A Workload-oriented Programming Model for Temporal Isolation with VBS*

Silviu S. Craciunas, Christoph M. Kirsch, and Ana Sokolova

Department of Computer Sciences
University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. Workload-oriented programming is a design methodology for specifying throughput and latency of real-time software processes in temporal isolation from each other on the level of individual process actions such as system or procedure calls. The key programming abstraction is that the type and amount of workload involved in executing a process action fully determines the action's response time, independently of any previous or concurrent actions. The model enables sequential and concurrent real-time process composition while maintaining predictability of each action's workload-determined real-time behavior. We show how the model can be implemented using variable-bandwidth servers (VBS) and discuss performance-related options for adequately configuring servers in the presence of non-zero scheduler overhead.

1 Introduction

Real-time software processes typically process data under known or estimated temporal application requirements and resource constraints. Application requirements are, for example, the usually throughput-oriented rates at which video frames in an MPEG encoder must be processed, or the mostly latency-oriented rates at which sensor data in a control system must be handled. Resource constraints might be the maximum rate at which data can be analyzed or written to a harddisk. Both application requirements and resource constraints in turn can often directly be related to the workload, in particular, the type [1] and amount of the involved data. For example, a real-time process that compresses video frames usually needs to process a given number of frames within some finite response time. Similarly, resource constraints may be characterized by the execution time it actually takes to process a given number of frames. Workload-oriented programming is a design methodology for expressing such workload-determined temporal application requirements and resource constraints (Section 2). Variable-bandwidth servers (VBS) [2] can be used to implement workload-oriented programming and, as a result, provide temporal predictability (Section 3), but require non-trivial server configuration for proper performance in the presence of non-zero scheduler overhead (Section 4).

* Supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the Austrian Science Fund No. P18913-N15 and V00125.

2 Programming Model

For an example of workload-oriented programming consider Figure 1, which shows the pseudo-code implementation of a real-time software process P. The process reads a video stream from a network connection, compresses it, and finally stores it on disk, all in real time. The `determine_rate` and `compress_data` procedures implement process functionality and are therefore referred to as process code. The other procedures are system code. In our process model, we call an invocation of process or system code an action of the invoking process. An action has an optional workload parameter. In the example, P has five actions with the same workload parameter, which specifies the number of frames to be handled. The parameter may be omitted in actions that only involve process code if the action's time complexity is constant or unknown. The `determine_rate` action does not have a workload parameter because the action always executes in constant time. For unknown complexity, an omitted workload parameter implicitly represents CPU time, see below for more details. For each action, there are two discrete functions, f_R and f_E, which characterize the action's performance in terms of its workload parameter. Figure 2 shows example functions for the `allocate_memory` action.

```
loop {
  int number_of_frames = determine_rate();

  allocate_memory(number_of_frames);
  read_from_network(number_of_frames);

  compress_data(number_of_frames);

  write_to_disk(number_of_frames);
  deallocate_memory(number_of_frames);
} until (done);
```

Fig. 1. A real-time software process P

The response-time (RT) function $f_R : R_D \to \mathbb{Q}^+$ characterizes the action's response time bound for a given workload, independently of any previous or concurrent actions. The domain R_D of the RT-function is the set of workloads, which is any linearly ordered set [1]. In our example, f_R is a linear function with $R_D = \mathbb{N}$, which shows that allocating memory, e.g., for 24 frames, may take at most 100ms, even when interrupted by other concurrent processes. RT-functions map any workload to a non-zero, positive bound of which the smallest is the action's intrinsic response delay d_R. In the example, $d_R = 4$ms.

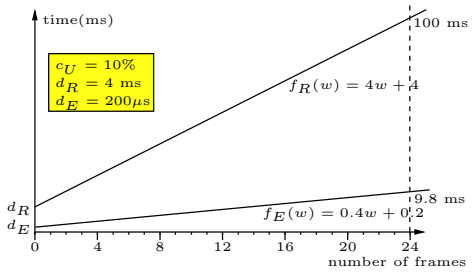

Fig. 2. Timing of the `allocate_memory` action

RT-functions help trading off throughput and latency. For example, if memory is allocated latency-oriented, say, just for a single frame, the response time is only at most 8ms but the resulting allocation rate merely provides enough memory for 125 frames per second (fps) if the action were invoked repeatedly as fast as possible. If memory is allocated throughput-oriented, say, for 24 frames at once, there could be enough memory for at least 240fps because d_R plays a smaller role, but the action's response time would also increase to 100ms in the worst case. For non-branching processes, one can derive an RT-function for the whole process, whereas RT-functions of processes with branching control flow may in general just be approximated, or described more accurately but in languages richer than plain arithmetics, c.f. [3].

An RT-function provides an upper bound on an action's response time. In order to trade off responsiveness for determinacy, one can also view the bound as both upper and lower bound, similar to the notion of logical execution time (LET) [4]. In this case, not considered here, one gets a logical-response-time (LRT) function.

