EDF Feasibility and Hardware Accelerators

Andrew Morton
University of Waterloo, Waterloo, Canada, arrmorton@uwaterloo.ca

Wayne M. Loucks
University of Waterloo, Waterloo, Canada, wmloucks@pads.uwaterloo.ca

A feasibility analysis is developed for embedded systems that use hardware accelerators to speed up critical portions of the software system. The scheduling policy analyzed is Earliest Deadline First. The concept of an accelerator-induced idle is introduced along with modifications in representation for tasks employing hardware accelerators. These modifications are incorporated into an existing algorithm for feasibility analysis of regular task sets. The changes introduced allow the benefits of the accelerators (parallel execution between processor and accelerator and reduced overall task execution time) to be accounted for, resulting in a less conservative analysis.

Keywords: Real-Time Scheduling, Theoretical Scheduling, Hardware Accelerators.

1 Introduction

This paper presents an algorithm for analyzing schedule feasibility under the Earliest Deadline First (EDF) policy. Specifically, EDF feasibility is examined for embedded real-time systems that use hardware accelerators to speed up parts of the software application.

For a hard real-time system, it is necessary that all tasks not only function properly but also meet their deadlines. Functional correctness without temporal correctness is failure and can have significant consequences. An algorithm to ensure schedule feasibility must be able to check that all tasks will meet their deadlines, every time.

The scheduling policy being examined in this paper is EDF. This is an important real-time scheduling policy for two reasons. First, it is optimal, in the sense that it will fail to meet a task’s deadline only if no other scheduling policy could meet the deadline. Second, a task’s priority is determined by the closeness of it’s deadline. This is a more natural way to schedule real-time tasks than the rate monotonic policy, where fixed task priorities are set to approximate task deadlines.

The problem of analyzing schedule feasibility can be exacerbated in embedded systems by the use of hardware accelerators to speed up selected “kernels” of software that have high computation cost. In this paper, the feasibility analysis algorithm proposed by Stankovic et al [3] is extended for tasks employing hardware accelerators for speed-up.

In the next section, feasibility analysis for tasks sets scheduled by EDF is introduced. This introduction is followed by motivating the extended analysis for hardware accelerators and then deriving the analysis. Lastly, the implications and usability of the extended analysis are considered and conclusions are drawn.

2 EDF Schedule Analysis

Under the EDF policy, of all ready tasks, the task with the earliest deadline is executed first. If another task arrives with an earlier deadline, it will preempt the currently executing task. Periodic tasks occur at regular intervals. Periodic task \( \tau_i \) has start time \( s_i \), period \( T_i \), relative deadline \( D_i \) and worst-case execution time \( C_i \). \( \tau_i \) is released at the start of each period at time \( r_i = s_i + mT_i \),
The task must complete by absolute deadline $d_i = r_i + D_i$. Sporadic tasks are not released at regular intervals but can be characterized by minimum inter-arrival time $T_i$. For the purpose of analysis, the sporadic task can be conservatively represented by a periodic task. A task set composed of periodic and sporadic tasks is a hybrid task set.

In Figure 1, $\tau_1$ is periodic and $\tau_2$ is sporadic. Task releases are indicated with an up arrow and task deadlines are indicated with a down arrow. $\tau_1$ has start time $s_1 = 2$, relative deadline $D_1 = 8$, period $T_1 = 12$ and worst-case execution time $C_1 = 4$. It has release dates $r_1 = 2, 14, \ldots$ and corresponding deadlines $d_1 = 10, 22, \ldots$ $\tau_2$ is released sporadically at times $t = 0, 16, 21$. At $t = 16$ $\tau_2$ preempts $\tau_1$ because its deadline is closer. $\tau_1$ is resumed at $t = 18$ after $\tau_2$ terminates. The minimum inter-arrival time for $\tau_2$, as shown in Figure 1, is $T_2 = 21 - 16 = 5$. An instance of a task is called a job. In Figure 1, $\tau_1$ has 2 jobs and $\tau_2$ has three jobs.

