REAL-TIME EMBEDDED SYSTEM MODELING BY INTRODUCING HARDWARE-IN-THE-LOOP CONCEPT TO SYSTEMC

by

Doğan Fennibay

BSc., Computer Engineering, Istanbul Technical University, 2006

BSc., Industrial Engineering, Istanbul Technical University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2010

REAL-TIME EMBEDDED SYSTEM MODELING BY INTRODUCING HARDWARE-IN-THE-LOOP CONCEPT TO SYSTEMC

APPROVED BY:

Assoc. Prof. Arda Yurdakul .....
(Thesis Supervisor)

Assist. Prof. Alper Şen .....

Assist. Prof. Faik Başkaya .....

DATE OF APPROVAL: 31.8.2010

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Arda Yurdakul, for providing invaluable insight to my work and her endless drive for perfection in review iterations. I also would like to thank Assist. Prof. Alper Şen, for his visionary contributions and his time; and Assist. Prof. Faik Başkaya for his participation at the examining committee.

I would like to thank the Corporate Technology division of Siemens Turkey, especially Ozan Özlü, for their support to my work in terms of infrastructure and for providing an industry perspective.

I would like to thank the CASLAB team, foremost Salih and Muhammet, for helping me in setting up the experimental environments. I would also like to give my special thanks to Suzan Bayhan for sharing a couple laughs at stressful times.

I would like to thank my family and dedicate this work to my mother Fatma Fennibay.

Last but not least, I thank Balca for cheering up and enlightening my world alone with her presence, not to mention her support and motivation during hard times and her joy.

ABSTRACT

REAL-TIME EMBEDDED SYSTEM MODELING BY INTRODUCING HARDWARE-IN-THE-LOOP CONCEPT TO SYSTEMC

As the demand for interaction of embedded systems with other systems is constantly increasing, the need to extend the model of the embedded system to include the other systems that are being interacted with is increasing, too. This results in degraded accuracy of the whole model and increased modeling effort. New modeling techniques have to be developed to reduce design effort without decreasing overall system accuracy.

On the other hand, complexity and time-to-market constraints demand early simulation, verification, and architectural exploration of systems. Hence, in this dissertation, a new design concept and new methods have been proposed to apply the hardware-in-the loop technique to the field of hardware/software co-design of industrial embedded systems using SystemC as the modeling environment. First of all, the hybrid channel has been conceptualized to clearly define the communication between real and virtual (modeled) subsystems. For real to virtual communication, novel methods have been developed for incorporating external events to the SystemC simulation. Additionally, a method has also been proposed for generating concurrent outputs from virtual to real subsystems as timely as possible. SystemC kernel has been patched for hard real-time execution and the underlying operating system has been ameliorated to guarantee an upper bound for the overall system latency. Furthermore, a mathematical model has been set up to estimate the execution performance of a given model. The performance of the proposed set of methods has been experimented on some industrial embedded systems. A stable operating frequency of 10 KHz and an I/O performance of sub-millisecond round-trip time over Ethernet have been observed. In an experiment to

observe the method's performance in a real-life environment, a non-timed transaction-level model of a BACnet Broadcast Management Device (BBMD) interacting with real devices outperformed a competing real system up to 80 times in maximum response time.

ÖZET

SYSTEMC DİLİNE DÖNGÜ İÇİNDE DONANIM KAVRAMININ GETİRİLMESİ YOLUYLA GERÇEK ZAMANLI GÖMÜLÜ SİSTEMLERİN MODELLENMESİ

Gömülü sistemlerin diğer sistemlerle etkileşme ihtiyacının sürekli artmasıyla beraber gömülü sisteme ait modelin etkileşilen diğer sistemleri içine alacak şekilde genişlemesi ihtiyacı da artmaktadır. Bu tüm modelin doğruluğunu azaltır ve modelleme eforunu arttırır. Genel model doğruluğunu azaltmadan tasarım eforunu düşürecek yeni tekniklerin geliştirilmesi gerekmektedir.

