Determining Average Program Execution Times and their Variance

Vivek Sarkar
IBM Research
T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598

Abstract

This paper presents a general framework for determining average program execution times and their variance, based on the program's interval structure and control dependence graph. Average execution times and variance values are computed using frequency information from an optimized counter-based execution profile of the program.

1 Introduction

It is important for a compiler to obtain estimates of execution times for subcomputations of an input program, if it is to attempt optimizations related to overhead values in the target architecture. In earlier work [SH86a, SH86b, Sar87, Sar89], we used estimates of execution times to facilitate the automatic partitioning and scheduling of programs written in the single-assignment language, SISAL, for parallel execution on multiprocessors.

In this paper, we present a general framework for estimating average execution times in a program. This approach is based on the *interval structure* [ASU86] and the *control dependence* relation [FOW87], both of which can be derived from an arbitrary reducible control flow graph. Therefore, this framework supports general, unstructured programs, rather than the special case of structured programs that was addressed earlier when dealing with SISAL. Average execution times are derived from frequency values, and from estimated execution times of primitive operations on the target architecture.

Besides showing how *average* execution times can be computed, we also introduce the notion of execution time *variance* which corresponds to the statistical meaning of variance (or standard deviation). We show how variance values can be computed in a manner similar to average execution times. Variance information can be used, for example, to help determine the *chunk size* of parallel loops [KW85].

In our framework, frequency values are obtained from a counter-based execution profile of the program. Rather than using the naive approach of incrementing a counter once per basic block, the control dependence relation is used to increment a counter once for each distinct control dependence branch condition. We use frequency information to estimate average execution times and variances, but this approach to execution profiling based on control dependence and counter variables would be useful for any code optimization that needs frequency information e.g. register allocation [Wal86], trace scheduling [FERN84], optimization of delayed branches [MH86].

This framework for estimating execution times has been implemented as part of the PTRAN (Parallel

Translation) project at IBM Research [ABC*87]. The PTRAN system contains a *program database* which can be conveniently used to store the frequency, execution time and variance information. Currently, the primary use of execution time information in PTRAN is in automatically partitioning the input program into tasks for parallel execution.

2 Program Representation

This section describes the program representations used for estimating execution times. These representations are used in PTRAN [ABC*87], and the approach described here is based on the PTRAN implementation.

The initial representation is assumed to be the traditional *control flow graph* [ASU86]. The nodes of the control flow graph may represent arbitrary units of computation — basic blocks, statements, operations or instructions. The only requirement is that the control flow graph should contain all control flow relations of interest. As in other code analysis and optimization techniques, we assume that the control flow graph is *reducible*. *Node splitting* [ASU86] is a standard approach that can be used to transform an irreducible control flow graph into a reducible control flow graph.

A reducible control flow graph has a unique depth-first spanning tree and hence a unique *interval structure* which can be easily computed from the control flow graph [Bur87, SS79]. The intervals identify the loops in the program. The other program representation we use is the *forward control dependence graph* [Hsi88, CHH89] based on the control dependence relation defined in [FOW87].

Definition 1 *A control flow graph $CFG = (N_c, E_c, T_c)$ consists of*

- *N_c, a set of nodes*
- *$E_c \subseteq N_c \times N_c \times L$, a set of labelled control flow edges, where L is the set of all possible labels.*
- *T_c, a node type mapping. $T_c(n)$ identifies the type of node n as one of the following values — $START$, $STOP$, $HEADER$, $PREHEADER$, $POSTEXIT$, $OTHER$.*

Since there can be multiple edges with different labels between the same pair of nodes in E_c, CFG is in general a *multi-graph*. The classification into node types specified by T_c is only used to help identify the interval structure in the forward control dependence graph computed later. The node type mapping does not change the semantics of the control flow graph in any way.

Figure 1 shows a Fortran code fragment along with the corresponding statement-level control flow graph, CFG. The labels T and F are used to identify the *true* and *false* branches respectively, for an IF statement. The label U is used to identify an *unconditional* branch from a node. All nodes in the original control flow graph have type $= OTHER$.

We first determine the *interval structure* of CFG [Bur87, SS79]. The intervals are summarized in a mapping called HDR, where $HDR(n) = h$ indicates that node h is the *header* of the interval containing node n. Interval *nesting* is stored in a mapping called HDR_PARENT, where $HDR\_PARENT(h_1) = h_2$ indicates that the interval with header node h_1 is an immediate *subinterval* of the interval with header node h_2. $HDR\_PARENT(h) = 0$ indicates[1] that the interval with header node h is the *outermost* interval (we assume that there is exactly one such interval — the one containing n_{first}, the first node to be executed in CFG). HDR_PARENT defines a directed tree on all header nodes in CFG. $HDR\_LCA(h_1, h_2) = h_3$ indicates that header node h_3 is the *least common ancestor* of header nodes h_1 and h_2 in this tree.

After determining the interval structure, the next step is to build an *extended* control flow graph,

[1]Assume that all nodes are numbered from 1 onwards.