The execution-time (ET) function $f_E : E_D \to \mathbb{Q}^+$ characterizes the action's execution time bound for workloads in the action's execution domain $E_D \subseteq R_D$, in the absence of any concurrent actions. In the example, f_E is also a linear function with $E_D = \mathbb{N}$, which states that allocating memory, say, again for 24 frames, may take up to 9.8ms if not interrupted by any other process. Similar to RT-functions, ET-functions map any workload to a non-zero, positive bound of which the smallest is the action's intrinsic execution delay d_E. In the example, $d_E = 200\mu$s since $f_E(w) \geq 200\mu$s for all $w \in E_D$, which means that allocating memory may take at least 200μs on any workload, if not interrupted.

The notion of worst-case execution time (WCET) can be seen as a special case of ET-functions that map to a constant execution time bound for all workloads. In turn, this means that determining ET-functions requires parametric forms of WCET analysis, e.g., as in [5]. Moreover, not all actions may be ET-characterized, i.e., compositional, on large execution domains. For example, the temporal behavior of write accesses to harddisks is known to be unpredictable just in terms of the workload. However, even such actions may be properly ET-characterized by limiting workloads to smaller execution domains.

The ratio between f_E and f_R induces a discrete (partial) utilization function $f_U : E_D \to \mathbb{Q}_0^+$ with:

$$f_U(w) = \frac{f_E(w) - d_E}{f_R(w) - d_R}$$

assuming there is zero administrative overhead for handling concurrency. In the example, f_U is a function that maps any workload $w \in \mathbb{N}^+$ to a constant $c_U = 0.1$ or 10% CPU utilization when allocating memory. In general, only workloads $w \in E_D$ with $0 \leq f_U(w) \leq 1$ (and ratios $0 \leq d_E/d_R \leq 1$) may be handled properly. We call the set of such workloads the action's utilization domain $U_D \subseteq E_D$. Even if $f_U(w)$ is not constant for all $w \in U_D$, there is still a minimal upper bound c_U such that $f_U(w) \leq c_U$ for all $w \in U_D$. This allows for a conservative but fast c_U-based schedulability test.

Recall that workload parameters are omitted in actions that only involve process code but have unknown time complexity. In this case, RT-functions directly determine the resulting CPU utilization. For example, consider a process that invokes process code with an unknown execution time. Then, the RT-function translates CPU time into real time by stating that, e.g., 10ms CPU time may take up to 100ms real time. The result is 10% CPU utilization since the ET-function is simply the identity function from CPU time to real time.

3 Variable-Bandwidth Server for Predictability

Figure 3 shows possible executions of the `allocate_memory` action on workloads of up to four frames. Consider the action when invoked on four frames where the response time is 20ms. The scheduler could in principle immediately release the action with a 20ms-deadline and apply earliest-deadline-first (EDF) scheduling. This strategy would, however, involve schedulability tests that depend on action invocation times and thus require analyzing process implementations and interactions. To avoid this and ensure programmable temporal isolation we use the concept of Variable-Bandwidth Servers (VBS) [2] for process scheduling. With VBS there is one server per process controlling the execution of process actions. Each action is assigned a virtual periodic resource R [6], which is a pair of a period π and a limit λ: an action using R will be scheduled for at most λ time units each π time units. The resource bandwidth is the ratio of limit over period. A VBS can change resources from one action to another under the restriction that the bandwidth of any used resource stays below a given per-server bandwidth cap. In the example, the resource used by the `allocate_memory` action has a period of 2ms and a limit of 200μs. Upon arrival, the action is delayed until the beginning of the next period, unless the arrival time coincides with a period instance. All released actions are then EDF-scheduled using their resources' periods as deadlines and their resources' limits as durations. When an action has exhausted the limit, it is delayed until the beginning of the next period when it is released again, and so on, until the action completes. In the example, the action executes for nine periods after being delayed in the first period and completes in the tenth period. Finally, the completed action must be terminated by the scheduler, which happens at the end of the period in which the action completed, i.e., the tenth period in the example. In total, the action is scheduled for up to 1.8ms, which is exactly its execution time bound for four frames.

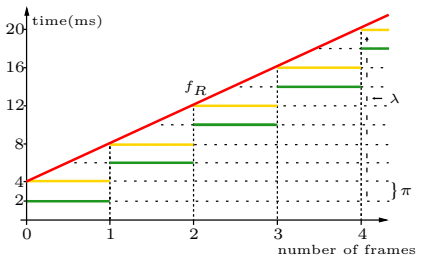

Fig. 3. The execution of `allocate_memory`

The duration from the action's arrival until its termination for all workloads is given by the scheduled-response-time (SRT) function f_S. For all $w \in E_D$, we have that:

$$\pi \cdot \left\lceil \frac{f_E(w)}{\lambda} \right\rceil \leq f_S(w) \leq \pi - 1 + \pi \cdot \left\lceil \frac{f_E(w)}{\lambda} \right\rceil \quad \text{if} \quad \sum_P \max_R \frac{\lambda}{\pi} \leq 1,$$

i.e., if the system utilization through the most-utilized resources R of each process P is less than or equal to 100% [7]. The upper bound (yellow step function in Figure 3) occurs if the action arrives one time instance after a period has begun. The lower bound for f_S (green step function in Figure 3) occurs if the involved action arrives exactly at the beginning of a new period.

