Chapter 20

REUSING SYSTEMS DESIGN EXPERIENCE THROUGH MODELLING PATTERNS*

Oana Florescu1, Jeroen Voeten1,2, Marcel Verhoef3, and Henk Corporaal1

1Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
o.florescu@tue.nl
h.corporaal@tue.nl

2Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
j.p.m.voeten@tue.nl

3Chess Information Technology B.V., P.O. Box 5021, 2600 CA Haarlem, The Netherlands
Marcel.Verhoef@chess.nl

Abstract Based on design experience for real-time systems, we introduce modelling patterns to enable easy composition of models for design space exploration. Our proposed approach does not require deep knowledge of the modelling language used for the actual specification of the model and its related analysis techniques. The patterns proposed in this paper cover different types of real-time tasks, resources and mappings, and include also aspects that are usually ignored in classical analysis approaches, like task activation latency or execution context switches. In this paper, we present a library of such modelling patterns expressed in the POOSL language. By identifying the patterns that are needed for the specification of a

*This work is being carried out as part of the Bodice project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.
340 ADVANCES IN DESIGN AND SPECIFICATION OF EMBEDDED SYSTEMS

KEYWORDS Real-time systems, modelling patterns, design space exploration models.

1. Introduction

Complex real-time embedded systems are usually comprised of a combination of hardware and software components that are supposed to synchronise and coordinate different processes and activities. From early stages of the design, many decisions must be made to guarantee that the realisation of such a complex machine meets all the functional and non-functional (timing) requirements.

One of the main problems to address concerns the choice of the most suitable architecture of the system such that all the requirements are met. To properly deal with this question, the common approaches are design space exploration and system-level performance analysis. An extensive overview of such methodologies is given in [2] and [14]. They range from analytical computation to simulation-based estimation. These are off-specialised techniques which claim that general purpose languages are ill-suited for system-level analysis. However, due to the heterogeneity and complexity of systems, for the analysis of different aspects different models need to be built and their coupling is often difficult. Therefore, a unified model, covering all the interesting aspects, is actually needed to speed-up the design process. This is how the Unified Modelling Language (UML) [20] came to be conceived. The language was designed mainly for object-oriented software specification, but recently it was extended (UML 2.0) to include (real-time) systems as well.

During the development of new systems, common problems are encountered again and again, and experienced designers apply the solutions that worked for them in the past [11]. These pairs of problem-solution are called design patterns and their application helps in getting a design “right” faster. With the increase in the development of real-time systems, design patterns were needed for dealing with issues like concurrency, resource sharing, distribution [6]. As UML has become the industry standard language for modelling, these patterns are described in UML. However, the semantics of the language is not strong enough to properly deal with the analysis of real-time system behaviour. Therefore, an expressive and formal modelling language is required instead in order to capture in a compact model timing, concurrency, probabilities and complex behaviour.

Design patterns refer to problems encountered in the design process itself, but problems appear also in the specification of components that are commonly encountered in complex systems [13]. Although components of the analysed systems exhibit some common aspects for all real-time systems (e.g. characteristics of tasks like periodicity or aperiodicity, processors, schedulers, and their overheads), they are built every time from scratch and similar issues are encountered over and over.

Contributions of the paper. To reduce the amount of time needed to construct models for design space exploration, we propose modelling patterns to easily compose models for the design space exploration of real-time embedded systems. These modelling patterns, provided as a library, act like templates that can be applied in many different situations by setting the appropriate parameters. They are based on the concepts of a mathematically defined general-purpose modelling language, POOSL [25], and they are presented as UML diagrams. These boilerplate solutions are a critical success factor for the practical application in an industrial setting and are a step towards the (semi-) automated design space exploration in the early phases of the system life cycle.

This paper is organised as follows. Section 2 discusses related research work. Section 3 briefly presents the POOSL language, whereas Section 4 provides the modelling patterns. The composition of these patterns into a model is discussed in Section 5 and their analysis approach in Section 6. The results of applying this approach on two case studies are presented in Section 7. Conclusions are drawn in Section 8.

