Selecting Server Parameters for Predictable Runtime Monitoring

Haitao Zhu, Steve Goddard, Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588
{hzhu,goddard,dwyer}@cse.unl.edu

Abstract—Application of runtime monitoring to maintain the health of an embedded real-time software system requires that anomalous behavior be detected within a bounded time while preserving the temporal guarantees of the underlying system. Existing results can compute bounds on the detection latency of runtime monitors that are realized as a deferrable server running at the highest priority. In this paper, we generalize those results to allow monitors to run at an arbitrary priority. We also present an analysis of queue length in predictable runtime monitoring, which allows one to compute an upper bound on queue length. When implementing predictable runtime monitoring, system engineers are presented with several challenges in configuring the parameters of monitor servers. To address those challenges, we explore the tradeoffs among key server parameters and make recommendations about how best to select those parameters to achieve system monitoring objectives.

I. INTRODUCTION

A recent trend in high-assurance embedded systems is the use of runtime monitoring to assure safe system operation by detecting and compensating for violations of critical system properties. In this approach a monitor executes alongside the program observing its behavior at selected points and comparing those observations against a correctness specification. For example, a recent proposal for assured flight control of unmanned air vehicles deploys a complex controller and a simple, statically verified, controller together on the aircraft [16]. A monitor observes vehicle parameters and detects when it is operating outside of the envelope for which the safety of the complex controller is known. Once the monitor detects such a safety violation the system switches to use the simpler controller whose safety has been verified.

Introducing runtime monitors is intended to enhance system correctness, but in doing so the monitors must neither logically nor temporally interfere with underlying system behavior. Moreover, as illustrated by the above example, the detection of violating system states must occur within a potentially stringent time bound in order to ensure that the system can compensate quickly enough to maintain safety. To address these challenges, we introduced a framework for predictable runtime monitoring which guarantees that violations will be detected within a specified time bound and that the timeliness guarantees of the underlying system are preserved [18].

Predictable runtime monitoring requires the explicit definition of a monitoring budget—the share of resources, e.g., CPU time and memory, that are dedicated to monitoring. The framework presented in [18] focuses on the resource of CPU time and defines the budget as a tuple \((e_m, p_m)\), where \(e_m\) is the execution time allocated to the monitor in a period of \(p_m\) time units. A deferrable server [17] is used to execute the monitor, with the server being assigned the highest priority in the system. While effective at minimizing the bound on violation detection latency, it is not always desirable to have a single monitor executing at the highest system priority. For example, an application may want to categorize violations due to criticality or differing violation detection latency requirements. In this case, it is desirable to have a separate monitor process each category. Even with a single monitor, an application may require one or more mission critical tasks have priority over runtime monitoring. Aside from CPU time, runtime monitoring also has storage requirement. For space-constrained applications, predictable runtime monitor requires the space occupied by the monitor to be bounded. The framework presented in [18] does not directly support these scenarios.

In this paper, we address these deficiencies. Given the dominating application of fixed-priority scheduling algorithms in most real-time OS (VxWorks, and LynxOS, etc.), we shall focus on fixed-priority systems in this paper. We also provide guidelines and the analytic tools necessary for engineers to select server parameters that meet application specific requirements without over-provisioning the monitor. Specifically, we address the following questions an engineer will face when implementing runtime monitoring using a deferrable server:

1) What execution budget \((e_m, p_m)\) should be assigned to each monitor?
2) Given a specified priority and server execution budget \((i.e., (e_m, p_m))\), what is the bound on the latency and storage requirements for detecting violations?
3) Given a specified latency for detecting a violation, what priority should be given to the monitor?
4) Given a specified amount of space for storing events,
This paper is organized as follows. In Section II, we provide an overview of our predictable runtime monitoring approach and describe how the work in this paper differs from previous work on this problem. In Section III, we define the foundations for predictable runtime monitoring at different system priorities. These foundations allow us to bound both the detection latency and the queue length used by the monitor. In Section IV, the concepts and properties related to a Most Stringent Execution Pattern are introduced. The analysis of the Maximum Queue Length and the Maximum Detection Latency are respectively presented in Sections V and VI. Then in Section VII we explore tradeoffs between different monitor server parameters and provide guidance as to the best strategies for selecting those parameters to resolve the key design decisions identified in Section I.

II. OVERVIEW AND BACKGROUND

In this paper we consider runtime monitoring to check safety properties — properties whose violations are witnessed by finite prefixes of system executions. For example, a regular expression can be used to encode the pattern of resource acquisition (acq) and release (rel)

\[(\text{rel}^*; \text{acq}; \text{rel}^+)\]^*

where a release must follow an acquire, consecutive acquires are disallowed, and extra releases are ignored.

Monitoring involves generating events that represent the execution of system behaviors that are relevant to the property of interest and then processing events to judge whether the sequence of generated events is a violation of the property. Figure 1 illustrates these activities for the acquire-release property formulated as a finite state automaton. Execution states of system tasks, which may be defined as predicates over program locations and data values, are used to define the occurrence of events that drive system monitoring. For example, a predicate might be defined to hold immediately after an acquire() call is performed on a resource. An instance of such a call would then trigger the generation of an acq event. Generated events are inserted into a queue and a monitor then processes queued events to track the sequence of relevant system events to determine when a violation has occurred.

Violations may trigger the system to execute compensating actions to recover from or adapt processing to any detected errors. While we do not consider the implementation of such actions here we note that there are frameworks that support their definition, e.g., [3].

Not all system events demand the same level of attention from a runtime monitor. We distinguish those events that can lead to violations, since they are the key to bounding the time and space needed for detecting violations.

**Definition II.1.** A Possible Violating Event (PVE) is an event that appears as the last event in an event sequence that violates a safety property specification.