The purpose of EDF analysis is, given task set $\tau$, determine if it can be proved that all $n$ tasks in the set can be scheduled by the EDF policy such that no job misses its deadline. Stankovic et al [3] have developed an analysis for task sets, which is based on processor demand. It is summarized here.

### 2.1 Algorithm Summary

A necessary but not sufficient condition for the feasibility of a task set scheduled by the EDF policy is that the processor utilization $U$ (defined by Liu and Layland [1]) does not exceed one:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

$U$ sums the fraction of processor time required per task.

The rest of the analysis assumes that the task set is synchronous: all tasks are released together at $t = 0$ (∀ $i$: $s_i = 0$). This is the most constraining scenario, otherwise known as the “critical instance”. After checking $U \leq 1$, the algorithm checks processor demand. Processor demand, $h(t)$, in the interval $[0, t]$ is defined:

$$h(t) = \sum_{D_i \leq t} \left( 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right) C_i.$$  

Processor demand at time $t$ measures the work required by all tasks having deadlines in $[0, t]$. Processor demand is applied in the following theorem:

**Theorem 1** (Processor Demand Condition [3]). Any given hybrid task set is feasible under EDF scheduling if and only if

$$\forall t : h(t) \leq t.$$
However, it is not necessary to test processor demand on all intervals in \([0, t)\); it is sufficient to test only when task deadlines occur [4]. Furthermore, an upper bound can be placed on the interval over which \(h(t)\) is checked. This is done by calculating the length \(L\) of the synchronous busy period:

**Theorem 2** (Liu and Layland [1]). *When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow.*

The interval of time preceding the first idle period is called the synchronous busy period. Stankovic et al describe the following iterative method for calculating the length \(L\) of the synchronous busy period. This algorithm converges in \(O(\sum_{i=1}^{n} C_i)\) time, if \(U < 1\) [2].

Apply recursively until \(L^{m+1} = L^m\):

\[
\begin{aligned}
L^{(0)} &= \sum_{i=1}^{n} C_i, \\
L^{(m+1)} &= W(L^{(m)}),
\end{aligned}
\]

where

\[
W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i.
\]

In essence, all tasks are released at \(t = 0\) and their cumulative execution times (busy period) are calculated \(L^{(0)}\). The algorithm iteratively checks how many jobs have been released in \(L^{(m)}\) and sums their execution times \(L^{(m+1)}\). When it reaches \(L^{(m+1)}\) where all tasks have finished executing before any new jobs are released, it terminates.

The set \(S\) of deadline events is defined over \([0, L)\), and for each event \(v\) in \(S\), it is checked that the processor demand doesn’t exceed processor time \((h(v) \leq v)\). Algorithm 1 lists the pseudocode for Stankovic et al’s feasibility analysis algorithm.

**Algorithm 1: EDF Feasibility Analysis**

```
if \(U > 1\) then return "not feasible";
S = \(\bigcup_{i=1}^{n} \{mT_i + D_i : m = 0, 1, \ldots\}\) \(\{v_1, v_2, \ldots\}\) // deadlines;
k ← 1;
while \(v_k < L\) do
    if \(h(v_k) > v_k\) then return "not feasible" // check processor demand;
k ← k + 1;
end
return "feasible";
```

### 3 Hardware Accelerators and Scheduling

A common platform for embedded systems is the System on Chip (Soc). In a SoC, a general purpose processor (CPU), RAM, ROM, I/O and application specific hardware are integrated on one IC. The application specific hardware typically consists of accelerators for portions, otherwise known as “kernels”, of the application that do not execute fast enough in software or consume too much processor time. Accelerators that execute independently of any software task are not considered here as they do not pose a problem for schedule analysis. However a software task that invokes a hardware accelerator and then waits, or “blocks” until the accelerator finishes before continuing is
of interest for schedule analysis. Such a software task is called an “accelerator-blocked” task and is illustrated in Figure 2.