Öte yandan, karmaşıklık ve piyasaya sürme süresi kısıtları; sistemlerin erken benzetim, doğrulama ve mimari keşifini gerektirmektedir. Dolayısıyla, bu tezde SystemC'nin modelleme ortamı olarak kullanıldığı gömülü sistemlerin donanım/yazılım ortak geliştirme alanında, döngü içinde donanım tekniğinin uygulanmasını sağlayacak yeni tasarım anlayışı ve metotlar önerilmiştir. Bunun için, öncelikle, gerçek ve sanal (modellenmiş) altsistemlerin arasındaki iletişimi açık bir şekilde tanımlayabilmek üzere melez kanal kavramı geliştirilmiştir. Gerçekten sanala iletişimde harici olayları SystemC benzetimine dahil edebilmek üzere özgün yöntemler öne sürülmüştür. Ek olarak, sanal altsistemlerden gerçek altsistemlere eşzamanlı çıktıların gerçeğe mümkün olan en yakın şekilde üretilmesi için bir yöntem önerilmiştir. Ayrıca, sıkı gerçek zamanlı çalışma için SystemC çekirdeği yamalanmıştır ve genel sistem gecikmesi için bir üst sınırı garantilemek üzere üzerinde çalışılan işletim sistemi iyileştirilmiştir. Ek olarak verili bir modelin çalışma başarımını tahmin etmek üzere bir matematiksel model geliştirilmiştir. Önerilen metotlar kümesinin başarımı bir dizi endüstriyel gömülü sistem üzerinde denenmiştir. 10 KHz kararlı çalışma frekansı ve Ethernet üzerinde bir milisaniyenin altındaki gidiş-dönüş süresinde bir Giriş/Çıkış başarımı sağlamıştır. Ayrıca yöntemin başarımını bir gerçek hayat ortamında gözlemek üzere yapılan deneyde,

gerçek aygıtlarla etkileşen bir BACnet Broadcast Yönetim Aygıtı'nın (BBMD) zamanlamasız işlem-seviyesi modeli, rakip gerçek sistemi azami yanıt süresinde 80 kata dek geçmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

A	Total number of time advance phases in the simulation
E_{ij}	Total number of evaluate phases before j'th update phase, which comes before i'th time advance phase
H	Total number of hybrid channels
O_i	Total number of outputs to real subsystems before i'th time advance phase
P	Total number of thread processes
T_S	Simulation clock
t_S	Current value of simulation clock
t_{Snew}	Next value of simulation clock
T_W	Wall clock, a.k.a. real-time clock
t_W	Value of wall clock when simulation clock is first set to t_S
$t_{Wactual}$	Value of wall clock after processing delta cycles and outputs
t_{Wcow_s}	Critical output window for a subset s of the set of all output channels, see t_{Wow}
t_{Wdelay}	Delay time necessary to keep the simulation clock and the wall clock synchronized
$t_{Wdeltaend}$	Value of wall clock at the end of delta-cycles
$t_{Wdeltacycles}$	Time needed for w.r.t the wall clock for processing delta-cycles
t_{Wnew}	Value of wall clock when simulation clock is first set to t_{Snew}
$t_{Woutput}$	Time needed for w.r.t the wall clock for processing outputs
t_{Wow}	Output window for the set of all output channels, i.e. the maximum amount of real-time to generate concurrent outputs
$t_{Wpassed}$	Time needed for w.r.t the wall clock for processing delta-cycles and outputs
U_i	Total number of update phases before i'th time advance phase

ADEOS	Adaptive Domain Environment for Operating Systems
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface

EV	Evaluate phase
FIFO	First in first out
gdb	GNU Debugger
GPOS	General-purpose Operating System
HDL	Hardware Description Language
PCI	Peripheral Component Interconnect
PLI	Programming Language Interface
POSIX	Portable Operating System Interface
RDI	Remote Debugging Interface
RTAI	Real-Time Application Interface
RTL	Register Transfer Level
RTOS	Real-time Operating System
SCSI	Small Computer System Interface
SoC	System-on-Chip
TA	Time advance phase
TAPI	Transaction Application Programming Interface
TLM	Transaction Level-Model/Modeling
UP	Update phase
UR	update real phase
Verilog	An HDL
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuits

1. INTRODUCTION

System-level modeling is a relatively new approach in the development process (from architectural exploration to verification) since systems are getting more integrated [1]. In this trend, hardware and software are also getting closer, hence hardware/software co-design is increasingly employed and system-level modeling comprises modeling of hardware and software together. Systems under development are usually complex. As a result, current design trend is the employment of models and modular/component-based strategies.

