Analysis of Software Remaining Execution time
Sarah Tasneem, Reda Ammar, Howard Sholl
U-155, Computer Science and Engineering Department
University of Connecticut
Storrs, CT 06269-3155.
sarah, reda, hasc@engr.uconn.edu

Abstract
We develop a method to calculate the distribution of remaining execution time of a software computation, \( R(t) \), after time \( t \) has passed towards its completion. Our approach finds an \( R(t) \) that depends on the path taken by the computation. The analysis can be done a priori (before run time).

With the recent growth in multimedia applications in computers, CPU schedulers face the burden of supporting batch, interactive and multimedia applications (real time) simultaneously. Traditional general-purpose operating systems lack the flexibility to support the mixture of diversified applications. Recognizing the need to provide better scheduling a number of policies have been proposed in this context. This paper focuses on calculating an accurate \( R(t) \), which is an important factor in effective scheduling. It is clearly shown that, when multiple tasks are concurrently scheduled with time quanta such that their scheduling is based on partial execution, a mean value (e.g., single-value) execution time characterization may end up with undesirable scheduling. Our method uses a distribution to evaluate a path dependent \( R(t) \) instead of a single expected value. We expect the prior work may show better performance when a more accurate method is used to estimate \( R(t) \).

Keywords: remaining execution time distribution, software performance, scheduling.

1. Introduction
This paper considers the estimation of execution time for use in the thread scheduling process. With the phenomenal development of generations of high-speed microprocessors, a growing and continuing number of multimedia applications are now available ranging from desktop video conferencing to sophisticated distributed simulation and virtual reality computing [6]. As a consequence a CPU must be able to schedule and run a combination of concurrent real-time and non-real-time applications [5,11]. When a user opens several applications on a computer, the application threads compete for the CPU time [3,4]. Scheduling CPU time for individual tasks at an appropriate rate and granularity can require analyzing not only the total computation time of the threads, but also their remaining execution time to complete. Often software performance [2,10] is characterized by a single value execution time (e.g., mean). However, when multiple tasks are concurrently scheduled with time quanta such that their scheduling is based on partial execution, a mean value execution time characterization is insufficient, and may result in undesirable scheduling.

For example, let \( E(T) = \sum_{i} p_i t_i \) be the overall expected execution time of a computation with \( r \) paths where \( p_i \) and \( t_i \) are the probability and execution time of path \( i \). Let \( \tau \) be the elapsed execution time. Now the remaining execution time to complete, \( \bar{R}(\tau) \), after time \( \tau \) has elapsed: \( \bar{R}(\tau) = E(T|\tau) \neq E(T)-\tau \), where, \( E(T|\tau) \) is the expected remaining time after elapsed time \( \tau \).

Consider a computation that takes either of two possible paths with equal probability and execution time \( T_i \) and \( T_s \), respectively. The expected execution time \( E(T) = 0.5T_i + 0.5T_s \). Assume \( T_i << T_s \). If \( T_i = 20 \) sec., \( E(T) = 10 \) sec. Here, the time taken by the longest path (20 sec.) is 100% more than the expected time (10sec). Moreover, at elapsed time \( \tau = T_s + \Delta \tau \), \( \bar{R}(\tau) = E(T|\tau) = 20 \) sec. Thus the remaining time is significantly greater than the mean total time. This example shows us the importance and justification of using a distribution for execution time instead of a single expected value.

Our approach includes different possible paths of a computation and provides an estimated execution time distribution. Each path has an associated probability and execution time cost.

At this point in time, however, there has not been an extensive study on calculating \( R(t) \) and its effect on scheduling. This paper focuses on calculating \( \bar{R}(t) \) accurately, which can be an important factor in effective CPU scheduling. Section 2 discusses the background and related work. Section 3 summarizes two methods to calculate \( \bar{R}(t) \). Section 4 presents the analysis of software execution time. In section 5 we develop one of the proposed methods and shown a thorough calculation of the \( \bar{R}(t) \) distribution. Section 6 presents methods to reduce the time and probability vectors.
2. Background

Traditional general-purpose operating systems lack the flexibility required to support and deal with a diverse workload (e.g., batch, interactive, and multimedia) of applications [7,11]. Recognizing the need to provide better scheduling, a number of resource management policies have been proposed [2,10,12-17]. A taxonomy of approaches is found in [16].

The scenario of this research is as follows: threads from several applications run concurrently and time quanta are allocated to the ready threads depending on the respective allocation policies. After execution for a time quantum an application thread progresses partially towards its completion.

For example, SMART: A Scheduler for Multimedia Applications [2] arranges tasks into a working schedule in increasing deadline order and they are serviced in that order. A schedule is said to be feasible, when tasks are serviced in the order defined by the working schedule and can be completed before their respective deadlines. The resource requirement of a periodic real-time task includes an estimate of the processing time. Neih et.al. [2] has calculated the estimated resource requirement of a task after time t, by subtracting E (the time the task has already spent running toward meeting its deadline) from Q (the processing time required by the task to meet its deadline). Their estimated resource requirement is similar to the remaining execution time, $R_t(t)$. Our method (below) uses a distribution to evaluate $R_t(t)$, and we expect that better performance estimation may show improved scheduling performance in this and other scheduling schemes.