4 Server Design for Performance

The server design problem is the problem of finding the right π for a given action. Given a value for π we then set $\lambda = \pi \cdot c_U$. The goal is that the scheduled response time f_S approximates the specified response time f_R best.

We have that $f_S(w) \leq f_R(w) + \pi$ if π divides d_R evenly, i.e., $\pi \mid d_R$, and

$$0 < \pi \leq d_R - \frac{d_E}{c_U}$$

This is true even if π does not divide d_R evenly, but only for π less than or equal to half of the upper bound. The constraint is a sufficient condition, which ensures that at least the d_E portion of an action's invocation will be completed within d_R time even when the first period of the invocation is not used. In our example, the upper bound is 2ms, which would come down to, say, 1ms if d_E were increased to 300μs. Note that, with d_E approaching 400μs, i.e., 10% utilization, π would have to become zero because of the potentially unused first period. In order to have that $f_S(w) \leq f_R(w)$ actually holds, π also needs to divide the remaining response time $f_R(w) - d_R$ evenly, i.e., $\pi \mid (f_R(w) - d_R)$ or equivalently $\lambda \mid (f_E(w) - d_E)$, which is true in the example. In general, checking this constraint may be difficult, in particular, on unbounded utilization domains, but is easy in case f_R or f_E are linear functions with $R_D = \mathbb{N}$. For example, if $f_R(w) = a_R \cdot w + d_R$, then $\pi \mid (f_R(w) - d_R)$ for all $w \in \mathbb{N}$ if and only if $\pi \mid a_R$.

No scheduler overhead. If we assume zero scheduler overhead, the best is to choose the smallest π possible, as this samples the specified response time best (f_S approximates f_R best). Moreover, as can be seen from the bounds, the scheduled-response-time jitter is $\pi - 1$, so smaller π mean less response-time jitter.

With scheduler overhead. In a real system with non-zero scheduler overhead, smaller π lead to more scheduler overhead. However, one can either account for the overhead in increased CPU utilization maintaining the scheduled response

time of actions (utilization accounting), or one can maintain CPU utilization by accounting for the overhead in increased response times of the actions (response accounting), or finally one can also combine both [8]. In all cases, it is important to estimate the number of scheduler invocations. For an action α with a virtual periodic resource (λ, π), the number of scheduler invocations in every period is bounded by $\lceil \pi / \gcd(\Pi) \rceil + 1$ where the Π denotes the set of all periods of all actions that may execute in parallel with α. Hence, the scheduler overhead is not only dependent on the period of an action but also on (almost) all other periods in the system, more specifically on the gcd of (almost) all other periods in the system. Thus, when choosing a period for an action we must consider the trade-off between response time approximation and scheduler overhead. If the CPU utilization leaves enough room for the scheduler overhead to be accounted for in increased CPU utilization, we can choose π for each action in such a way that the scheduled response time best approximates the specified response time. Otherwise, the scheduled response time of each process will increase as a result of response accounting. Intuitively, it is desirable to have large harmonic periods in the system which are small enough to approximate the specified response time.

A way out may be a higher-level scheduler that diverges from standard VBS scheduling (with a fixed resource per action). The higher-level scheduler would select a small period for the first part of an action (because the first period is waiting time) and a large period for the remaining part. In the example given in Figure 3, the first period would remain 2ms but then, depending on the workload, the remaining periods would be lumped together into one large period. This both maintains the scheduled response time of the action and may decrease the number of scheduler invocations, however, at the expense of potentially increased response-time jitter.

References

1. Chakraborty, S., Kirsch, C. Generalized from [7] inspired by private communication (2009)
2. Craciunas, S., Kirsch, C., Payer, H., Röck, H., Sokolova, A.: Programmable temporal isolation through variable-bandwidth servers. In: Proc. SIES. (2009)
3. Chakraborty, S., Thiele, L.: A new task model for streaming applications and its schedulability analysis. In: Proc. DATE. (2005)
4. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: A time-triggered language for embedded programming. Proc. of the IEEE **91**(1) (2003) 84–99
5. Bernat, G., Burns, A.: An approach to symbolic worst-case execution time analysis. In: Proc. 25th Workshop on Real-Time Programming. (2000)
6. Shin, I., Lee, I.: Periodic resource model for compositional real-time guarantees. In: Proc. RTSS. (2003)
7. Craciunas, S., Kirsch, C., Röck, H., Sokolova, A.: Real-time scheduling for workload-oriented programming. Technical Report 2008-02, University of Salzburg (2008)
8. Craciunas, S., Kirsch, C., Sokolova, A.: Response time versus utilization in scheduler overhead accounting. Technical Report 2009-03, University of Salzburg (2009)