The task $\tau_i$ has three parts, $\tau_{i,1}$, $\tau_{i,2}$, $\tau_{i,3}$. Part $\tau_{i,2}$ has been identified for speed-up in hardware. In Figure 2(a) it executes in software and in Figure 2(b) $\tau_{i,2}$ is replaced by a hardware accelerator. A simple approach to the analysis is to calculate the worst-case execution time of task $\tau_i$ as the sum $C_i = C_{i,1} + C_{i,2} + C_{i,3}$ and use Algorithm 1 without modification. This may be a reasonable approach when $C_{i,2} \ll C_{i,1} + C_{i,3}$. The primary drawback of this approach is that the parallel execution between processor and accelerator is ignored. In other words, while $\tau_i$ is waiting for $\tau_{i,2}$ to execute on the hardware accelerator, another task could be using the processor. The algorithm could rule a task set infeasible that is actually feasible. The goal of the extended analysis is to account for the parallel execution between the processor and hardware accelerators.

### 4 Extended Analysis

The analysis is initially developed for a task set in which one task blocks on an accelerator once during its execution. Extension to a task set in which one task blocks multiple times on an accelerator during its execution will then be briefly discussed.

To aid in the extended analysis, the execution time of each task is represented by vector $\vec{C}_i$. For a regular task the size of the vector is $|\vec{C}_i| = 1$. For the accelerator-blocked task from Figure 2(b), $|\vec{C}_i| = 3$. $C_{i,2}$ is execution time on the accelerator.

In order to perform the analysis, the accelerator-blocked task $\tau_i$ is replaced by three ordered subtasks as described in Table 1. The deadlines are modified so that $\tau_{i,1}$ finishes in time for $\tau_{i,2}$ and $\tau_{i,3}$ to finish. Likewise, $\tau_{i,2}$ finishes in time for $\tau_{i,3}$ to finish. Because EDF schedules tasks with earlier deadlines first, the subtasks will execute in the correct order.

The task set $\tau$ is modified by replacing the accelerator-blocked task $\tau_i$ with its two software subtasks $\tau_{i,1}$ and $\tau_{i,3}$. For now $\tau_{i,2}$, which executes in hardware, is ignored. It will be accounted for later.

Now consider the example task set in Figure 3. If $\tau_{1,1}$, $\tau_{1,3}$ and $\tau_2$ are used as the task set in the analysis of Algorithm 1, it will pass the analysis and be deemed feasible. That is because there is enough processor time to meet the processor demand at all deadline events $S = \{2, 7, 11\}$ in the synchronous busy period. The problem arises with the second jobs of $\tau_1$ and $\tau_2$. There is enough processor time to fulfill the software tasks’ demands but $\tau_{i,3}$ cannot use some of the available time.

---

1In [3], the upper bound is chosen from the minimum of three values. The other two upper bounds are omitted here as they are not used in the extended algorithm in Section 4.
because it is waiting for $\tau_{i,2}$ to finish on the accelerator. This causes an “accelerator-induced” idle that is not accounted for by the analysis. The goal of the extended analysis is to determine whether an accelerator-induced idle can result in a missed deadline.

**Lemma 1.** Given a task set $\tau$ in which one task $\tau_x$ blocks on an accelerator once, with logical subtasks $\{\tau_{x,1}, \tau_{x,2}, \tau_{x,3}\}$, the accelerator-induced idle results in a critical instance when:

1. $\tau_{x,3}$ has no slack, and
2. all other tasks are released synchronously at the end of the accelerator-induced idle (i.e. at $t = D_{x,2}$).

A second critical instance has been introduced. The first is the synchronous release of all tasks in $\tau$. The second is an accelerator-induced idle during which no other tasks are ready to execute but at the end of which all tasks (except $\tau_{x,1}$) are released.

Before proving Lemma 1, loading factor must be introduced. Loading factor on the interval $[t_1, t_2]$ is defined as

$$u_{[t_1, t_2]} = \frac{h_{[t_1, t_2]}}{(t_2 - t_1)},$$

where $h_{[t_1, t_2]}$ is the sum of work released not earlier than $t_1$ with deadline not later than $t_2$:

$$h_{[t_1, t_2]} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k,$$

where $r_k$, $d_k$ and $C_k$ are release, deadline and worst-case execution time for jobs in that interval. Loading factor is the fraction of the interval required to meet its processor demand. The absolute loading factor is the maximum loading factor of all possible intervals:

$$u = \sup_{0 \leq t_1 < t_2} u_{[t_1, t_2]}.$$

The proof of Lemma 1 now follows.

