# Performance Estimation for Real-Time Distributed Embedded Systems

Ti-Yen Yen, *Member*, *IEEE*, and Wayne Wolf, *Fellow*, *IEEE*

**Abstract**—Many embedded computing systems are distributed systems: communicating processes executing on several CPUs/ASICs. This paper describes a performance analysis algorithm for a set of tasks executing on a heterogeneous distributed system. Tight bounds are essential to the synthesis and verification of application-specific distributed systems, such as embedded computing systems. Our bounding algorithms are valid for a general problem model: The system can contain several tasks with hard real-time deadlines and different periods; each task is partitioned into a set of processes related by data dependencies. The periods of tasks and the computation times of processes are not necessarily constant and can be specified by a lower bound and an upper bound. Such a model requires a more sophisticated algorithm, but leads to more accurate results than previous work. Our algorithm both provides tighter bounds and is faster than previous methods.

**Index Terms**—Embedded systems, distributed systems, real-time systems, performance analysis, periodic tasks

## 1 Introduction

This paper describes new methods for the performance analysis of an embedded application executing on multiple processors. The performance analysis algorithm is useful in the design of any distributed system in which the task structure is known in advance. Accurate performance analysis is especially important for the **hardware/software co-synthesis** of distributed embedded systems [1] in which hardware and software are designed together to meet performance and cost goals [2]. Performance analysis is a critical step in the inner loop of co-synthesis algorithms, where it is used to judge the quality of a scheduling and allocation of processes in the embedded system. Since performance analysis is performed many times during the course of co-synthesis, it must be efficient. Performance analysis must also be accurate to ensure high-quality results from co-synthesis.

An embedded computing system is an *application-specific* computing system. The hardware platform consists of one or more *processing elements* (*PEs*)—programmable CPUs and application-specific integrated circuits (ASICs). The software executing on the programmable CPUs is, in general, split into multiple concurrent processes. An embedded system is designed to execute a relatively stable body of code, unlike general-purpose systems which may execute code which was not available to the system designers for characterization. Many embedded systems have *hard real-time performance constraints*. Shin and Ramanathan [3] provide a good survey of real-time computing. A real-time application is usually composed of a set of periodic **tasks**. Each task has a **deadline** by which it must complete the execution. The application-specific nature of the design problem means that, unlike in time-sharing operating systems, the job characteristics are known when the system is designed. This favors static design and analysis over dynamic analysis for predictability and reducing overhead.

Given the computation time of the uninterrupted execution of processes, the allocation of processes, and the priority assignment for process scheduling, the goal of our analysis algorithm is to statically estimate the worst-case delay of a task to see whether the deadline is satisfied. A sophisticated performance analysis algorithm is important for hardware/software co-design because many embedded systems are distributed heterogeneous computing engines: Modern automobiles include up to 60 microcontrollers of various sizes; many cellular phones contain multiple CPUs which execute both signal processing and control-oriented code; many 35mm cameras include several microprocessors. Rosebrugh and Kwang [4] described a pen-based system built from four processors of different types.

While the scheduling and allocation of a nonperiodic task consisting of predefined processes is NP-complete [5] (multiprocessor scheduling), the performance analysis problem is polynomially solvable given an allocation and schedule of processes. On the other hand, for periodic tasks running on a distributed system, we can show that even the analysis problem is NP-hard based on the results proved by Leung and Whitehead [6]. This paper presents a new performance analysis algorithm for multirate tasks on multiprocessors which is both faster and more accurate than previous methods. Given the complexity of the problem, we adopt *conservative* delay estimation algorithms which give *strict upper bounds* on delay, but which also give *tight* bounds on delay. In some cases, we can prove that our schedule bounds are tighter than those provided by other algorithms, since our algorithm uses a more general model of the set of tasks. Many algorithms for hard real-time distributed systems enumerate all the occurrence of processes in a cycle with a length equal to the least common multiple

- *T.-Y. Yen is with Quickturn Design Systems, 55 W. Trimble Rd., San Jose, CA 95131-1013. E-mail: tyen@quickturn.com.*
- *W. Wolf is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. E-mail: wolf@princeton.edu.*

*Manuscript received 8 June 1995.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 101201.*

of all the task periods. This approach is not practical for cosynthesis, as we will mention in Section 2. Our algorithm does not rely on unrolling the schedule to the least common multiple of the periods. Our algorithm is also considerably faster on practical problems than previous methods, in part because it extends previous constraint-generation methods and uses multiple constraint-solution algorithms to solve the system of constraints which describes the multirate scheduling problem.

Our algorithm has several advantages: It handles both upper and lower bounds on process execution time [7], it considers data dependencies between processes, and it handles preemption and task pipelining. In general, the initiation times for processes may vary from iteration to iteration; these variations ripple throughout the schedule and complicate the analysis. The traditional way to handle this problem is to unroll the schedule to the least-common multiple of the periods of the tasks, but that method both takes a large amount of CPU time and creates obstacles for scheduling heuristics. Our algorithm can compute a tight bound on the schedule without unrolling. Previous publications [8], [9] provide earlier descriptions of this algorithm.

The next section surveys previous work on performance analysis for hard real-time systems. Section 3 gives our formulation of the problem. Section 4 describes our algorithm in detail. Section 5 describes the results of experiments with that algorithm.

## 2 Previous Work

The performance analysis of a single program has been studied by Park [10], Ye et al. [11], and Li and Malik [7], among others. These papers describe the calculation of bounds on the running time of uninterrupted execution of a sequence of code. In most embedded systems, several processes can run concurrently on the same CPU and interfere with one another according to a certain scheduling criterion. In this case, the *response time*—the time between the request and the finish of a process—consists of the whole execution time of the process itself and portions of the execution time of some other processes. The previous work at the program level provides a foundation for the performance analysis at the process level; in this paper, we assume the *computation time*—the uninterrupted execution time of each process—has been given in advance. We need to bear in mind the computation time for a process is usually not a constant and should be modeled by a bounded interval due to factors such as conditional behavior in the control flow, cache access, and the inaccuracy of the analysis techniques [7].

Rate-monotonic scheduling [12] (RMS) is a foundational scheduling method for real-time systems. It gives a fixed-priority assignment for periodic processes on a single processor. RMS assumes that the processes are *independent* with no data dependencies between them. Each process $P_i$ is characterized by a computation time $c_i$ and a period $p_i$. The deadline of a process is always at the end of its period. The processes on the PE may have different periods. Liu and Layland showed that the CPU was optimally utilized when processes are given priorities according to their rates. The deadlines of $n$ processes can be met if the processor utilization

$$U \equiv \sum_{j=1}^{i} c_j / p_j \le n\left(2^{1/n} - 1\right). \tag{1}$$

RMS has been generalized in several ways as surveyed by Sha et al. [13].

There is a large amount of literature on real-time distributed systems. Some of this work uses models which are not suitable for embedded systems. For instance, Liu and Shyamasundar [14] and Amon et al. [15] assumed that there can be only one process on each processor; Chu et al. [16] applied probabilistic approach which is not appropriate to handle hard deadlines. Many scheduling or allocation algorithms for distributed systems focus only on a single nonperiodic task and cannot handle the RMS model. Chu and Lan [17] reviewed some of these works and categorize them as graph-theoretic, integer 0-1 programming, and heuristic approaches. However, in most real-time embedded systems, different tasks running in different rates mix together. For instance, Chiodo et al. [18] gave a seatbelt example to demonstrate multiple-rate tasks in embedded systems.