2. Related Research

An extensive overview of performance modelling and analysis methodologies is given in [2] and [14]. They range from analytical computation, like Modular Performance Analysis [26] and UPPAAL [3], to simulation-based estimation, such as Spade [18] or Artemis [21]. The techniques for analytically computing the performance of a system are exhaustive in the sense that all possible behaviours of the system are taken into account. For this reason, the models are built at a high level of abstraction and they abstract away from the non-deterministic aspects, otherwise the state space explosion problem is faced.

On the other hand, simulation of models allows the investigation of a limited number of all the possible behaviours of the system. Thus, the obtained analysis results are estimates of the real performance of the system. To obtain credible results, these techniques require the models created to be amenable to mathematical analysis (see [23]), using mathematical structures like Real-Time Calculus [5], timed automata [1] or Kahn process networks [16]. As in general, analytical approaches do not scale with the complexity of the industrial systems, simulation-based estimation of performance properties is used more often. In this context, the estimation of performance is based on statistical analysis of simulation results.
The analysis technique described in this paper is based on simulation of models expressed in the formally defined modelling language POOSL. Due to the semantics of the language, analytical computation of the properties of a real-time system is possible. However, the type of models that we shall present throughout the paper, though compact, they incorporate non-determinism. Due to the state space explosion problem, estimation of the system performance is used instead of analytical computation.

With respect to the idea of using patterns for building a model based on previous experience with modelling systems from a particular application area, this is not new. As an example, in [13], the authors propose patterns to deal with the complexity of system models by reusing structures expressing expert modelling experience at a higher level of design and abstraction than the basic elements. In a similar manner, we propose modelling patterns for real-time systems that capture their typical components and characteristics, like tasks, computation and communication resources, schedulers, and input and output devices. The use of them prevents the building of system models from scratch over and over and enable the possibility of automatic generation of such models based on a textual representation.

3. POOSL Modelling Language

The Parallel Object-Oriented Specification Language [25] lies at the core of the Software/Hardware Engineering (SHE) system-level design method. POOSL contains a set of powerful primitives to formally describe concurrency, distribution, synchronous communication, timing, and functional features of a system into a single executable model. Its formal semantics is based on timed probabilistic labelled transition systems [22]. This mathematical structure guarantees a unique and an unambiguous interpretation of POOSL models. Hence, POOSL is suitable for specification and subsequently, verification of correctness and evaluation of performance for real-time systems.

POOSL consists of a process part and a data part. The process part is used to specify the behaviour of active components in the system, the processes, and it is based on a real-time extension of the Calculus of Communicating Systems [19]. The data part is based on traditional concepts of sequential object-oriented programming. It is used to specify the information that is generated, exchanged, interpreted, or modified by the active components. As mostly POOSL processes are presented in this paper, Fig. 20.1 presents the relation between the UML class diagram and the POOSL process class specification. The name compartment of the class symbol for process classes is stereotyped with <<process>>. The attributes are named <<parameters>> and allow parameterising the behaviour of a process at instantiation. The behaviour of a process is described by its <<methods>> which may include the specification of sending (!) and/or receiving (?) of <<messages>>. More details about the UML profile for POOSL can be found in [23].

The SHE method is accompanied by two tools, SHESim and Rotalumis. SHESim is a graphical environment intended for incremental specification, modification and validation of POOSL models. Rotalumis is a high-speed execution engine, enabling fast evaluation of system properties. Compared with SHESim, Rotalumis improves the execution speed by a factor of 100. Both tools have been proved to correctly simulate a model with respect to the formal semantics of the language in [12].

4. Modelling Patterns

One of the approaches for performing systematic design space exploration is the Y-chart scheme, introduced in [17]. This scheme makes a distinction between applications (the required functional behaviour) and platforms (the infrastructure used to perform this functional behaviour). Although we are concerned only with the realisation of the software part of a real-time system, the hardware part must also be taken into account in the analysis in order to predict the behaviour of the system as a whole and the impact each part may have on the others. Moreover, as real-time systems are typically reactive systems, meaning that there is a continuous interaction with the outside world, in [10] we added the model of the environment to the Y-chart scheme, as depicted in Fig. 20.2. The design space can be explored by evaluating different mappings of applications onto platforms.

