HAPI: An Event-Driven Simulator for Real-Time Multiprocessor Systems

Philip S. Kurtin § Joost P.H.M. Hausmans § Marco J.G. Bekooij § ¶
philip.kurtin@utwente.nl
§ University of Twente, Enschede, The Netherlands ¶ NXP Semiconductors, Eindhoven, The Netherlands

ABSTRACT

Many embedded multiprocessor systems have hard real-time requirements which should be guaranteed at design time by means of analytical techniques that cover all cases. It is desirable to evaluate the correctness and tightness of the analysis results by means of simulation. However, verification of the analytically obtained results is hampered by the lack of a fast high level simulation approach that supports task scheduling and that does not produce pessimistic simulation traces.

In this paper we present HAPI, an event driven simulator for the evaluation of the results of real-time analysis techniques for task graphs executed on multiprocessor systems that support processor sharing. HAPI produces simulation traces that are pessimistic to reality and optimistic to temporal analysis. It can be consequently used to detect optimistic, i.e. incorrect, analysis results.

Several task scheduling policies are supported by HAPI such as fixed priority preemptive, time-division multiplex and round-robin. Preemptive task scheduling decisions are simulated which enables to study the cause of delayed task finishes and thereby helps to identify overly pessimistic analysis results.

We demonstrate the applicability of the simulator using a number of didactic examples and a WLAN 802.11p application.

1. INTRODUCTION

Many embedded systems have hard real-time requirements. The design of these systems requires the use of analytical techniques because it is usually nearly impossible to find and trigger the worst-case situation by means of testing. However, testing remains useful because it might reveal flaws in the analysis techniques and can give insight into corner cases that are coarsely over-approximated and thereby unnecessarily limit the tightness of analysis results.

Some confidence in the correctness of the temporal analysis results can be obtained by testing the software on the physical system. This is usually complemented by testing using a simulation because simulation offers non-intrusive monitoring of all signals in the system, which is hard to achieve with a physical system.

Simulation can be performed at different abstraction levels which results in a trade-off between accuracy and simulation speed. In [1] the register transfer, cycle true, Programmers View (PV) and Communicating Processes (CP) abstraction levels are distinguished, with the register level the lowest abstraction level and CP the highest. The use of a higher abstraction level results in a higher simulation speed because less events need to handled by the simulation kernel. The use of the more abstract simulation models reduces the accuracy of the simulation results and can cause the simulation results to be pessimistic compared to the physical implementation. However, temporal analysis results usually only hold for the physical realization and not necessarily for the potentially more pessimistic simulation models. As a consequence it is not possible to verify analytically obtained results by comparing them with simulation results. More specifically, from a later production of data by a task in the simulator than determined by analysis it cannot be concluded that the analysis results are incorrect.

In [2, 3] it has been shown that simulation models at the CP level can be created for which it is guaranteed that the simulation results are temporally pessimistic. At the CP level there is no notion of an instruction set nor a notion of registers. Creation of these simulation models requires that the application can be modeled as a deterministic dataflow process network and that the multiprocessor hardware allows to determine Worst-Case Execution Times (WCETs) that are independent of the schedule of the tasks. This is for instance possible using the multiprocessor hardware architectures described in [4, 5]. In some cases the same dataflow model that is used for simulation can be used for analysis. In these cases a violation of an analytically computed deadline in the simulator indicates a flaw in analysis method.

However, the approaches of [2, 3] are only applicable in case so-called budget schedulers [3] are used for task scheduling. An example of a budget scheduler is Time Division Multiplex (TDM) scheduling. For this class of schedulers it is possible to compute worst-case response times of tasks independent from schedules and execution times of other tasks. Given the worst-case response times of tasks upper bounds on the production times of these tasks can be found by means of simulation. However, this does not give much insight in the accuracy of simulation results because preemption of tasks by other tasks is not simulated. Furthermore, it is not possible to detect that the used worst-case response times are overly pessimistic or even incorrect be-

DOI: http://dx.doi.org/10.1145/2906363.2906381

cause they are input parameters of the simulation. Another important limitation is that the approaches of [2,3] are unsuitable for systems that use schedulers not belonging to the class of budget schedulers, such as Fixed Priority Preemptive (FPP) task scheduling, which severely limits the scope of their applicability.

In this paper HAPI, an event-driven simulation approach of dataflow graphs on multiprocessor systems, is presented, which is suitable for the evaluation of the correctness of real-time analysis results. This is enabled by formally guaranteeing that analytical results are pessimistic compared to simulation results. The approach is capable of simulating any scheduler type, but the current implementation of the simulator only supports fixed priority scheduling, time-division multiplex scheduling, as well as cooperative Round-Robin (RR) scheduling.