A useful result by Lehoczky et al. [19] bounds the response times for a set of independent processes; we will take advantage of this result in Section 4 to compute bound the response times of processes with data dependencies. Suppose $P_1$, $P_2$, ... are a set of priority-ordered processes allocated on the same CPU, with $P_1$ being the process with the highest priority. There is no data dependency between one another so the initial phase of the processes can be arbitrary. For a process $P_i$, its minimum period is $p_i$, and its longest computation time on the CPU is $c_i$. Let the worst-case response time from a request of $P_i$ to its finish be $w_i$. Lehoczky et al. showed that $w_i$ is the smallest nonnegative root of the equation

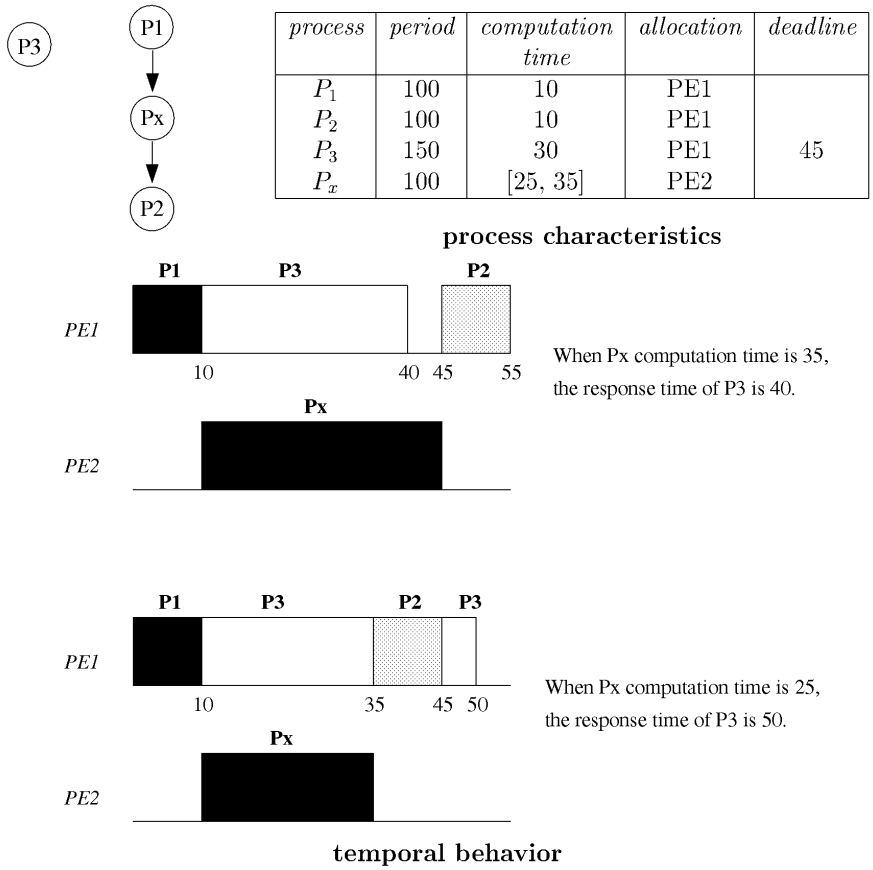
$$x = g(x) = c_i + \sum_{j=1}^{i-1} c_j \cdot \left\lceil x / p_j \right\rceil. \tag{2}$$

The function $g(x)$ represents the computation time required for higher priority processes and for $P_i$ itself: If the response time is $x$, there are at most $\lceil x/p_j \rceil$ requests from $P_j$. The total computation time for these requests is $c_j \cdot \lceil x/p_j \rceil$, so $g(x)$ includes these terms for all $j$ as well as the computation time $c_i$ for $P_i$ itself. The iteration technique used to solve this equation has been mentioned by Sha et al. [13]. It can be stated as a *fixed-point iteration* technique:

1) $x = \left\lceil c_i \Big/ \left(1 - \sum_{j=1}^{i-1} c_j/p_j\right) \right\rceil$;
2) **while** $(x < g(x))$ $x = g(x)$;

It is assumed that $\left(1 - \sum_{j=1}^{i-1} c_j/p_j\right) > 0$; otherwise, the schedule must be infeasible and such an allocation should be given up by the synthesis algorithm because the processor utilization $U \equiv \sum_{j=1}^{i} c_j/p_j > 1$. We can prove that the value of $x$ must converge to $w_i$ in finite steps.

EXAMPLE 1. Suppose that $p_1 = 5$, $c_1 = 1$, $p_2 = 37$, $c_2 = 3$, $p_3 = 51$, $c_3 = 16$, $p_4 = 134$, $c_4 = 42$. Fixed-point iteration tells us that we only need four steps to know that $w_4 = 128$; the $x$-values during iterations are 104, 120, 126 and 128.




| *process* | *period* | *computation time* | *allocation* | *deadline* |
|---|---|---|---|---|
| $P_1$ | 100 | 10 | PE1 | |
| $P_2$ | 100 | 10 | PE1 | |
| $P_3$ | 150 | 30 | PE1 | 45 |
| $P_x$ | 100 | [25, 35] | PE2 | |

Fig. 1. Using worst-case execution times in unrolled schedules may lead to erroneous results.

The restricting case occurs when the deadline is equal to the period. By rate monotonic analysis in (1) the processor utilization for this case is

$$U = \frac{1}{5} + \frac{3}{37} + \frac{16}{51} + \frac{42}{134} = 0.908 > 4\left(2^{1/4-1}\right) = 0.76$$

and the schedule may be misjudged to be infeasible. But fixed-point iteration determines that the deadlines are satisfied.

Pure RMS is not, however, often used to schedule distributed systems with data dependencies. Surveys of scheduling algorithms for periodic tasks in real-time distributed systems can be found in several valuable survey papers [3], [20]. Many algorithms [21], [22], [23] solve the problem by *unrolling* the schedule—forming a single large task whose length is the least common multiple (LCM) of all the periods. The LCM period allows the scheduler to evaluate interactions between different-rate processes. We decided that it was important to design an algorithm that did not require the periods to be unrolled for several reasons. First, since the analysis algorithm is run in the inner loop of a synthesis algorithm, efficiency is important and we believe that unrolling is inherently less efficient. Second, schedule unrolling cannot easily handle cases in which the period and computation time are bounded but not constant. If we use the upper bound of the computation time in the simulation over the length of the LCM, a deadline thought to be satisfied may actually be violated, as shown in the example of Fig. 1. In this example, the three processes $P_1$, $P_2$, and $P_3$ share the same CPU and $P_3$ has lowest priority. The deadline of $P_3$ is satisfied when $P_x$ runs for 35 time units, but not if $P_x$ runs for 25 time units. This phenomenon was also mentioned by Gerber et al. [24]. Using a static table [20] to fix all the process request time in the length of LCM can handle nonconstant computation times, but cannot handle nonconstant periods (as in variable-speed engines and motors) and can lead to large tables which are expensive to implement in memory-limited embedded systems.

Leinbaugh and Yamani [25] derived analytic bounds for response times of a set of tasks. However, their approach uses some pessimistic assumption so the bounds are not tight enough. D'Ambrosio and Hu [26] used simulation to judge the feasibility of a schedule during co-synthesis. Extensive simulation is often time consuming and not guaranteed to prove feasibility.

Prakash and Parker [27] formulated distributed system co-synthesis as an integer linear program (ILP). Their ILP formulation cannot handle periodic processes and preemptive scheduling. A great deal of recent work has studied hardware-software partitioning, which targets a one-CPU-one-ASIC topology. The Vulcan system of Gupta and De Micheli [28] and the COSYMA system of Ernst et al. [29] move operations between software and hardware. They schedule and allocate operations inside a process to satisfy timing constraints between operations but handle only single-rate problems and single-threaded implementations.