Two additional trends in embedded systems are that [a] they are increasingly connected with other embedded systems and [b] they increasingly contain off-the-shelf components to shorten the time-to-market and reduce development costs. Other embedded systems and off-the-shelf components are referred as *real subsystems*. These trends imply that real subsystems also have to be modeled even though their implementations exist. Modeling of such elements has no added value as it multiplies the intrinsic disadvantages of modeling, i.e. inaccuracy due to abstraction and additional effort.

This thesis argues that real subsystems should not be modeled in the systems under development because they are already implemented and these implementations should be integrated with the system-under-development. A well-known method which is used in the test of implemented embedded systems is hardware-in-the-loop [4]. The thesis introduces the same concept to the hardware/software co-design of embedded systems. Novel communication mechanisms between virtual and real subsystems are defined to implement hardware-in-the-loop (Figure 1.1) method for hardware/software co-design. Real-time behavior for virtual subsystems is also introduced as most communication mechanisms between subsystems rely on a common notion of time such as a timeout mechanism in a communication protocol.

Real-time behavior consists of synchronizing the clocks of the virtual subsystems

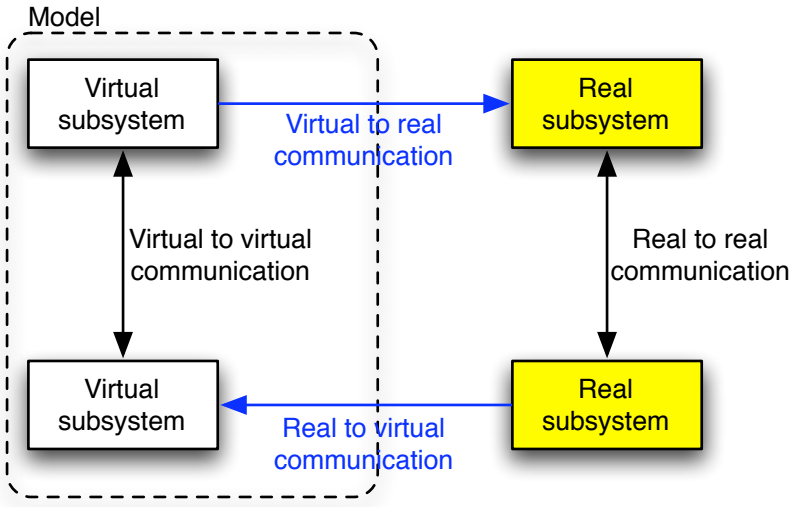


Figure 1.1. Types of communication in a hardware-in-the-loop setup

with the clocks of the real subsystems and achieving determinism in the overall system. Non-deterministic behavior is present in all computing platforms with an operating system and decreases the accuracy of the model by introducing latency and by increasing the response time randomly. Achieving determinism in the overall system requires bounding such latencies to limit the effects on the accuracy of the models.

SystemC is deemed as the most appropriate platform to introduce the hardware-in-the-loop technique to hardware/software co-design, because it is widely used and standardized [5]. This study introduces the *hybrid channels* concept by extending SystemC's channel construct to clearly specify the communication between real and virtual subsystems. Additionally, the SystemC kernel is adapted to execute the simulation in real-time and improve the underlying operating system to limit the system latency. In addition, special mechanisms are added to properly manage timing of communication between virtual and real subsystems.

The proposed method is evaluted in the domain of industrial applications, more specifically industrial communication. The basic reasons are: [a] there exist industrial communication standards that have very strict hard real-time constraints such as PROFINET [6, 7]; [b] the data exchange rate in industrial communication ranges

up to 10 KHz, and this is achievable with current computing systems executing the models; and [c] system-level modeling is starting to be a design trend in this domain. Our method can also be applied directly to the design of SoCs if computation power of the modeling platforms becomes sufficient for executing the complex SoC models in real-time. There is a vast amount of research to speed up simulation of SystemC models via parallelization or optimization. In addition, the increasing use of system-level modeling for software, which has lower execution speed, will also be an application area for our method.

The organization of this thesis is as follows: The next chapter will provide an overview on the preliminaries of hardware/software co-design, which is followed by another chapter covering related work. The fourth chapter explains our solution and the fifth chapter provides information on the experimental evaluation of the proposed solution. The final chapter will conclude the thesis by discussing the obtained results and listing future work.