To reduce the amount of time needed to construct models for design space exploration, we propose modelling patterns to easily compose models of real-time embedded systems. They are provided as a library and act like templates that can be applied in many different situations by setting the appropriate parameters. These modelling patterns emerged from previous experience with real-time systems modelling (see [9], [7], and [24]). Moreover, they reflect
the approach assumed in classical scheduling analysis [4] for modelling such systems. The library of patterns contains templates for different types of tasks, resources, schedulers, and input and output devices, which are presented in Fig. 20.3. This figure shows the Y-chart components to which each of these patterns belongs, the names of the patterns and their parameters. Each of these patterns are explained in the remainder of this section.

Application Model

The functional behaviour of a real-time embedded system is implemented through a number of tasks that may communicate with each other. Task activation requests can be periodic (time-driven), being activated at regular intervals equal to the task period T, or aperiodic (event-driven), waiting for the occurrence of a certain event. There are three types of uncertainties, showed in Fig. 20.4, that may affect a task:

- Activation latency: It may be caused, for example, by the inaccuracies of the processor clock that might drift from the reference time because of temperature variations. For event-driven tasks, the performance of the run-time system, which cannot continuously monitor the environment for events, may also have influence.

- Release jitter: It may be caused by the interference of other tasks that, depending on the scheduling mechanism, may impede the newly activated task to start immediately its execution.

- Output jitter: It may be caused by the cumulated interference of other tasks in the system, the scheduling mechanism that may allow pre-emption of the executing task, the variation of the activation latency, and even of the execution time of the task itself, which may depend on the input data.

In classical real-time scheduling theory [4], the release jitter and, to some extent, the output jitter can be computed, but the activation latency is usually ignored. As in control-oriented systems the effect of this latency might be of significant importance, the POOSL specification of the task modelling patterns overcomes this problem. The two task patterns that we have conceived are visualised using the POOSL equivalent of UML class diagrams in Fig. 20.5.

The PeriodicTask pattern is to be used whenever a periodic independent task is required. Its parameters are the period T, the deadline D, the load, which represents the worst-case value of the number of instructions the task imposes on a target processor and can be obtained based on previous experience, the activation latency l specified as a distribution, and the number of iterations for the case the task is not infinitely running. The AperiodicTask pattern should be applied for the specification of a task triggered by an event from the environment or by a message from another task in the system. Its parameters are the deadline D, the load, and the activation latency l. Each of these two patterns has two methods. One is called Periodic, and respectively Aperiodic, and contains the specification of the task according to the type of triggering it has,
such as waiting for the next period, respectively for the next incoming event, to be activated, whereas the Behaviour method contains the specification of the actual computation the task needs to perform. In the templates provided with our library, the specification of this method is empty for two reasons. The first one is that it depends on the application what a task is supposed to compute, hence the designer who is using this library has to supply the right specification. The second reason is that for the type of analysis we are interested at a high level of abstraction, which will be discussed in Section 6, the actual computation of a task is not important and can be left out.

Platform Model

The modelling patterns we have conceived for describing the platform part of the Y-chart model of a system provide a unified way of specifying communication and computation resources by exploiting their common characteristics. This modelling approach is possible as at a high level of abstraction there is no large conceptual difference between a processor and a bus: they both receive requests, execute them (either by transferring the bits of a message or executing the instructions of an algorithm) and send back a notification on completion. As a resource is typically shared, a scheduler is needed in order to arbitrate the access to a resource.

Figure 20.6 visualises the Scheduler and the Resource modelling patterns that are needed for the specification of the platform model. The scheduler has one parameter which is the name of the scheduling policy desired to be used on a certain resource. Amongst the scheduling policies that we provide within the POOSL library are: earliest deadline first, rate monotonic, first come first served, and round-robin. The resource is characterised by a throughput, which is the number of instructions a processor can execute per time unit or the transfer bit rate on a bus, and an initial latency, which incorporates the task context switch time or the communication protocol overhead.