To illustrate, in the acquire-release example, acq is a PVE, since it labels a transition to the viol state, but rel is not; note that once a property is violated subsequent events are ignored. PVEs can be calculated by a simple offline analysis of the property; for finite state automata, the symbols that label transitions to a trap state, i.e., a non-accept sink state, are PVEs. While runtime monitoring requires that both PVEs and non-PVEs be processed to drive state transitions, processing of multiple non-PVEs can be optimized to minimize their storage and computational requirements. In our example, a sequence of multiple consecutive occurrences of rel can be collapsed into a single occurrence based on an analysis of the structure of the property automaton. Non-PVEs can always be collapsed and associated with the preceding PVE [7], thus it suffices to consider only PVEs.

In predictable runtime monitoring PVEs play a special role. Whenever a PVE occurs a property may have been violated. Ideally, the property monitor would be notified of a PVEs occurrence immediately so that it can determine whether a violation has occurred and, if so, notify the system. PVE notification may be delayed, however, due to, for example, ongoing high-priority system processing. Predictable monitoring requires that this delay is explicitly managed by the run-time system.

The detection latency (DL) of a property violation is the time between when the violating event is generated by the system and when the monitor detects this violation. Since PVEs are queued before being processed, the detection latency is closely related to the length of that queue. The queue length (QL) is the number of PVEs stored in a monitor’s queue at a point during system execution.

For each property, a monitor and a queue are defined to process the events that are relevant for the property. When a PVE occurs for a property, it isn’t immediately obvious that a violation has occurred since property violation depends on the state of the monitor. Moreover, one can’t just consult the current monitor state since there may be queued events that are waiting to be processed to determine the appropriate state for processing the latest PVE. Nevertheless, since the occurrence of a PVE may be the first evidence of a violation in the system it serves as the reference point relative to which predictable monitoring must be defined.

**Definition II.2.** The Maximum Detection Latency (MDL) of a property is the longest detection latency for any violation of
the property that might be observed during system execution.

Since PVEs represent potential violations, the MDL may be bounded by analyzing the latency for processing all PVEs generated by the system.

Definition II.3. The Maximum Queue Length (MQL) of a property is the maximum number of events that may be stored in the property monitor’s queue at any point during system execution.

Similarly, by focusing on PVEs the MQL may be bounded by analyzing the number of generated, but unprocessed PVEs at any point in the system.

Definition II.4. Given a property p with an MDL of d and an MQL of l, runtime monitoring is predictable if (a) the monitor can detect all the violations of p in a system within d time units after their occurrence; (b) the monitor never stores more than l events in its queue; and (c) the execution of the monitor does not affect the original system’s schedulability.

Previous Work. In [18] we defined predictable monitors using a deferrable server running at highest priority in the system. This allowed us to bound the maximum detection latency based on the arrival rate of PVEs. While this provides one to the predictable monitoring problem it also limits the applicability of monitoring unnecessarily.

Some applications may have very stringent timeliness requirements that require a task to run at the highest priority on a given execution platform. In such cases, it would be impossible to place monitors at the highest priority without violating system timeliness guarantees. Even if the application does not require use of the highest priority task, the detection latencies for monitoring may be lax enough to permit monitors to run at a lower priority. To run such monitors at the highest priority wastes processor bandwidth which could possibly make lower priority tasks, that were schedulable without the monitor, unschedulable. Finally, when multiple properties in a system have very different detection latency requirements mapping them onto a single highest priority server is inefficient. Defining two monitor servers and assigning them appropriate priorities will allow for better use of system resources. Thus, a key contribution of this paper is the generalization of predictable monitoring to support different server priorities.

Our previous work focused exclusively on the MDL, but predictable monitoring, especially on space-constrained platforms, may only be possible if queue lengths can be bounded at modest values. Thus, a key contribution of this paper is the explicit investigation of the time-space resource relationship in predictable monitoring.

III. A MODEL OF SYSTEM AND MONITOR EXECUTION

Our system model consists of a set of target tasks some or all of which are instrumented to generate PVEs, and a set of monitor tasks each of which is associated with an event queue. The PVEs impose processing demand, which is processed by the monitor tasks. The processing demand within an interval is the amount of CPU time needed to process the jobs released within this interval and the jobs that were released but unfinished before this interval began. Similarly, the processing demand at time t is the amount of CPU time needed to process the jobs released before or at t but unfinished at t. When no confusion will arise, we shall refer to processing demand as demand.

Each target task $\tau_i$ has a period $p_i$ and execution time $e_i$. In a fixed-priority system, each monitor task $m_i$ can be scheduled by a deferrable server $s^i$. The deferrable server $s^i$ has a replenishment period $p^i_{m_i}$, and capacity $e^i_{m_i}$ that constrains the maximum CPU time that can be used to process jobs in a replenishment period. As a deferrable server executes, its capacity decreases, and it is suspended when its capacity decreases to 0. At the beginning of the next replenishment period, its capacity is replenished to $e_{m_i}$. When higher priority tasks and servers are running, a lower priority server cannot use CPU time to process pending demand, even if it has a non-zero capacity. We denote as $t^i_p$ the maximum amount of time during which $s^i$ cannot use CPU time due to the execution of higher priorities tasks and servers. When no confusion will arise, we shall refer to a deferrable server as a server, and omit the superscript $i$ and simply use $s, p_m, e_m$ and $t_p$.

To guarantee that a server $s$ with an arbitrary priority can execute for $e_m$ time units in each replenishment period, it is necessary and sufficient to guarantee that, during each replenishment period, the total processing demand of higher priority tasks and servers are no more than $(p_m - e_m)$. By calculating the processing demand of all the higher priority tasks and servers within $p_m$ time units, we can bound $t_p$, as follows.

$$t_p \leq \sum_{\forall j \in S(s)} \left( \frac{p_m + J_j}{p^j_{m_j}} \right) e^j_{m_j} + \sum_{\forall j \in T(s)} \left( \frac{p_m}{p^j_{m_j}} \right) e^j_{m_j} \leq p_m - e_m \quad (1)$$

where $S(s)$ denotes the set of servers with priorities higher than $s$. $J_j$ is the release jitter of the server $s^j, p^j_{m_j} - e^j_{m_j}$ for a deferrable server and 0 for a sporadic server [1]. $T(s)$ denotes the set of target tasks with priorities higher than $s$.