3. Methods to calculate $R_t(t)$

We propose two possible methods for calculating $R_t(t)$. Method 1: Let us assume software codes are written with some kind of associated look up table. The format of the table may look like this:

<table>
<thead>
<tr>
<th>$R_t(t)$ distr.</th>
<th>Time elapsed</th>
<th>path taken</th>
</tr>
</thead>
</table>

Moreover, assume a software application has the capability to signal a flag indicating the exact path it took as it progressed during the time of execution. From the information of the signal flag and the look up table one can estimate $R_t(t)$ of the application. However, this is a demanding approach, and software developers may not have provided this information. Method 2: This approach uses an a-priori performance model (time distribution) for a software application. We show how one can use this model during scheduling, along with the current time of execution, $t$, to calculate the distribution of remaining time $R_t(t)$. We start with the Computation Structure Model (CSM) [1,9], which describes the detailed time-execution behavior of computations, and extend the analysis to calculate the behavior of the remaining execution time distribution. The following sections describe this method in detail.

4. Analysis of software execution time

4.1. Computation Structure Model

4.1.1 Definition

A CSM is a set of two directed graphs: the Data Flow Graph (DFG) and the Control Flow Graph (CFG)[1,9]. These two graphs, together, model the time and space requirements of a computation.

The DFG is used to model the storage requirements of a computation, and the CFG is the representation of the execution paths of a computation. The focus of the paper is on the execution time behavior, so we will not consider the DFG any further.

The CFG is a directed graph that can be used to model any computation (task). The CFG shows the way in which different subtasks fit together in a task. It is a common tool for modeling the control flow of a computation, and is used to organize the calculation of execution time (the time a program will take to complete)[1]. The CFG consists of nodes (elements) that represent actions in the computation that consume time, and edges that depict how the thread of execution reaches these nodes. Each node is given a time cost (a single value or a distribution) and each edge a control flow count. The control flow through an element is defined as the number of times that the element is activated, on the average, for each execution of the computation represented by the CFG.

4.1.2. Stochastic vs. deterministic operators

A deterministic operator represents a task whose execution time has a finite, deterministic value. A stochastic operator represents a task with several possibilities of execution time length $l_i$ (e.g., as a branch-merge structure [8]), where each possibility has a specified probability $p_i$ (a discrete probability distribution). Execution time distributions are, by nature, discrete, although they might sometimes be modeled otherwise.

4.2 Time equivalent CFG transformation steps

A method is described in [8] that provides a set of transformation steps which can be used to transform a given CFG, composed of elements and control flows, into a simpler, time equivalent CFG. The transformations steps are termed as time equivalent transformation steps, because, two CFG segments representing the “before” and “after” of a proposed transformation step have equivalent execution time distributions. Rules are defined to make sure that the execution time probability distribution of the two CFGs will be identical.

The following examples will show the procedure of applying the time equivalent transformation steps.
4.2.1 Merging two stochastic operations

Consider two independent stochastic processes $S_1$ and $S_2$ in sequence. Let us assume the following execution time probability distribution for $S_i$ and $S_{-i}$:

$$T_i = \{t_{i1}, t_{i2}, \ldots, t_{im_i}\}$$

$$P_i = \{p_{i1}, p_{i2}, \ldots, p_{im_i}\}$$

Where process $S_i$ takes any one of time $t_{ij}$ to execute with probability $p_{ij}$. We shall apply the step-by-step time equivalent transformation steps on to this sequence. The distribution of the sequence can then be described as [8]:

$$T_{12} = (t_{11} + t_{21}, t_{12} + t_{22}, \ldots, t_{1m_i} + t_{2m_j}, t_{1m_i} + t_{2m_j})$$

$$P_{12} = \{p_{11}p_{21}, p_{11}p_{22}, \ldots, p_{11}p_{2m_j}, p_{12}p_{21}, p_{12}p_{22}, \ldots, p_{12}p_{2m_j}, p_{m_i}p_{m_j}, \ldots, p_{m_i}p_{2m_j}\}$$

Thus, we see that a sequential merge of two stochastic operators yields a single operator with the joint distribution of the individual distributions.

4.2.2. Unfolding a loop structure

Fig. 1a shows a common loop structure. We unfold the loop structure as shown in fig. 1b. The steps to unfold a loop structure can be written as the following algorithm:

Algorithm for unfolding a loop

- Expand the loop, as shown in Fig. 1b, to represent each possible path through CFG segment in fig. 1a.
- As a next step merge (applying the steps from sequential merge as explained before) two sequential operators at a time in fig. 1b until a single operator in each of the branches is obtained.

The effect of the transformation is to divide the flow through each operator into all the possible fractional flows through it. The original flow is obtained after adding these fractional flows, since the sum of the probabilities of all the branches in fig. 1b will add up to unity.

