Predicting the Behavior of a Highly Configurable Component Based Real-Time System

Scott A. Hissam[1], Gabriel A. Moreno[1], Daniel Plakosh[1],
Isak Savo[2], Marcin Stelmarczyk[2]

[1] *Software Engineering Institute (SEI)*
Carnegie Mellon University
4500 Fifth Avenue, Pittsburgh, PA 15213-2612
{shissam, gmoreno, dplakosh} @sei.cmu.edu

[2] *ABB Corporate Research, Forskargränd 7, SE-721 78 Västerås, Sweden*
{isak.savo, marcin.stelmarczyk}@se.abb.com

Abstract

Software components and the technology supporting component based software engineering contribute greatly to the rapid development and configuration of systems for a variety of application domains. Such domains go beyond desktop office applications and information systems supporting E-Commerce, but include systems having real-time performance requirements and critical functionality. Discussed in this paper are the results from an experiment that demonstrates the ability to predict deadline satisfaction of threads in a real-time system where the functionality performed is based on the configuration of the assembled software components. Presented is the method used to abstract the large, legacy code base of the system software and the application software components in the system; the model of those abstractions based on available architecture documentation and empirically-based, runtime observations; and the analysis of the predictions which yielded objective confidence in the observations and model created which formed the underlying basis for the predictions.

1. Introduction

In this paper, we discuss an experiment that was performed on a commercial device that is a PowerPC based embedded system running the VxWorks real-time operating system with many built-in monitoring and control interfaces. The purpose of this device is to perform real-time monitoring and control of equipment connected to those interfaces which are used in a variety of industrial settings. The device reacts to specific conditions detected through its monitoring interfaces based on settings configured through a unique combination of application software components available for the device. The specific conditions and accompanying reactions vary based upon its intended use and are specified using a configuration table.

The application software on the device is designed to be highly configurable through the dynamic instantiation and composition of software components at runtime (during startup) as specified by the configuration table. However, the device's system software was not specifically designed for predictability with respect to any one configuration, thus the only way to determine if a configuration will meet its performance specification is by physically measuring the device's response times while running with a configuration.

The purpose of this experiment was to determine if it was possible to predict the worst and average case latency of critical threads running on the device based upon settings in the configuration table, dependencies between components, and measurements of component worst and average case execution times. Ultimately, the goal is to predict the device's response time for future configurations. However, this experiment was focused solely on the ability to predict the latency of critical threads, a first crucial step to achieve this goal.

After an initial inspection of the software and accompanying documentation it was concluded that the experiment was going to require much more effort than initially anticipated. This conclusion was based upon the following initial findings:

1068-3070/08 $25.00 © 2008 IEEE
DOI 10.1109/ECRTS.2008.7

IEEE computer society

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

1. The device's system software was much larger than initially expected. The software source code, written in C and C++, is more than one million source lines of code.
2. Documentation describing the software architecture, design and implementation was limited.
3. While the device was designed to be highly configurable through the composition of software components, the rest of the system software that supports all of the other needed functionality was not component based thus complicating modeling.
4. Modeling parts of VxWorks operating system had increased difficulty due to lack of detailed documentation and source code.

The rest of this paper is structured as follows. Section 2 provides a brief background on our research and our approach to modeling and prediction. Section 3 outlines the approach used to understand, model and predict the behavior of the system. Section 4 presents the prediction results. Section 5 discusses limitation and issues associated with our approach and Section 6 summarizes our conclusions.

2. Background

The theories and concepts used in this experiment are based on our work in the Predictable Assembly from Certifiable Components (PACC) initiative at the Software Engineering Institute (SEI). PACC builds on software architecture technology, software component technology, and a growing body of theory for predicting the quality attributes of software systems (for example performance, security, safety). Architectural design constraints that satisfy the assumptions of quality attribute theories ("smart constraints") are enforced at construction time and run time by software component technology. Analysis is automated by automatic generation of analysis models from assembly specifications. The complexity of this automatic generation, and of the underlying analytic theories, is packaged in a reusable form called a reasoning framework. The resulting predictions have an established and verifiable statistical or formal basis for objective confidence.

One of the core technologies developed by the PACC initiative are the performance reasoning frameworks, which are a combination of a property theory, an automated reasoning procedure, and a validation procedure that is used to predict assembly properties. These frameworks are founded on the principles of General Rate Monotonic Analysis (GRMA) [1] for predicting the average and worst-case latency of periodic and stochastic tasks in what is typified as embedded, real-time control systems.

ABB Corporate Research (CRC) is the central research unit for ABB, conducting research in industrial areas including both power technology and automation technology. It works in partnership with ABB business units and leading universities to connect academic research and concrete product development through more applied industrial research, technology scouting and adaptation of technology to the needs of ABB business units. The SEI and CRC have previously performed successful work together in the area of component-base software, with a track record covering, among other things, predictable assembly.

3. Approach

Based upon an initial evaluation of the system software (discussed in the introduction), it was concluded that, for this experiment, a prediction model of the software for the entire system with all possible configurations could not be produced due to time and budget constraints. Thus, the experiment was scaled down to a manageable and relevant scope.

As part of the methodology used to scale down the system, all software in the device was categorized into three primary software component abstractions:

1. Software without source code such as the operating system VxWorks
2. Application support software that provides:
   a. the ability to dynamically configure an application with specific functionality through the dynamic creation and composition of components at runtime (during startup)
   b. support functions to the system software components
3. Application software components[1]

Next, so as not to model every possible software component, the number of application software components was limited by defining only two candidate configurations which included two inputs with defined ranges and rates and one output. One configuration was a positive configuration that was known to be schedulable and the other one was a negative configuration that was known not to be schedulable. This approach allowed the determination of the component abstractions that would actually be instantiated on the system for either configuration. Then, using this information, the threads that would be running on the device could be determined.