Adding servers into the target system will change the original target system’s schedulability, and we assume that the original system is still schedulable after servers are added. Schedulability analysis on a fixed priority system with deferrable servers is presented in [11], [17].

In our system model, a server is dedicated to execute only one monitor task, and the PVEs processed by the same server have the same detection cost $c_{pve}$. If a server needs to process PVEs with different detection cost, $c_{pve}$ is set to be the maximum value of these PVEs detection costs. If the PVEs processed by the same server are generated by different target tasks, the release of these PVEs exhibits an aperiodic pattern [18].

In a practical system, the number of PVEs that could occur within an interval is usually bounded from above. We assume

1Sporadic servers can also be used to schedule monitor tasks. However, they have higher implementation complexity, and do not necessarily achieve significant performance improvement over deferrable servers [1]. In this paper we only present an analysis of deferrable servers, but the results hold whether or not sporadic servers exist in the system.
that, during any sliding window of \( w \) time units, the number of PVEs entering a server’s event queue is no more than \( N_{pve} \). Let \( N(t) \) be the number of PVEs entering a server’s event queue within the interval \([0, t]\) for \( t \geq 0 \), we have
\[
\forall t \in \mathbb{N}_0 : N(t + w) - N(t) \leq N_{pve}
\]
which bounds the demand imposed on the server within an interval of \( w \) time units. Given that all the PVEs in a server’s event queue are assumed to have same (maximum) detection cost of \( c_{pve} \), the demand within \( w \) time units is \( A_{pve} = N_{pve} \cdot c_{pve} \). Let \( A(t) \) be the demand imposed on the server within the interval \([0, t]\) for \( t \geq 0 \), we have
\[
\forall t \in \mathbb{N}_0 : A(t + w) - A(t) \leq A_{pve}
\]
The PVE workload can be specified by \((N_{pve}, c_{pve}, w)\) or \((A_{pve}, w)\). In a long-running system, the smallest share of CPU resource needed for processing all the PVEs is equal to \( \frac{A_{pve}}{w} \). To bound the MQL and the MDL from above in a long-running system, the server must have enough budget to process all the PVEs, that is, \( \frac{A_{pve}}{w} \leq \frac{e_m}{p_m} \leq 1 \).

Since the server is dedicated to process only one monitor task, it is generally undesirable to allocate more server budget than needed. Instead, a desirable approach is to give the server a budget equal to the smallest share of CPU resource needed for processing all the PVEs, which is formulated as
\[
\frac{A_{pve}}{w} = \frac{e_m}{p_m} \leq 1
\]
which is the case we consider in the rest of this paper.

In addition, we assume in this paper that the minimum time unit is 1, so \( A_{pve} \), \( w \), \( e_m \) and \( p_m \) are all integers. Furthermore, the first server replenishment occurs before or at the time when the first PVE is released. This assumption is feasible and desirable in practice, as delaying the first server replenishment does not decrease the MQL and the MDL, and may even increase the MQL and the MDL.

IV. THE MOST STRINGENT EXECUTION PATTERN

In this paper, we calculate the MQL and MDL of a monitor task in the case where demand arrives as quickly as possible, and the CPU time a server can use is provided as slowly as possible. This execution pattern, which we refer to as the the Most Stringent Execution Pattern (MSEP), is defined as follows

**Definition IV.1.** The MSEP is an execution pattern characterized by the following properties: For all \( t > r_1 > 0 \) within the interval \([r_1, t]\) where \( r_1 \) is the release time of the first PVE,

- the first server replenishment occurs before or at \( r_1 \), and
- the demand is no less than that in the same interval under any other execution pattern, and
- the CPU time a server can use is no more than that in the same interval under any other execution pattern.

Fig. 2 shows the MSEP of an arbitrary priority server and a highest priority server which is a special case of arbitrary priority servers. In Fig. 2, an empty arrowhead denotes a simultaneous release of \( N_{pve} \) PVEs, and a solid arrowhead denotes a server’s replenishment. Clearly, when all the \( N_{pve} \) PVEs are released simultaneously and are separated by \( w \) time units, they will impose the maximum demand on a long-running system.

For a lower priority server, \( t_p > 0 \). Because of the execution of higher priority tasks and servers, a PVE may not be processed immediately after being released, even if the server has non-zero capacity. In Fig. 2 (a), within \([0, p_m - t_p]\), no PVEs are released, and no higher priority tasks and servers are executing. At time \((p_m - t_p)\), the first \( N_{pve} \) PVEs are released. However, due to the \( t_p \) time units of execution of higher priority tasks and servers, the server cannot execute within \([p_m - t_p, p_m]\). For the same reason, the server cannot execute either until \((p_m + t_p)\). The longest interval during which there is no-zero demand but the server cannot execute is \( 2 \cdot t_p \). If, starting from the third server period, the server cannot execute during the first \( t_p \) units within each replenishment period, the CPU time the server can use will be no more than that in any other execution pattern within \([r_1, t]\), \( 0 < r_1 < t \), where \( r_1 \) is the release time of the first \( N_{pve} \) PVEs, and \( t \) is any instant later than \( r_1 \).

