation while the operation completes. Examples of optimizations based on this idea are software prefetching [MLG92, CKP91] and message hoisting/vectorization [HKT92, Ger90]. Consider the following example:

```
do i = 1, n
   statement 1
   statement 2
   statement 3
   do j = 1, 100
      a(j,i) = ... c(j,kk,n) ...
   enddo
enddo
```

Suppose that we are compiling this loop nest for a computer with a deep memory hierarchy and hardware support for compiler-directed prefetching. The compiler for this machine would like to be able to program the prefetch unit to bring locations into the cache prior to the point where they are used, in order to mask main memory latency. Suppose that the compiler needs to prefetch the region accessed by the reference "c(j,kk,n)" into the cache, and it decides that it can place the prefetch anywhere within the "i" loop prior to the point where the values are used (note that statements 1, 2, and 3 can be arbitrary – they may be assignments, subroutine calls, loops, etc.). This placement choice is an important one. If the compiler places the prefetch too close to the "j" loop, the data will not arrive in time to be in the cache when the references take place. If the compiler places the prefetch too far away from the "j" loop (for example, prior to statement 1) then the prefetched data may displace other cached values which are required by the intervening code. What the compiler really needs to know is how long each of the three statements will take to execute. Static performance estimation is means of providing this information.

The compiler can use performance estimates for each of the statements to place the prefetch in the best location – just far enough ahead of the access to mask the latency.

The longer the memory latencies of the machine, the larger the regions of code for which the compiler needs performance estimates. In the case of compiling for distributed-memory multiprocessors, for example, references to off-processor data may result in communications latencies which are orders of magnitude larger than those encountered for a shared-memory multiprocessor. In order to hide these latencies, the compiler would like to be able to estimate the execution time of much larger regions of the program; such regions may include loops, procedure calls, and so on.

1.2 Other Applications

Whole-program performance estimation can be used to guide programmers or the compiler to regions of a program which are especially important. By ranking the subroutines and loops which account for the largest fraction of the computation performed in a program, performance estimation provides a mechanism for focusing either the compiler or a user on the most computational intensive portions of a program [HHK+93]. Performance estimation for a complete application requires interprocedural analysis.

A variant of static performance estimation can also be used to determine execution time lower and upper bounds for real-time systems [Sha89, Par92]. These bounds can be used at run-time when a program is ready to be scheduled to determine if it is possible to meet real-time constraints in the form of deadlines.

1.3 Profiling

Running a program using a profiler can provide similar information to that provided by static performance estimation. It is true that empirically measuring a program is more accurate for a particular input than making compile-time estimates, since a compiler must be conservative and very often has to make guesses when confronted with branches, control flow, and unknown variables. On the other hand, profiling information may be difficult and time consuming to obtain for some programs. To profile a program, it is first compiled, instrumented and executed one or more times. It is then compiled with the profile information and executed again. For a large scientific program, which operates on large data sets and runs for significant amounts of time on supercomputers, it may be difficult to justify this compile-profile-run-compile-run cycle.

An orthogonal issue is that of keeping profile information up to date. It seems unacceptable to require programmers to re-profile the program each time a change is made. Profiling information which has become out of date due to program changes may be worse than no profiling information at all.

1.4 Outline

This paper is organized into four major sections, followed by a description of related work and conclusions. Section 2 contains background material about the ParaScope parallel programming environment and the types of analysis that are used by the performance estimator. In Section 3 we describe the design of the performance estimator. Section 4 contains experimental results.

2 Background

2.1 ParaScope

The ParaScope programming environment is an integrated tool set designed to assist users in developing parallel programs for shared-memory multiprocessors [CCH+88, KMT91a]. ParaScope gathers and computes the information necessary for program parallelization. For example, it performs interprocedural constant propagation and dependence analysis [GKT91]. This analysis determines if parallelization is *safe*, *i.e.*, if it preserves the meaning of the program. The program compiler uses the analysis to determine program parallelization and optimization. Performance estimation is used in concert with this analysis in the compiler to determine if parallelization is *profitable* [McK92].