The outline of this paper is as follows. In Section 2 the HAPI simulation approach is compared with alternative approaches. The HAPI simulation approach and its properties is presented in Section 3. The internals of the simulator are presented in Section 4. In Section 5 the capabilities of the simulator are demonstrated using a number of small examples. Finally, the conclusions are stated in Section 6.

2. RELATED WORK

In this section a number of discrete event simulation approaches are compared to HAPI, with a focus on application scopes.

Transaction Level Modeling (TLM) [1] is an approach to build simulation models of multiprocessor systems at various levels of abstraction. The SystemC kernel [6] is used for simulation of these models. Modeling at a higher level of abstractions improves simulation speed, but results in an approximation of the temporal behavior. Building the required TLM models takes usually a significant amount of time. A HAPI simulation model can be seen as a TLM model that operates between the CP and PV abstraction level because there is no notion of registers and instructions, but preemption of tasks by an operating system kernel is simulated. The HAPI simulation model is a dynamic dataflow model [7] in which the execution of actors can be suspended as a results of preemption. The simulated dynamic dataflow model is abstract in the sense that is does not directly reflect the structure of the system. There is for example no notion of a communication bus or network in HAPI. As a result of this high level of abstraction a relatively low number of events need to be handled by the simulation kernel and simulation is typically fast.

Platform architect [8] is a commercial simulation tool developed by Synopsys for the evaluation of the performance of several multiprocessor system architecture configurations in early design phases. Platform architect is based on the SystemC simulation kernel, simulates at the PV abstraction level and has a graphical user interface. Models of communication buses and memory controllers are provided and describe what happens during transactions instead of per each clock cycle. The use of these transaction level models improves simulation speed at the cost of accuracy. Instruction set simulation models are used to model processors. A key difference between Platform Studio and the HAPI simulation approach is that HAPI requires that worst-case execution times of tasks can be determined independent of other tasks in the system. Therefore HAPI is not suitable for analyzing increased task execution times due to contention on memory ports that can occur on both read and write accesses of processors.

The Parallel Object Oriented Specification Language (POOSL) [9] is a system-level modeling language based on a small set of language primitives. POOSL enables a precise representation of a system based on mathematically defined semantics. It consists of a process part and a data part. The process part is based on a real-time probabilistic extension of the process algebra Calculus of Communicating Systems (CCS). The data part is based upon the concepts of traditional sequential object-oriented programming languages like C++. POOSL descriptions are simulated using the discrete event simulator Rotalumis or executed in an interpretative way by the SHESim tool. The underlying formal model of POOSL is the Timed Probabilistic Labeled Transition System (TPLTS). A TPLTS can be transformed into a Markov chain whose equilibrium distribution can be computed analytically. However the Markov chains obtained after transformation have typically a very large number of states which results in a prohibitive run-time of the algorithms to compute the equilibrium distribution. As a consequence it is typically only practical to simulate a POOSL descriptions to obtain an estimate of the equilibrium distribution. The equilibrium distribution corresponds to long running average performance metrics. A key difference between HAPI and POOSL is that HAPI has dataflow as underlying formal model instead of TPLTS. For a subclass of dataflow models the minimum throughput and maximum latency can be computed analytically with computational efficient algorithms, which is not possible for the probabilistic TPLTS models. Another conceptual difference is that HAPI has built-in support for a number of schedulers, while POOSL abstracts from scheduling.

The Y-chart Application Programmers Interface (YAPI) [10] is an event driven simulator for Kahn Process Networks (KPNs) [11]. The YAPI simulator does simulate functional behavior like the HAPI simulator, but does not support a notion of time. Furthermore, unlike the HAPI simulator, the YAPI simulator does not support auto-concurrent execution. The YAPI simulator also does not simulate sharing of processors such as preemptive task scheduling.

A number of SystemC [6] based simulation approaches have been developed [12, 13] to deal asynchronous events such as interrupts. For that matter preemption points are introduced by splitting tasks in smaller fragments. The tasks are split because they can communicate with global variables during their execution. At these preemption points the scheduler checks whether an asynchronous event has occurred. The use of the preemption points introduces blocking times in the simulator which are not present in reality. This blocking causes an unpredictable amount of additional delay in the simulator between the arrival of asynchronous events and their handling, which does not occur in reality. Such an additional delay can make simulation results more pessimistic than analytical results. The HAPI simulation approach does not make use of such preemption points.