2. PRELIMINARIES OF HARDWARE/SOFTWARE CO-DESIGN

2.1. SystemC

SystemC is a language developed for system-level modeling used mostly for system-on-chip (SoC) systems. Models created with SystemC can be used for various kinds of simulation ranging from non-timed functional simulation, where timing is not modeled at all, to cycle-accurate simulation, where every clock cycle is modeled. Fundamentals of SystemC are explained completely in [1] and it will be used throughout this section.

SystemC is devised as the merge of register transfer level (RTL) models mostly written in an hardware description language (HDL) and transaction-level models (TLM) being developed in C/C++. It is focused on system-level modeling but the model can be refined, too. SystemC can model the whole system with different abstraction levels inside the system. For example, a model can consist of sub models with abstractions at RTL and transaction-level; furthermore some subsystems might even be real as in our approach, or be modeled in some other language like VHDL.

In comparison to the C/C++ models, SystemC provides a more structured approach. Instead of varying kinds of abstraction structures depending on the developers choices, SystemC offers a clear set as seen in Figure 2.1.

- Modules function as containers for other language elements including other *modules*. So hierarchical structures - via composition - can be built with SystemC.
- Events (*sc_event* from here on) are the basic means of synchronization between *processes*. They provide the *wait* and *notify* methods.
- Processes are the execution paths inside SystemC. They have a notion of *sensitivity*, which is basically a list of *sc_events*, from which the *processes* want to be notified in case of changes or other events. Two important *process* types

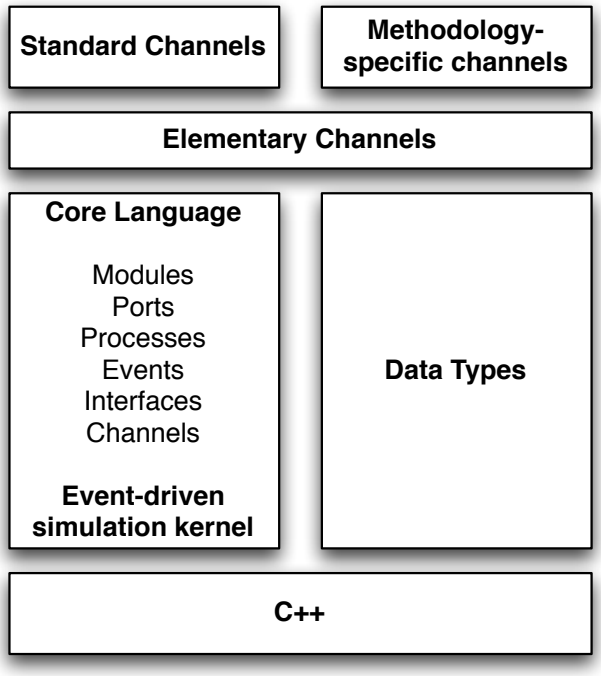


Figure 2.1. SystemC Architecture [1]

are *method* and *thread* processes. *Method processes* can only execute due to an *sc_event* in their sensitivity list and may not *wait* during the execution. *Thread processes* on the other hand are independent flows of execution, so they are allowed to make *waits*.

- Channels are the formal means of communication between *modules*. Employing other means for inter-*module* communication harms the reusability of the model. Communications that can be modeled with a *channel* ranging from a very simple mutual exclusion mechanism to very complex hierarchical communications, e.g. a PCI (Peripheral Component Interconnect) bus.
- Interfaces are entry points to *channels* from *modules*; they serve to hide the unnecessary information from the *channels*.
- Ports are connection points of modules to *channels*; they explicitly specify the communication needed by the *module*.