$ECFG = (N_e, E_e, T_e)$, as follows:

1. Initialize, $N_e \leftarrow N_c; E_e \leftarrow E_c; T_e \leftarrow T_c$
2. For each header node h in CFG:
   (a) Create a new preheader node, ph, add it to N_e, and mark it as h's preheader
   (b) For each control flow edge (u, h, l) in CFG if $HDR\_LCA(HDR(u), h) \neq h$ then (we have an interval entry):
      i. Replace (u, h, l) by (ph, u, l) in $ECFG$
   (c) Add an unconditional branch from ph to h
3. For each control flow edge (u, v, l) in CFG if $HDR\_LCA(HDR(u), HDR(v)) \neq HDR(u)$ then (we have an interval exit):
   (a) Create a new postexit node, pe, and add it to N_e
   (b) Replace edge (u, v, l) by edges (u, pe, l) and (pe, v, U)
   (c) Add a *pseudo* control flow edge from the preheader node of node u's interval, to the new postexit node pe
4. Add a special $START$ node with an unconditional branch to n_{first}, the first node to be executed in CFG
5. Add a special $STOP$ node with an unconditional branch from n_{last}, the last node to be executed in CFG
6. Add a *pseudo* control flow edge from $START$ to $STOP$

Figure 2 shows the extended control flow graph, $ECFG$, corresponding to the control flow graph from Figure 1. The *pseudo* control flow edges introduced in steps 3.c and 6 have a special label ($Z1$ or $Z2$ in Figure 2) to indicate that the corresponding branch can never be taken in the original program. However, the insertion of these pseudo edges provides a convenient structure to the control dependence graph, as described later. For convenience, we assumed that CFG has a unique *first* node (n_{first} in step 4) and a unique *last* node (n_{last} in step 5. If there is more than one "first" node (e.g. due to multiple entry points) then step 4 should be modified to insert an edge from $START$ to each such node. Similarly, if there is more than one "last" node (e.g. due to RETURN statements), then step 5 should be modified to insert an edge from each such node to $STOP$.

We now turn to the notion of *control dependence* as defined in [FOW87]:

Definition 2 *Let x and y be nodes in a control flow graph. y is control dependent on x with label l if and only if*

1. *y does not post-dominate x*
2. *there exists a directed path P from x to y with all intermediate nodes post-dominated by y*
3. *there exists an edge with label l from node x to the second node in path P*

In other words, there is some edge from x that definitely causes y to execute, and there is also some path from x to $STOP$ that avoids executing y.

Based on this definition, we can build a *control dependence graph*, CDG, containing exactly the edges of the form (x, y, l) that satisfy the above conditions [FOW87, CF87, CFR*89]. However, we will find it more convenient to use an *acyclic* form of the control dependence graph obtained by ignoring all *back edges* in CDG. This is the *forward control dependence graph* [Hsi88, CHH89].

Figure 3 shows the forward control dependence graph, $FCDG$, corresponding to the extended control flow graph, $ECFG$, shown in Figure 2. The tuples enclosed in < ... > and [...] brackets provide frequency and execution time values, which are discussed later. The pseudo control flow edge inserted from $START$ to $STOP$ caused all nodes in $ECFG$, except $STOP$, to be directly or indirectly control dependent on $START$.

Therefore, the forward control dependence graph is rooted and connected. Similarly, the pseudo control flow edges introduced from the *PREHEADER* node to the *POSTEXIT* nodes caused all nodes in the corresponding interval (and subintervals) to be directly or indirectly control dependent on the *PREHEADER* node. The pseudo control flow edges were inserted in *ECFG* so as to obtain this nested structure of intervals in *FCDG*.

3 Automatic Run-time Profiling

In this section, we define the *execution frequency* of a control dependence edge in *FCDG*. Execution frequencies include conditional branch probabilities as well as loop frequencies. These frequency values may be determined by program analysis, or may be obtained from an execution profile of the input program. We believe that program analysis is feasible for only a few restricted cases (e.g. a Fortran DO loop with constant bounds and no conditional loop exits, an IF condition that can be computed at compile-time, etc.), and should be complemented by execution profile information wherever compile-time analysis is unsuccessful.

Definition 3 *Given an edge (u, v, l) in the forward control dependence graph, $FCDG = (N_e, E_f, T_e)$, the execution frequency of the edge is defined as:*

1. *(when u is a preheader node and l is the label connecting u to its header node in ECFG)*
   $FREQ(u, l)$ = average number of times u's header node is executed in one execution of u's interval. In this case, $FREQ(u, l) \geq 0$ represents the loop frequency for u's interval.
2. *(all other cases)*
   $FREQ(u, l)$ = average number of times that node u takes the branch labelled l in one execution of node u. In this case, $0 \leq FREQ(u, l) \leq 1$ represents the branch probability of label l in node u.

Many execution profilers are based on run-time sampling of the program counter, typically done by the operating system. The output of a sampling-based profiler is of the form "Procedure P was found executing x% of the time", which gives an approximate but realistic measure of the relative *execution time* spent in each procedure, for that particular execution run. However, the coarse granularity of the sampling interval makes this approach unsuitable for determining execution frequencies of individual statements, or even small procedures. So, we propose using a counter-based profiler instead, where counter variables are incremented in the compiled code and recorded in a program database at the end of each program execution. The output of a counter-based profiler is of the form "Statement S was executed n times", which gives an exact measure of the *execution frequency* of each statement. This approach is very flexible, in that the frequency information can be generated on any machine, and can be used to estimate execution times for different optimizations/transformations of the program on different target architectures.

An important efficiency advantage of the counter-based approach is that the profiling code is directly compiled in with the program, without requiring any calls to the operating system. However, the potential overhead of counter-based profiling is still a concern if the counter variables need to be incremented too frequently. The naive approach would be to maintain one counter per basic block. In this section, we outline three optimizations for counter-based profiling, based on the interval structure and control dependence graph. With these optimizations, we believe that counter-based profiling is a practical approach. This belief is supported by experimental results presented at the end of this section. Throughout the following, we assume that the program completes execution with no abnormal termination, so that all procedure calls are followed by corresponding returns.