3. SIMULATION IN THE CONTEXT OF REFINEMENT THEORY

Temporal analysis using dataflow models is based on refinement theories. The refinement theory in [14] defines that a component X' refines another more abstract component X, i.e. $X' \sqsubseteq X$, if the same values are produced later by

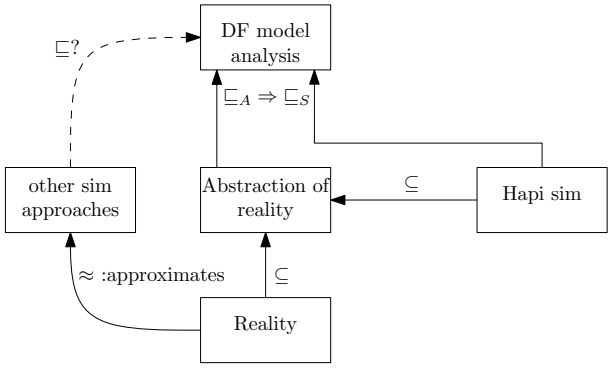


Figure 1: The HAPI verification approach.

X than by X', given equal arrival times of data at both components. Moreover, a component is called temporally monotone if an earlier arrival of inputs at a component cannot result in a later production at its outputs. The refinement theory states that if all components of a graph refine the components of another graph and if the abstract components are temporally monotone, then the refinement of components is lifted to the refinement of graphs. This implies that conservative analysis results can be computed using an abstract dataflow model. Note that an analysis result is called conservative if all arrival times are not earlier than in reality.

However, demonstrating that a component X' refines a component X is usually done by means of case distinction [15–18], which can be non-trivial as a result of the large number of cases that must be distinguished, as well as the lack of formal methods guaranteeing that indeed all cases are identified. Therefore it is desirable to simulate the models used at lower levels of abstraction in order to gain confidence that the abstract analysis models and the analysis results are indeed correct. Confidence in the correctness of the model and the analytically obtained analysis results is thereby increased if none of the production moments found during simulation are later than the ones computed at design time. In contrast, for the case that a production moment found during simulation is later than the moment computed by analysis one would like to conclude that the analysis is incorrect.

However, such a conclusion cannot be drawn in general. The reason is that the model of reality used in the simulator is not necessarily a refinement of the one used to compute analytical results. It is not guaranteed that the simulation model is a refinement of the dataflow model, as indicated by the dashed arrow in Figure 1. As a consequence simulation results can be more pessimistic than analysis results, which makes it impossible to draw conclusions by comparing arrival times of data. Note that the arrow in Figure 1 from the reality block towards the abstraction-of-reality block indicates that at least all traces of reality can occur in the abstraction-of-reality (trace-inclusion, "$\subseteq$").

Analytically obtained results can be verified with the HAPI simulator because it is taken care that the dataflow model used by the HAPI simulator is a refinement of the dataflow model used for analysis. The only difference between the two models is that the analysis model uses constant worst-case firing durations of the dataflow actors while the firing durations of the actors in the simulation model can vary per firing and are obtained during simulation. The firing durations in the simulation model depend on whether actors are preempted. For instance, for a FPP scheduler preemption of actors corresponding to lower priority tasks can occur if higher priority tasks must be executed. Preemption of tasks may result in scheduling anomalies [19] because an earlier arrival of data for a task may result in a later production of another task. The anomalies are bounded from above by a correct dataflow analysis approach that uses a dataflow analysis model with constant firing durations.

A simulation with the HAPI simulator produces traces that are valid for the used abstract model of reality, as shown in Figure 1. This model of reality is the conceptual model of reality in the head of the engineers that create both dataflow analysis models and dataflow simulation models. It must hold that the model of reality is a refinement of the dataflow analysis model, which is indicated by $\sqsubseteq_A$ in Figure 1. Because HAPI produces traces that are valid for the same model of reality it follows according to Figure 1 that also $\sqsubseteq_S$ must hold, i.e. $\sqsubseteq_A \Rightarrow \sqsubseteq_S$. From this we can conclude that if $\sqsubseteq_S$ does not hold that also $\sqsubseteq_A$ does not hold. In other words HAPI simulation results can be used to falsify analytical results.

4. THE HAPI SIMULATOR

The HAPI dataflow model simulator is created using the SystemC library. The SystemC library provides an event-driven simulation kernel that processes events in the order they are generated by components. The SystemC kernel does not actually perform preemptive scheduling of components, but can be used to simulate preemptive scheduling. We will show in this section that this does not require the so-called preeemption points as mentioned in the related work section. Therefore, the HAPI simulator does not introduce an unpredictable amount of additional delay, which makes it possible to use HAPI simulation results to falsify analytical results.