The ParaScope Editor, PED, is an interactive parallel programming tool that assists users in parallelizing programs [KMT91a, KMT91b, HHK+93]. It provides the analysis and transformation capabilities of a parallelizing compiler in a powerful editor. In a recent study, researchers using PED often desired more assistance in navigating their way through their programs [HHK+93]. In particular, they wanted PED to guide them to the most time consuming portions of their applications in a methodical fashion. This guidance would enable them to concentrate their efforts on the program parts most likely to yield a high payoff.

We intend to use performance estimation to annotate the program with a ranking of the subroutines and loops by their relative frequency of execution. This information will then be used to assist users in examining their programs in ParaScope.

2.2 Analysis

2.2.1 Dependence Analysis

Dependences describe a partial order between statements that must be maintained to preserve the meaning of a program with sequential semantics. A dependence between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink*. There are two types of dependence: data dependence and control dependence.

Data Dependence A *data dependence* between statements S_1 and S_2, written $S_1 \delta S_2$, indicates that S_1 and S_2 read or write a common memory location in a way that requires their execution order to be preserved [Ber66]. The compiler uses dependence information to determine if a loop's iterations can safely execute in parallel. A dependence is *loop-carried* if its endpoints lie in different iterations of a loop [All83, AK87]. Loop-carried dependences inhibit safe parallelization of the loop.

Control Dependence Intuitively, a *control dependence*, $S_1 \delta_c S_2$, indicates that the execution of S_1 directly determines whether S_2 will be executed and is precisely what performance estimation requires. The following formal definitions of control dependence and the postdominance relation on G_f, the control flow graph, are taken from the literature [FOW87, CFS90]:

Definition 1 A statement x is *postdominated* by a statement y in the control flow graph if every path from x to the exit node of G_f contains y.

Definition 2 Given two statements x, $y \in G_f$, y is *control dependent* on x if and only if:

1. $\exists$ a non-null path p, $x \rightarrow y$, such that y postdominates every node between x and y on p, and

2. y does not postdominate x.

Based on these definitions, a control dependence graph G_{cd} can be built with the control dependence edges $(x,\ y)_l$ where l is the label of the first edge on path $x \rightarrow y$. Performance estimation uses the G_{cd} to determine which decisions effect the execution of a particular statement for programs with arbitrary control flow.

2.2.2 Constant Propagation

ParaScope combines local and interprocedural constant propagation to determine when the values of scalar variables are constant. Interprocedural constant propagation determines the values of scalars on entry to each procedure and as a result of executing each procedure [CCKT86]. The local constant propagation phase determines the values of scalars at particular references. When these values are constants, dependence analysis and performance estimation often produce much better results.

2.2.3 Augmented Call Graph

The program representation for our work on whole program optimization and parallelization requires an *augmented call graph*, G_{ac}, to describe the calling relationships among procedures and to specify loop nests [HKM91]. For this purpose, the program's call graph, which contains the usual *procedure nodes* and *call edges*, is augmented to include special *loop nodes* and *nesting edges*. The loop nodes contain loop header information. If a procedure p contains a loop l, then there will be a nesting edge from the procedure node representing p to the loop node representing l. If a loop l contains a call to a procedure p, then there will be a nesting edge from l to p. Any inner loops are also represented by loop nodes and are children of their surrounding loop. The performance estimator uses this representation to assist in the construction of estimates. The performance estimator also uses this information to map execution frequencies for each loop and procedure in the source.

3 Design

The implementation of the performance estimator was designed with the following goals:

- machine independence
- accurate estimates
- efficiency

In order to provide the most machine-independence, our implementation relies on the use of Fortran *training sets*, which allow a variety of uniprocessors and shared-memory multiprocessors to be modeled. The bulk of the performance estimator is basically machine-independent, allowing it to be easily ported. The drawback of the training set approach is reduced accuracy in the estimates, because of the unmeasured affects of optimizations, and register and cache usage. However, by using advanced analysis such as local and interprocedural constant propagation, we believe that the estimates will be accurate enough for use in a compiler.