Let $\{(u,l) | (u,v,l) \in E_f\}$ be the set of *control conditions* in *FCDG*. The first optimization is to maintain one counter per control condition, rather than one per basic block. This optimization is based on the observation that identically control dependent nodes must have the same execution frequency. In this way, one counter could serve for several identically control dependent basic blocks. For example, in the following Fortran code fragment, the I=1 and K=3 statements are identically control dependent on the C1=*true* condition, though the statements belong to different basic blocks in the control flow graph. Therefore, only one counter is needed to track the execution frequency of both statements.

```
IF ( C1 ) THEN
     I = 1
     IF ( C2 ) THEN
          J = 1
     ELSE
          J = 2
     ENDIF
     K = 3
ENDIF
L = 4
```

The second optimization applies to any node, u, that has at least one edge (u,v,l) in *FCDG* for each branch labelled l out of u in *CFG*. Of course, it is not necessary for every branch label from u to appear as a control condition in *FCDG*. For example, in the above Fortran code fragment, the L=4 statement is not control dependent on the C1 condition, so there will be no edge in *FCDG* corresponding to C1=*false*. However, if all the branch labels from node u (say n of them) are present as control conditions in *FCDG*, then we observe that the sum of the total execution frequencies of all n labels must equal the total execution frequency of node u. Therefore, we only need to maintain counters for $(n-1)$ of the n labels. This represents a 50% savings when $n = 2$. This optimization can also be applied to loops by making the following observations:

1. The sum of the total execution frequencies of all loop exits must equal the total execution frequency of the loop preheader (i.e. the total number of times the loop is entered from outside).
2. The sum of the total execution frequencies of all control conditions (u,l) that transfer control back to the loop header must equal the difference between the total execution frequencies of the preheader and header nodes.

The two optimizations discussed above are purely syntactic, in that they are derived from the structure of the control flow graph without examining the nodes' contents. If we limit ourselves to syntax-based schemes for counter-based profiling, then these two optimizations will yield the minimum possible number of counter variables and operations. To do better, one would have to use semantic information which could involve exhaustive symbolic analysis of all branch conditions and other parts of the program.

Our third optimization uses limited semantic information, and has been chosen because it provides a large payoff for little effort! The optimization applies to a DO loop with no loop exits, where the number of iterations can be added to the counter variable once at the start of the loop, instead of incrementing the counter by 1 in each iteration. Further, if the number of iterations is known to be a compile-time constant, then it is not necessary to maintain a counter variable at all. In the absence of interval structure information, the best we can do is to perform this optimization only when the DO loop body consists of straight-line code. However, with interval structure information, we can check if any branch in the body of the DO loop is targetted to a node that is not contained (directly or indirectly) in the DO loop's interval. If so, the branch represents a loop exit and the optimization cannot be applied. This is a simple test to do in $FCDG$, where we can just look for an edge to a $POSTEXIT$ node, since the interval structure information was already incorporated in $ECFG$ by adding the $PREHEADER$ and $POSTEXIT$ nodes.

For each control condition (u, l) in $FCDG$, the output of execution profiling is $TOTAL\_FREQ(u, l)$, the *total* number of times the branch labelled l was taken from node u. The *relative* frequency values, $FREQ(u, l)$ from Definition 3, can be computed from the total frequency values by a single top-down pass over $FCDG$. First, we know that $TOTAL\_FREQ(START, U)$ gives the total execution frequency of the procedure or subroutine containing CFG, since it is the total number of times the $START$ node was executed. Let $NODE\_FREQ(u)$ be the average execution frequency of node u, for a single execution of u's subroutine. Then, all $FREQ(u, l)$ values can be computed in a single top-down pass of $FCDG$ by using the following recurrence equations[2]:

1. $NODE\_FREQ(START) = 1$
2. $FREQ(u, l) = \frac{TOTAL\_FREQ(u,l)}{TOTAL\_FREQ(START,U) \times NODE\_FREQ(u)}$
3. $NODE\_FREQ(v) = \sum_{(u,v,l) \in E_f} NODE\_FREQ(u) \times FREQ(u, l)$

Since the only use of $TOTAL\_FREQ$ values is as a *ratio* in equation 2, we can just as well use $TOTAL\_FREQ$ values that are accumulated over several program runs. In fact, it is a good idea to accumulate the $TOTAL\_FREQ$ values (as a sum or average) from different program executions in the program database, so as to get a more representative set of frequency values.

[2]One must take care to avoid a division by zero in equation 2. It can only happen if $TOTAL\_FREQ(START, U) = 0$ or $NODE\_FREQ(u) = 0$, in which case the numerator, $TOTAL\_FREQ(u, l)$ must also $= 0$, so we can correctly set $FREQ(u, l) = 0$ without performing the division.

The $< ... >$ tuples in Figure 3 contain $FREQ$ and $TOTAL\_FREQ$ values respectively, for each edge in $FCDG$. These values correspond to an execution of the code in Figure 1, in which the IF statement with label 10 is executed 10 times, and the loop is exited by taking the $IF(N.LT.0)$ branch. Note that the pseudo edges with labels Z1 and Z2 have zero frequencies.