For a highest priority server, \( t_p = 0 \). By letting \( t_p = 0 \), we can derive from Fig. 2 (a) the MSEP for a highest priority server, as illustrated in Fig. 2 (b). Under this circumstance, the first release of \( N_{pve} \) PVEs occurs at time \( r_1 = p_m \). Since during the first server replenishment period \([0, p_m]\), no demand exists, and the server is idle throughout this interval, we can take \( r_1 \) as the time origin, and then obtain the MSEP of a highest priority server as shown in Fig. 2 (c). Note that
Fig. 2 (b) and (c) are different representations of a highest priority server’s MSEP, and whichever representation is used does not affect the results of MQL and MDL analysis. In later discussion, we shall use the representation in Fig. 2 (b). This way we can unify the analysis of lower priority servers and highest priority servers.

A. Casting the Problem Under MSEP

Under an MSEP, the monitor task can be treated as a sporadic task $\bar{T}$ with a period $w$ and execution time $A_{pve}$. The problem of finding the MDL of the PVEs under an MSEP is then cast into the problem of calculating the maximum response time of the task $\bar{T}$ scheduled by a deferrable server under an MSEP. With this mapping, each empty arrowhead at each top line in Fig. 2 (a) and (b) denotes the release of a job. In the rest of this paper, we use $r_k$ to denote the release time of the $k$-th, $k \in \mathbb{N}_1$, job. $r_k$ is given by

$$r_k = p_m - t_p + (k - 1) \cdot w$$ (5)

The MDL of a long-running system is equal to the maximum response time of $\bar{T}$. In the rest of this paper, the simultaneous release of $N_{pve}$ PVEs will be referred to as a job.

B. Queue Length and Pending Demand

The queue length at some time $t > 0$ is the number of PVEs currently in the server’s event queue, and bounds the demand at $t$. On the other hand, if we know the demand at $t$, we can obtain the queue length at $t$. We use $Q(k)$ to denote the demand at time $(r_k - \epsilon)$ where $r_k$ is defined by (5) and $\epsilon > 0$ is an infinitely small number. In addition, we use $\hat{Q}(k)$ to denote the demand at time $r_k$. That is, $\hat{Q}(k) = Q(k) + A_{pve}$ for all $k \in \mathbb{N}_1$.

Assuming that all the PVEs in the event queue have the same (maximum) detection cost of $c_{pve}$, the queue length at $r_k$, where $r_k$ is defined by (5), is given by $\lceil \frac{Q(k)}{c_{pve}} \rceil$. Therefore, the analysis of queue length can be done by analyzing $Q(k)$ or $\hat{Q}(k)$.

C. CPU Time and Demand under MSEP

A deferrable server is an eager server [5] in the sense that whenever demands arrive, the server will start processing them once it obtains the CPU\(^2\). The server’s capacity within a replenishment period decreases from $e_m$ as the server executes. If, at the end of a replenishment period, a server still has non-zero capacity, say, $e'$ units, the $e'$ units capacity will be discarded, and the server is replenished with $e_m$ units capacity. When this occurs, we say that the server loses $e'$ units of CPU time.

To simplify later discussion, we define the following functions.

**Definition IV.2.**

$$L(k) = (k - 1) \cdot w - 2 \cdot t_p$$ (6)

\(^2\)We assume in this paper that the only resource needed by a server is the CPU.

If $L(k) \leq 0$, the $k$-th job is released before or at $(p_m + t_p)$, and if $L(k) > 0$, the $k$-th job is released after $(p_m + t_p)$. This is illustrated in Fig. 2 (a). Intuitively, the absolute value of $L(k)$ gives the length of the interval between time $(p_m + t_p)$ and $r_k$ where $r_k$ is defined by (5).

**Definition IV.3.** $P$ is the number of jobs released in an interval of $LCM(p_m, w)$ units, where $LCM(p_m, w)$ denotes the least common multiple of $p_m$ and $w$. Under an MSEP,

$$P = \frac{LCM(p_m, w)}{w}$$ (7)

**Definition IV.4.** $LCT(k)$ is the lost CPU time before the $k$-th job is released.

Under an MSEP, if (4) holds, a server will not lose any CPU time after the server starts processing the first job, which is explained as follows. First, consider a highest priority server. A highest priority server can process demands immediately after they arrive. (4) implies that the server budget is equal to the smallest share of CPU resource needed for processing all the PVEs. Therefore, no CPU time will be lost by a highest priority server after it starts processing the first job.

For a lower priority server, after the first job is released, the server can use no more CPU time than the highest priority server does within an interval of the same length. That is, if an MSEP, to provide the same amount of CPU time, the time needed by a lower priority server is no less than that by a highest priority server. For a deferrable server, CPU time is lost at the end of a replenishment period only when demand arrives so late that the allocated CPU time cannot be completely consumed before the end of a replenishment period. Thus, with the same $p_m$, $e_m$ and job release pattern, if a highest priority server does not lose any CPU time after the server starts processing the first job under an MSEP, then a lower priority server will not lose any CPU time.

By the above argument, the only lost CPU time of a lower priority server under an MSEP is the $e_m$ units of CPU time before the server starts processing the first job. Therefore,

$$\forall k \in \mathbb{N}_1 : LCT(k) = e_m$$ (8)

Note that, when we use Fig. 2 (b) to represent a highest priority server’s MSEP, then (8) also applies to a highest priority server.

**Definition IV.5.** $ACT(k)$ is the available CPU time a server can provide within $[0, r_k)$ where $r_k$ is defined by (5).