**Proof.** The worst-case schedule feasibility occurs when:

- **Part 1:** $\tau_{x,3}$ has no slack ($d_{x,3} = C_{x,3}$)
  If $\tau_{x,3}$ has no slack, then no other work can be done before $d_{x,3}$. (i.e. if there exists another task with deadline before $d_{x,3}$, then at $t = d_{x,3}$, $h(t) > t$ which makes the schedule infeasible.) Adding slack would only improve schedule feasibility.
• **Part 2:** all other tasks are released synchronously with \( \tau_{x,3} \)

The remaining part of this proof is the same, in essence, as the proof of Lemma 3.1 in [3] except for modifications to the accelerator-blocked task as described next.

Let \( \tau \) be the asynchronous task set \( (\exists i \mid s_i \neq 0) \) and \( \tau' \) the corresponding synchronous task set and let the accelerator-blocked task, \( \tau_x \), be excluded from \( \tau \) and \( \tau' \). When the processor demands, \( h_{[t_1, t_2]} \) and \( h'_{[t'_{1}, t'_{2}]} \), of \( \tau \) and \( \tau' \) are computed, they are augmented by the accelerator-blocked task as follows:

- In calculating the processor demand, a unit of work \( C^* \) is released at \( t_1 \) with duration \( C^* = \bar{C}_{x,3} \), representing the part of the accelerator-blocked task that must execute after the accelerator-induced idle. The deadline of \( C^* \) is \( D^* \) which is equal to \( \bar{C}_{x,3} \) (no slack). Furthermore \( \tau_x \) is replaced by \( \tau_{x,1} \) and \( \tau_{x,3} \). Since \( C^* \) represents the remaining work of the previous job of \( \tau_x \) which has a deadline at \( t = \bar{C}_{x,3} \), the next job of \( \tau_x \) is released at \( T_x - D_x + \bar{C}_{x,3} \) after \( t_1 \). Tasks \( \tau_{x,1} \) and \( \tau_{x,3} \) are said to have phases \( \phi_{x,1} = \phi_{x,3} = T_x - D_x + \bar{C}_{x,3} \). Only an accelerator-blocked task is said to have phase; for all other tasks \( \phi_i = 0 \) (note that phase and start time are not related). Figure 4 shows the relationship between \( \tau_x \) and \( C^* \) and \( \tau_{x,1} \) and \( \tau_{x,3} \) with phases.

In order to prove that synchronous release of all other tasks but the accelerator-blocked task is the worst-case scenario, it is sufficient to prove that the absolute loading factor (“loading factor”) \( u \) of \( \tau \) is bounded above by the loading factor \( u' \) of \( \tau' \). It is therefore sufficient to prove that \( \forall [t_1, t_2] \) there exists an interval \( [t'_{1}, t'_{2}] \) such that \( h_{[t_1, t_2]} \leq h'_{[t'_{1}, t'_{2}]} \), where the accelerator-blocked task is included as described in the previous paragraph. For any interval \( [t_1, t_2] \), the processor demand for \( \tau \) is bounded above by:

\[
h_{(t_1, t_2)} \leq C^* + \sum_{D_i + \phi_i \leq t_2 - t_1} \left( 1 + \left( \frac{t_2 - t_1 - (D_i + \phi_i)}{T_i} \right) \right) C_i.
\]

Addition of \( \phi_i \) to this equation only affects \( \tau_{x,1} \) and \( \tau_{x,3} \).

Now consider the synchronous case, \( \tau' \), and let \( t'_{1} = 0 \) and \( t'_{2} = t_2 - t_1 \). Processor demand is
Processor demand of the synchronous case, $h(t'_{1}, t'_{2})$, equals the upper bound of the non-synchronous case, $h(t_{1}, t_{2})$. Therefore:

$$h(t_{1}, t_{2}) \leq h(t'_{1}, t'_{2}).$$

To apply this result, the analysis of Algorithm 1 is done twice: once for each critical instance (synchronous release and accelerator-induced idle). The synchronous busy period and processor demand calculations need to be redefined for the second critical instance. To facilitate this, the phase of the accelerator-blocked task is redefined.