[1] These are not commercial of-the-shelf but in-house developed components.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

Given that the goal for this experiment was to predict the worst-case and average-case latency of the application threads (which are responsible for the execution of the application software components), all other lower priority threads that would not interfere with the execution of the application threads were eliminated from the model.

3.1. Finding Thread Dependencies

In order to build a model of the system, detailed information about how threads communicate with each other was needed. Semaphores and shared memory are used as the primary mechanism for threads to communicate with each other. The semaphores are used as *kickers*, meaning a consuming thread blocks on a semaphore which is released by the producing thread once enough data has been produced. Some of the threads used message queues for communication. Dependencies between communicating threads were found by analyzing those threads that shared common semaphores and message queues.

In order to locate the common semaphores, the tracing facilities of VxWorks were used to log every system call that operates on a semaphore. Tracing such calls (and therefore analyzing them) can be done using WindView, the Tornado logic analyzer for real-time applications [12]. These traces contain timestamp, type of access and the id of the thread using the semaphore.

WindView traces of the running device configuration were analyzed using a custom parser to create a list of the threads that accessed a particular semaphore, and how it was accessed. For each semaphore in the system, the traces revealed the threads that took or released the semaphore and the associated access count. This information was further analyzed in order to find thread dependencies. A trace showing two threads, one taking and the other giving the same semaphore the same number of times suggests a thread dependency – one thread is most likely waiting for data from the other before it can do its job. A trace showing the same thread taking and giving the same semaphore signifies that the semaphore is used to protect a critical section of the code and is therefore not a communication dependency between threads.

When a possible thread dependency had been identified it had to be verified. This was accomplished by checking the source code, reading available architecture documentation and interviewing the developers. Once a thread dependency had been verified, it was added to the core model as described in the Section 3.3.

3.2. Measuring component execution time and application latency

For each software component in the system, only two specific runtime characteristics were of interest: period and CPU execution time. Period is defined as the reoccurring interval of time at which the software component is scheduled to run. CPU execution time is defined as the total uninterrupted time the software component is being executed in the CPU during its period.

VxWorks tracing facilities, whose output is viewable using the WindView tool, provided much of the needed insight to obtain this information. Specifically, WindView depicts running and blocked states on a per thread basis along with a variety of scheduling states. However, this information, absent of the context as to what the software components are doing at any moment in time, was not sufficiently accurate to determine periods and CPU execution times.

Key in knowing this information was to learn the execution lifecycle of the application support threads and software components of interest (e.g., startup, steady state, reaction state, shutdown, etc.) and being able to identify those states within execution traces. Through interviews, it was learned that all threads in the system, in steady state, were designed to yield the CPU until such time that there was work for each thread to perform (either through timers, data arrivals, or other coordination mechanisms). This steady state activity was performed by a specific task body in each thread.

```
// Task A task body
…
for (;;) {
   if (waitForResource(X) == OK)
   {
     wvEvent(BEGIN, 0, 0);
     //
     // perform task body work
     //
     wvEvent(END, 0, 0);
   }
   else
     break;
}
…
```

Tagged Task body

```
// software component exec chain
…
for (;;) {
   wvEvent(BEGIN,0,0);

   while (next = nextSC()!=NULL) {
     wvEvent(SC_BEGIN, 0, 0);
     next->execute();
     wvEvent(SC_END, 0, 0);
   }

   wvEvent(END,0,0);

   if (signalled)
     break;
   waitForNextPeriod();
}
…
```

Tagged SC Execution Chain

Figure 1. Example of how code was tagged

Given that the task body was known, and that source code for these task bodies was available, each task body was tagged by inserting WindView user event markers in the source code. These markers delineated in the trace when a specific task body was actually running in the CPU and when that task body had yielded. Figure 1 illustrates how these markers are used to delineate these task body traces.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

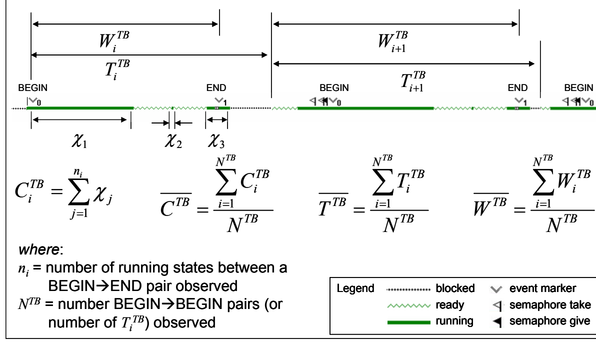

Figure 2. Period and CPU execution

With these markers, the actual periods and CPU execution times for the thread could be measured. For thread periods (T_i^{TB} in Figure 2) the time interval between each `BEGIN` marker was used (or the immediately preceding RUNNING or READY state—which ever occurred first). For CPU execution time (C_i^{TB} in Figure 2), time intervals of running states observed were accumulated between each `BEGIN` and `END` marker (while ignoring any READY or BLOCKED state time intervals occurring between those same markers). This approach only applied for those threads where source code was available. For all the remaining threads, which were the native VxWorks' threads, the average periods ($\overline{T^{VX}}$) and CPU execution times ($\overline{C^{VX}}$) were estimates based on untagged observations.

Similarly, WindView event markers (`SC_BEGIN`, and `SC_END`) were used to tag the execution of each of the software components (SC) assembled to carry out the application's functionality. The only difference in this case was that only CPU execution time was measured for each individual software component ($\overline{C^{SC_*}}$ in Figure 3, where '*' matches a specific software component). Figure 3 illustrates two sequential traces for an application thread task body (denoted by periods T_i^{TB} and T_{i+1}^{TB}). Each trace shows that there were four software components executed (denoted *SC_A* through *SC_D*) by the task body during those periods.