To make performance estimation practical for use in a compiler, we use the interprocedural framework in ParaScope [CCH+88] which divides interprocedural problems into two phases, a *local* phase and an *interprocedural* phase [CKT86, Hal91].[1] ParaScope runs the local phase automatically immediately following an editing session. It determines the immediate interprocedural effects of each edited procedure and stores the results in a database. This summary information includes a local performance estimate. The interprocedural phase uses the local summary information to determine estimates for the individual procedures and an estimate the entire program *without* inspecting the Fortran source. Its efficiency is further improved because the two-phase design provides the information needed from each procedure

[1] This framework also naturally supports a modular programming style.

at all times and does not need to be derived on every invocation of the program compiler.

3.1 Overview

3.2 Local Performance Estimation

After an editing session in ParaScope, the local phase is automatically invoked on the changed file. For each changed procedure, it first constructs the procedure's calling interface and the local augmented call graph with all the loops, procedure calls, and their relative positions. It then performs the local phase for each requested variety of interprocedural analysis, among them performance estimation. The local phase of performance estimation has two parts: estimating the time of basic operations via training sets and estimating the execution times for branches and loops.

3.2.1 Using Training Sets to Estimate Basic Operations

To predict the execution time of basic operations such as a multiply or a compare, the performance estimator recognizes the individual language constructs and then looks up the execution time of the construct in a table of performance data for the target architecture. Estimates for statements and basic blocks are then generated by summing the estimates of their components.

The table of performance data is collected using a *training set* [BFKK91]. A training set contains a benchmark code for each operation designed to measure its average execution time. When the training set is run on a target machine, it generates a table of data that includes execution times for most computation-related language constructs. For example, the table contains entries for arithmetic operations such as addition, subtraction, and divide (for each of the various Fortran data types), the cost to perform multidimensional array references, the cost of executing various Fortran intrinsics, and the cost of making a subroutine call.

The training set does not attempt to measure I/O times, nor does it gather any kind of information on the memory hierarchy of the machine. It does not attempt to predict if accesses are to registers, cache or main memory. Instead, the training sets only measure access times for items in cache. This simplification eliminates the need for a precise machine model, but may lead to inaccuracies[2].

Creating a training set is a fairly mechanical and machine independent task. Many of the operations and language constructs in Fortran are benchmarked in a similar fashion. For example, the code to measure floating point division is very similar to the code that measures integer division. As a result, we designed a training set generator rather than just a training set. The generator writes a set of Fortran programs which can be run on a variety of target architectures to produce raw performance data. This design makes generating training sets for new architectures much easier.

3.2.2 Branches and Loops

The local phase of the performance estimator operates on the AST (Abstract Syntax Tree) representation of the procedure in ParaScope. A control dependence graph, G_{cd}, is built from the AST, where nodes in the G_{cd} represent statements in the original program. The local phase of the performance estimator is implemented using a bottom-up pass over the G_{cd}. Nodes in the graph are visited in reverse depth-first order. Using this ordering insures that when a node N is visited, all other nodes which are control-dependent on N have already been visited. In other words, when an estimate is built for any node N, any estimates which contribute to the estimate for N are available.

Consider Figure 1. Statement S_1 is control dependent on the `do j` loop, which

[2]Pfister and Norton discuss detecting memory effects in detail, but do not address the issue of predicting when memory contention or "hot spots" will occur [PN85]. They do address mitigating the effects of memory contention once it is detected. They find that even a single hot spot can severely degrade performance. Goldberg and Hennessy also address these issues [GH91].

in turn is control dependent on the `do i` loop. We first compute the cost expression for S_1, then use it to compute the cost expression for the `do j` loop, which becomes a component used in the estimate of the `do i` loop.

The performance estimator uses local (intraprocedural) symbolic analysis within ParaScope to get more information about variables appearing in guard expressions that affect control flow. Information from local constant propagation is used to see if a given scalar has a constant value. If all of the variables involved in a guard expression are constant, then the control flow can be determined at compile time.

For variables that are not constant, the compiler tries to determine if they are formal parameters of the routine, and if a clear path exists from the entry of the procedure to the use in the guard expression. If the guard expression consists only of constants and formals, then it is recorded in symbolic form, since it is possible that some unknowns in the expression will be resolved during the interprocedural phase. It is fairly common to have loop upper bounds which are formal parameters. As a result, it is particularly important to record the symbolic expression for a loop as opposed to immediately making a guess as to the number of iterations in the loop.