To calculate $ACT(k)$ under an MSEP, we consider two cases: 1) $L(k) \leq 0$ and 2) $L(k) > 0$, where $L(k)$ is defined by (6). If $L(k) \leq 0$, the $k$-th job is released before or at $(p_m + t_p)$, as illustrated in Fig. 2 (a). Before time $(p_m + t_p)$, the server can use $e_m$ units of CPU time which is only available within $[0, p_m - t_p]$. If $L(k) > 0$, the server can also use $e_m$ units of CPU time within $[0, p_m + t_p]$. After time $(p_m + t_p)$, the server can use $e_m$ units of CPU time every $p_m$ time units, as illustrated in Fig. 3 (a) and

\(^3\)But the server does not use this $e_m$ units of CPU time under MSEP because no jobs are released within $[0, p_m - t_p]$. 


an infinitely small number, is $PD(k) - (ACT(k) - e_m)$. If $PD(k) \leq ACT(k) - e_m$, then $PD(k)$ units of demand has been finished before or at time $(r_k - \epsilon)$, so no demand is pending at time $(r_k - \epsilon)$, i.e., $Q(k) = 0$. In summary, $Q(k)$ is given by (11).

$$Q(k) = \max\{PD(k) - (ACT(k) - e_m), 0\} \quad (11)$$

$Q(k)$ has a periodic property formulated by Lemma IV.1. As we shall see in Section VI, Lemma IV.1 plays a key role in determining the MDL under an MSEP.

**Lemma IV.1.** When (4) holds,

$$\forall k, L(k) > 0 : Q(k + P) = Q(k) \quad (12)$$

where $L(k)$ is defined by (6).

**Proof:** By substitution, it can be verified that $\text{mod}(L(k), p_m) = \text{mod}(L(k), p_m)$ where $L(k)$ is defined by (6). For all $k$, $L(k) > 0$, by (4), (9), (10) and (11), it can be verified via substitution that $Q(k + P) = Q(k)$. This finishes the proof. ■

In the next section, the MQL analysis is presented by deriving the upper bound of $Q(k) = Q(k) + A_{pve}$.

**V. MQL Analysis**

The analysis of queue length can be done by analyzing $\hat{Q}(k)$, which is defined in Section IV-B and indicates the pending demand when the $k$-th job is released. Since after a job is released, the pending demand will not increase until the next job is released, the maximum pending demand during system execution can be obtained by calculating the upper bound of $\hat{Q}(k)$ for all $k \in \mathbb{N}_1$.

First consider a highest priority server. Based on the results in [18], each job will be finished before the next job is released when (4) holds, so no previous demand will be pending when a job is released, and thus the pending demand during system execution is no more than $A_{pve}$.

Now consider a lower priority server under MSEP as illustrated in Fig. 2. Within $[r_1, p_m + t_p)$ where $r_1$ is the release time of the first job, the pending demand is no more than $\lceil \frac{2t_p}{w} \rceil \cdot A_{pve}$. Starting from time $(p_m + t_p)$, during the first $(p_m - t_p)$ time units of each interval with a length of $p_m$, the server does not suffer preemption, and can use $e_m$ units of CPU time every $p_m$ time units, so it behaves like a highest priority server whose first replenishment occurs at time $(p_m + t_p)$, as illustrated in Fig. 3. If there is no queued demand before $(p_m + t_p)$, then each job released after $(p_m + t_p)$ can be finished before the next job is released. Under this assumption, the pending demand is no more than $A_{pve}$. Taking into account the queued demand before $(p_m + t_p)$, the pending demand in the system is no more than $\lceil \frac{2t_p}{w} \rceil \cdot A_{pve} + A_{pve}$. A formal analysis follows.

**Lemma V.1.** When (4) holds,

$$\forall k \in \mathbb{N}_1 : \hat{Q}(k) \leq \frac{2t_p}{w} \cdot A_{pve} + A_{pve} \quad (13)$$

**Proof:** Consider the following 3 cases.
• If \( L(k) \leq 0 \), by (6), (9), (10) and (11),
\[
Q(k) = (k - 1) \cdot A_{pve} \leq \frac{2 \cdot t_p}{w} \cdot A_{pve}
\]

• If \( L(k) > 0 \) and \( \text{mod} (L(k), p_m) \geq e_m \), by (4), (6), (9), (10) and (11),
\[
Q(k) = (k - 1) \cdot A_{pve} - \left( \left\lceil \frac{L(k)}{p_m} \right\rceil + 1 \right) \cdot e_m
\]
\[
(\leq (k - 1) \cdot A_{pve} - \frac{L(k)}{p_m} \cdot e_m = \frac{2 \cdot t_p}{w} \cdot A_{pve}
\]

• If \( L(k) > 0 \) and \( \text{mod} (L(k), p_m) < e_m \), by (4), (6), (9), (10) and (11),
\[
Q(k) = (k - 1) \cdot A_{pve} + (p_m - e_m) \cdot \left( \left\lceil \frac{L(k)}{p_m} \right\rceil - L(k) \right)
\]
\[
\leq (k - 1) \cdot A_{pve} + (p_m - e_m) \cdot \left( \frac{L(k)}{p_m} - L(k) \right)
\]
\[
= \frac{2 \cdot t_p}{w} \cdot A_{pve}
\]

Since \( Q(k) = Q(k) + A_{pve}, \) \( \hat{Q}(k) \leq \frac{2 \cdot t_p}{w} \cdot A_{pve} + A_{pve} \).

**Theorem V.2.** Assuming all the PVEs have the same detection cost of \( c_{pve} \), the MQL under an MSEP is given as follows.
\[
MQL = \left\lceil \frac{2 \cdot t_p}{w} \cdot N_{pve} \right\rceil + N_{pve} \quad (14)
\]

**Proof:** After the \( k \)-th job is released at \( r_k \), the pending demand will not increase before \( (r_k + w) \), the instant when the next job is released, so \( \hat{Q}(k) \), the pending demand at time \( r_k \), is the maximum pending demand within \( [r_k, r_k + w) \). The pending demand is \( \hat{Q}(k) \), so the queue length is given by \( \left\lceil \frac{\hat{Q}(k)}{e_{pve}} \right\rceil \), and the MQL is given by \( \left\lceil \frac{\max(\hat{Q}(k))}{e_{pve}} \right\rceil \). By Lemma V.1, the following holds.
\[
MQL = \left\lceil \frac{2 \cdot t_p}{w} \cdot A_{pve} + A_{pve} \right\rceil = \left\lceil \frac{2 \cdot t_p}{w} \cdot N_{pve} \right\rceil + N_{pve} \quad (15)
\]

This finishes the proof.