Period for software components was not needed since these software components were executed linearly in the context of the period of the application thread.

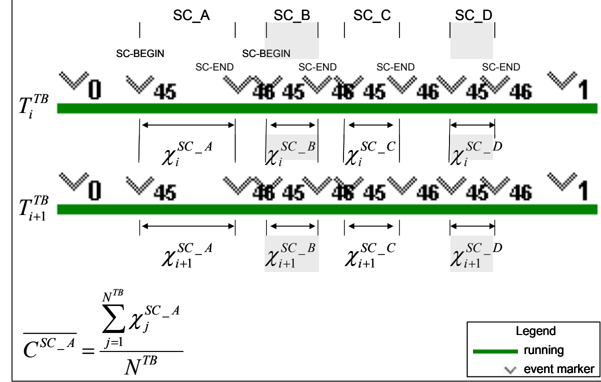

Figure 3. Software component execution chain

Measurement Overhead and Verifying Execution Time

WindView's State Summary feature provides the total running, blocked, and ready time measures for all threads in the system. Using these independent summaries as a check, the total accumulated CPU execution time for a task body (C^{TB}) computed using the event markers could be verified—that is C^{TB} should be close to 100% of that reported by WindView. However, comparisons between these two measures were consistently off by 5 to 7%.

The major source for this difference was the overhead introduced by the injection of the event markers into the task bodies. That is, C^{TB} is the accumulation of the running states between the `BEGIN` and `END` markers (see Figure 2), however, there remains CPU execution time between the `END` and `BEGIN` sequences that is not accounted for—this is the average measurement overhead ($\overline{CM}$) that had to be included.

One other minor source for this difference was that measures were captured during steady state. The system was already executing when collection of WindView traces commenced. Conversely, the system was also executing in steady state when collection ceased. As a result, not every task body trace conveniently started with a `BEGIN` marker or ended with an `END` marker. Any pre-`BEGIN` or post-`END` running states were discarded when computing C^{TB} as these were not needed for software component measurements, but had to be accounted for when verifying the computations.

To verify C^{TB} of the marked task bodies against the total running states reported by WindView (C^{WV}), the equation (1) was used.

$$delta = \frac{C^{TB} + \left(\overline{C^{TB}} + \left(\overline{CM} \bullet \left(N^{TB} + 1\right)\right)\right)}{C^{WV}} \quad \textbf{(1)}$$

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

Here, C^{TB} is increased by the total number of times the task body is executed (N^{TB}) multiplied by the average $\overline{CM}$ calculated (which was on the order of 300ns) and is increased by one additional average task body execution (with overhead, hence '+1' in equation (1)) which is added to account for any potentially discarded `BEGIN` and `END` state. This resulted in a delta of less than 1% of that reported by WindView and that computed for the task body—thus verifying the accounting for C^{TB}.

This same approach was also repeated for determining the overhead for executing the software components execution chain, also resulting in the same success. Overhead in executing the software component chain not only accounted for the injection of the event markers (`SC_BEGIN` and `SC_END`) but also the runtime executive loop that iterated through and executed the software component chain (right diagram in Figure 1).

Observed Best, Average, and Worst-case Periods and CPU Execution Time

The ability to predict the worst case latency of the application threads was predicated on the ability to know the worst case execution time of the threads and software components in the system. Although it was known that various effects (e.g., caching) inject non-determinism into knowing the true worst-case execution time (WCET) [2], the approach used was to observe WCET over a fixed interval of time that matched the device's testing methods and standards. This meant that the device was subjected to the maximum allowable data rates on all input channels for a fixed interval of time.

At the conclusion of that testing period, threads were analyzed to compute best case, average case, and worst case periods and CPU execution times based on the tagged WindView traces captured. The analysis also included a histogram plot of the observed period and execution traces as a means to analyze the behavior of each of the threads and software components in the system.

Figure 4 shows two examples of the roughly 500 histograms generated from the traces (one a VxWorks thread, and the other a tagged thread, Task A). Histograms that exhibited one peak were indicative of threads that conformed to a simple, data-independent behavior while running in steady state. Other histograms that exhibited more than one peak were indicative of more complex, data-dependent behavior that warranted further investigation (see below). For example, the VxWork thread (top in Figure 4) is the interrupt handler for input data. It was learned through interviews, that the two peaks seen in the histogram for the period are representative of the two different type of input data rates supported by the configured device, channel 1 (left peak) and channel 2 (right peak). Likewise, it was further determined that the Task_A thread (bottom in Figure 4) handled the processing of those channels and did behave differently depending if the data was from channel 1 (left peak) or for channel 2 (right peak).