If an array reference or other unknowns appear in a guard expression, then the performance estimator simply records an expression of $\perp$ and guesses a value for the probability that the branch in question is taken (the guess varies depending on the construct, see Section 3.3.1).

In the case of a Fortran `DO` loop, the execution time of the loop is a function of the bounds and step:

$$(UpperBound - LowerBound + 1)/Step * BodyCost$$

Loops which are marked as explicitly parallel are handled by table lookup. The table can be thought of as a function which maps a tuple (P, N, C) to a tuple (B, T),

where

- P is the number of processors available
- N is the number of iterations of the loop
- C is the time to execute the loop body once
- B is the estimated best number of processors for the loop
- T is the estimated total execution time of the loop

This scheme takes into account the overhead of starting and finishing a parallel loop, but ignores ignores any memory contention between the parallel tasks. In particular, it will specify a parallel loop to be run sequentially if there is not sufficient computation.

```
      procedure R(m, n, s, q)
        integer i, j, m, n
        real s(100), q(100)
        do i = 1, m
          do j = 1, 100
S1          s(i) = s(i) + 1
          enddo
          read *, n
          if (n .eq. 0) then
            q(i) = s(i)
          else
            call V(m, q, s)
          endif
        enddo
      end
```

Figure 1: Example of local phase

Consider the program in Figure 1, the local estimate for the execution time of procedure `R` would be

$$\mathtt{m} * ((100 * C_1) + ((0.5 * C_2) + (0.5 * \mathtt{V(m,}\perp\mathtt{,}\perp\mathtt{)}))).$$

Notice that the local estimate preserves for the interprocedural phase the program structure surrounding the procedure call to `V` and when possible, determines constants and symbolics for the actual parameters. Also, the local estimate for procedure `R` is completely independent of procedure `V`.

For a given procedure, the end result of the local phase is a table of (possibly symbolic) execution time estimates, one for each loop in the procedure and one for the entire procedure. This table is stored for use later by the interprocedural phase.

3.3 The Interprocedural Phase

To solve any interprocedural problem, the system begins by building the augmented call graph, G_{ac}. The G_{ac} provides the structure for solving all the other interprocedural problems. The performance estimator uses the results of other interprocedural analyses such as constant propagation, MOD and ALIAS to improve its accuracy. These analyses are therefore performed first. The interprocedural phase of the performance estimator begins by annotating each loop node and procedure node in the G_{ac} with the corresponding local estimate's symbolic expression. The estimator then proceeds as follows:

1. Using the results of interprocedural constant propagation, the expression for each node is simplified when possible.

2. A single backward pass is made over the augmented call graph in reverse depth-first search order. For every call edge $u \rightarrow v$, the symbolic expression for v is substituted into the expression for u in the appropriate place. During this substitution, the formals of v are translated into the actuals at the call site in u. The expression for u is then simplified.

Consider the example in Figure 2. The local estimate of `R` is "`k`$*C_1$" and the local estimate of `Q` is "`50`$*$(`R`(`50`,$\perp$, $\perp$)+`R`($\perp$,$\perp$, $\perp$))". After interprocedural propagation of performance estimates, the estimate of `Q` becomes "`50` $* (50 * C_1 + \perp\ * C_1)$".

```
procedure Q(s, q)
 integer i, j, m, n
 real x(100,100), p(100,100)
 m = 50
 do i = 1, m
     call R(m, s, p)
     call R(i, s, p)
 enddo
end

procedure R(k, s, p)
 integer j
 real x(100,100), p(100,100)
     do j = 1, k
         s(j, k) = s(j, k) + p(k, j)
 enddo
end
```

Figure 2: Interprocedural phase example

3.3.1 Guessing

In some programs, the performance estimator will have to make numerous guesses about control-flow decisions, so it is important that the guesses be made as carefully as possible. Guesses can be refined by looking for certain hints and clues in the program.