We end this section with some execution time measurements for the overhead of counter-based profiling. Table 1 gives the CPU time in seconds for the following cases:

1. Original code — time taken for a complete execution of the original program
2. Smart profiling — time taken for a complete execution of the original program, augmented with counter-update operations as dictated by our optimized approach
3. Naive profiling — time taken for a complete execution of the original program, augmented with counter-update operations for the naive approach of one counter per basic block, with the DO loop optimization applied only when the body consists of straight-line code by our optimized approach

These measurements were made on an IBM 3090, using the VS Fortran compiler, Version 2, Release 3. The "Compiler optimization ON" heading stands for full optimization and vectorization, and the "Compiler optimization OFF" heading stands for no optimization or vectorization. The benchmark programs used are:

- LOOPS — a program that executes all 24 Livermore Loops [McM86].
- SIMPLE — a benchmark for computational fluid dynamics and heat flow [CHR78]. The problem size used was 100 x 100, with $NCYCLES = 10$.

The execution times in Table 1 show that the overhead of counter-based profiling is small compared to the performance difference between optimized and unoptimized code, and that "smart profiling" is noticeably more efficient than "naive profiling".

4 Computation of Average Execution Times

In this section, we describe how average execution times can be computed for all nodes in the forward control dependence graph. Having obtained frequency values as described in the previous section, the other important set of values is the execution times of *primitive* operations on the target architecture. We will not discuss in detail the possible techniques for obtaining the costs of primitive operations, as they vary widely for different architectures. A simple approach is to simply count the number of instructions required to implement a primitive operation. A more careful estimation is required when considering pipelined architectures, vector instructions or the effects of cache usage. For the purpose of this work, it is assumed that the (average) local execution time of each node, u, in $FCDG$ has already been estimated, and is stored as $COST(u)$. This section describes the computation of node u's *total* execution time, $TIME(u)$, which includes $COST(u)$ and the frequency-weighted execution times of all of node u's descendants in the forward control dependence graph.

The computation of $TIME(u)$ is based on two simple rules:

1. $TIME(u) =$
   $COST(u) + \sum_{(u,v,l) \in E_f} FREQ(u,l) \times TIME(v)$
   This rule assumes that the node v's execution time is independent of which conditional branch caused it to execute. With this assumption, we can use the same average value, $TIME(v)$, for all predecessors of node v in $FCDG$.

2. If node u is a procedure or function call, then $COST(u) = TIME(START)$, where $START$ is the start node from the callee function's forward control dependence graph. This rule assumes that the execution time of a procedure call is independent of the call site, so that the same average execution time of the procedure, $TIME(START)$, is used as the average execution time of each call to that procedure. This assumption is commonly made in execution profilers e.g. the Unix profiler [GKM82].

Rules 1 and 2 implicitly dictate how the execution time values should be computed. Rule 2 requires that the procedures be visited in a bottom-up traversal of the call graph, so that the root procedure (main program) is visited last. We do not address the issue of cost estimation for recursive procedures in this paper. In [Sar87, Sar89] we described an approach for handling recursive procedures in structured programs, which can easily be extended for use with the forward control dependence graph for unstructured programs. Analogous to rule 2, rule 1 requires that the nodes be visited in a bottom-up traversal of $FCDG$.

The first two values in the [...] tuples in Figure 3 contain the $COST$ and $TIME$ values respectively, for each node in $FCDG$. These values were obtained assuming $COST = 0$ for the $START$, $CONTINUE$, $PREHEADER$ and $POSTEXIT$ nodes, $COST = 1$ for the IF nodes, and $COST = 100$ for the $CALL$ node. The entire program has an estimated execution time of $TIME(START) = 920$.

5 Computation of Variance

This section generalizes the computation of average execution times to the computation of variance. An interesting application of variance information is in determining the optimal *chunk size* [KW85] for the execution of parallel loops on multiprocessors. Intuitively, when the execution time of the loop body has zero variance, we would prefer to use a chunk size of $\lceil N/P \rceil$ for N iterations on P processors, since that provides perfect load balancing with the smallest overhead. However, when the variance is large, we have to move to smaller chunk sizes to get better load balancing, at the cost of increased overhead due to a larger number of chunks. The techniques presented in this section show how variance can be estimated at compile-time, allowing the compiler to choose smaller chunk sizes only when it is really necessary (when the variance is large).

To define variance precisely, let T be the random variable corresponding to the execution time of node v (say). In the previous section, we computed $TIME(v) = E(T)$, the expected value of T. Variance is defined according to the well-known equation, $VAR(v) = E(T^2) - E(T)^2$, and the standard deviation is $STD\_DEV(v) = \sqrt{VAR(v)}$.

The computation of average execution times in the previous section freely used the rules
$E(A + B) = E(A) + E(B)$, and
$E(A \times B) = E(A) \times E(B)$.

Computing variance is more complicated because $VAR(A \times B) \neq VAR(A) \times VAR(B)$ in general, though $VAR(A + B) = VAR(A) + VAR(B)$ is always true. To compute $VAR(A \times B)$, we instead need to use the identities

$$\begin{aligned} E(A^2 \times B^2) &= E(A^2) \times E(B^2) \\ VAR(A) &= E(A^2) - E(A)^2 \end{aligned}$$

to obtain

$$\begin{aligned} VAR(A \times B) &= VAR(A) \times VAR(B) + \\ &\quad E(A)^2 \times VAR(B) + \\ &\quad E(B)^2 \times VAR(A) \end{aligned}$$

The relationship between node execution times in $FCDG$ was stated in the previous section as:

$$\begin{aligned} TIME(u) &= COST(u) \\ &\quad + \sum_{(u,v,l)\in E_f} FREQ(u,l) \times TIME(v) \end{aligned}$$

Based on this relationship between average execution times, $TIME(u)$ and $TIME(v)$, we now derive the relationship between the variance values, $VAR(u)$ and $VAR(v)$.