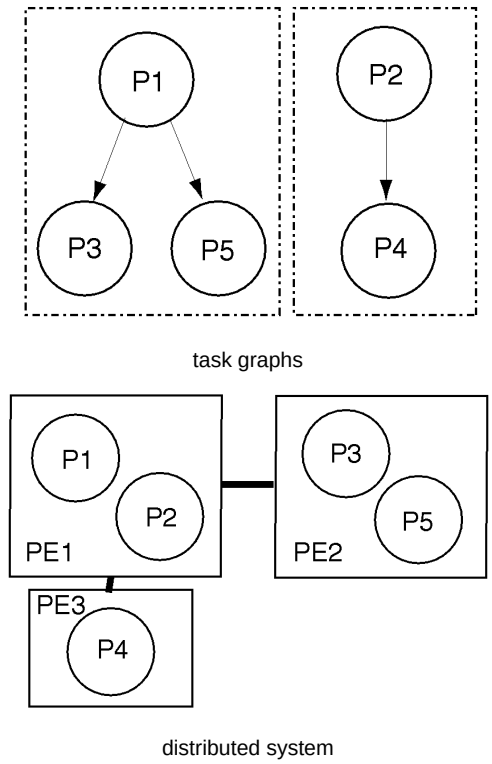



Fig. 2. A task graph and its implementation.

Our work may be applied to one-CPU-one-ASIC model but also holds for more general system topologies.

## 3 Problem Formulation

Our problem formulation is similar to those used in distributed system scheduling and allocation problems. A **process** $P_i$ is a single thread of execution, characterized by bounds on its **computation time**—the bounds may be written as $\left[c_i^{lower}, c_i^{upper}\right]$, where $c_i^{lower}$ is the lower bound and $c_i^{upper}$ is the upper bound. These bounds are a function of the processor type to which $P_i$ is allocated. A **task** is a partially ordered set of processes. The **task graph** is a directed acyclic graph which represents the structure of a task (or a set of tasks). A directed edge from $P_i$ to $P_j$ represents that the output of $P_i$ is the input of $P_j$. A process is not initiated until all its inputs have arrived; it issues its outputs when it terminates. (Delay processes may be added to the model to accommodate data which arrives or leaves the main computation processes at different times.) A problem specification may contain several concurrently running tasks. (An embedded system may perform several tasks which are nearly decoupled, such as running several relatively independent peripherals; in such cases, it may be useful to describe the system as unconnected subsets of processes.)

Processes are illustrated on the top of Fig. 2 as nodes labeled with their names. The processes and data dependency edges form tasks. Each task has two implicit nodes—the start node and the end node. There is a dependency from the start node to each process, and a dependency from each process to the end node. The start node and the end node are not drawn in the examples and will be used only to explain the algorithm. Each task is given two characteristics: a **period** (also known as a **rate constraint**) which is the time between two consecutive initiations; and a **deadline**, the maximum time allowed from initiation to termination of the task. If a task $\tau$ is not issued in a constant rate, the period is modeled by an interval $\left[p_\tau^{lower}, p_\tau^{upper}\right]$. A task's deadline is satisfied if it is greater than or equal to the worst-case delay through the task. Release times and multiple deadlines can be modeled by inserting dummy processes—processes with delay but not allocated on any physical PE—in the task graph, as described in Fig. 3. We therefore assume each task has a single deadline.

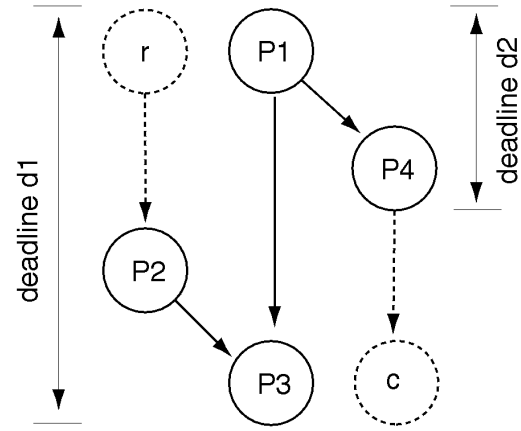



Fig. 3. How to transform release times and multiple deadlines into single-deadline form.

Synthesis produces an embedded system architecture. As illustrated in the bottom of Fig. 2, the *hardware engine* architecture is a labeled graph whose nodes represent **processing elements (PEs)** and whose edges represent communication links. A PE may represent a CPU executing software or an ASIC which can execute a particular function. The edges and nodes of the graph are labeled with the type of link or PE, as appropriate. The **allocation** is given by a mapping of processes onto PEs. The **schedule** of processes is an assignment of priorities to processes. Some processes may be implementable by either a CPU or an ASIC; we assume that the processes have been partitioned so that they do not cross CPU-ASIC or CPU-CPU boundaries.

We assume static allocation of processes—a process does not move to a different PE during the course of execution. Since processes may share a CPU, we must model the interaction of co-allocated processes. Each process is given an integral, fixed priority; the CPU always executes the highest-priority ready process, which continues execution until it completes or is preempted by a higher-priority process. We neglect operating system overhead in this analysis; with fixed priority scheduling, OS overhead is frequently negligible, but its effects can be straightforwardly incorporated into the computation time of a process.

We can show that the analysis problem under this model is NP-hard, even though the schedule and allocation of processes have been given.

THEOREM. *Given the task graphs and an embedded system architecture* (*the hardware engine, schedule, and allocation of processes*), *the problem to decide whether the deadline of each task is satisfied is NP-hard.*

PROOF. Leung and Whitehead [6] proved that deciding whether a priority assignment schedule is valid for an asynchronous system on a single processor is NP-hard. In their formulation, an asynchronous system contains a set of processes. Each process $P_i$ has a period $p_i$, an initial request time $s_i$, a deadline $d_i$ measured relative to the initial request time, and a computation time $c_i$.

We prove that our analysis problem is NP-hard by showing that Leung and Whitehead's problem is polynomial-time reducible [30] to our analysis problem. For each process $P_i$ in the asynchronous system, create a task graph $\tau_i$ as follows: Task $\tau_i$ contains $P_i$, another process $Q_i$, and an directed edge from $Q_i$ to $P_i$. The computation time of $Q_i$ is $s_i$ and the computation time of $P_i$ is $c_i$; the period of $\tau_i$ is $p_i$ and the deadline of $\tau_i$ is $d_i + s_i$. Schedule and allocate the processes such that each $Q_i$ on a different PE such that only $Q_i$ is executed on that PE, all the processes $P_i$s in the original asynchronous system on the same PE, the priority assignment for the original asynchronous system are used to schedule the processes $P_i$s. The deadline of task $\tau_i$ is satisfied if and only if the deadlines of process $P_i$ in the original asynchronous system is satisfied. Because the analysis problem of an asynchronous system is NP-hard, the performance analysis of task graphs on an embedded system architecture is also NP-hard. □

The theorem is different from the NP-completeness of the multiprocessor scheduling problem [5] because:

- scheduling and allocation is known a priori;
- processes are periodic;
- preemptive scheduling is allowed.