On top of the SystemC library several dataflow elements are defined in HAPI, such as actors and unbounded queues. Actors are enabled when there is a predefined number of input tokens in their input queues. After an actor is enabled it fires immediately during so-called self-timed execution. HAPI simulates self-timed execution of dataflow graphs. Tokens are consumed from input queues at the moment that an actor fires. Each actor has a firing duration ρ which is also often called execution time. An actor produces tokens in its output queues ρ time units after its firing is started. The number of tokens consumed and produced is defined by the quanta of the actors. For ease of understanding, all the quanta of the actors in the examples presented in this paper are equal to one. The actors model tasks that only communicate using FIFO buffers with each other.

Using the Application Programming Interface (API) of HAPI a dataflow graph can be defined in C++ in which dataflow elements are instantiated and connected. Furthermore, it can be specified which actors share the same processor and the parameters of the schedulers can be set, such as the scheduling policy that should be applied and for instance priorities that are assigned to the actors. During simulation the production times of actors depend on whether preemption has occurred during their execution.

With a g++ compiler an executable is created from the C++ program in which the dataflow graph is specified. Running this executabled results in a vcd file in which traces with events are stored. These traces can be viewed with a vcd

viewer such as gtkwave [20]. In the traces it is visible when actors are preempted, leading to a delayed token production.

The HAPI simulator supports so-called auto-concurrent execution of actors during which multiple instances of the same actor can execute at the same moment in time. The HAPI simulator also supports non-deterministic execution because an actor can check how many tokens are in an input queue. With this feature so-called non-sequential firing rules can be defined that can result in schedule-dependent execution behavior [7].

SystemC components generate events by executing the *notify()* SystemC call. Simulation time can only be advanced in the simulator by executing the *wait(Interval, EventList)* SystemC call. This call does not return until *Interval* simulation time has passed or one of the events in *EventList* has occurred.

Preemptive scheduling of tasks on a processor can be simulated in SystemC by making use of *wait* and *notify* calls. In Figure 2 two components are depicted of which one components models a TDM scheduler and the other the execution of an actor. Both components exchange start and stop events which creates a kind of handshaking protocol between these components. The TDM scheduler component performs time-slicing and keeps track of the remaining budget for the actor in the current replenishment interval of the scheduler. The *ExecuteActor* component models the execution of a task and handles events from the scheduler indicating that execution of the actor should be stopped in order to model the effects of preemption.

The interaction between the *ExecuteActor* component and the *TDMscheduler* component is as follows. The actor waits until it can acquire the required input data from its input queues. After that it reads the input data and notifies the *TDMscheduler* component that it wants to execute by sending the *startActE* event. If there is budget for the actor left the scheduler notifies the actor by sending the *startSchedE* event that it can advance time till the budget is depleted or the actor completes its execution. A depleted budget is indicated by the scheduler by sending the *stopSchedE* event. That the execution of the actor is finished is indicated by sending the *stopActE* event by the *ExecuteActor* component. If there is remaining execution time then the inner while loop in the *ExecuteActor* component is repeated a number of times. If an actor has depleted its budget then it will receive a new budget in the next replenishment interval of the TDM scheduler. When the remaining execution time is zero a computation is performed on the input data and the results are written to the output queues, after which this output data is released which results in a notification for the consuming tasks. These operations do not advance the simulation time.

Other types of scheduling policies can be implemented without changing the *ExecuteActor* component, as shown in Figure 3.

In the FPP scheduling component *FPPscheduler* all actors are tested in a descending priority order whether they are enabled. If an actor is enabled it is notified with the *startSchedE* event that it can execute till a higher priority actor gets enabled or the execution of the actor is completed. After that the outer while loop is repeated and it is tested again for the highest priority actor that is enabled.

In the non-preemptive RR scheduling component *RRscheduler* the actors are executed in a predefined order. The scheduler component sends the *startSchedE* event to indicate that the actor can execute and waits till the *stopActorE* event before the next actor is allowed to execute.

```
ExecuteActor(){
  while(true){
    actor.acquire();
    actor.readData();
    set(requested);
    notify(startActE);
    remainingET=actor.ET();
    do{
      wait(startSchedE);
      startT=currT();
      wait(remainingET,stopSchedE);
      remainingET-=(currT()-startT);
    }while(remainingET>0);
    clear(requested);
    notify(stopActE);
    actor.processData();
    actor.writeData();
    actor.release();
  }
}
```

(a) Actor execution.

```
TDMscheduler(){
  while(true){
    startR=currT();
    foreach(actor){
      B=actor.getbudget();
      while(B>0){
        startT=currT();
        if(requested){
          notify(startSchedE);
          wait(B,stopActE);
        }else{
          wait(B,startActE);
        }
        B=currT()-startT;
      }
      notify(stopSchedE);
    }
    D=currT()-startR;
    wait(proc.RI()-D);
  }
}
```

(b) TDM scheduler.