By letting \( t_p = 0 \), we can apply the above result to a highest priority server. Under this case, \( MQL = N_{pve} \).

**VI. MDL Analysis**

When the jobs are processed in FIFO order, the response time of the \( k \)-th job with release time \( r_k \) is the time needed to process the pending demand at \( r_k \), which is given by \( \hat{Q}(k) \). The response time analysis presented here does not rely on the relative values of \( p_m \) and \( w \), i.e., \( p_m > w \) or \( p_m \leq w \), neither does it rely on the relative values of a task’s deadline and period. This is distinguished from existing work, e.g., [5], [6], [15], that rely on some of these assumptions.

Consider two cases: 1) \( L(k) \leq 0 \) and 2) \( L(k) > 0 \).

Case 1: \( L(k) \leq 0 \). As shown in Fig. 4, the \( k \)-th job is released within \([p_m - t_p, p_m + t_p]\), wherein no CPU time can be used by the server. When the \( k \)-th job is released, \( \hat{Q}(k) = k \cdot A_{pve} \); the demand must wait for \( p_m + t_p - r_k = 2 \cdot t_p - (k - 1) \cdot w \) time units until it starts to be processed. After \((p_m + t_p)\), the server can use \( e_m \) units of CPU time every \( p_m \) time units. To finish \( Q(k) \) units of demand, it takes \( \left\lceil \frac{Q(k)}{e_m} \right\rceil - 1 \) complete intervals each of which has a length of \( p_m \), and after these complete intervals, the time to process the residual demand is \( \left\lceil \frac{Q(k)}{e_m} \right\rceil - 1 \cdot e_m \). The total time to process \( Q(k) \), i.e., response time of the \( k \)-th job, is then given by (16).
\[
2 \cdot t_p - (k - 1) \cdot w + \left( \left\lceil \frac{Q(k)}{e_m} \right\rceil - 1 \right) \cdot p_m + \hat{Q}(k) - \left( \left\lfloor \frac{Q(k)}{e_m} \right\rfloor \right) e_m
\]
\[
L(k) \leq 0 \) implies \( k \leq \frac{2 \cdot t_p}{w} + 1 \), and since \( k \in \mathbb{N}_1 \), by evaluating (16) for all \( k \in \mathbb{N}_1 \), \( 1 \leq k \leq \frac{2 \cdot t_p}{w} + 1 \), we can find the maximum value of (16).

Case 2: \( L(k) > 0 \). We define the following functions.
\( f(k) \) is the length of the interval \( \left[ \left\lfloor \frac{\hat{Q}(k)}{p_m} \right\rfloor \cdot p_m, \left\lceil \frac{\hat{Q}(k)}{e_m} \right\rceil \cdot p_m \right] \). \( f_m \) is the instant when the last server replenishment before time \( r_k \) occurs, and \( r_k \) is defined by (5). \( \Phi(k) \) is given by
\[
\Phi(k) = r_k - \left( \frac{r_k}{p_m} \right) \cdot p_m
\]
If \( t_p \geq \Phi(k) \), then when the \( k \)-th job is released the server cannot execute due to the execution of higher priority tasks and servers, and the server must wait for \( (t_p - \Phi(k)) \) time units to start execution. If \( t_p < \Phi(k) \), the server will not suffer preemption until the current replenishment period ends. Therefore, when the \( k \)-th job is released, the server may decide to execute for execution due to the execution of higher priority tasks and servers is given by \( \max(t_p - \Phi(k), 0) \).

\( e_r(k) \) denotes, when the \( k \)-th job is released, the CPU time a server can use within the current replenishment period. To calculate \( e_r(k) \), we consider two cases: 1) \( t_p \geq \Phi(k) \) and 2) \( t_p < \Phi(k) \). If \( t_p \geq \Phi(k) \), the server cannot execute due to the execution of higher priority tasks and servers, so no CPU time replenished in the current replenishment period has been consumed, and thus the remaining CPU time the server can use within the current replenishment period is \( e_m \). If \( t_p < \Phi(k) \), suppose the \( k \)-th job is released in the \( j \)-th replenishment period, the total available CPU time the server can use within the first \( j \)-th complete replenishment period is \( \left( \frac{r_j}{p_m} \right) \cdot e_m \), which includes the unused CPU time of \( e_m \) in the first period. There are two sub-cases of the case \( t_p < \Phi(k) \): 1) \( Q(k) > 0 \) and 2) \( Q(k) = 0 \). If \( Q(k) > 0 \), the demand by the first \((k - 1)\)-th jobs have not been completely processed at time \( r_k \), which is defined by (5). By the discussion in Section IV-C, the server loses \( e_m \) units of CPU time before it starts processing the first
job, and never loses its CPU time after it starts processing the first job, so \((ACT(k) - e_m)\) out of the \(ACT(k)\) units of CPU time is used to process the jobs. Therefore \(e_r(k)\) under this case is \(\left[\frac{\Delta}{p_m}\right] \cdot e_m - e_m - (ACT(k) - e_m) = \left[\frac{\Delta}{p_m}\right] \cdot e_m - ACT(k)\). If \(Q(k) = 0\), the demand has been finished before or at time \(r_k\). The finished demand, \(PD(k)\), is equal to the used CPU time. By subtracting the amount, \(e_m\), of the lost CPU time and the amount, \(PD(k)\), of the used CPU time from \(\left[\frac{\Delta}{p_m}\right] \cdot e_m\), we have \(e_r(k)\) for this case: \(\left[\frac{\Delta}{p_m}\right] \cdot e_m - e_m - PD(k)\).