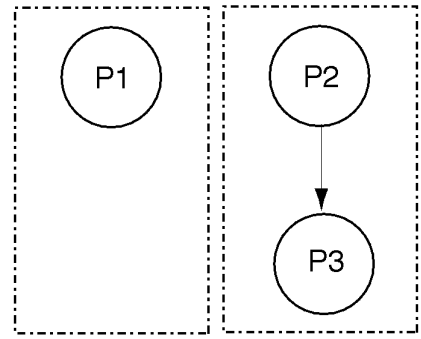
## 4 Our Algorithm

This section describes our core algorithm, which uses a *fixed-point iteration* based on *phase adjustment* and *separation analysis* to compute tight bounds on the worst-case delay through a task graph executing on multiple PEs, including complications caused by data dependencies in the task graph.

### 4.1 Algorithm Overview

Our algorithm iteratively applies two techniques—phase adjustment and separation analysis—to tighten the bounds on process delay. A simple example illustrates the need for both types of analysis.

EXAMPLE 2. Consider the task graph of Fig. 4 in which the three processes share the same CPU and $P_1$ have highest priority. If we ignore data dependencies between $P_2$ and $P_3$, as is done in (2), their worst-case response times are 35 for $P_2$ and 45 for $P_3$. But the worst-case total delay along the path from $P_2$ to $P_3$ is actually 45 instead of 80 (the sum of 35 and 45) because of two conditions. First, $P_1$ can only preempt either $P_2$ or $P_3$, but not both in a single execution of the task. Second, $P_2$ cannot preempt $P_3$.




| | period | computation time |
|---|---|---|
| P1 | 80 | 15 |
| P2 | 50 | 20 |
| P3 | 50 | 10 |

Fig. 4. An example of worst-case task delay estimation.

This example shows that the delays among processes in disjoint tasks are not independent, unlike many critical-path problems in CAD. Since processes in separate tasks may be co-allocated to the same processor, the execution times of processes in separate tasks can affect each other. As a result, we cannot use simple longest-path algorithms [30] to solve for the worst-case delays of the processes. We must take into account combinations of process activation's to generate tight bounds on worst-case delays.

Fig. 5 outlines our complete algorithm. $G_i$ is a subgraph of the task graph which contains process $P_i$ and its successors. Each connected subgraph of the complete task graph is referred to as a $G_i$ which contains a process $P_i$ and all of its successors in the task graph. The result of the algorithm is a set of upper and lower bounds on process separation times $maxsep[\cdot, \cdot]$ which provide the bounds on process execution times. The algorithm repeatedly applies two steps to tighten the bounds, starting with bounds of $\infty$:

- `EarliestTimes()` and `LatestTimes()` use a modified longest-path algorithm to take into account data dependencies, handling the first phenomenon observed in Example 2.
- `MaxSeparations()` uses a modified max-constraint algorithm to take into account preemption, handling the second phenomenon of Example 2.

The algorithm terminates when the bounds do not change over the course of an iteration. It is easy to show that the algorithm terminates, since the bounds always tighten at each iteration and termination occurs when the tightening of every bound is zero. We may also set a limit on the number of iterations if faster delay estimation is desirable.

The algorithm computes *phase adjustments* which model the constraints imposed by data dependencies in the task graph. It computes two types of phases:

- The *request phase* $\phi_{ij}^r$ between processes $P_i$ and $P_j$ captures the smallest interval between the request time of the current $P_i$ and the first request time of $P_j$ following it.

```
1    maxsep[·,·] = ∞; /* initialize separations */
2    step = 0; /* count tightening iterations */
3    do {
4      for (each P_i) { /* longest path */
5        EarliestTimes(G_i);
6        LatestTimes(G_i);
7      }
8      for (each P_i) /* max constraints */
9        MaxSeparations(P_i);
10     step++;
11   } while (maxsep[·,·] is changed and step < limit);
```

Fig. 5. Our delay estimation algorithm.

```
1  LatestTimes(a task graph G)
2  /* Compute latest[P_i.request] and latest[P_i.finish] for all P_i in a task graph G. */
3  {
4    for (each process P_i) { /* initialize */
5      latest[P_i.request] = 0;
6      for (each process P_j) φ^r_ij = 0;
7    }
8    for (each process P_i in topologically sorted order) {
9      w_i = worst-case response time of P_i with phase adjustment of φ^r_ij
10       for each process P_j with higher priority than P_i;
11     latest[P_i.finish] = latest[P_i.request] + w_i;
12     Calculate the phases φ^f_ij relative to latest[P_i.finish] for each j;
13     for (each immediate successor P_k of P_i) {
14       δ = latest[P_k.request] − latest[P_i.finish];
15       if (latest[P_k.request] < latest[P_i.finish])
16         latest[P_k.request] = latest[P_i.finish];
17       Update phases φ^r_kj for each process P_j according to φ^f_ij and δ;
18     }
19   }
20 }
```

Fig. 6. The LatestTimes algorithm.

- Similarly, the *finishing phase* $\phi_{ij}^f$ between processes $P_i$ and $P_j$ captures the smallest interval between the finish time of the current $P_i$ and the first request time of $P_j$ following it.

These phases capture the minimum distance between a preempted process and a preempting process. As we saw in Example 2, the phases of a pair of processes may be influenced by both data dependencies and preemptions caused by co-allocation of processes to a PE. In a pure rate-monotonic model, phase relationships between processes do not exist since there are no constraints on the relative timing of pairs of processes. However, given the existence of data dependencies which limit the timing relationships between processes, determining the relative phase of a pair of processes allows the algorithm to tighten the schedule bounds.

The next two sections describe each step of the algorithm in more detail. We will then summarize the relationships between these steps. Finally, we will show how our algorithm can be used to analyze pipelined task sets.

### 4.2 Latest/Earliest Times Calculation

**`LatestTimes`**, one of the two algorithms used to analyze data dependencies, is listed in Fig. 6. This procedure computes the *latest request time* and *latest finish time* of each process $P_i$ in a task. (Given the **`LatestTimes`** procedure, it is easy to transform that program into the **`EarliestTimes`** procedure to compute earliest request and finish times.) If the times are relative to the start of the task, the latest finish time of the end node of the task is the worst-case delay of the whole task.

**`LatestTimes`** is a modified longest-path algorithm—it traces through the processes in topological order, as is typically done in a longest-path algorithm, but it uses the current bounds, which include preemption information, to compute new bounds on execution time.

The first step in the algorithm is to initialize the request phase relationships and latest request times for each process. The algorithm considers the processes in topological order as defined by the task graph. Line 9 in Fig. 6 calculates the worst-case response time of a process $P_i$ using the fixed-point iteration for $w_i$ given by (2). However, the terms $\lceil x/p_j \rceil$ in $g(x)$ shown in (2) are modified into

$$\left\lceil \left(x - \phi_{ij}^r\right) / p_j \right\rceil,$$

where $\phi_{ij}^r$ is the phase of $P_j$ relative to the request time of $P_i$. In other words, $x$ is adjusted by the request phase $\phi_{ij}^r$.