Figure 2: TDM scheduling in HAPI.

5. CASE STUDY

A number of small didactic examples, as well as the dataflow graph of a WLAN 802.11p transceiver application, are presented in this section which illustrate the use and the capabilities of the HAPI simulator.[1]

The first example is the Homogeneous Synchronous Dataflow (HSDF) model shown in Figure 4 which models a producer and consumer task. The actor v_0 models the producer and v_1 the consumer. The actors have firing durations of 2 ns and 3 ns, respectively. The actors in the model have self-edges with a single token which prevents auto-concurrent execution because tasks cannot start their execution before their previous execution has been finished. The FIFO communication buffer between the tasks is modeled by two unbounded queues. The buffer is assumed to

[1]The HAPI simulator together with the C++ code of the examples in this paper are made available under the GPL licence.

```
FPPscheduler(){
  while(true){
    sortPriority(actorList);
    foreach(actor){
      if(actor.requested()){
        notify(startSchedE);
        Elist=actor.HPeventList();
        Elist=Elist OR stopActE;
        wait(Elist);
        notify(stopSchedE);
        break;
      }
    }
  }
}
```

(a) FPP scheduler.

```
RRscheduler(){
  while(true){
    foreach(actor){
      if(requested){
        notify(startSchedE);
        wait(stopActE);
      }
    }
  }
}
```

(b) RR scheduler.

Figure 3: FPP and RR scheduling in HAPI.

be initially empty and its capacity of 3 data containers is modeled by 3 initial tokens on the edge e_{10}.

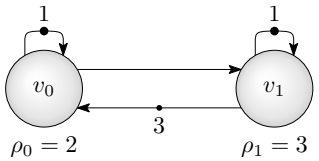


Figure 4: Producer-consumer example.

Both actors execute concurrently during self-timed execution in the HAPI simulator. This is also apparent in the simulation trace shown in Figure 5. Initially actor v_0 can execute every 2 ns because there are a sufficient number of tokens on the edge e_{10}. However after 10 ns its firings are delayed because it has to wait for productions by actor v_1.

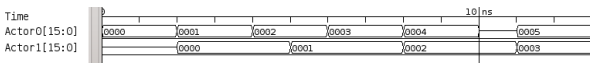


Figure 5: Trace of producer-consumer example.

A second example is shown in Figure 6 in which actor v_1 is forced to execute strictly periodically every period of 4 ns. An error is reported in the case that the actor is not enabled before it is executed. It is not enabled if there is less than one token in the queue e_{01}. The actor v_0 has a firing duration which varies randomly every execution between 1 and 5 ns. In this example it can occur that an error is reported because the firing duration of v_0 can be larger than the period of v_0.

An example of a task graph in which task τ_0 is scheduled by a TDM scheduler is shown in Figure 7. The task τ_0 has a computation time C_0 of 3 ns, a budget S_0 of 9 ns and a replenishment interval Q_0 of 18 ns. Task τ_1 is executed

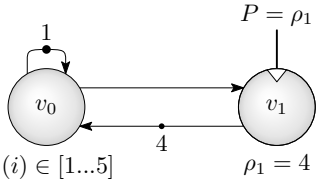


Figure 6: Dataflow graph with a periodic sink.

on a separate processor. Whether task τ_0 can continue its execution depends on the availability of remaining budget in the current replenishment interval.

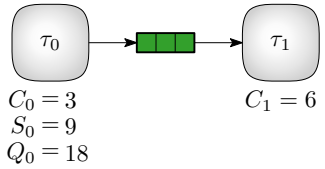


Figure 7: Task graph scheduled using TDM.

The simulation trace for the task graph in Figure 7 is shown in Figure 8. This figure shows that the processor becomes idle after τ_0 has finished its execution at $t = 3$ ns. In the interval between 3 ns and 9 ns τ_0 cannot execute, although it has budget left, because it is not enabled (indicated by "xxxx"). And in the interval between 9 ns and 18 ns τ_0 is enabled, but does not have budget left (indicated by "Xxxx"). Therefore it has to wait for a new budget in the second replenishment interval, which begins at $t = 18$ ns, to continue its execution.