Environment Model

The model of the environment is composed by input generators and output collectors, as shown in Fig. 20.7. The input generators model the generation of environmental events of a certain eventType with a certain generation pattern which can be chosen amongst periodic, periodic with jitter, or sporadic...
with a certain distribution of occurrence, such as uniform or normal. These events trigger the activation of tasks in the application model and are collected by output collectors which model the output devices in the environment. The collector receives the events of a certain event type exiting the system and compares the end-to-end delay of the system against the desired one expressed as the throughput.

5. Model Composition

To build a model of a real-time system for design space exploration, its specific components that correspond to the modelling patterns described in the previous section must be identified together with their parameters. The names of the necessary patterns and their parameters, together with the specification of the mapping (which task is scheduled on which processor, etc.) and the layout of the platform (which processor is connected to which bus) can be provided as the configuration of the system. From such a configuration, the POOSL model of the system can be automatically generated based on the library of modelling patterns and fed to SHESim or Rotalumis tools for analysis.

As an example, a producer-consumer system is showed in Fig. 20.8(a). The system is made of a periodic task producer, TASK1, and an aperiodic task consumer, TASK2, whose activation is triggered by the production of a new item by TASK1. The specification of the system may look like the one in Fig. 20.8(b) structured along the Y-chart scheme, expressing the application components, the platform, and its interconnections, and the mapping. The structure of the generated model is showed in Fig. 20.8(c). As it can be seen, it differs somewhat from the original system in the sense that the model generation tool is able to detect that the communication between the two tasks is done over a bus, hence a buffer to contain the message and to transport it across the communication medium is required.

For design space exploration, different configurations must be compared. To do this, changes in the initial configuration may be done and the POOSL model regenerated in order to analyse them. To specify a different mapping, the Map specifications must be changed according to the new task-to-resource mapping. To change the architecture components, simply change the Resource specifications and/or their parameters. Similarly, the interconnections of the platform can be changed in the Connection specification tags. In this way, the model can be easily tuned to specify different possibilities in the design space without any knowledge about the underlying formal model that will be generated in accordance with the description of the new configuration.

6. Model Analysis

By composing together the necessary modelling patterns, the complete model of a system can be built and validated. For each configuration specified and generated, during the execution of the model, the scheduler, which also acts as a monitor for the system schedulability, can report if there are any tasks that miss their deadlines. Furthermore, based on the formal semantics of POOSL, it can be analysed if there is any deadlock in the system. If all the deadlines are met and there is no deadlock, then the corresponding architecture is a good candidate that meets the system requirements.

However, for soft real-time systems, it is allowed that some deadlines are missed (usually there is a given threshold). Therefore, in this case, it is especially useful that in the specification of the tasks their computations are decoupled from the activation mechanism, in the sense that the analysis of the model could handle tasks with multiple active instantiations that are likely to miss their deadlines. The percentage of deadlines missed can be monitored and checked against the requirements if, according to this criterion, the underlying platform is suitable.

To correctly dimension a system (the required CPUs and BUSes performance) such that it works in any situation, the worst-case behaviour of the system must be analysed. This usually means to consider the worst-case execution times for all the activities in the system. On the other hand, the analysis of the average behaviour, based on probabilities, which can be enabled in the proposed patterns, as showed in [8], gives a measure of the suitability of the design. If the dimension of the system, needed for the worst-case situation that appears only once in a while, is far bigger than the one needed in average, that could give useful hints for a redesign (e.g. split tasks into smaller ones in order to spread the load onto different CPUs).
Some other useful results the analysis of the proposed model can provide are the release and the output jitter of tasks, which is useful for control applications, the number of active instances of a task type, which influences the missed of deadlines, the throughput of the system, important in streaming applications.