After $w_i$ is computed, line 12 calculates the phases relative to *latest*[$P_i$.*finish*], the $\phi_{ij}^f$s. If $P_j$ preempts $P_i$,

$$\phi_{ij}^f = \left(\phi_{ij}^r - w_i\right) \bmod p_j.$$

We subtract $w_i$ from $\phi_{ij}^r$ because *latest*[$P_i$.*finish*] = *latest*[$P_i$.*request*] + $w_i$. Otherwise,

$$\phi_{ij}^f = \max\left(\phi_{ij}^r - w_i, 0\right).$$

Updating the request phases (the $\phi_{kj}^r$s) is more complex—the algorithm must look ahead one iteration to determine the request time for a process on its next iteration. Line 17 uses the finishing phases and $\delta$ calculated at line 14 to update the request phases; it does so by examining each immediate successor $P_k$ of $P_i$. If $P_i$ is the first visited immediate predecessor of $P_k$, $\phi_{kj}^r = \phi_{ij}^f$ for each $j$ because *latest*[$P_k$.*request*] = *latest*[$P_i$.*finish*]. If $\delta > 0$, there is slack between *latest*[$P_i$.*finish*] and *latest*[$P_k$.*request*]; that slack is used to increase the finishing phase $\phi_{ij}^f$. If $\delta < 0$, increase $\phi_{kj}^r$ similarly. Finally, $\phi_{kj}^r = \min\left(\phi_{kj}^r, \phi_{ij}^f\right)$. We choose the smaller phase, which may give longer delay to $P_k$ for the worst case. The final $\phi_{kj}^r$ values are used to adjust the phases in fixed-point iteration to get more accurate value for $w_k$ when $P_k$ is visited.

We can modify **`LatestTimes`** as follows to obtain the earliest request time *earliest*[$P_i$.*request*] and the earliest finish time *earliest*[$P_i$.*finish*]:

- Replace *latest*[·] with *earliest*[·]. Replace the ceiling operators $\lceil \cdot \rceil$ with the floor operators $\lfloor \cdot \rfloor$ in (2) for fixed-point iteration. Let $c_i = c_i^{lower}$ and $p_i = p_i^{upper}$ for the best-case delay estimation.
- At line 12, set $\phi_{ij}^f$ to 0 if $P_j$ are not allocated to the same CPU as $P_i$. Otherwise, calculate $\phi_{ij}^f$ in a way similar to that in Section 4 but keep it in the range $(-p_j, 0]$. At line 17, make $\phi_{kj}^r$ equal to $\phi_{ij}^f$ only when

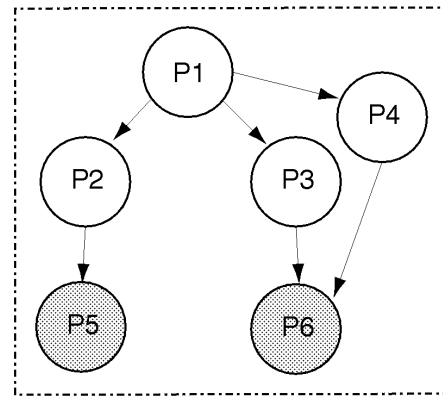
  *earliest*[$P_k$.*request*] = *earliest*[$P_i$.*finish*].

  Otherwise, leave $\phi_{kj}^r$ unchanged.

Suppose the modified algorithm is **`EarliestTimes`**.

Processes $P_i$s or $P_k$s visited by **`LatestTimes`** belong to the same task graph. However, the worst-case delay of the task graph may be affected by processes $P_j$s from the other tasks, if those processes share a CPU. For instance, the delay of the task graph composed of $P_2$ and $P_3$ can be lengthened by $P_1$ in another task.

The following example explains how **`LatestTimes`** algorithm solves the problem in Fig. 4 and Example 2.




| | case 1 | case 2 | case 3 | case 4 |
|---|---|---|---|---|
| P1 | [12,15] | → | → | → |
| P2 | [5,6] | [20,25] | → | → |
| P3 | [20,25] | [5,6] | [5,12] | [5,6] |
| P4 | [8,10] | → | → | [30,35] |
| P5 | [10,10] | → | → | → |
| P6 | [10,10] | → | → | → |

Fig. 7. Several different schedules for a task showing different combinations of execution time overlaps.

EXAMPLE 3. The situation is illustrated in Fig. 4. Initially, $\phi_{21}^r = 0$. When the algorithm **`LatestTimes`** visits $P_2$, by solving the equation

$$x = g(x) = 20 + 15 \cdot \lceil x / 80 \rceil,$$

it determines that *latest*[$P_2$.*finish*] = *latest*[$P_3$.*request*] = $w_2 = 35$ and

$$\phi_{31}^r = \phi_{21}^f = \left(\phi_{21}^r - w_2\right) \bmod p_1 = -35 \bmod 80 = 45.$$

If we know $P_2$ will not preempt $P_3$, then the phase adjustment is described by

$$x = g(x) = 10 + 15 \cdot \lceil (x - 45) / 80 \rceil.$$

In this case, we get $w_3 = 10$ and *latest*[$P_3$.*finish*] = 35 + 10 = 45, which is the worst-cast task delay we expect.

### 4.3 Separation Analysis

The **`MaxSeparations`** procedure determines what combinations of process activation's cannot occur, since one process cannot preempt another if they cannot be activated simultaneously. The role of max constraints in describing limitations on preemptions was illustrated in Example 2. While it is easy to know that a process will not preempt its predecessors or successors, such as with $P_2$ and $P_3$ in Fig. 4, it is not obvious how to decide whether two processes in disjoint paths of a task graph can overlap. This problem is illustrated in the following example.

EXAMPLE 4. Fig. 7 gives four different combinations of process computation times, with the lower and upper bounds listed as [*l*, *u*]. $P_5$ and $P_6$ are on the same CPU, where $P_5$ has higher priority. Even if the processes are not allowed to be allocated to different PEs, we can see substantial changes in their computation times as the schedule changes. In Cases 1 and 2, $P_5$ will not preempt $P_6$ because they are *separated*—that is, they

```
1 MaxSeparations(a process P_i)
2 /* Compute maxsep[P_i.request, P_j.finish] for all P_j in the same task graph */
3 {
4   for (each process P_j in topologically sorted order) {
5     Enqueue(Q, P_j); /* initialize queue */
6     tmp = ∞; /* initialize time */
7     while (Q is not empty) {
8       P_k = Dequeue(Q); /* get node to evaluate */
9       for (each immediate predecessor P_l of P_k) {
10        if (P_l is a predecessor of P_i)
11          tmp = max(tmp, upper[P_k.request, P_j.finish]
12            −lower[P_l.finish, P_i.request]);
13        if (P_j is a predecessor of P_i)
14          tmp = min(tmp, −lower[P_j.finish, P_i.request]);
15        if (maxsep[P_i.request, P_l.finish] == −lower[P_l.finish, P_i.request])
16          continue;
17        if (P_l ∉ Q) Enqueue(Q, P_l); /* keep traversing */
18      }
19      if (tmp == −lower[P_j.finish, P_i.request]) break;
20    }
21    maxsep[P_i.request, P_j.finish] = tmp;
22  }
23}
```

Fig. 8. The MaxSeparations algorithm.

cannot execute simultaneously. But in Cases 3 and 4, $P_5$ can preempt $P_6$. This possible overlap must be taken into account during worst-case task delay estimation.

The separation analysis illustrated in Example 4 depends on, but is not the same as, the phase analysis performed by **LatestTimes**. Phase analysis looks at the data dependencies between processes, but does not fully take into account the effects of preemption. The purpose of **MaxSeparations** is to use the results of phase analysis, along with allocation information, to determine the separations between pairs of processes.

The relationship between the start of a process and its predecessors in the task graph can be modeled by *max* constraints [31], [32], [15]. *Max* constraints are introduced by the co-allocation of processes on a processing element. The initiation time of a process is computed by a *max* function of the finish times of its immediate predecessors in the task graph. For example, the completion of a lower priority process depends on both the execution time of that process and the possible execution of higher priority processes. *Max* constraints, unlike data dependencies, are nonlinear and the combinatorics of solving systems of *max* constraints is more challenging than that of a system of linear constraints. *Max* constraints cannot be efficiently processed using the critical-path algorithms exemplified by **LatestTimes**. McMillan and Dill's $O(ne)$ algorithm [32] can deal with *max* constraints and calculate pairwise separations. However, in their algorithm, the delay along a path is the sum of the bounds of individual constraints, which is not true in our case, as mentioned in Example 2.