For example, when trying to estimate the execution time of the **if** statement in Figure 3(a), the performance estimator can do little but guess arbitrarily at the frequency of the two arms of the conditional branch. In Figure 3(b) however, the array "`a`" is declared to have 10 elements. By recognizing that the induction variable "`i`" is used to index through this array, the performance estimator can make a more precise guess for the upper bound of the **do** loop.

Without any hints, we assume conditional branches are taken equally often and loops are performed 50 times. If the local phase can determine that the induction

variable for a given loop is used as an array index (often the case), then it will guess a number of iterations equal to the dimension size of the array in question. We are experimenting with other ways to improve these guesses.

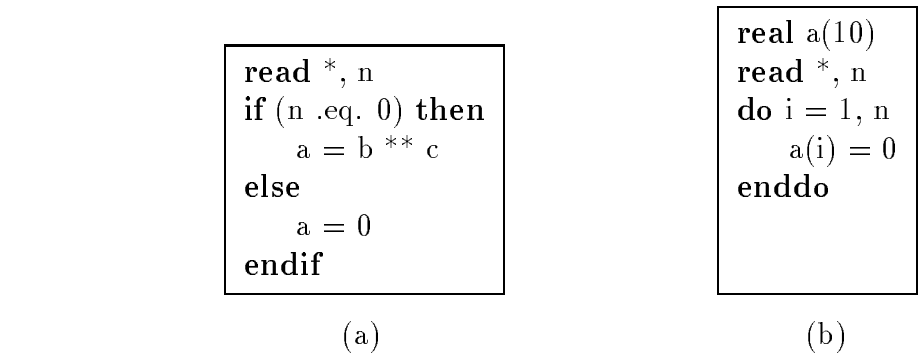

Figure 3: Intraprocedural control flow

3.4 Discussion

Our interprocedural algorithm has a couple of drawbacks. First, it does not support recursion. For scientific Fortran programs, this limitation is not a serious disadvantage, but it restricts the applicability of this technique to other languages. The algorithm can be extended in a fairly straightforward manner to support recursion by locating the strongly connected regions within the call graph, running the algorithm recursively on just the region, compressing the region into a single node, and then continuing.

The second drawback of this approach is the potential for exponential growth in the size of the expressions being maintained during the backward pass over the graph. Due to the style in which many people write scientific Fortran programs, we do not anticipate that this will be a problem in practice. However, the design

includes a cut-off mechanism which imposes an arbitrary limit on the size of the expressions to insure that they do not grow too large.

An additional extension for this algorithm is a forward pass over the call graph to refine interprocedural constant propagation.[3] In this "refinement" phase, instead of trying to find whether a particular parameter has a constant value on every entry to a procedure, a maximum and minimum value for the parameter would be determined, if possible. This approach could be considered an interprocedural version of range propagation and analysis [Har77]. Range information could then be used to derive better guesses for loop upper bounds.

4 Experimental Results

We have completed an implementation of the performance estimator within ParaScope. This implementation consists of the local phase, the interprocedural phase, and a training set for the Sequent Symmetry. This section describes three experiments we performed to test our implementation. The first demonstrates the ability of the performance estimator to correctly identify the relative importance of computationally intensive subroutines for targeting optimization and for interactively guiding users to interesting subroutines. The second seeks to test its effectiveness in determining when to parallelize loop nests. The third experiment demonstrates the importance of interprocedural constant propagation for determining appropriate estimates.

4.1 Target Architecture

All of our experiments were run on a 20-processor Sequent Symmetry S81. Each processor on this machine has a 2-way set-associative 64k cache with a 4-word cache line size and a write-back policy. Coherency is maintained with a snoopy bus scheme.

[3] Nodes are visited in depth-first search order.

4.2 Test Programs

We have tested the performance estimator on the four programs in Figure 4; we plan to also use a number of larger programs, including several more PERFECT benchmarks [CKPK90]. Figure 4 gives the size of each program in lines (excluding comments), number of procedures, and sequential execution time in seconds on the Sequent. **Erlebacher** is a 3-D tri-diagonal solver for the calculation of variable derivatives written by Thomas Eidson at ICASE, NASA-Langley. **Control** computes solutions for linear-quadratic optimal control problems [Wri91]. **Seismic** checks the adjointness of two routines and was written by Michael Lewis at Rice University. **Trfd** is a chemical and physical modeling code from the PERFECT benchmark suite.