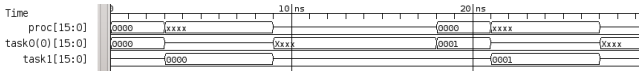
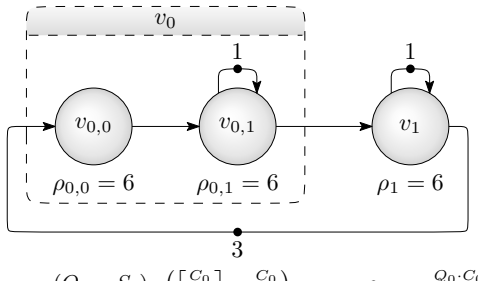


Figure 8: Trace of the TDM example.

The fourth example is the dataflow graph in Figure 9 which uses a latency-rate dataflow component that can accurately model the effects of TDM scheduling. Actor $v_{0,0}$ does not contain a self-edge and can therefore execute auto-concurrently. The firing durations of the actors correspond to the parameters in Figure 7.



$$\rho_{0,0} = (Q_0 - S_0) \cdot \left(\left\lceil \frac{C_0}{S_0} \right\rceil - \frac{C_0}{S_0} \right) \qquad \rho_{0,1} = \frac{Q_0 \cdot C_0}{S_0}$$

Figure 9: Latency-rate model of a TDM scheduled actor

The trace generated by HAPI given the dataflow graph in Figure 9 is shown in Figure 10. In this figure there is a separate trace for the latency and the rate actor that model the behavior of v_0. The counter of the latency actor $v_{0,0}$ is updated when its next firing starts. At $t = 0$ the counter value

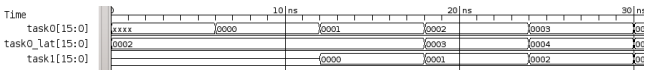


Figure 10: Trace for the latency-rate model.

equals to 2 because firing 0, 1 and 2 happen concurrently as a result of the 3 initial tokens.

The fifth example is the task graph shown in Figure 11. In this task graph the task τ_0 has a lower priority than task τ_2 and both tasks are executed on the same processor. Therefore, τ_2 can preempt τ_0.

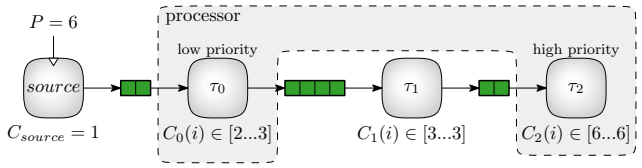


Figure 11: Task graph scheduled using FPP scheduling.

The simulation trace for this task graph is shown in Figure 12. The *proc* trace in this figure shows which task is executed on the processor at every point in time. From the trace it becomes apparent that τ_0 is preempted at $t = 9$ ns.

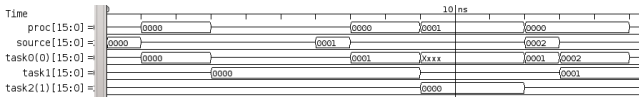


Figure 12: Trace of the FPP example.

Another example which makes use of FPP scheduling is shown in Figure 13. This example illustrates that tasks can have multiple inputs. The tasks have HSDF actor like semantics because they can only execute when there is at least one data container available in all their input buffers.

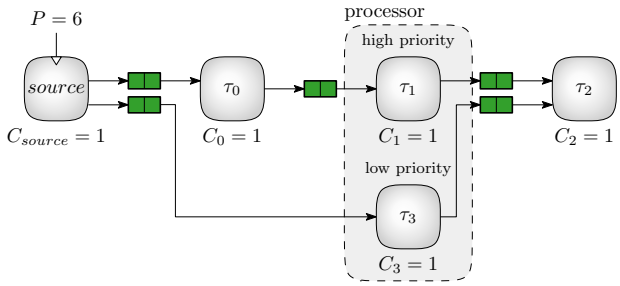


Figure 13: Tasks graph with a task that has multiple inputs.

In our last example we consider a WLAN 802.11p transceiver application. It is used in the automotive domain as part of safety-critical applications, which defines its hard real-time context. Furthermore, it is executed on a multiprocessor system for performance reasons. The application consists of multiple modes, of which we only consider the packet decoding mode whose dataflow graph is depicted in Figure 14.

Symbols arrive strictly periodic with a frequency of $f = 125$ kHz. Received symbols are first passed through a filter and then processed by a Fast Fourier Transformation (FFT), an equalizer, a demapper, a deinterleaver and finally a viterbi decoder. Furthermore, stability is improved by compensation of the domain-specific channel variation.