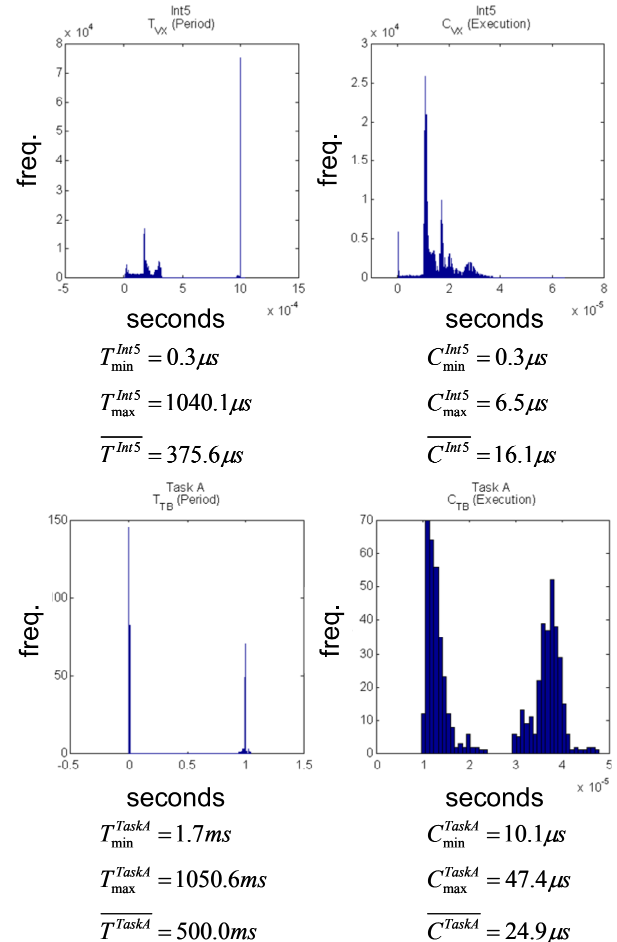

Figure 4. Period and Execution Histograms

Application Latency

Lastly, the application threads (responsible for iterating and executing the software component chains) were measured using event markers according to the same approach for measuring the system threads. Only in this case, the ability to measure the latency of the application thread was the focus.

Figure 2 also illustrates how these markers are used to delineate these application thread task body traces for latency in WindView. For application thread

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

latency (W_i^{TB}) the time interval between each T_i^{TB} and the next `END` marker was used.

3.3. Modeling

The end goal of the modeling effort was to create a performance model that supported predicting worst-case latency of the application threads in the device. Performance modeling can be accomplished by identifying the sources of events and the responses to them [13, 14]. These events can be external (e.g., arrival of a data packet) or internal (e.g., timer expiration). A response is the work carried out by the system as a reaction to an event. Responses are composed of actions, which are portions of code that execute at a fixed priority level and have no synchronization with other parts of the system [1]. In most large systems, and especially in component-based ones, responses are implemented as a composition of components that interact with each other using both synchronous (call-return) and asynchronous (event-based) communication. Figure 5 shows the component and connector view of an example of such a response.

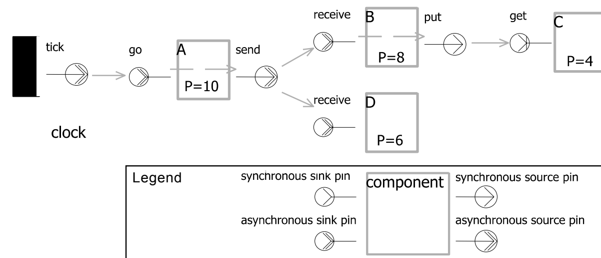

Figure 5. Response with asynchronous communication

Since performance analysis methods require that the responses be expressed as a sequence of actions with no internal concurrency, responses that in the software architecture look like the one in Figure 5 need to be flattened in the performance model. This process is not trivial, especially when the response includes asynchronous communication, which creates concurrency within the response. In this example, based on the priorities and connector types, the corresponding response to an event on `A's "go"` in the performance model would be <A, B, D, C>.

Creating a performance model is an expensive task that requires expertise [4]. In addition, small changes in the design of the system may have greater impact on the performance model, incurring additional maintenance costs. For example, changing the priority of component B in Figure 5 from 8 to 2 changes the response in the performance model to <A, D, B, C>. The effort required for performance modeling and analysis can be reduced by using a reasoning framework, a combination of theory, an interpretation that can translate designs into attribute specific analysis models, and an evaluation procedure to compute predictions [3].

Lambda$_{WBA}$ is a reasoning framework that predicts worst-case latency using RMA [1]. It includes an automated interpretation that transforms a design model into a performance model suitable for analysis with RMA. The evaluation of the model is done using MAST [6], a tool that implements an RMA technique for analyzing tasks with varying priorities [5].

Considering the aforementioned reasons, the modeling effort was focused on creating a model of the software architecture instead of directly creating a performance model. In that way, changes due to the different configurations and even the evolution of the application support software could be handled at a higher level of abstraction.

The model was created using the *Construction and Composition Language* (CCL), a language for specifying components and their assemblies [7]. Components in CCL have *sink pins* through which they receive stimuli from other components. Upon receiving a stimulus, the *reaction* associated with the sink pin is executed. In order to interact with other components, reactions can in turn emit stimuli through the component's *source pins*. Reactions can be threaded if they execute in their own thread context, or unthreaded if they execute in the caller's thread context. In order to support different modes of component interaction (call-return, event-based), pins can be either synchronous or asynchronous. The following code snippet shows the specification in CCL for the type of component B in Figure 5.

```
Component ComponentB() {
    sink asynch receive();
    source synch put();
    threaded react theReaction (receive,
                                put) {}
}
```

This code defines a component type *ComponentB* with an asynchronous sink pin *receive*, a synchronous source pin *put*, and a threaded reaction *theReaction* in which both pins participate. Even though CCL allows specifying data signatures in pins and state machines in reactions, the model was kept at this level of detail, a level sufficient for performance analysis. Component instances are then assembled together by connecting pins as shown in the following code.

```
clock:tick ~> A:go;
A:send ~> { B:receive, D:receive }
B:put ~> C:get;
```

Thus far, the CCL code shown describes the components and their composition. However, in order to do performance analysis additional information is needed, namely, event interarrival distributions ($\overline{T^{VX}}$