Let $L(u) = \{l | (u,v,l) \in FCDG\}$ be the set of labels from node u in $FCDG$, and $C(u,l) = \{v | (u,v,l) \in FCDG\}$ be the set of u's children in $FCDG$ with label l. Then the relationship between average execution times can be rewritten as

$$\begin{aligned} TIME(u) &= COST(u) + \\ &\quad \sum_{l\in L(u)} FREQ(u,l) \times \sum_{v\in C(u,l)} TIME(v) \end{aligned}$$

We consider two cases:

<u>*Case 1: u is a preheader node*</u>

There is only one label l from u that is of interest — the label of the $FCDG$ edges connecting u to nodes in the loop body (which includes the header node). All other labels from u go to postexit nodes and have zero execution frequencies. For simplicity, we also assume that $COST(u) = 0$, since preheader node u is a special node with no local computation. Therefore, the above expression for $TIME(u)$ becomes

$$TIME(u) = FREQ(u,l) \times \sum_{v\in C(u,l)} TIME(v)$$

Using the identity for $VAR(A \times B)$ discussed above, we get

$$\begin{aligned} VAR(u) &= FREQ(u,l)^2 \times (\sum_{v\in C(u,l)} VAR(v)) + \\ &\quad VAR(FREQ(u,l)) \times (\sum_{v\in C(u,l)} TIME(v))^2 \\ &\quad + VAR(FREQ(u,l)) \times (\sum_{v\in C(u,l)} VAR(v)) \end{aligned}$$

In this case, $FREQ(u,l)$ represents the average number of iterations in the interval, and the corresponding variance, $VAR(FREQ(u,l))$, can be determined by assuming an appropriate distribution for the number of iterations. If we do not wish to assume a particular distribution for the number of loop iterations, the variance term can also be computed by obtaining $E(FREQ(u,l)^2)$ from execution profile information. If we decide to ignore the variance in $FREQ(u,l)$ and consider $VAR(FREQ(u,l)) = 0$, then the above equation simply becomes

$$VAR(u) = FREQ(u,l)^2 \times \sum_{v \in C(u,l)} VAR(v)$$

Case 2: u is not a preheader node

In this case, $FREQ(u,l)$ represents a branch probability from node u. For convenience, we assume that $VAR(COST(u)) = 0$, i.e. the local execution time at a node has zero variance, though a sophisticated approach to estimating the execution time of primary operations may provide a variance value (e.g. dependent on a cache miss ratio). Similarly, we also assume that $VAR(FREQ(u,l)) = 0$ for all branch probabilities, otherwise we would have to deal with the complication of computing variance when the probability values themselves are random variables with non-zero variance!

Let

$$TIME_C(u) = \sum_{l \in L(u)} FREQ(u,l) \times \sum_{v \in C(u,l)} TIME(v)$$

be the total cost of u's children.

Therefore, $TIME(u) = COST(u) + TIME_C(u)$, and

$$\begin{aligned} VAR(u) &= VAR(COST(u)) + VAR(TIME_C(u)) \\ &= VAR(TIME_C(u)) \\ &= E[TIME_C(u)^2] - E[TIME_C(u)]^2 \end{aligned}$$

But $E[TIME_C(u)^2]$ can be computed as follows:

$$\begin{aligned} & E[TIME_C(u)^2] \\ = & \sum_{l \in L(u)} FREQ(u,l) \times E\Big[\big(\sum_{v \in C(u,l)} TIME(v)\big)^2\Big] \\ = & \sum_{l \in L(u)} FREQ(u,l) \times \Big(VAR\big(\sum_{v \in C(u,l)} TIME(v)\big) \\ & + E\Big[\big(\sum_{v \in C(u,l)} TIME(v)\big)\Big]^2\Big) \\ = & \sum_{l \in L(u)} FREQ(u,l) \times \Big(\sum_{v \in C(u,l)} VAR(v) \\ & + \big(\sum_{v \in C(u,l)} TIME(v)\big)^2\Big) \end{aligned}$$

The dependence between $VAR(u)$ and $VAR(v)$ is similar to that of average execution times, in that it requires a bottom-up traversal of $FCDG$. The last three values in the [...] tuples in Figure 3 contain the $E[TIME(v)^2]$, $VAR(v)$ and $STD\_DEV(v)$ values for our examples. For simplicity, we assumed that $VAR(FREQ(u,l)) = 0$ for the loop frequency as well. Therefore, the only contribution to variance arises from the conditional branches. The estimated standard deviation in execution time for the entire program is $STD\_DEV(START) = 300$ (the average execution time was 920).

6 Related Work

There has been a long-standing interest in measuring execution frequencies, and using the information as feedback to the programmer. An early study of execution frequencies in Fortran program was reported in [Knu71], which discusses both the sampling-based and counter-based approaches to execution profiling. [CK74] presented an approach for determining average execution frequencies from transition probabilities in a control flow graph. It is only recently that automatic program optimizations have been proposed that use frequency information e.g. trace scheduling [FERN84], register allocation [Wal86], optimization of delayed branches [MH86], partitioning and scheduling of parallel programs [SH86a, SH86b]. Given its growing importance, execution profile information ought to become an indispensable component of future programming systems, and the availability of the frequency information will no doubt motivate its use in new optimizations.