4.3. Probability distribution of execution time

Applying these ideas in an orderly algorithmic manner, one can obtain the overall distribution of execution time of a computation with the following form:

$$T = \{t_1, t_2, t_3, \ldots, t_n\}$$

$$P = \{p_1, p_2, p_3, \ldots, p_n\}$$

The in-depth work for all the different kinds of transformations is beyond the scope of this paper. Readers are suggested to read [8] for a more detailed description.

5. Remaining time estimation algorithm

To find the probability distribution of remaining execution time, $Rt(t)$, we start with time equivalent CFG segment obtained after applying the transformation steps to the original CFG. We assume that the initial distribution is ordered, such that, $t_i > t_{i+1}$, and the $p_i$ terms are rearranged accordingly.

**Staring point:** $T_i = \{t_1, t_2, t_3, \ldots, t_n\}$

$$P_i = \{p_1, p_2, p_3, \ldots, p_n\}$$

Assume time $t_{e1}$ has elapsed. We have no knowledge regarding the location in CFG. The remaining execution time distribution after time $t_{e1}$ can be calculated as follows:

**Step1:**

$$T_1 = \{t_1 - t_{e1}, t_2 - t_{e1}, \ldots, t_n - t_{e1}\}$$

$$P_1 = \{p_1, p_2, \ldots, p_n\}$$

**explanation:**

Some of the entries in $T_1$, where $t_i \leq t_{e1}$, may be negative or zero, indicating that, those paths were not taken in the execution. Next, these distribution components need to be removed and the distribution be normalized. Let $p_e = \sum_{t \leq 0} p_t$. The RHS adds all the values of probability for which the corresponding time is either 0 or negative. $1 - p_e$ gives the summation of the probabilities without zero or negative time.

**Step2:**

Rewrite $T_1$ and $P_1$ as $T'_1$ and $P'_1$, respectively, to normalize the distribution.

**explanation:**

From $T_1$, remove those entries that have zero or negative time and rename $T_1$ as $T'_1$.

From $P_1$ remove the corresponding entries, normalize the entries in $P_1$, and rename $P_1$ as $P'_1$, where

$$T'_1 = \{t_1, t_2, \ldots, t_{n'}\}$$

$$P'_1 = \{p_{1'}, p_{2'}, \ldots, p_{n'}\}$$

The remaining time distribution is complete.

Assume time $t_{e2}$ has passed after time $t_{e1}$. This procedure can be applied iteratively.

**Step 3:**

The mean of remaining time, $Rt(t) = \sum_{t} p_t t_i$ at the $r_j$ cycle of iteration, where, $p_t \in P'_n$ and $t_i \in T'_n$. 

221
**6. Vector Reduction Methods**

For a large task (computation) the distribution vectors may be large and time-expensive to manipulate (of course, the initial distribution is determined at compilation time). Therefore, we are investigating approaches to reduce the vector size. Several rules are defined below to reduce the size of the distribution vectors:

1) Merge entries that have identical or a very close values of execution time. If the time vector $T$ has two entries and such that $|t_j - t_{j+1}| \leq \alpha_1$, delete $t_j$ and $p_j$ from the time and probability vector, respectively, and replace $p_{j+1} = p_j + p_{j+1}$ (where, $\alpha_1$ is a user specified threshold value). Repeat the process until no more merging can be done.

2) Drop entries from probability and time vectors $P_i, T_i$ where $p_i << \alpha_2$, a user defined threshold value.

Let $p_i = \sum$ all probabilities but $p_i$.

Normalize all the $p_j$'s for $j \neq i$ as follows: $p_j = p_j / p_i$.

Normalize the time vector to maintain the original distribution mean:

Let $\beta = E(T)/E(T)'$, where $E(T), E(T)'$ are the means before and after the above adjustment.

Set the new time vector to $T' = \beta T'$

3) Delete entries for $t_i$ where $(p_j / \sum p_i) < \alpha_3$, because $p_j$ will have negligible effect on the time-probability distribution graph. Since deleting $t_i$ itself will also change the average remaining execution time, we also normalize the resulting distribution to maintain time integrity as shown above. Any one or a combination of the above rules can be applied to reduce the vector, depending on the type of application under consideration.

**7. Conclusions/ Future Work**

In this paper, we describe a step-by-step calculation of the remaining execution time distribution of a software computation, after execution time $t$ has passed. The analysis can be done a priori (before run time). Execution time for a software computation varies depending on the path the flow graph takes during execution. Expected execution time, $E(T)$, is calculated taking into account of the distribution of all the possible times for execution, where some of the execution times are higher than the average and some of them are lower than the average. As a result, there are cases where the expected remaining execution time, $R(t)$, given time $t$ has elapsed can be even greater than the initial expected execution time $E(T)$.

We have proposed two methods to calculate $R(t)$.
The first is very demanding and may not be practical. We have shown the second method in detail which is based on a remaining time distribution. Furthermore, we have presented three different ideas for reducing the size of the time and probability vectors. As a part of our future work we will evaluate the effect of these results in scheduling algorithms. Of course, use of a more accurate remaining time assessment may be helpful in areas other than thread/process scheduling, such as control systems, task migration, routing, quality of service.

**References**


---

Fig. 1a. A common loop structure.

Fig. 1b. After unfolding the loop in fig. 1a.