To sum up, \(e_r(k)\) is given by (18).

\[
e_r(k) = \begin{cases} 
    e_m, & \text{if } t_p \geq \Phi(k) \\
    \left[\frac{\Delta}{p_m}\right] \cdot e_m - ACT(k), & \text{if } t_p < \Phi(k) \text{ and } Q(k) > 0 \\
    \left[\frac{\Delta}{p_m}\right] \cdot e_m, & \text{if } t_p < \Phi(k) \text{ and } Q(k) = 0 \\
    -e_m - PD(k), & \text{if } t_p < \Phi(k) \text{ and } Q(k) < 0 
\end{cases}
\]  

(18)

The response time, \(R(k)\), is then calculated as follows. Consider two cases: 1) \(e_r(k) \geq \hat{Q}(k)\) and 2) \(e_r(k) < \hat{Q}(k)\). If \(e_r(k) \geq \hat{Q}(k)\), \(Q(k)\) units of demand can be processed within the current period, but before the server starts execution, it must wait for \(\max(t_p - \Phi(k), 0)\) time units. If \(e_r(k) < \hat{Q}(k)\), the response time of the \(k\)-th job consists of 3 components. The first component is the waiting time for the next replenishment, which is given by \((p_m - \Phi(k))\). The second component is the \((\left[\frac{\Delta}{p_m}\right] - 1)\) complete replenishment periods. The third component is the execution time for the residual demand in the last replenishment period, plus \(t_p\), the time during which the server cannot execute due to higher priority tasks and servers.

To sum up, \(R(k)\) is given by (19) where \(QE(k) = \hat{Q}(k) - e_r(k)\).

\[
R(k) = \begin{cases} 
    \max(t_p - \Phi(k), 0) + \hat{Q}(k), & \text{if } e_r(k) \geq \hat{Q}(k) \\
    p_m - \Phi(k) + \left((\frac{\Delta}{p_m}) - 1\right) \cdot p_m + QE(k), & \text{otherwise} \\
    \left((\frac{\Delta}{p_m}) - 1\right) \cdot e_m + t_p, & \text{if } e_r(k) < \hat{Q}(k) 
\end{cases}
\]  

(19)

Theorem VI.1. For all \(k\), \(L(k) > 0\), \(R(k + P) = R(k)\) where \(L(k)\) is defined by (6) and \(P\) is defined by (7).

Proof: It can be verified that \(\Phi(k)\) and \(e_r(k)\) are all periodic functions with the same period \(P\). In addition, all these functions have the same domain determined by \(L(k) > 0\). Thus, the condition that determines the form of \(R(k)\), i.e., \(e_r(k) \geq \hat{Q}_k\), is also periodic with a period of \(P\). By Lemma IV.1, \(\hat{Q}(k) = \hat{Q}(k) + A_{pve}\) also has a period of \(P\) for all \(k\), \(L(k) > 0\). By substitution, it can also be verified that \(R(k)\) has a period of \(P\).

\(L(k) > 0\) implies \(k > 2t_p - \frac{w}{2} + 1\). By Theorem VI.1, to find the maximum value of (19), it suffices to evaluate (19) for all \(k \in \mathbb{N}_1: \frac{2t_p}{w} + 1 \leq k \leq \frac{2t_p}{w} + P\).

Combining the results in the above two cases, the MDL is then given by (20).

\[
MDL = \max\{m1, m2\}
\]  

(20)

where \(m1\) is the maximum value of (16), and \(m2\) is the maximum value of (19).

By letting \(t_p = 0\), (20) can be also applied to highest priority servers, and this produces the same results as the analysis in [18] does. Therefore, (20) generalizes the result in [18] to arbitrary priority servers.

VII. EVALUATION AND RECOMMENDATIONS

This section evaluates the variation in MQL and MDL with changing server parameters. Based on that evaluation, we recommend guidelines for engineers to use in designing predictable runtime monitoring using a deferrable server. Specifically, we answer the questions put forward in Section I.

A. Evaluation Results

We have run multiple evaluations by varying server parameters and PVE workloads. In all cases, we found that the plots exhibit nearly identical shapes, so in our presentation we focus on one evaluation whose results are shown in Fig. 5 and Fig. 6. In this evaluation, the server utilization \((\frac{\Delta}{p_m})\) is 0.2, and \(p_m\) increases from 75 to 350 with a step of 5. For each value of \(p_m\), \(t_p\) increases from 0 to \((p_m - e_m)\). The PVE workload is \(w = 100\) and \(A_{pve} = 20\).

Both the MQL and MDL have a common overall shape. A region beginning above the line where \(t_p = p_m - e_m\) and sloping downward to the \(p_m\) axis where \(t_p = 0\). The plots show a zero value elsewhere, when \(t_p \geq p_m - e_m\), but in reality those values are undefined due to the requirements of (1).

Fig. 5 shows how the MQL changes as \(p_m\) and \(t_p\) change. For the same \(t_p\), the MQL does not change as the value of \(p_m\) changes along the sloping surface. Indeed, by (14), the MQL only depends on \(t_p\), \(A_{pve}\) and \(w\) when (4) holds. Note that, although \(p_m\) does not determine the MQL directly, it may affect the value of \(t_p\) and thus the MQL indirectly.

Fig. 6 shows how the MDL changes as the values of \(p_m\) and \(t_p\) change. First, with a fixed value of \(t_p\), the minimum MDL is achieved when \(p_m\) is a multiple of \(w\). The same observation for a highest priority server is presented in [18], and here we generalize it to an arbitrary priority server. This finding suggests that, if the server’s priority is fixed in the system (and thus \(t_p\) is fixed), \(p_m\) should be selected as a multiple of \(w\).

As we expected, for the same value of \(p_m\), the MDL increases as \(t_p\) increases, which is shown in Fig. 6.