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

and $\overline{T^{TB}}$), CPU execution time distributions ($\overline{C^{VX}}$, $\overline{C^{TB}}$, and $\overline{C^{SC_*}}$), priorities, and deadlines. This information can be added to the specification via the CCL annotation mechanism. For sources of events, the event interarrival distribution is specified with the following annotation on the source pin, where the value of *eventDistribution* in this case indicates a constant interarrival time of 100 units.

```
annotate clock:tick {"lambda*",
    const string eventDistribution =
            "C(100)" }
```

Optionally, the deadline for the response to that event can be specified. If omitted, it is assumed to be at the end of the period for periodic events.

```
annotate clock:tick {"lambda*",
        const float deadline = 90.0 }
```

Threaded reactions must be annotated with their priority as in this example.

```
annotate B:theReaction {"Pin",
        const int priority = 8 }
```

Each sink pin will trigger some component code to be executed, so their execution time must be specified. In this example, the execution time has a general distribution with known minimum, average, and maximum.

```
annotate A:go {"lambda*",
    const string execTime =
            "G(9.5, 10.0, 10.5)" }
```

Building the Model

The model for the device was divided in four modules:

- Core model
- Configuration model
- Core execution times
- Configuration execution times

The core model contains the specification of the part of the system that is common to all configurations, which includes interrupt handlers, data distribution services, and other supporting functions. The configuration model contains the assembly of software components that carry out the specific functions that a given configuration provides. All the execution time annotations for both parts of the model are kept in separate modules so that the model can be reused for different target hardware.

The core model was the one that required the most effort to create, mainly because it was not component based and documentation describing the software architecture, design, and implementation was limited. An initial model was created using one model component for each thread in the system. For those few threads whose triggering events were known (e.g., interrupt handlers), specific sources of events were created. For the rest of the threads, the initial approach was to model them as driven by timers whose periods were set according to the measured period of the thread. This resulted in a very flat model with no dependencies and little causality, similar to the one shown in Figure 6, albeit much larger.

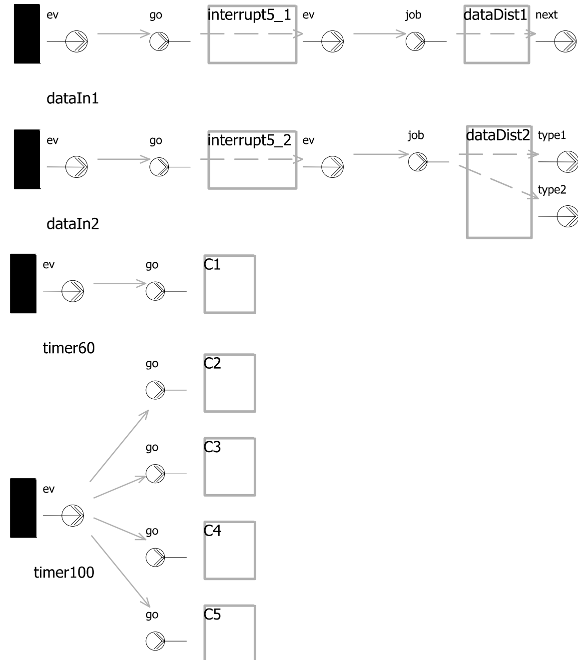

Figure 6. Initial model with few thread dependencies

Using the thread dependency information mined with the approach described in Section 3.1, it was possible to refine the model by combining several seemingly independent threads into one response composed of multiple model components.

Threads having the same period were likely to have dependencies among them and therefore, they were further investigated. Figure 7 shows that components *C2*, *C4*, and *C5* from the model in Figure 6 were combined in one response modeling the discovered dependencies. However, Figure 7 also shows that component *C3* was kept as a separate response because no real dependency was found, and despite having the same period, the offset or phase of this component was different than for the rest of the components having the same period.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

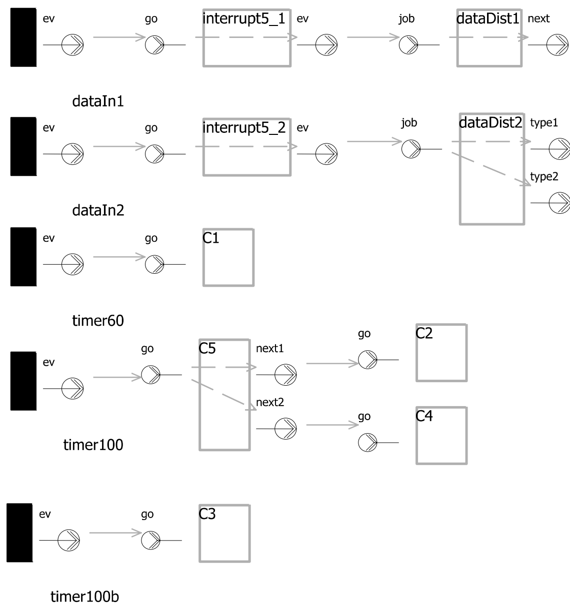

Figure 7. Model with components combined into responses

In most cases, a one to one mapping from threads to components was created. However, in a few cases, the histograms of interarrival times and CPU execution times showed two behaviors. When two distinct statistical modes were observed in the interrarival times, it was concluded that the component was being triggered by two different sources of events (see Int5 period in Figure 4). Similarly, when the execution time histogram showed two distinct statistical modes, it meant that the component was taking two different execution paths, possibly performing two different functions (see Task_A execution in Figure 4). In this case, the thread was modeled as two separate components. In all cases, confirmation for the modeling decision was sought either by code inspection and/or from the system developers.

Thus far, all the interaction between the components in the model had been asynchronous. In some parts of the system, a greater fidelity of the model was achieved by breaking up some monolithic components into several components with synchronous interactions, reflecting the calls to specific functions performed within one thread.