The algorithm **MaxSeparations**, listed in Fig. 8, finds the maximum separations from $P_i$ to all other processes in the same task graph. It is a modified version of McMillan and Dill's maximum separation algorithm. Like McMillan and Dill's algorithm, **MaxSeparations** traverses the task graph from sink to source to evaluate the maximum separation constraints. The efficiency of a sink-to-source traversal when evaluating max constraints becomes clearer when one considers the alternative of source-to-sink traversal, which would require node values to be repeatedly pushed forward in time to satisfy max constraints.

**MaxSeparations** uses the information computed by **LatestTimes** and **EarliestTimes** to calculate two time instants for a process *P*—*P.request* is the request time of *P* relative to the beginning of the task graph and *P.finish* is the time *P* finishes its execution. Given two time instants *x* and *y*, *upper*[*x*, *y*] is an upper bound on the delay from *x* to *y* and *lower*[*x*, *y*] is a lower bound on the delay from *x* to *y*. Let $G_i$ be a subgraph composed of $P_i$ and all its successors. The modifications to McMillan and Dill's algorithms are required because we are not computing a single value for a node, but rather the upper and lower bounds on request and finish times.

We compute bounds on delays based on request and finish times as follows. After calling **LatestTimes**($G_i$) we can assign

$$upper[P_i.request, P_j.request] = latest[P_j.request]$$

$$upper[P_i.request, P_j.finish] = latest[P_j.finish]$$

for all successors $P_j$ of $P_i$. Similarly, we can call **EarliestTimes**($G_i$) and let

$$lower[P_i.request, P_j.request] = earliest[P_j.request]$$

$$lower[P_i.request, P_j.finish] = earliest[P_j.finish]$$

The bound from a finish time can be derived from the bounds from the request time. For instance,

$$lower\left[P_i.\,finish, p_j.\,request\right] = \max_k lower\left[P_k.\,request, P_j request\right],$$

where $P_k$ is an immediate successor of $P_i$. These bounds are used in **MaxSeparations** to derive the maximum separations.

Given a source node $i$ in McMillan and Dill's algorithm, the maximum separation $maxsep[i, j]$ can be calculated only after $maxsep[i, k]$ is known for all predecessors $k$ of $j$ through recursive calls. In Fig. 8, line 4 searches the task graph in topological order for the same reason. In McMillan and Dill's algorithm, for each immediate predecessor $k$ of $j$,

$$maxsep[i, j] = \min\left(\max_k\left(maxsep[i, k] + upper[k, j]\right), -lower[j, i]\right).$$

Similarly, we apply the $\max_k$ and min operators at lines 11 and 14, respectively. But when a path delay is not the sum of bounds, it is not accurate enough to consider only the immediate predecessors and we use the formula $maxsep[i, k] + upper[k, j]$ to calculate upper bounds. Instead, we try to consider all the predecessors of $P_j$ by a backward breadth-first-search in lines 7-20. The breadth-first-search is trimmed at line 15 and terminated at line 19 when the lower bounds determine the maximum separation value; it is not necessary to trace further back to know the upper bounds along a path. The maximum separation between two request times can be calculated by:

$$maxsep\left[P_i.\,request, P_j.\,request\right] = \max_k maxsep\left[P_i.\,request, P_k.\,finish\right],$$

where $P_k$ is an immediate predecessor of $P_j$.

### 4.4 Iterative Improvement

Separation analysis (performed by **MaxSeparations**) and phase analysis (performed by **LatestTimes** and **EarliestTimes**) compute distinct but related pieces of information about the system schedule. The complete analysis algorithm uses the result from one to improve the results obtained by the other. We use maximum separations to improve delay estimation in **LatestTimes**, but we need to call **LatestTimes** to derive maximum separations. Therefore, we get successively tighter delay bounds and maximum separations iteratively. Initially, to be pessimistic but feasible, we set the maximum separations to ∞.

Maximum separations are used to improve delay estimation in **LatestTimes** as follows:

- If $maxsep[P_i.request, P_j.finish] \le 0$ or $maxsep[P_j.request, P_i.finish] \le 0$, the execution of $P_i$ and $P_j$ will not overlap. The corresponding terms are eliminated from the function $g(x)$ in (2) when the worst-case response time $w_i$ is computed at line 9 in Fig. 6.
- In **LatestTimes**,

  $$\phi_{ij}^r = \max\left(\phi_{ij}^r, -maxsep\left[P_j.\,request, P_i.\,request\right]\right)$$

  for phase adjustment. Similarly, in **EarliestTimes**,

  $$\phi_{ij}^r = \min\left(\phi_{ij}^r, -maxsep\left[P_j.\,request, P_i.\,request\right] - p_j\right)$$

  for phase adjustment.

The use of improved phase estimates to tighten separations computed by **MaxSeparations** is more straightforward—lines 11 through 20 in Fig. 8 directly use the latest values of the finish and request times to update the separations.

### 4.5 Period Shifting and Task Pipelining

This section shows how our algorithm automatically handles pipelined task execution—pipelining causes more complex, nonconstant relationships between process executions which our algorithm automatically handles *without* unrolling to the least-common multiple of the periods. Our earlier examples assumed for simplicity that the period of a process is the same as that of the task to which the process belongs. While this is true for a process with no predecessors in a task graph, this assumption is not accurate in general. The delay for the predecessors may vary from period to period, making the request period of a process different from the period of the task.

EXAMPLE 5. Consider the task graphs and a system implementation shown in Fig. 9. In this figure, an upward arrow represents the request of a process, while a downward dotted arrow stands for a data dependency. $P_1$ and $P_2$ share one CPU, with $P_1$ having higher priority; $P_3$ and $P_4$ share the other CPU, with $P_3$ having higher priority. If we consider the period of $P_3$ to be 70, which is the same as that of $\tau_2$, the worst case delay of $\tau_3$ should be 50. However, the worst case delay of $\tau_3$ can be 80. Note a process cannot start before its request arrives or before its predecessors in the task graph finish.

Suppose there is no task pipelining, so that every process finishes before its next request. In (2), the maximum number of requests for the processes $P_j$ is $\lceil x / p_j \rceil$, where $p_j$ is the period of the task containing $P_j$. Before any preemption from $P_j$ occurs, the term for the number of requests should be modified into

$$\lceil (x + latest[P_j.request] - earliest[P_j.request]) / p_j \rceil$$

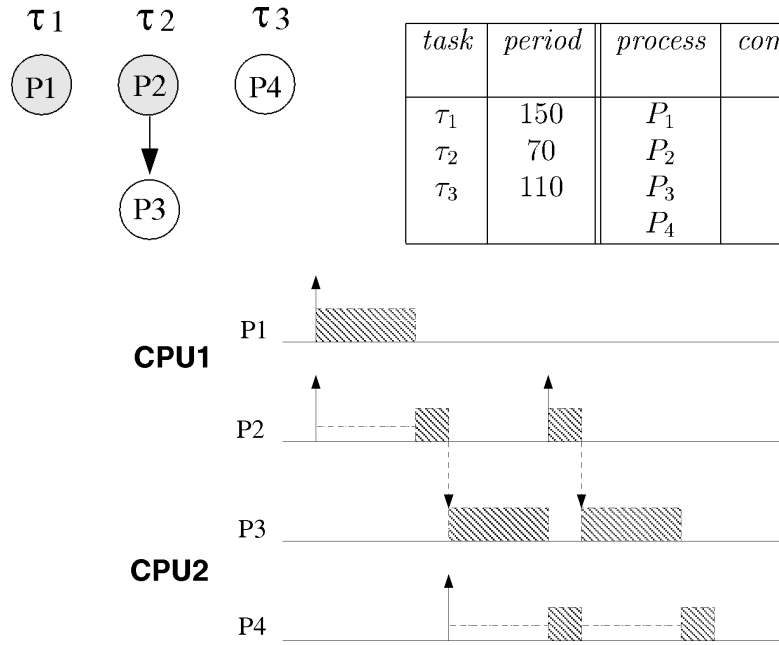
We call such modification *period shifting*. Similarly, when we calculate the earliest times, the minimum number of requests should be