B. Recommendations

In Section I we put forward several fundamental questions to be considered when designing predictable runtime monitoring with a deferrable server. Those questions are explored below. These answers also suggest desirable guidelines for engineers to implement predictable runtime monitoring with a deferrable server.

What execution budget \((e_m, p_m)\) should be assigned to each monitor?
A and \( p \) is a multiple of \( p \) and \( t \) and \( w \). A. \( p \) is a divisor of \( w \) and the capacity by (1). Then we calculate the \( m \) to be equal to a large capacity, and reduces the probability of periodic interference caused by the server replenishment and thus reduces the average response time of aperiodic jobs. However, In predictable runtime monitoring where the MDL is of major interest, \( p_m \) should be chosen to be a multiple of \( w \) to minimize the MDL.

This conclusion is also different from a recent study in [6] where the authors suggest the server replenishment period \( p_m \) be selected to be a divisor of the task period \( w \). This is correct when \( p_m \leq w \), and our study confirms this conclusion. To better illustrate this we use another set of server parameters and PVE workload, and create a plot in Fig. 7. In Fig. 7, the server budget is still 0.2, but \( p_m \) increase from 60 to 1000 with a step of 5, and \( t_p \) is arbitrarily set to be 30, \( w \) is set to be 480. As illustrated in Fig. 7, the MDL is locally minimized when \( p_m \) is a divisor of \( w \), e.g., when \( p_m = 60, 80, 120, \) or 160 in Fig. 7. If, however, \( p_m \) can be no smaller than \( w \), and it can be increased in a way that \( t_p \) will remain the same value, the MDL will be globally minimized when \( p_m \) is a multiple of \( w \), e.g., \( p_m = 480 \) in Fig. 7.

With the same server utilization, a large \( p_m \) affects the whole system’s schedulability more than a small value does. In this sense, setting \( p_m \) to be equal to \( w \) is a desirable strategy.

**Given a specified priority and server execution budget what is the bound on the latency and storage requirements for detecting violations?**

First, we need to calculate \( t_p \) by (1). Then we calculate the bound on latency by (20) and the storage requirement by (15), assuming each PVE has the same known storage requirement.

**Given a specified latency for detecting a violation, what priority should be given to the monitor?**

This priority can be found as follows. Starting from the lowest priority in the system with an increase in the priority of 1, test whether the MDL calculated by (20) is longer than the specified latency under the current priority. If yes, then repeat this process until the highest priority is reached. If no, then the lowest priority that produces a MDL no longer than the specified latency should be assigned to the monitor.

**Given a specified amount of space for storing events, what priority should be given to the monitor?**

Given a specified amount of space for storing events, we can first calculate the largest possible value of \( t_p \), say, \( t_p^\text{m} \), by (15). Then by (1) we can search in the system for the lowest priority that will not lead to a value of \( t_p \) greater than \( t_p^\text{m} \). If, by assigning this lowest priority to the monitor, the whole system is still schedulable, then the monitor can be assigned this priority. Otherwise, the system must be redesigned, e.g., by increasing the available space for storing events.

**VIII. RELATED WORK**

Multiple researchers have explored the problem of runtime monitoring. In the context of real-time systems, Chodrow et al. implemented a run-time environment based on Real-Time Logic [8] for specifying and monitoring real-time system [4]. Mok and Liu extended Chodrow et al.’s work by giving an
algorithm for violation detection and developing a Java runtime timing constraint monitor [12], [13].

One of the first general runtime monitoring frameworks was Java-MaC [9] which is based on the Monitoring and Checking (MaC) architecture [10]. MaC provides a two-level event specification language that permits the separation of programming language dependent and programming language independent portions of the specification [10]. In more recent work Rosu and colleagues have developed a more sophisticated and efficient runtime monitoring framework termed monitor oriented programming [3]. Their approach supports property specifications in a number of languages including: regular expressions, context free grammars, and temporal logics. It has been implemented for the Java language [2] and, with hardware assistance, to monitor bus traffic [14]. We believe that these approaches when applied to real-time systems can and should, but currently do not, incorporate support for predictable monitoring.

For sporadic tasks with deadlines no longer than periods, Saewong et al [15] provide response time analysis for sporadic tasks scheduled by deferrable or sporadic servers. Davis and Burns [6] improved the earlier work by providing tighter analysis on deferrable servers. It is pointed out by Cuijpers and Bril [5] that the analysis in [6] is pessimistic rather than exact if the server’s resources will not be consumed by a soft real-time lowest priority task (which is the case considered in our system), or if more information on this soft real-time lowest priority task is known. For sporadic tasks with deadlines longer than periods, Cuijpers and Bril [5] show the upper bound of the response time of a special class of sporadic tasks. To apply their approach, the tasks and the server must satisfy some conditions (See the conditions of Theorem 1 in [5]) that do not hold under an MSEP. To sum up, all these existing techniques either cannot be applied in our system settings, or suffers pessimistic results.

IX. CONCLUSIONS

In this paper, we provide a general framework for program monitoring that permits an engineer to select server parameters that minimize schedulability issues, while controlling the time and space resources required to achieve predictable monitoring. We have shown that this framework generalizes the special case of a highest-priority server monitor from [18]. We presented an analytic framework for calculating the MQL and MDL for a given monitoring problem and then explored the tradeoffs among key parameters of the problem. This led us to recommend a set of design guidelines for predictable monitoring that can be applied by engineers when designing their system.

In future work, we plan to exploit our multi-priority server approach by clustering properties with similar MQL and MDL and assigning them to a common monitor server task. By using separate cluster servers, we have the opportunity to then exploit multiple execution cores to reduce monitoring processing overhead. We are also exploring several implementation frameworks that will allow predictable monitoring solutions to be deployed in robotics and sensor network applications.

REFERENCES