7. Case Studies

In this section, two case studies are presented for which worst-case analysis and design space exploration have been performed using the modelling patterns proposed in this work. The characteristics of the systems and the results of their analysis follow.

A Printer Paper-Path

The first case study is inspired by a system architecture exploration for the control of the paper-path of a printer.

The high-level view of the system model, visualised using SHESim tool, is given in Fig. 20.9. User’s printing requests arrive at the high-level control (HLC) of the machine which computes which activities need to take place and when in order to accomplish the request. The HLC tasks activate the tasks representing the low-level control (LLC) of the physical components of the paper path, like motors, sensors, and actuators. As HLC tasks are soft real-time, whereas LLC tasks (Fig. 20.10) are hard real-time, a rather natural solution was to consider a distributed architecture. LLC can be assigned to dedicated processor(s) and connected through a network to the general-purpose processor that runs HLC.

Under these circumstances, the problem was mainly to find an economical architecture for LLC, whose task parameters are showed in Fig. 20.11. For the models of the time-driven tasks of type T1, T3, and T4, we took into account a latency of up to 10% of their period. Although tasks of type T2 are activated based on notifications from HLC, they behave completely periodic until the next notification arrives. Therefore, their dynamical behaviour was captured using an aperiodic task which triggers a periodic task with a finite number of activations. Tasks of type T5 are event-driven; therefore, a model of the environment was needed (PhysicalComponents), for which we considered event streams with a uniform distribution in [1, 20] ms.

Given the frequency of events and the task execution times, we have analysed three commercially available low-end processors, a 40 MIPS, a 20 MIPS, and a 10 MIPS, and compared their utilisations under different schedulers. Figure 20.12 presents the results obtained using the earliest deadline first scheduling algorithm [4]. Although the 10 MIPS processor seems to be used the most efficiently (close to its maximum capacity), the analysis of the model...
showed that some of the deadlines are missed; thus this processor is not a good candidate. For the other two, all the deadlines are met. Due to the fast execution engine Rotationalis, tens of hours of system behaviour could be covered in less than one minute simulation. Moreover, the analysis of the model gave the values of the maximum release jitter, respectively output jitter of the tasks (for the 20 MIPS they are showed in Fig. 20.13) which could be checked against the expected margins of errors of the control design.

An In-Car Navigation System

The second case study is inspired by a distributed in-car navigation system described in detail in [26]. The high-level view of the system is presented in Fig. 20.14(a). There are three clusters of functionality, as the picture suggests: the man–machine interface (MMI) that handles the interaction with the user; the navigation functionality (NAV) that deals with route-planning and navigation guidance; the radio (RAD) which is responsible for basic tuner and volume control, as well as receiving traffic information from the network. Three application scenarios are possible. Users are allowed to change the volume (scenario I) and to look addresses up in the maps in order to plan their routes (scenario II); moreover, the system needs to handle the navigation messages received from the network (scenario III). Each of these scenarios has its own individual time-line requirements that need to be satisfied. They all share the same platform, however, not all three of them can run in parallel due to the characteristics of the system (only I with III, or II with III). The characteristics of tasks for each scenario are given in Fig. 20.15. They are all periodic tasks with infinite

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Task</th>
<th>Load [instr]</th>
<th>T [s]</th>
<th>D [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>T1</td>
<td>1E5</td>
<td>1/32</td>
<td>1/32</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>1E5</td>
<td>1/32</td>
<td>1/32</td>
</tr>
<tr>
<td></td>
<td>T3</td>
<td>5E5</td>
<td>1/32</td>
<td>1/32</td>
</tr>
<tr>
<td>II</td>
<td>T4</td>
<td>1E5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T5</td>
<td>5E5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T6</td>
<td>5E5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>III</td>
<td>T7</td>
<td>1E6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T8</td>
<td>5E6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T9</td>
<td>5E5</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>
on the resources are much larger, as this case study combines control with data streaming.