The other part of the model, the configuration model, was in the most part generated automatically from the configuration table. This is important because this part of the model not only changes from product to product, but also has the largest number of components. The CCL code modeling all the instances and their connections was automatically generated using the information in the configuration. There was one small fraction of this model that was not automatically generated. This portion of the model had to do with protected access to a shared repository that is done at the beginning and end of each application response. Although this was written manually in this experiment, it would be possible to generate this as well.

Using the Model

The model of the architecture was analyzed using the $\text{Lambda}_{\text{WBA}}$ reasoning framework. $\text{Lambda}_{\text{WBA}}$ is fully integrated into the PACC Starter Kit (PSK) [8], so once the model is created and all the required annotations are in place, predicting the worst-case latency is just a matter of selecting the performance analysis option on the CCL specification created during the modeling process.

The same specification can be used to predict different scenarios or operation modes, for example to predict latency when data is read from one input channel and when data is read from both input channels. This is accomplished by using scenario annotations in the model.

```
annotate channel1:data { "scenario",
         const int scenario = SCN_MODE1}
```

This annotation indicates that the source pin *channel1:data* will emit events only when the scenario SCN_MODE1 is selected for analysis. Pins can participate in more than one scenario and if they do not have a scenario annotation, they are assumed to participate in all the scenarios.

When the analysis with $\text{Lambda}_{\text{WBA}}$ is invoked, the reasoning framework performs the interpretation, creating a performance model that is then translated to the specific syntax of MAST for final evaluation. The results of the evaluation include the best and worst latencies for each of the responses modeled in the system. Additionally, those responses that do not meet their deadline are visually flagged with red color.

As a worst-case reasoning framework, $\text{Lambda}_{\text{WBA}}$ will compute the worst-case using the worst-case execution time for each component and accounting for the worst preemption and blocking effects by other threads. Since it is unlikely that all these worst cases will happen simultaneously, the predicted worst-case latency may be very difficult to reproduce in testing, making it difficult to gauge the accuracy of the model. Section 4 describes the results of the prediction and how alternative prediction methods where used to judge the correctness of the model.

4. Prediction Results

In this experiment, the system was considered schedulable if the predicted worst-case latency for each

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

of the application threads would occur before the end of the thread's period. If all threads were schedulable, then the prediction result was presented by Lambda$_{WBA}$ in the affirmative. As discussed in this section, the system as configured was predicted and shown to be schedulable. However, additional insight was needed to ascertain to the 'goodness' of the predictions themselves, as well as the approach (i.e., the measurements, model, and tools) that was used to make that prediction. This was accomplished in two ways: a comparison of the predicted worst-case latency to that actually observed in the system; and a comparison of the average-case latency predicted to that actually observed.

Schedulability Analysis: Worst Case Latency for Application Threads

Predictions for and observations of worst-case latencies are shown in Table 1. Based on the WCET measured for the threads in the device ($C_{\max}^{TB}, C_{\max}^{VX}$, and $C_{\max}^{SC_*}$) and the model of thread priorities and their inter-dependencies, Lambda$_{WBA}$ predicted that application thread, Task A's, worst case latency would be just over 2.7ms and that application Task B would be just over 4.6ms. Given that both of these predictions are less than the deadline (3ms and 8ms respectively) for each of these threads, the system was considered schedulable.

Table 1. Predicted and observed WC latency using WCET

Thread	Deadline	Predicted WC Latency	Observed WC Latency
Task A	3000μs	2764.50μs	1357.13μs
Task B	8000μs	4635.00μs	1797.58μs

At first glance, this was certainly good news—that is, based on the WCET for all components and a task phasing which exhibit the worst preemption and blocking among threads the device would be (and was) schedulable. However, as noted above, to actually achieve such conditions in the actual device and observe all these most unfortunate conditions at once would be difficult to achieve. Had Lambda$_{WBA}$ produced the predictions in the table or any higher predictions less than the deadline, the system would have been considered schedulable nonetheless.

A more likely explanation of the wide gap between the predicted and observed worst case latency in the device is that the actual CPU execution times being experienced by the threads in the system most of the times were closer to the average case execution times (ACET) than WCET. A second prediction was generated with Lambda$_{WBA}$, however this time the ACETs ($\overline{C^{TB}}, \overline{C^{VX}}$, and $\overline{C^{SC_*}}$) were used as a basis for the predictions. The second predictions are shown in Table 2.

Table 2. Predicted and observed WC latency using ACET

Thread	Deadline	Predicted WC Latency	Observed WC Latency
Task A	3000μs	2415.90μs	1357.13μs
Task B	8000μs	2893.80μs	1797.58μs

As expected the predicted worst case latency was closer to that observed in the device supporting the notion that the threads were operating closer to their ACET most of the time, instead of the more unlikely situation predicted as shown in Table 1, which requires that the threads and components exhibit their WCET all at the same time.

The last test attempted for this device's configuration was a negative test—that is, if Lambda$_{WBA}$ predicted that the device, as configured, was *not* schedulable then the device should be observed to fail. One of the software components in the chain was a component introduced as a control (SC_CTRL). In the positive configuration, this software component was known to burn approximately $\overline{C^{SC_CTRL}} = 166.01\mu s$ of CPU execution time. In the negative configuration, the same software component, was reconfigured to increase its CPU execution time by 13 to $\overline{C^{SC_CTRL}} = 2158.16\mu s$. This value when combined with the remaining software components should put the device just over the deadline for the application thread causing the device to fail. Lambda$_{WBA}$ reported that utilization was too high and that the negative configuration was not schedulable. When the device was tested in the lab using the control component in the negative configuration, it did fail.