4.3 Experiment 1: Identifying important functions

We compared the results of the performance estimator with profiling data gathered using the Unix profiling tool "`gprof`" to determine our ability to identify important functions. The performance estimator was run for each program, and the performance estimates were used to select the 5 procedures containing the most computation. The programs were then run and profiled using "`gprof`" on the Sequent, and the data was inspected by hand to determine the 5 routines containing the most computation.

The data for these experiments is given in Figures 5, 6, and 7. All times in the tables are expressed as a percentage of the total program execution time. The E_{total} column represents the time to execute a call to the specified routine, *including* the time taken by other functions subsequently called during its execution. The E_{local} column represents the time to execute a single call to the routine *minus* the time spent in any descendants. "Count" represents the estimated number of times the routine is called. Finally, the T^P_{total} and T^E_{total} columns represents the total fraction

of the execution time attributable to the routine itself, which is likely to be the most useful number [4]. For the profiling data, T^P_{total} represents the actual percentage of the programs execution time spent in the specified routine.

4.3.1 Discussion

In **erlebacher** (see Figure 5), both the performance estimator and the profiler reported the function "`gensoln`" as the one with the most computation. The profiler and the performance estimator agree on the next four important functions, but order them differently. It is worth noting that the estimated execution times for the routines `tridvpi` and `tridvpj` are both off by an order of magnitude from their actual execution times, but their relative importance (as compared to other routines in the program) is within a much more acceptable margin of error.

The results for **control** are not quite as satisfactory (see Figure 6). Although the performance estimator is able to successfully identify "`lqpdp`" as the single most computation-intensive function, it misses four unlisted subroutines whose share of the total computation ranged from 0.10 to 0.05.

Figure 7 shows the results for the program **trfd**. In this program, the performance estimator and the profiler agree for the first two routines, which account for 99.99 percent of the overall computation. The estimator incorrectly guesses the relative importance of the next three routines, but given their minuscule contribution to total execution time, it is probably not significant.

In **seismic** (see Figure 8), the performance estimator's choice for the most important routine, "`chgadj`" is incorrect. There are couple of reasons for this mistake. First, although the estimate for the number of times the routine is called is correct, the local estimate is artificially high due to a poor choice for the branch frequency of one "`if`" statement in the inner loop of triple nest. The performance estimator guesses that the branch is taken 50 percent of the time, when in reality the

[4] For the estimation case, $T^E_{total} = Count * C_{local}$.

Program	Lines	Functions	Time
control	1902	30	18
erlebacher	615	19	16
seismic	566	20	160
trfd	424	8	2960

Figure 4: Test Program Characteristics

	Performance estimates				Gprof
Function	E_{total}	E_{local}	Count	$\boldsymbol{T}^E_{total}$	$\boldsymbol{T}^P_{total}$
gensoln	0.606	0.606	1	0.606	0.25
genvar	0.212	0.212	1	0.212	0.08
compare	0.123	0.123	1	0.123	0.12
tridvpi	0.015	0.015	1	0.015	0.14
tridvpj	0.015	0.015	1	0.015	0.13

Figure 5: Performance estimates for **erlebacher**

	Performance estimates				Gprof
Function	E_{total}	E_{local}	Count	$\boldsymbol{T}^E_{total}$	$\boldsymbol{T}^P_{total}$
lqpdp	0.016	0.016	1	0.797	0.52
rimain	0.176	0.172	1	0.172	0.01
daxpy	0.000	0.000	23184	0.010	0.02
ddot	0.000	0.000	5996	0.005	0.03
riqy	0.005	0.005	1	0.005	0.00

Figure 6: Performance estimates for **control**

	Performance estimates				Gprof
Function	E_{total}	E_{local}	Count	$\boldsymbol{T}^E_{total}$	$\boldsymbol{T}^P_{total}$
olda	0.307	0.307	3	0.921	0.97
intgrl	0.014	0.014	3	0.042	0.02
trfprt	0.012	0.008	3	0.025	0.00
trfout	0.000	0.000	8	0.012	0.00
trfd	1.000	0.000	1	0.000	0.00

Figure 7: Performance estimates for **trfd**

percentage is much lower. Second, the performance estimator incorrectly handled a conditional loop exit in the same inner loop due to a bug. However, the performance estimator and the profiler have identical choices for the next four routines.