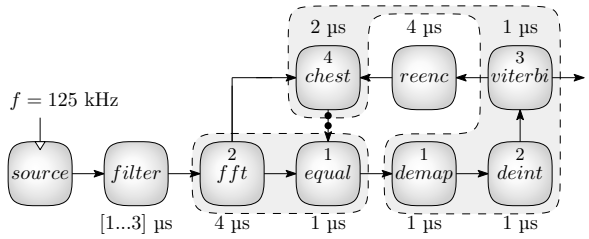


Figure 14: Dataflow graph of WLAN 802.11p transceiver.

This is realized by adjusting the equalizer for the reception of symbol n by an estimate of the channel during the reception of symbol $n-2$. Such a channel estimate is obtained by re-encoding the error-corrected bits of symbol $n-2$ and comparing them to the received symbol $n-2$. This results in the depicted feedback loop, limiting the maximum throughput of the packet decoding mode.

The tasks of the packet decoding mode are executed on four different processors. If multiple tasks share one processor then an FPP scheduler is used. Figure 14 depicts one possible task-to-processor mapping. The tasks *filter* and *reenc* are executed on separate processors and consequently cannot get preempted. The other tasks, however, use shared processors, as indicated by the dashed boxes. The priorities of the tasks executed on shared processors are denoted inside the corresponding actors, with 1 being the lowest and 4 being the highest.

Denoted next to the actors are the WCETs of the tasks (and also the Best-Case Execution Time (BCET) if it is differs from the WCET) and that include the time needed to send their data to tasks on other processors and which represent theoretical bounds obtained on the so-called Starburst platform [21]. As data and resource dependencies are the same as in the realization, whereas BCETs and WCETs are under-approximated and over-approximated, respectively, it holds that the model depicted in Figure 14 represents a valid abstraction of reality that complies to the refinement theory discussed in Section 3.

Simulating the presented example using HAPI produces traces that can occur in that abstraction of reality, of which one is depicted in Figure 15. As one can see the execution time of task *filter* varies between 1 and 3μs, resulting in different preemption patterns and consequently different end-to-end latencies (the times between executions of *source* and finishes of *viterbi*). The maximum end-to-end latency determined using simulation equals to 14μs. This latency is for instance determined for iteration 6 in Figure 14, with task *filter* having an execution time of 3μs and task *demap* being preempted for 2μs by task *chest* in iteration 5.

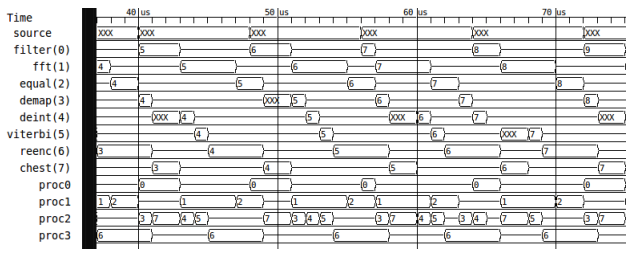


Figure 15: Trace of the WLAN 802.11p example.

As discussed in Section 3 the model used for analysis must be an abstraction of the model used for simulation.

The analysis technique presented in [22] is capable of using the same model as used in simulation, except that only the bounds on BCETs and WCETs are used, as opposed to the usage of all in-between values during simulation. For this example temporal analysis determines the same maximum end-to-end latency as our HAPI simulation. This shows on the one hand that in this case analysis is so accurate that its result matches the one obtained with simulation. On the other hand this also implies that HAPI simulation cannot be only used to determine average latencies, but can also trigger the worst-case.

Now consider that one makes a mistake when constructing the analysis model from the abstraction of reality. For instance assume that in the analysis model the priority of task *chest* is not set to the highest, but to the lowest on the shared processor. Then this faulty temporal analysis determines a maximum end-to-end latency of only 12 instead of $14\mu s$. The reason for this difference can be observed in Figure 16, which depicts a simulation trace with the same mistake: Now task *chest* in iteration 5 cannot preempt any of the tasks in iteration 6, but is itself preempted. As *chest* is not on the critical path a smaller end-to-end latency is determined.

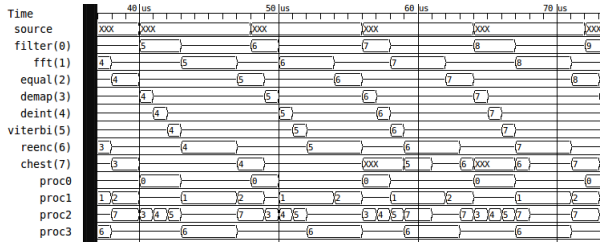


Figure 16: Trace of the WLAN 802.11p example with wrong priority of task *chest*.

Finally note that the HAPI simulation using the correct model determines an end-to-end latency of $14\mu s$, whereas the faulty analysis determines an end-to-end latency of only $12\mu s$. According to Figure 1 the higher simulated latency allows to conclude that the analysis must be indeed faulty.