$$\lfloor (x - latest[P_j.request] + earliest[P_j.request]) / p_j \rfloor$$

In the iterative tightening procedures of Fig. 5, we use as an initial worst-case estimate $latest[P_j.request] - earliest[P_j.request] = p_j - c_j$ because, otherwise, $P_j$ may not finish before its next request. Later on, as the values for $latest[P_j.request]$ and $earliest[P_j.request]$ are tightened, period shifting is modeled more and more accurately.

The other phenomenon handled by our algorithm is *task pipelining*. Although the computation time of a process should be smaller than the period of the task containing the process, we allow the deadline or the delay of a task to be longer than its period. As a result, some processes may not finish before the beginning of some processes in the next execution of the same task. To make the techniques discussed so far valid in spite of task pipelining, we require two conditions to be satisfied:

- If two processes $P_i$ and $P_j$ belong to the same task with a minimum period $p$ and are allocated on the same CPU, it




| *task* | *period* | *process* | *computation time* |
|---|---|---|---|
| $\tau_1$ | 150 | $P_1$ | 30 |
| $\tau_2$ | 70 | $P_2$ | 10 |
| $\tau_3$ | 110 | $P_3$ | 30 |
| | | $P_4$ | 20 |

Fig. 9. An example of period shifting.

TABLE 1
THE BOUNDED COMPUTATION TIME ON i960 FOR SEVERAL ROUTINES (AFTER LI AND MALIK)

| function | description | [lower bound, upper bound ] |
|---|---|---|
| checkdata | Park's example | [32, 1039] |
| piksrt | Insertion Sort | [146, 4333] |
| des | Data Encryption Standard (DES) | [42302, 604169] |
| line | line drawing | [336, 8485] |
| circle | circle drawing | [502, 16652] |
| jpegfdct | JPEG forward discrete cosine transform | [4583, 16291] |
| jpegidct | JPEG inverse discrete cosine transform | [1541, 20665] |
| recon | MPEG2 decoder reconstruction | [1824, 9319] |
| fullsearch | MPEG2 encoder frame search routine | [43082, 244305] |
| matgen | matrix generating routine | [5507, 13933] |

TABLE 2
THE BOUNDED COMPUTATION TIME ON DSP3210 FOR SEVERAL ROUTINES (AFTER LI AND ONG)

| Function | Description | [lower bound, upper bound] |
|---|---|---|
| arccos | arc cosine | [166, 706] |
| sqrt | square root | [460, 460] |
| gran | random number generator | [1128, 1128] |
| matmul | matrix multiplication | [810, 810] |
| fft | fast Fourier transform | [103688, 103688] |

is not allowed that *latest*[$P_i$.*finish*] > *latest*[$P_j$.*request*] + *p*. If this requirement is not satisfied, $P_i$ will delay the request time of $P_j$, which may in turn delay the request time of the next iteration of $P_i$ further and there is chance that the delay will continue to grow.

- We will avoid *latest*[$P_i$.*finish*] > *earliest*[$P_j$.*request*] + *p*. When this does happen, we implement a dummy process with a delay *latest*[$P_i$.*finish*] – *p* between the start of the task and the initiation of $P_j$. If this requirement is not satisfied, when $P_i = P_j$, the peak frequency of $P_i$ may get too high due to period-shifting effect, endangering the deadlines of other tasks.

As a matter of fact, such requirements are conservative. However, in most practical pipelining designs, different stages are allocated to different resources (PEs). The execution time of each stage is smaller than the period and it is unlikely that a process will overlap the next execution of the same stage, so these requirements are reasonable in practice.

## 5 EXPERIMENTAL RESULTS

Li and Malik estimated the computation time for some real programs on an Intel i960 [7]and Li and Ong performed a similar task for a Lucent DSP3210 [33]. We repeat their data in Table 1 and Table 2. Their data reveals that, in many real

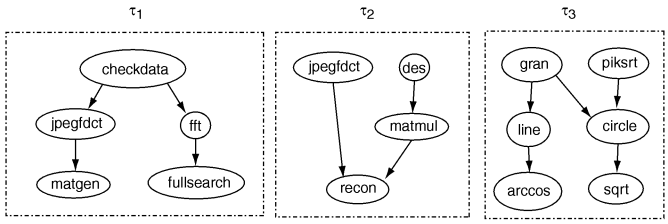



Fig. 10. Three task graphs and their periods.

TABLE 3
THE ALLOCATION AND SCHEDULE OF THE PROCESSES IN FIG. 10

| PE | processes (priority-ordered) |
|---|---|
| i960-1 | piksrt, line, circle, jpegidct |
| i960-2 | jpegfdct, matgen, fullsearch, checkdata |
| i960-3 | des, recon |
| DSP3210 | sqrt, arccos, matmul, gran, fft |

TABLE 4
THE DELAY ESTIMATION FOR THE THREE TASKS IN THE EXAMPLE CREATED FROM LI AND MALIK'S DATA

| method | $\tau_1$ | $\tau_2$ | $\tau_3$ | CPU time |
|---|---|---|---|---|
| our algorithm | 356724 | 615464 | 29930 | 0.12s |
| simulation | 355914 | 615004 | 29930 | 641.76s |

problems, computation time is not constant and is sufficiently large to make the LCM method inefficient. We constructed three task graphs containing these processes as shown in Fig. 10. The processes are allocated in four CPUs: one DSP3210s and three i960s. The allocation and priority assignment of the processes is shown in Table 3. We compared our algorithm to extensive simulations using an interval of length equal to the LCM of the periods, the lower bound value for each period, and the upper bound value for each computation time. The results are given in Table 4 with the CPU time for our analysis algorithm on a Sun SS20 workstation. The results show that simulation requires much more CPU time than our analysis, even though we did not perform exhaustive simulation, which would require possible values between the lower bound and upper bound should be used. Such an exhaustive simulation is too expensive to be implemented.

Some examples in other literature are not directly comparable: The example by Ramamritham [34] did not use static allocation; the example by Peng and Shin [21] used synchronization to make three tasks equivalent to a single nonperiodic task. Both examples have small periods and the LCM of the periods happen to be equal to the largest period. We compare our algorithm with Leinbaugh and Yamini's algorithm [25] using their two examples. In each of their examples, all the tasks have the same period, but the initial phases can be random. The results are given in Table 5. Our algorithm gave better bounds because it does not rely on their pessimistic assumptions. For example, they assume a process with high priority can preempt a task during the whole interval of the task's execution, even though the task only spends a portion of time on the same CPU the high-priority process is allocated to.