We find these results quite encouraging; in most cases, the performance estimator does an acceptable job of finding routines which are computation-intensive. It should be noted that the study we performed focuses exclusively on relative execution time and not absolute execution time. Our experience was that the performance estimator did a poor job at determining the absolute execution time, and was occasionally off by as much as an order of magnitude. This result did not have however an adverse effect on the computation of relative execution times.

4.4 Experiment 2: Estimate-based parallelization

The following study was designed to illustrate the effects of introducing unprofitable parallelism. We first obtained a set of hand-coded parallel programs for various architectures. Each hand-coded program was transformed into a "nearby" sequential version by changing all parallel loops to sequential loops. The nearby sequential programs could then be parallelized using two strategies. In the first strategy, called "Brute-force", all loops which could legally be made parallel were made parallel [5]. In the second strategy, called "Estimate-based", loops were made parallel only if they were legally parallelizable *and* the performance estimator found that it would be profitable to run them in parallel. In this study, only loops which did not contain subroutine calls were considered for parallelization.

Figure 9 presents the results for **erlebacher**, **control** and **seismic**. For **erlebacher**, the estimate-based parallelization produced was identical to the brute-force parallelization. In **control**, the only difference between the two was a single loop which was parallelized by the brute-force strategy and left sequential by the estimate-based strategy. Because this loop did not contribute significantly to

[5] This system will not produce multiple levels of parallelism, only a single level.

the overall execution time, no discernible difference in execution time occurred. In **seismic**, several parallel loops were performed sequentially due to insufficient computation and although each of these executed more quickly than their parallel counterpart the overall execution time increased. The results reveal that the rest of the program executes faster when these loops are parallel. Perhaps the loops which were not estimated to perform better in parallel are preloading the caches of the individual processors for later parallel loops and thus making them execute faster.

These findings indicate that when parallel loop overhead is small and it is the only factor, overall program performance is not effected very much by executing small loops either way. Loops without enough computation to merit parallelization are unlikely to contribute significantly to total execution time. Unfortunately, other and harder to predict factors may also need to be considered. This type of estimation, of course, increases in important for shared-memory machines with high startup costs.

4.5 Experiment 3: Effects of interprocedural constant propagation

Figure 10 shows some of the effects of interprocedural constant propagation on the accuracy of the performance estimates. In this study, estimates for each program were computed with and without interprocedural constant propagation. For each of the top 5 computation-intensive procedures (as determined by profiling), the figure shows whether interprocedural constants produced any improvement or degradation in the accuracy of the estimates. Changes of less than 1 percent were not recorded. As can be clearly seen, interprocedural constant propagation has very little effect on **erlebacher** and **trfd**, but for the other two programs, it improved the results. In the case of **seismic**, the improvement was quite drastic. Without interprocedural constant propagation, the estimates for several of the most important procedures were significantly flawed, by as much as 10 to 15 percent of the total execution time.

Function	Performance estimates E_{total}	E_{local}	Count	T^E_{total}	Gprof T^P_{total}
chgadj	0.006	0.006	50	0.288	0.04
afold	0.007	0.007	25	0.180	0.31
fold	0.007	0.007	24	0.173	0.31
chgvar	0.001	0.001	98	0.137	0.06
sdvtt	0.003	0.001	48	0.060	0.04

Figure 8: Performance estimates for **seismic**

Execution times in seconds			
Program	Sequential	Brute-Force	PE-based
Erlebacher	15.770	3.020	3.020
Control	17.410	17.210	17.210
Seismic	155.860	12.496	12.572

Figure 9: Program execution times

	Number of procedure estimates changed	
Program	Improved	Degraded
control	1	0
erlebacher	0	0
seismic	4	1
trfd	0	0

Figure 10: Change due to Interprocedural Constant Information

We believe interprocedural constant propagation and the interprocedural structure of our estimator are key to the quality of our implementation.