Previous efforts in obtaining execution frequencies were typically extended to estimate the total program execution time as the sum,

$$TOTAL\_TIME = \sum_{basic\ block\ B} freq(B) \times time(B)$$

To the best of our knowledge, our work is the first to extend this approach to estimate each statement's *total* execution time by considering the contribution of all statements "contained" within it. For the structured SISAL programs considered in [Sar89], "containment" was defined by lexical nesting; in this paper, we considered unstructured programs where "containment" is defined by the control dependence relation. Also, to the best of our knowledge, this work is the first to define the notion of execution time variance and to provide an algorithm for computing variance.

A far more ambitious approach than the methods presented in this paper is to automatically perform a symbolic complexity analysis of the program, and thus estimate execution times without relying on execution profile information. The complexity analysis problem is, of course, undecidable in general, but there have been a few efforts to solve the problem for restricted cases.

[Weg75] describes Metric, a prototype system that analyzes simple Lisp programs and produces closed-form symbolic expressions for execution times as a function of the length of the arguments, costs of primitive operations, and branch probabilities. Recursion is handled by mapping each recursive procedure into a recursive cost expression, which is mapped into a set of recursive equations, which in turn are mapped into a set of difference equations, which are then finally solved for the integer-valued complexity measures of interest. The approach enforces several approximations and restrictions at each step e.g. all branch probabilities are assumed to be statistically independent, and the call graph is required to be acyclic with possible self-loops to permit only direct recursion. The approach works reasonably well for simple Lisp functions like REVERSE and UNION, but it appears that it would be too restrictive to be useful for larger programs.

[FS87] and [HC88] discuss approaches for automatic average-case analysis of special classes of programs. The work in [FS87] is applicable to recursive descent procedures over recursively defined data structures that can be expressed in their language *PL-tree* e.g. tree matching, binary search. [HC88] describes approaches for simple probabilistic programs and a simple functional programming language.

An interesting question that arises with symbolic complexity analysis is how will the information be used, even if it can be derived for real programs? There is a danger of flooding the programmer or compiler with too much information, when providing symbolic expressions with several variables. The most common usage of complexity analysis in program optimization would be a less-than or greater-than comparison between two different symbolic expressions. Such a comparison is hard to resolve when the expressions contain more than one variable. Even if all symbolic expressions are somehow rewritten in terms of a single variable, some range information is necessary to answer questions like "Is $100 \times N < N^2$?" or "Is $N > (log_2 N)^3$?". The range information would somehow have to represent a "typical" program input, making the whole approach come closer to that of using execution profiles. It seems that automatic symbolic analysis is better suited to asymptotic analysis where constants can be ignored, than to compiler optimizations where real numbers are needed.

7 Conclusions

We have presented a general framework for the estimation of average execution times and variance in a program. Our approach is based on the control dependence relation, and can be used for unstructured programs. Execution profile information is used to obtain the frequency values necessary for cost estimation. The cost of primitive operations is assumed to be a function of the target architecture, so that the same frequency information can be used to estimate execution times on different architectures. The average execution times and variance values can be computed in a single, linear time, bottom-up traversal of the forward control dependence graph.

This framework has been implemented in the PTRAN system, where the execution time and variance values will be used to guide the automatic partitioning of the input program into parallel tasks. We believe that several new optimizations that use average execution times will naturally evolve, now that the information is available in the program database.

Acknowledgements

The author would like to thank members of the PTRAN group — Fran Allen, Michael Burke, Ron Cytron, Jeanne Ferrante and Dave Shields — for their comments and suggestions on the paper, and their assistance in the implementation of this work. The author especially thanks Ron Cytron for suggesting the use of control dependence for efficient profiling. Finally, the candid comments of the reviewers served as valuable feedback in improving the paper and its coverage of related work.

References

[ABC*87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, Vol. 5, No. 5, pp. 617-640.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bur87] Michael Burke. *An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data Flow Analysis.* Technical Report, IBM Research, August 1987. Research report RC12702, submitted to ACM Transactions on Programming Languages and Systems.

[CF87] Ron Cytron and Jeanne Ferrante. *An Improved Control Dependence Algorithm.* Technical Report, IBM, 1987. Tech. Report RC 13291.

[CFR*89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 25–35, January 1989.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of dag parallelism. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 1989. To appear.

[CHR78] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The SIMPLE code.* Technical Report, Lawrence Livermore Laboratory, 1978. UCID 17715.

[CK74] John Cocke and Ken Kennedy. *Profitability Computations on Program Flow Graphs.* Technical Report, IBM, 1974. Tech. Report RC 5123.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. *Proceedings* *of the ACM Symposium on Compiler Construction*, 37 – 47, June 1984.

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

[FS87] Philippe Flajolet and Jean-Marc Steyaert. A complexity calculus for recursive tree algorithms. *Math. Systems Theory*, 19:301–331, 1987.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126, June 1982.

[HC88] Timothy Hickey and Jacques Cohen. Automating program analysis. *JACM*, 35(1):185–220, 1988.

[Hsi88] Wilson C. Hsieh. *Extracting Parallelism from Sequential Programs*. Technical Report, Massachusetts Institute of Technology, May 1988. Master's thesis.

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

[KW85] Clyde Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10), October 1985.

[McM86] F. H. McMahon. *L.L.N.L. FORTRAN Kernels: MFLOPS*. Technical Report, Lawrence Livermore National Laboratory, March 1986.

[MH86] S. McFarling and J. Hennessy. Reducing the cost of branches. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7), July 1986.