In the proof above, the portion of the accelerator-blocked task remaining after the accelerator-induced idle was represented by $C^{*}$ and the next job of the task had phase $\phi_{x,1} = \phi_{x,3} = T_{x} - D_{x} + C_{x,3}$. This can also be represented by modifying the phase of $\tau_{x,3}$: $\phi_{x,3} = -(D_{x} - C_{x,3})$. By doing this $\tau_{x,3}$ has an initial deadline at $t = C_{x,3}$. Since $D_{x} - C_{x,3} = D_{x,2}$, the phases of the two subtasks of the accelerator-blocked task $\tau_{x}$ become:

$$\phi_{x,1} = T_{x} - D_{x,2} \quad \phi_{x,3} = -D_{x,2},$$

and every other task has phase $\phi_{i} = 0$. The change in $\phi_{x,3}$ can be seen by comparing Figures 4 and 5. In Figure 5, $C^{*}$ has been incorporated into $\tau_{x,3}$.

The synchronous busy period is now calculated as follows:

Apply recursively until $L'_{m+1} = L'_{m}$:

$$
\begin{align*}
L'^{(0)} & = \sum_{\phi_{i} \leq 0} C_{i}, \\
L'^{(m+1)} & = W'(L'^{(m)}),
\end{align*}
$$

where

$$W'(t) = \sum_{\phi_{i} \leq t} \left\lceil \frac{t - \phi_{i}}{T_{i}} \right\rceil C_{i}. \quad (6)$$
When \( \phi_{x,1} = \phi_{x,3} = 0 \), then \( L' = L \). When \( \phi_{x,1} = T - D_{x,2} \) and \( \phi_{x,3} = -D_{x,2} \), then the result is different: at \( t = 0 \) all of the regular tasks are released as well as accelerator-blocked subtask \( \tau_{x,3} \). The processor demand calculation is also modified for phase:

\[
h'(t) = \sum_{D_i + \phi_i \leq t} \left( 1 + \left[ \frac{t - (D_i + \phi_i)}{T_i} \right] \right) C_i.
\]  

(7)

4.1 Algorithm Analysis

Algorithm 1 is run twice: once with \( \phi_{x,1} = \phi_{x,3} = 0 \), and once with \( \phi_{x,1} = T_x - D_{x,2} \) and \( \phi_{x,3} = -D_{x,2} \). In both instances, \( \tau_x \) is replaced by \( \tau_{x,1} \) and \( \tau_{x,3} \) in \( \tau \). As a result, the execution on the hardware accelerator is omitted from the processor demand analysis. The implication is that the parallel execution between processor and accelerator is incorporated into the feasibility analysis. However to do this, a second critical instance is added that looks at the worst-case scenario that can follow an accelerator-induced idle. This puts a limitation on the type of task set that will pass the analysis: all regular tasks must have enough slack to permit \( \tau_{x,3} \) to execute at \( t = 0 \). Since accelerator execution times are generally shorter than task execution times, it is expected that this limitation will not be onerous.

Extending the algorithm to task sets in which one task blocks multiple times is described only briefly due to space limitations. The length of the execution time vector \( C_x \) of the accelerator-blocked task would be incremented by two for each \{accelerator,software\} pair added to it’s execution as in Figure 6. Each time the task blocks on a accelerator, it may result in a accelerator-induced idle. Thus each accelerator block in the vector is associated with a critical instance. The analysis of Algorithm 1 is performed once with all phases equal zero and then is repeated once per block with phases modified to simulate the appropriate accelerator-induced idle.

5 Conclusions

Task sets in which a task blocks on an accelerator have been shown to add a second critical instance for feasibility analysis. This is done by including only the parts of that accelerator-blocked task that execute on the processor in \( \tau \). Phases for this subtasks were introduced to facilitate analysis of the second critical instance. The analysis incorporates the parallel execution between processor and accelerator but impose some limitations on the characteristics of task sets that can pass the analysis. Analysis of task sets with multiple accelerator-blocked tasks is left for future work.

References