The problem related to this system was to find suitable platform candidates that meet all the timing requirements of the application. For exploration of the design space, a few already available platforms (see Fig. 20.14(b)) were proposed for analysis. Two approaches have been applied for the analysis of this system, Modular Performance Analysis (MPA) in [26] and UPPAAL in [15].

MPA is an analytical technique in which the functionality of a system is characterised at a high level of abstraction by quantifying the incoming and outgoing event rates, message sizes, and execution times. Based on Real-Time Calculus, hard upper and lower bounds of the system performance are computed. Although these bounds are always hard, they are in general not tight, meaning that the technique derives conservative estimates of worst and best case.

The UPPAAL model checker is a tool for modelling and verifying networks of timed automata. The analysis results obtained by applying this technique are exact computations of the performance properties. Nevertheless, the method suffers severely from the state space explosion problem. Limitations, stating for example that tasks can be preempted only up to a certain number of times, are necessary, otherwise model checking is not possible anymore. Moreover, combination of scenarios with large difference in the time scale of the requirements (milliseconds versus seconds) proved to be another problem for the model checker.

In the rest of this section, we show that the model of the in-car navigation system, which can be easily built using the modelling patterns (see Fig. 20.16), can also be accurately analysed. All the tasks in the system are event-driven, hence we could easily use just the AperiodicTask pattern with the parameter values showed in Fig. 20.15 for the construction of the application model, whereas for the environment we assumed streams of events with periodic arrival patterns. The end-to-end delay for each scenario on each of the proposed platforms, in the presence or absence of other scenarios, was monitored. The analysis shows that all the timing requirements are met for all scenarios in all configurations. As an example, the results obtained for the worst case end-to-end delay for different combinations of scenarios on architecture A are presented in Fig. 20.17 next to the results obtained using MPA and UPPAAL techniques. As MPA is an analysis technique which finds hard upper bounds, this explains why its results are larger than the other techniques. On the other hand, the results computed by UPPAAL are exact values of the worst case end-to-end delay. It is interesting to observe that our results are very close to UPPAAL (~1% difference which also represents the accuracy of the results), except for scenario III for which the difference is 7%. For this situation, there was a mismatch in the conceiving of the models with respect to the modelling of jitter in the incoming events.

Furthermore, the processor(s) and bus(es) utilisations were monitored and, as an example, Fig. 20.18 shows the results obtained for architecture A. All together, such results help the designer in detecting if there is any scenario likely to miss its deadline, or which processor or bus might be a bottleneck, and in choosing an appropriate platform.

Due to the easiness of using the patterns and going to different configurations in the design space by just changing their parameters, the construction of models for each of the proposed combinations took several minutes. Moreover, as mentioned for the previous case study as well, due to Rotalumis, the engine for the model execution, the analysis results could be obtained also fast.
8. Conclusions

In this paper, we have presented a library of modelling patterns, specified using the Parallel Object-Oriented Specification Language, that enables the automatic construction of models for the design space exploration of real-time embedded systems. To build such models, knowledge about the POOSL language itself is not needed as system models consisting of real-time tasks, computation and communication resources and their associated schedulers are specified in terms of the necessary patterns and the values of their parameters. Due to the expressiveness of POOSL, important aspects like task activation latencies and context switches can be taken into account, enabling the building of realistic models without sacrificing their conciseness. Moreover, due to this reason, the analysis can provide more realistic results than the classical scheduling techniques can.

The use of the patterns presented in this paper reduces both the modelling and the analysis effort. Although completeness cannot be claimed, the efficiency of the model simulation allows exploration of a substantial part of the design space. As future work, we aim at extending the modelling patterns to cover for complex platforms like networks-on-chip, by taking into account memory components, routing algorithms and even batteries for the analysis of energy consumption.

References


<table>
<thead>
<tr>
<th>Scen. I</th>
<th>Scen. II</th>
<th>Scen. III</th>
<th>MMI %</th>
<th>NAV %</th>
<th>RAD %</th>
<th>Bus %</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>87</td>
<td>0</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>88</td>
<td>2</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

*Figure 20.18. Processors and bus utilizations in architecture A.*