[Sar87] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. PhD thesis, Stanford University, April 1987. Tech. Report CSL-TR-87-328.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[SH86a] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7):17–26, July 1986.

[SH86b] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. *ACM Conference on Lisp and Functional Programming*, 202–211, August 1986.

[SS79] J. T. Schwartz and M. Sharir. *A Design for Optimizations of the Bitvectoring Class*. Technical Report, Courant Institute of Mathematical Sciences, New York University, September 1979. Courant Computer Science Report No. 17.

[Wal86] D. W. Wall. Global register allocation at link time. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7), July 1986.

[Weg75] Ben Wegbreit. Mechanical program analysis. *CACM*, 18(9):528–539, 1975.

	Compiler optimization ON			Compiler optimization OFF		
Program	Original code	Smart profiling	Naive profiling	Original code	Smart profiling	Naive profiling
LOOPS	0.05	0.06	0.08	0.24	0.24	0.26
SIMPLE	3.8	4.2	4.4	17.0	17.8	18.6

Table 1: Sequential execution times with and without profiling

```
10      IF (M .GE. 0) THEN
            IF (N .LT. 0) GOTO 20
        ELSE
            IF (N .GE. 0) GOTO 20
        ENDIF
        CALL FOO(M,N)
        GOTO 10

20      CONTINUE
```