Table 6 shows the analysis results for three designs in D'Ambrosio and Hu's example [26]. In this table, "yes" means

TABLE 5
THE ESTIMATED TASK DELAYS FOR LEINBAUGH AND YAMINI'S EXAMPLES

| Example | Method | Task 1 | Task 2 | Task 3 | Task 4 | CPU time |
|---|---|---|---|---|---|---|
| #1 | L & Y | 1116 | 1110 | 1114 | | NA |
| | ours | 1093 | 1084 | 1072 | | 0.03s |
| #2 | L & Y | 959 | 845 | 863 | 912 | NA |
| | ours | 655 | 586 | 820 | 637 | 0.03s |

TABLE 6
DEADLINE SATISFACTION RESULTS FOR D'AMBROSIO AND HU'S EXAMPLE

| PEs | cost | D&H's simulation | Our algorithm | CPU time | Our simulation |
|---|---|---|---|---|---|
| P1, PIO | 3.00 | no | no | 0.04s | no |
| MC2 | 3.25 | yes | no | 0.04s | no |
| MC1 | 3.50 | yes | yes | 0.04s | yes |

that the deadlines of all nine processes were satisfied, while "no" means that at least one was not. In their example, the largest period is the LCM of all the periods. There are no data dependencies between processes, but each process has both a release time and a deadline. We also ran a simulation in a length equal to the largest period and compare the results.

## 6 CONCLUSIONS

Distributed computers are often the most cost-effective means of meeting the performance requirements of an embedded computing application. We have proposed an algorithm for tight yet easy-to-compute timing bounds on the tasks. This work does not address all the modeling issues in distributed embedded systems. In other work, we have studied the problem of communication link contention [35], however, more work remains in integrating communication link scheduling with processing element scheduling. We have also not considered the effects of preemption overhead. We have used this analysis algorithm to develop a hardware/software co-synthesis algorithm [1] which simultaneously designs a hardware topology and allocates and schedules processes on CPUs to meet hard real-time deadlines. Once again, there is more work to be done in this area. We believe that algorithms such as this are an important tool for the practicing embedded system designer.

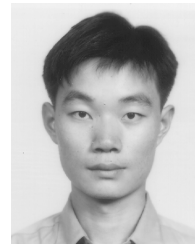
## ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under Grant MIP-9424410.

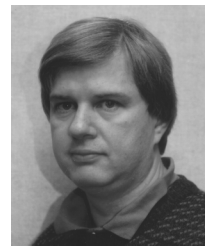
## REFERENCES

[1] T.-Y. Yen and W. Wolf, "Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems," *Proc. Eighth Int'l Symp. System Synthesis*, pp. 4–9, 1995.
[2] W. Wolf, "Hardware-Software Co-Design of Embedded Systems," *Proc. IEEE*, vol. 82, no. 7, July 1994.
[3] K.G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proc. IEEE*, vol. 82, no. 1, Jan. 1994.
[4] C. Rosebrugh and E.-K. Kwang, "Multiple Microcontrollers in an Embedded System," *Dr. Dobbs J.*, Jan. 1992.
[5] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[6] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, vol. 2, 1982.
[7] S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *Proc. Design Automation Conf.*, 1995.
[8] T.-Y. Yen and W. Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems," *Proc. IEEE Int'l Conf. Computer Design*, 1995.
[9] T.-Y. Yen and W. Wolf. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Norwell, Mass.: Kluwer Academic, 1996.
[10] C.Y. Park, "Predicting Deterministic Execution Times of Real-Time Programs," PhD thesis, Univ. of Washington, Seattle, Aug. 1992.
[11] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," *Proc. IEEE Int'l Conf. Computer Design*, 1993.
[12] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, Jan. 1973.
[13] L. Sha, R. Rajkumar, and S.S. Sathaye, "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," *Proc. IEEE*, vol. 82, no. 1, Jan. 1994.
[14] L.Y. Liu and R.K. Shyamasundar, "Static Analysis of Real-Time Distributed Systems," *IEEE Trans. Software Eng.*, vol. 16, no. 4, Apr. 1990.
[15] T. Amon, H. Hulgaard, S.M. Burns, and G. Borriello, "An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems," *Proc. IEEE Int'l Conf. Computer Design*, 1993.
[16] W.W. Chu, C.-M. Sit, and K.K. Leung, "Task Response Time for Real-Time Distributed Systems with Resource Contentions," *IEEE Trans. Software Eng.*, vol. 17, no. 10, Oct. 1991.
[17] W.W. Chu and L.M.-T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Trans. Computers*, vol. 36, no. 6, June 1987.
[18] Chiodo, Guisto, Hsieh, Jurecska, Lavagno, and Sangiovanni-Vincentelli, "Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems," *Proc. Int'l Workshop Hardware-Software Co-Design*, 1993.
[19] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. IEEE Real-Time Systems Symp.*, 1989.
[20] K. Ramamritham and J.A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proc. IEEE*, vol. 82, no. 1, Jan. 1994.
[21] D.-T. Peng and K.G. Shin, "Static Allocation of Periodic Tasks with Precedence Constraints," *Proc. Int'l Conf. Distributed Computing Systems*, 1989.
[22] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 4, Apr. 1995.
[23] C.J. Hou and K.G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.*, 1982.
[24] R. Gerber, W. Pugh, and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks," *IEEE Trans. Computers*, vol. 44, no. 3, pp. 471-479, Mar. 1995.
[25] D.W. Leinbaugh and M.-R. Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," *Proc. Real-Time Systems Symp.*, 1982.
[26] J.G. D'Ambrosio and X. Hu, "Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems," *Proc. Int'l Workshop Hardware-Software Co-Design*, 1994.
[27] S. Prakash and A.C. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *J. Parallel and Distributed Computing*, vol. 16, 1992.
[28] R.K. Gupta and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29-41, Sept. 1993.
[29] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Co-Synthesis for Microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 61-75, Dec. 1993.
[30] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. McGraw-Hill, 1990.
[31] P. Vanbekbergen, G. Goossens, and H. De Man, "Specification and Analysis of Timing Constraints in Signal Transition Graphs," *Proc. European Conf. Design Automation*, 1992.
[32] K. McMillan and D. Dill, "Algorithms for Interface Timing Verification," *Proc. IEEE Int'l Conf. Computer Design*, 1992.
[33] S. Li and P.-W. Ong, "Cinderella for DSP3210," unpublished report, 1994.
[34] K. Ramamritham, "Allocation and Scheduling of Complex Periodic Tasks," *Proc. Int'l Conf. Distributed Computing Systems*, 1990.
[35] T.-Y. Yen and W. Wolf, "Communication Synthesis for Distributed Embedded Systems," *Proc. Int'l Conf. Ccomputer-Aided Design-95*, pp. 288-294, 1995.



**Ti-Yen Yen** received his BS degree in electrical engineering from National Taiwan University in 1989 and his PhD degree in electrical engineering from Princeton University in 1996. He served as an electronic engineering officer in the Navy of Taiwan from 1989 to 1991. He joined Quickturn Design Systems, San Jose, California, in 1995 and is now the HDL-ICE project manager. His research interests include synthesis, design verification, hardware/software co-design, emulation, and reconfigurable computing.



**Wayne Wolf** received the BS, MS, and PhD degrees in electrical engineering from Stanford University in 1980, 1981, and 1984, respectively. He was with AT&T Bell Laboratories from 1984 through 1989. He joined the Department of Electrical Engineering at Princeton University in 1989, where he is now a professor. His research interests include hardware/software co-design and embedded computing, multimedia computing systems, and video libraries. He is a fellow of the IEEE and a member of the ACM and SPIE.