6. CONCLUSION

In this paper we introduced the HAPI dataflow simulator which can be used to verify worst-case temporal dataflow analysis results. This is useful because the applied models and analysis techniques are created manually and can therefore contain errors. Verification of these results is possible as the analysis model and simulation model are derived from the same abstraction of reality. Furthermore, the HAPI simulator does not make use of periodic preemption points to handle asynchronous events and therefore does not introduce additional pessimism in the simulation results. Therefore it must hold that the traces obtained with simulation must refine the traces computed with analysis. If this is not the case then it can be concluded that the dataflow analysis results are optimistic and thus incorrect.

The HAPI simulator operates between the CP level and the PV level because it simulates dataflow process networks and additionally also preemptive scheduling decisions. This allows to use the simulator for getting insight in the cause of delayed productions. An impression of the over-estimation made by analytical techniques can be obtained by comparing the difference in production times found with the simulator and the computed analytical results.

With a number of small didactical examples, as well as the dataflow graph of a WLAN 802.11p transceiver application, we demonstrated features of the HAPI simulator.

7. REFERENCES

[1] A. Donlin, "Transaction level modeling: Flows and use models," in *Proc. Int'l Symposium on Hardware/Software Codesign (CODES)*, 2004, pp. 75–80.

[2] M. Bekooij, S. Parma, and J. van Meerbergen, "Performance guarantees by simulation of process networks," in *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2005.

[3] M. Wiggers, M. Bekooij, and G. Smit, "Monotonicity and run-time scheduling," in *Proc. Int'l Conf. on Embedded Software (EMSOFT)*, 2009, pp. 177–186.

[4] A. Hansson, K. Goossens, M. Bekooij, and M. Wiggers, "CoMPSoC: A composable and predictable multi-processor system on chip template," *ACM Transactions on Design Automation of Electronic Systems*, 2009.

[5] B. Dekens, M. Bekooij, and G. Smit, "Low-cost guaranteed-throughput dual-ring communication infrastructure for heterogeneous MPSoCs," in *Design and Architectures for Signal and Image Processing (DASIP)*, 2014.

[6] OSCI, "SystemC," http://www.systemc.org.

[7] E. Lee and T. Parks, "Dataflow process networks," in *Proceedings of the IEEE*, May 1995.

[8] Synopsis, "Platform Architect," http://www.synopsys.com.

[9] B. D. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten, "Software/hardware engineering with the parallel object-oriented specification language," in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 2007, pp. 139–148.

[10] E. Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers, "YAPI: Application modeling for signal processing systems.," in *Proc. Design Automation Conference (DAC)*, Los Angeles, June 2000, pp. 402–405.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings IFIP Congress*, 1974, pp. 471–475.

[12] K. Yu, "Real-time operating system modelling and simulation using SystemC," Ph.D. dissertation, University of York, 2010.

[13] H. Posadas, E. Villar, F. Blasco, P. Tecnológico, and S. Paterna, "Real-time operating system modeling in systemc for hw/sw co-simulation," in *Proc. Conference on Design of Circuits and Integrated Systems*. Citeseer, 2005.

[14] J. Hausmans, "Abstractions for aperiodic multiprocessor scheduling of real-time stream processing applications," Ph.D. dissertation, University of Twente, 2015.

[15] M. Wiggers, M. Bekooij, and G. Smit, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2007, pp. 11–22.

[16] M. Steine, M. Bekooij, and M. Wiggers, "A priority-based scheduler with conservative dataflow model," in *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2009, pp. 37–44.

[17] J. Staschulat and M. J. G. Bekooij, "Dataflow models for shared memory access latency analysis," in *Proc. Int'l Conf. on Embedded Software (EMSOFT)*, 2009, pp. 275–284.

[18] A. Lele, O. Moreira, and P. J. Cuijpers, "A new data flow analysis model for TDM," in *Proc. Int'l Conf. on Embedded Software (EMSOFT)*, 2012, pp. 237–246.

[19] R. L. Graham, *Bounds on the performance of scheduling algorithms*. John Wiley and Sons, 1976, pp. 165–227.

[20] http://gtkwave.sourceforge.net/.

[21] B. Dekens, P. Wilmanns, M. Bekooij, and G. Smit, "Low-cost guaranteed-throughput communication ring for real-time streaming MPSoCs," in *Design and Architectures for Signal and Image Processing (DASIP)*, 2013.

[22] P. Kurtin, J. Hausmans, and M. Bekooij, "Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, accepted for publication.