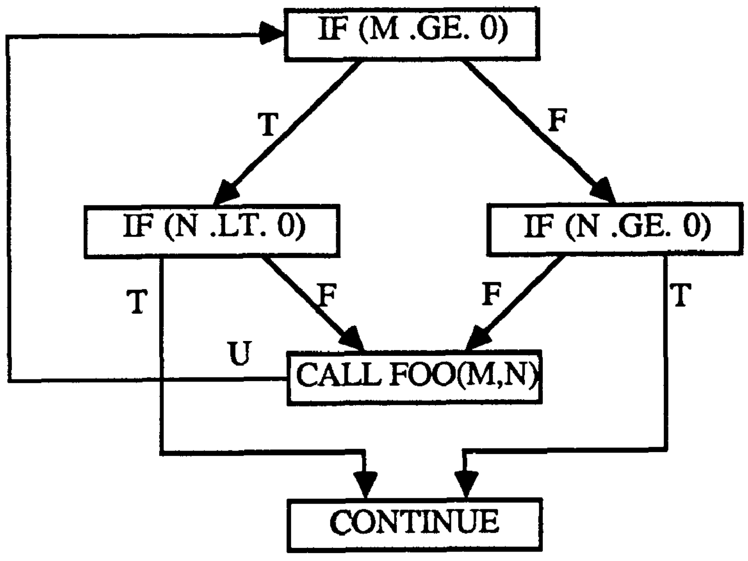


Figure 1: Original Control Flow Graph, *CFG*

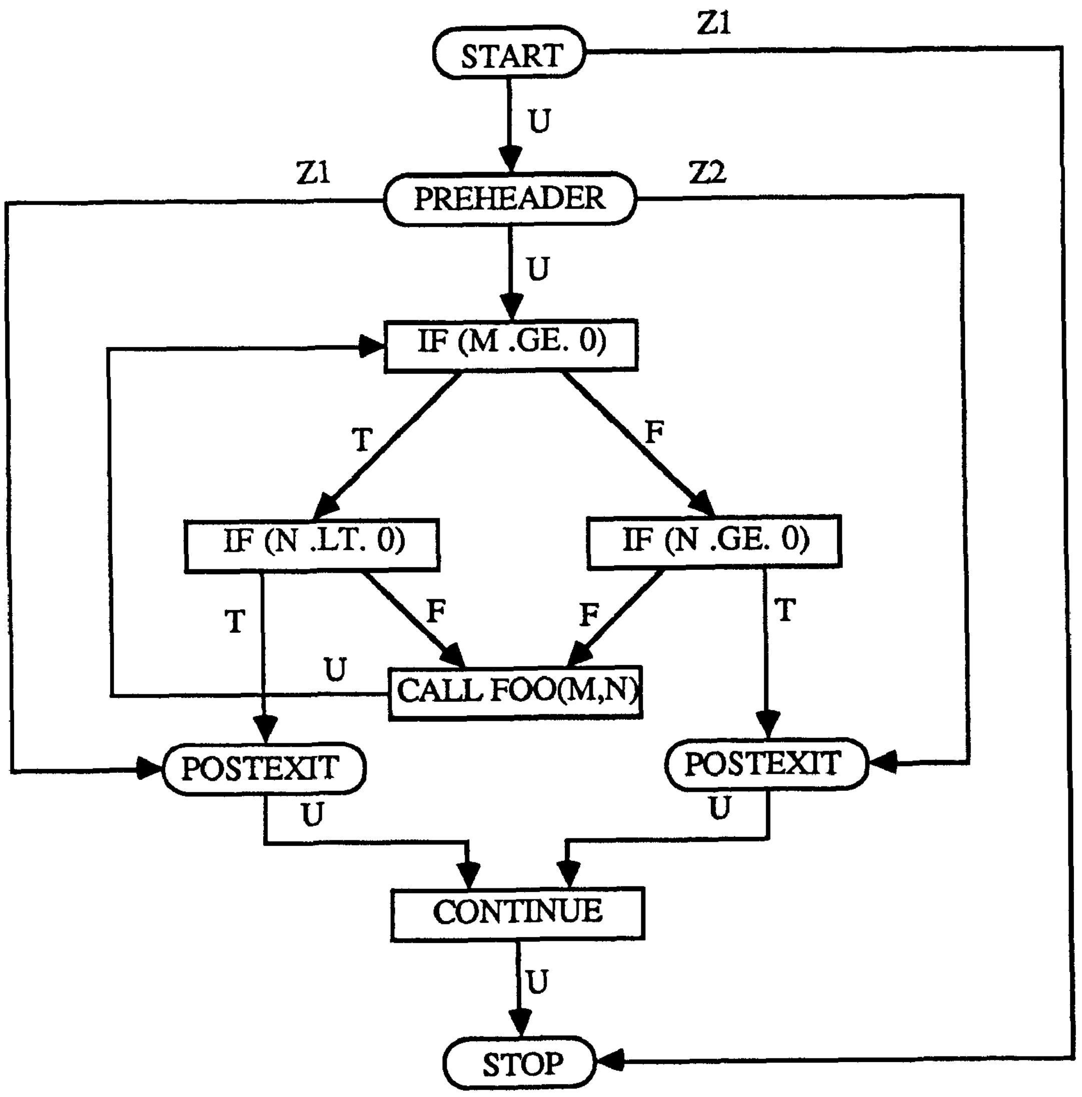
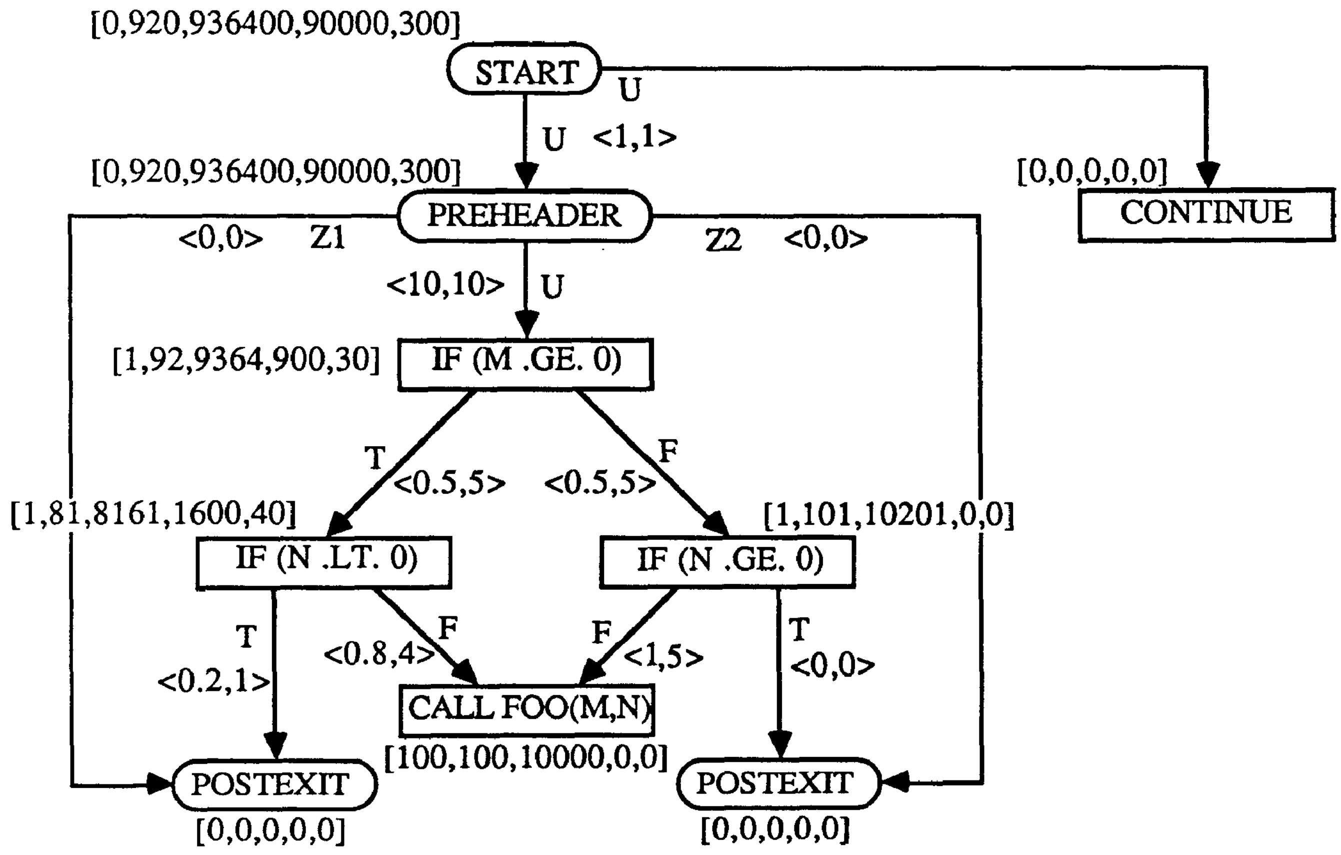


Figure 2: Extended Control Flow Graph, *ECFG*



<FREQ,TOTAL_FREQ> = edge frequency values

[A,B,C,D,E] = node execution times as follows:
A = local node execution time, COST(n)
B = expected total node execution time, E(T) = TIME(n)
C = expected value of time squared, E(T*T)
D = VAR(n) = E(T*T) - E(T)*E(T) = C - B*B
E = STD_DEV(n) = √VAR(n) = √D

Figure 3: Forward Control Dependence Graph, *FCDG*