5 Related Work

Balasundaram *et al.* introduce a static performance estimator for distributed memory machines which pioneered the use of training sets [BFKK90, BFKK91]. They use training sets to create a cost model for an architecture by summarizing empirically obtained data, as opposed to using a theoretical machine model. Our approach is modeled on this one, but offers an efficient and practical solution for programs that contain procedure calls.

Atapattu and Gannon describe a performance predictor as part of a general multiprocessor programming environment [AG89]. In their system, a user can interactively request a performance estimate for a particular loop or procedure. Their estimator works by disassembling the object code for the procedure or loop in question, and then generating an expression which represents its estimated execution time. This expression, which may involve symbolics, is displayed to the user to illustrate what the compiler is doing with the code. By looking at the actual assembly language output of the compiler, they can generate fairly accurate estimates, but at the price of making their estimator very architecture specific.

Both Polychronopoulos and Sarkar have also used machine models in their research which estimate the amount of computation in a loop [Pol86, Sar89]. These approaches are similar to Atapattu and Gannon's in that they are very architecture specific.

Gabber, Averbach and Yehudai generate estimates of execution times of loops in their compiler as a means of deciding when to make loops parallel [GAY91]. Their estimates are calculated in a way that is similar to the method we are proposing, however their compiler does not perform interprocedural analysis. Additionally, they

have a fixed policy for guessing unknowns. For example, all branches are assumed to be taken 50 percent of the time and all loops with unknown bounds are assumed to run from 1 to 50.

Fahringer, Blasko, and Zima use performance prediction as part of the Vienna Fortran Compilation System to assist in automatic support for data distribution [FBZ92]. Their approach provides good precision, but at the cost of requiring an initial profiling run to generate values for unknown symbolics.

6 Future work

Much work remains to be done on the ParaScope performance estimator implementation. Planned improvements include better use of the ParaScope symbolic analysis, training sets for additional architectures, and a better model for predicting the cost of memory accesses.

In recent evaluations of the ParaScope Editor, users requested performance estimation or profiling be integrated into the tool in order to guide them to the computation-intensive portions of their program [HHK+93]. As we demonstrated in the introduction, if user edits or transformations are performed, profiling information is insufficient. The additional flexibility offered by performance estimation and its relative accuracy serve this purpose well.

If profiling information is available at the loop and basic block level, then a hybrid approach which uses actual execution times as the initial estimates might provide the most accuracy and flexibility. If an edit or transformation occurs, then the performance estimator could build an estimate for the changed module and invalidate profiled times for any affected module. This approach would work well with our modular two-phase design.

7 Summary

The ParaScope performance estimator has a number of aspects which will contribute to its effectiveness. Its use of training sets isolate machine-dependent code and allow it to be ported to new architectures with a minimum of effort. By using interprocedural analysis, it can operate on complete applications in an efficient and practical manner. Control dependence analysis allows it to handle functions which use unstructured control flow, greatly enlarging the class of programs which can be analyzed. By collecting static estimates, our design can deal with situations where profiling data is unavailable or when it loses meaning following transformations. Our performance estimator has shown promise in the area of guiding the user and the compiler to the most computation-intensive subroutines in a program, and we hope that subsequent experiments will prove it useful for assisting in selecting transformations during parallel code generation.

References

[AG89] D. Atapattu and D. Gannon. Building analytical models into an interactive performance prediction tool. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[CFS90] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[CKPK90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[FBZ92] T. Fahringer, R. Blasko, and H. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GAY91] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice & Experience*, 2(3):171–193, September 1990.

[GH91] A. Goldberg and J. Hennessy. Mtool: A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Dept. of Computer Science, Rice University, April 1991.

[Har77] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[HHK+93] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: an interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[HKM91] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[KMT91a] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMT91b] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[Knu71] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[McK92] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[Par92] Change Yun Park. *Predicting deterministic execution times of real-time programs.* PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1992.

[PN85] G. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Pol86] C. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1986.

[Sar89] V. Sarkar. *Partition and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[Wri91] S. J. Wright. Partitioned dynamic programming for optimal control. *SIAM Journal of Optimization*, 1(4):620–642, November 1991.