SystemC specification offers a non-deterministic scheduler, which allows modeling concurrent systems without any third party methods. It is a non-preemptive, non-deterministic scheduler without a notion of priority. The scheduled *process* is

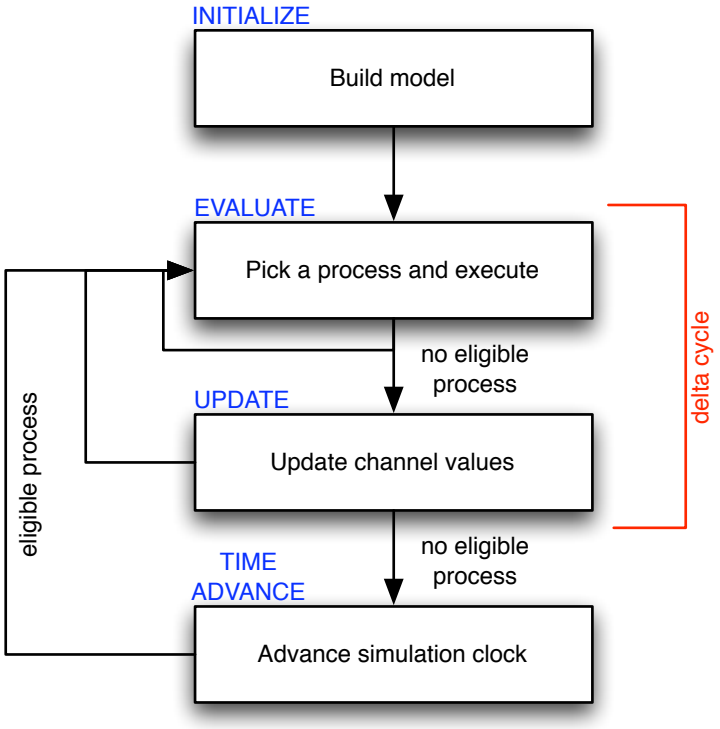


Figure 2.2. Flow of SystemC scheduler [1]

executed until it willingly gives up the execution resource via a *wait* or termination (non-preemptive). Among active[1] *processes*, the one, which will be selected for execution, is unspecified to the model developer (non-deterministic) and the developer cannot affect this decision by any means (no notion of priority).

SystemC kernel's execution consists of four main phases: *initialize*, *evaluate*, *update* and *time advance* (Figure 2.2). *Evaluate* and *update* phases form a *delta-cycle*. Only an infinitesimal amount of time is assumed to pass in a delta-cycle, so it is a "zero time advance" cycle. The result of operations done in the *evaluate* phase are not updated until the *update* phase. This mechanism allows the single-threaded simulation kernel to model concurrent operations.

SystemC simulation kernel is a discrete event simulator (Figure 2.3) [1], the *simulation clock* is advanced in discrete time intervals and changes in the state only happen

[1]A process not waiting for a time period to elapse or for an event to be notified is active, i.e. eligible for execution.

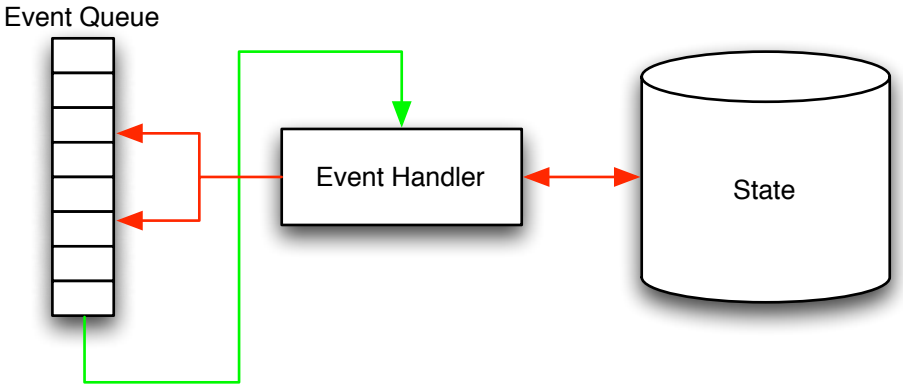


Figure 2.3. Basic architecture of a discrete event simulator

at discrete points in time. An *event* happens at a point in time, specifies changes to the *state* of the simulation and adds/removes elements to/from the *event queue*. Event queue holds an ordered list of events according to their time stamps. *Event handler* processes events in order and executes changes specified by events. The simulation clock is part of the state and is advanced according to the time stamp of the next event in the event queue. [8].

It should be stressed that a *SystemC event* is not the same thing as an *event in a discrete event simulator*. A SystemC event only represents an occurrence in the simulation model, so that *processes* in the model can *notify* others via SystemC events or *wait* on SystemC events. To avoid confusion, a SystemC event will always be referred with *sc_event*, and an event in a discrete event simulator will be referred with only *event*.