Lambda$_{WBA}$ passed both the positive and negative configuration tests. Additionally, this success was an early indicator that the model produced was capturing the behavior of the device as learned through available documentation, measurement, inter-dependency analysis and supporting investigation. Additional analysis of the model was needed to make any objective conclusions as to the goodness of the model and the accuracy of the reasoning framework to make predictions based on that model.

Average Case Actual vs. Predicted

After looking at the results using the ACET for worst-case latency prediction, the hypothesis was formed that if the model was an accurate representation of the

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

device, average-case latency ($\overline{W^{TB}}$) prediction would be close to average-case latencies observed in the device. The reasoning framework $Lambda_{ABA}$ [10], which preceded $Lambda_{WBA}$, predicts average-case latency by first transforming the design model to a performance model with linear transactions and then using discrete event simulation to evaluate it. $Lambda_{ABA}$ has been objectively validated to produce predictions with less than 1% error, at least 80% of the time [11]. $Lambda_{ABA}$ can use different discrete-event simulators to evaluate the model; in this case, SIM-MAST was used [9]. The predicted and observed average-case latencies, $\overline{W^{TB}}$, shown in Table 3, were very close (around 1% MRE[2]), providing evidence that the model was indeed a good model of the actual system.

Table 3. Predicted and observed average latency

Thread	Deadline	Predicted Average Case	Observed Average Case	MRE
Task A	3000μs	686.33μs	691.53μs	0.8%
Task B	8000μs	530.35μs	537.26μs	1.3%

Confidence Intervals

Statistically, results from $Lambda_{WBA}$ (schedulability pass/fail) and $Lambda_{ABA}$ (MRE) provided initial indicators as the correctness of model created as a result of this experiment. This, in of itself, is not necessarily sufficient to provide concrete empirical evidence as to the 'goodness' of this model. Confidence intervals were selected as the approach to provide that objective confidence given the encouraging MREs that were being produced as a result of the average-case predictions in the previous section. Furthermore, in order to produce the confidence intervals, additional observations of the device would be needed. If the interval produced was within a narrow range for a large population of those observations, this would attest to the repeatability of the behavior modeled and increase the confidence in the results produced by this experiment.

Two different instances of the device were stressed and measured over the course of a few days to obtain the additional observations. To shorten the time required to capture and analyze these results, two devices were situated in two different laboratories by two different teams using the same measurement procedure and device configuration. In total, 29 sets of observations were recorded and coalesced between the two devices. Each set of observations consisted of statistical measures for period, CPU execution time, and latency (best-, average-, and worst-case as discussed in Section 3) for 72 threads and 109 software components.

The average-case latency MRE from each set was computed from the data for each of the application threads. This resulted in 29 sample MREs comparing predicted average-case latency with observed average-case latency for Task A and 29 samples for Task B. Each of the 29 samples used in computing the confidence interval passed the Shapiro-Wilks[3] test for normality. This was an important step, as the tool used for computing the confidence interval, `StInt.exe` [15], is based on the assumption that the data (the MREs in this case) fit a normal (Gaussian) distribution.

To compute the interval, the population percentage was fixed ($\rho = 0.99$), and two target confidence levels were selected ($\gamma = 0.95$ and 0.99). This meant that we were looking for γ confidence in the upper bound on the expected MRE for future observations and that we expect ρ of those future observations to be less than that upper bound. The resulting upper bound (Ub MRE) confidence intervals are show in Table 4.

Table 4. Prediction confidence intervals

Thread	Population (ρ)	Ub MRE where $\gamma = 0.95$	Ub MRE where $\gamma = 0.99$
Task A	99%	8.75%	9.42%
Task B	99%	6.03%	6.44%

Consider Task_A, in 99% of future observations for average-case latency, the MRE (predicted v. actual) will be no greater than 9.42% and that there is 99% ($\gamma = 0.99$) confidence in that upper bound. Selecting a slightly less confidence level, 95%, the upper bound MRE will be 8.75% for Task_A.

From the statistical result generated from $Lambda_{WBA}$ and $Lambda_{ABA}$, including the negative configuration tests, and the repeatability of the observed results from average-case latency testing which formed the basis of the confidence intervals, the model was sound for the configuration of the device used in this experiment.

5. Limitations & Issues

Software Architecture and Design

There were some difficulties associated with modeling certain parts of the application support software due to

[2] Magnitude of Relative Error

[3] The Shapiro-Wilks' W test is used in testing for normality. If the W statistic is significant, then the hypothesis that the respective distribution is normal should be rejected. [Shapiro, Wilk, & Chen, 1968].

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

its architecture and design. This primarily was due to the use of few designs patterns that can not be modeled accurately for the purposes of prediction. Some issues encountered where:

1. Dynamic thread priority changes
2. Polling threads. Threads that poll have a nondeterministic execution time and also tend to waste CPU cycles
3. Non-Deterministic phasing among threads and consistency of phasing between device restarts
4. Threads with varying execution times which are actually performing more than one task
5. Tasks being shared by a group of threads. This causes the execution time of these threads to be nondeterministic.
6. Non-harmonic scheduling among threads

In these cases, the best possible approximations were made; however, these approximations do have a negative impact on the accuracy of the prediction model which may account for the 10% upper bound MRE for average-case latency predictions. Correcting these issues in the device's system software will improve the real-time response of the software and the model's prediction results.

Measurement

The software components used in this device are somewhat special as they tend to only have a single path of execution. This unique attribute makes it much easier to determine the WCET of the components due to the absence of multiple paths of execution within the component. Measurement observations were used to estimate the average and worst case execution times. These observations include any observed caching effects as well any measurement errors. The true accuracy of this approach is unknown because little effort was put into determining the true number of observations needed to obtain an accurate result. Thus, the determination of the sample size was, for the most part, ad hoc. However, in this case, given that components only have a single path of execution, the errors associated with this approach are most likely minimal. Work is ongoing in this area to improve and quantify the accuracy of this approach.

End-To-End Latency

There is an obvious difference between the latency of application threads and end-to-end latency for the device itself. Thread latency only covers the time it takes to execute the internal function logic. End-to-end latency includes thread latency plus the time it takes to collect and process an input before it reaches the application, and the time it takes for the thread's response, if required, to be presented on the output channel. From a perspective external to the device, the aspects of internal functions are of little interest. It does not matter how the device manages to perform its control and/or monitoring function, as long as it does it on time. Ultimately, the ability to predict end-to-end latency is needed.

The first step in predicting end-to-end latency is the ability to predict the latency of the application threads. For this experiment, the worst-case and average-case latencies of the application threads were predicted. Using this information it can be determined whether or not an application thread will meet its deadline. By design, if an application thread cannot meet its deadline the device will not be able to meet its end-to-end response time. Thus, further calculations are needed to predict the end-to-end response time for the application threads that meet their internal deadlines. In the next phase of this experiment, the prediction of end-to-end latency will be addressed.

6. Conclusion

The work presented in this paper shows that it is possible to use an empirical approach to start imposing predictability of real-time behavior into highly configurable software, designed without a focus on predictability. The approach is based on white-box measuring of runtime behavior and limited architectural knowledge that does not require full scale architecture documentation. Empirical data combined with real time theory can be used to predict the real-time behavior with objective confidence.

This method can be used as a fast start when architecture documentation is limited, particularly for legacy systems that have evolved over long periods of time. It is possible to adjust the coverage and accuracy which implies the possibility to work iteratively. By looking at dynamic behavior such as semaphore usage patterns and observed thread execution times it is possible to point out key areas where further detailed system information is needed. Furthermore, by comparing predicted and observed thread response time, it is possible to refine the knowledge of the system behavior as well as verify the understanding of it. As a bonus, the method can indicate design limitations which affect the overall determinism of a system's behavior which can bring focus to well targeted architecture improvements.

In a domain of large and complex legacy systems having real-time critical functionality, not always built, *a priori*, with predictability as a top requirement, the empirical approach demonstrated in this paper shows promise for a wide application area, where architecture

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.

documentation is limited, complete source code review and architecture reverse engineering is impractical but the source code is available and tooling exists to get needed insight.

References

[1] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonzalez Harbour, M. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Boston, MA: Kluwer Academic Publishers, (1993).

[2] S.Basumallick and K.D.Nilsen. Cache Issues in RealTime Systems. ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, http://mack.ittc.ku.edu/basumallick94cache.html (1994)

[3] Bass, L., Ivers, J., Klein, M., Merson, P. 2005. Reasoning Frameworks. Technical Report CMU/SEI-2005-TR-007, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA(2005).

[4] Woodside, M., Franks, G., and Petriu, D. C. 2007. The Future of Software Performance Engineering. In 2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 171-187 (2007).

[5] Gonzalez Harbour, M., Klein, M. H., and Lehoczky, J. P. 1994. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. IEEE Trans. Softw. Eng. 20, 1 (Jan. 1994), 13-28 (1994).

[6] M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, J.M. Drake Moyano, MAST: Modeling and Analysis Suite for Real Time Applications, ECRTS, p. 0125, 13th Euromicro Conference on Real-Time Systems (ECRTS'01), (2001).

[7] K.Wallnau and J. Ivers. Snapshot of CCL: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, June (2003).

[8] Ivers, J. and Moreno, G. A. Model-driven Development with Predictable Quality. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, OOPSLA '07. Montreal, Quebec, Canada, October 21 - 25, (2007).

[9] P. López Martínez, SIM-MAST, Simulator of MAST Models, Grupo de Computadores y Tiempo-Real, Departamento de Electrónica y Computadores, Universidad de Cantabria, Spain. http://mast.unican.es/simmast/

[10] Hissam, S.; Hudak, J.; Ivers, J.; Klein, M.; Larsson, M.; Moreno, G.; Northrop, L.; Plakosh, D.; Stafford, J.; Wallnau, K.; & Wood, W. *Predictable Assembly of Substation Automation Systems: An Experiment Report*, Second Edition (CMU/SEI-2002-TR-031, ADA418441). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, (2003).

[11] Moreno, G.; Hissam, S.; & Wallnau, K. *Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling*. Proceedings of the 5th International Workshop on Component-Based Software Engineering, in conjunction with the 24th International Conference on Software Engineering (ICSE2002). Orlando, Florida, May 19-20, (2002).

[12] Wind River Systems, Inc., "Tornado Getting Started Guide, 2.2, Windows Version", Wind River Systems, Inc., Alameda, CA, (2002).

[13] Dasdan, A., Ramanathan, D., and Gupta, R., *A Timing-Driven Design and Validation Methodology for Embedded Real-Time Systems*, ACM Transactions on Design Automation of Electronic Systems (TODAES), ACM Press 3, 4 (October 1998), 533-553, (1998).

[14] Thome, B., Glas, B., and Nahm, R.,Validation of Real-Time Systems: From "Soft" to "Hard", 1994 Tutorial and Workshop Systems Engineering of Computer-Based Systems. IEEE Conference Proceedings (May 24-27, 1994), 152-158, (1994).

[15] Hahn, G., Meeker, W., *Statistical Intervals: A Guide for Practitioners*, New York: John Wiley & Sons, Inc., 1991.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on June 29,2020 at 14:49:38 UTC from IEEE Xplore. Restrictions apply.