2.2. Real-Time Simulation

Two different clocks are involved in a discrete event simulator: the *simulation clock* representing the virtual clock of the model and the *wall clock* (a.k.a. real-time clock) representing the physical time passing during the execution of the simulation model [8]. Real-time simulation consists of establishing the relationship given in Equation (2.1) between the simulation clock T_S and the wall clock T_W, so that the advance of simulation clock is bound to the wall clock [9]. It should be noted here that the

previously mentioned notion of delta-cycle poses a difficulty for real-time simulation as it is impossible to suspend the real-time clock to allow cycle executions where no time is assumed to pass.

$$\frac{T_S - T_{Sstart}}{T_W - T_{Wstart}} = 1 \tag{2.1}$$

As mentioned, SystemC uses an event-driven simulation kernel. However, all events are in the simulation model. There is no interface for an external event, e.g. new data arriving via a hybrid channel to the model, in the current implementation. For example, if the event queue contains events with time stamps 00:02 and 00:06; the simulation clock will be advanced from 00:02 to 00:06 directly. So, if data has arrived from real subsystems to the model at 00:04, it will first be received at 00:06 by the model. This issue decreases accuracy of the model and should be addressed.

A factor affecting accuracy is the nondeterministic behavior of the system. Due to inherent latencies, all computing platforms demonstrate a deviation from real-time, so the response time of the platform cannot be determined 100%. Assuring deterministic behavior consists of guaranteeing an upper bound for the system latency, which can be defined as the time needed for a system to react whenever there is an action on it. Real-time schedulers guarantee that routines handling actions are executed when necessary, but regions preventing schedulers to manage resources such as interrupt service routines, a.k.a. non-preemptible sections increase system latency. Solutions proposed in [10] and [11] decrease the system latency by increasing preemptibility.

2.3. Transaction-Level Modeling

As mentioned earlier, SystemC has abstraction levels ranging from TLM to RTL. TLM is good for fast model execution with low modeling effort, while RTL models are highly accurate and can also be used as a replacement of hardware description

languages like VHDL or Verilog. Another positive aspect of SystemC is the plug and play behavior: previously designed models can be plugged to each other for building larger models. Furthermore different parts of models can have different abstraction levels. Finally, SystemC does not foresee a division of software and hardware; this is merely an issue of realization of the modeled system.

According to [12], TLM is a preferred way of modeling, especially, because of its ability to provide a platform for development at an early point. Therefore it is widely used for system-level design exploration/verification and afterwards for block-level models which are derived from the system level models. Most TLMs and the TLM proposed by [12] are *untimed*, i.e. the model does not take the time in account and focuses on the functionality.

The approach of [13] proposes an intermediate method: Result oriented modeling allows to remain at the transaction level, yet still increase the model accuracy. This is achieved by estimating the effects of the inner workings by looking at the current state, e.g. the bus transaction times are estimated according to the load situation on the bus. Thereby transaction level models can be changed to be *timed* without slowingdown. However the model of the estimator should be either known beforehand or developed [13].

3. RELATED WORK

Related work is addressed under four categories: [a] existing hardware-in-the-loop techniques, [b] studies integrating different modeling environments, [c] studies related to timing concerns and [d] studies with the objective of improving platform's determinism.

3.1. Current Hardware-in-the-Loop Techniques

As a well-established technique, hardware-in-the-loop has well-established tools. Most famous ones among them are MathWorks' solutions xPC Target [14] and Real-Time Windows Target [15], where the model is executed on a dedicated system or on a Windows system, respectively. However, the modeling language provided by these solutions has not been designed for hardware/software co-design purposes, so it lacks the necessary constructs and mechanisms that are already present in SystemC. Our work is orthogonal to such existing methods, as it is proposing to introduce the technique to the field of hardware/software co-design with a much more powerful modeling language, namely SystemC.

3.2. Integration of Different Environments

There have been several studies regarding the integration of different environments, i.e. enabling different modeling environments to interact with each other and also enabling interactions between models and real systems. Figure 3.1 shows an overview of them.

3.2.1. Integrating Different Virtual Platforms

These methods introduce concepts and software mechanisms to integrate different modeling platforms. As there are no real subsystems involved, there is no need for real-time behavior, but the need for synchronizing executions of virtual environments.

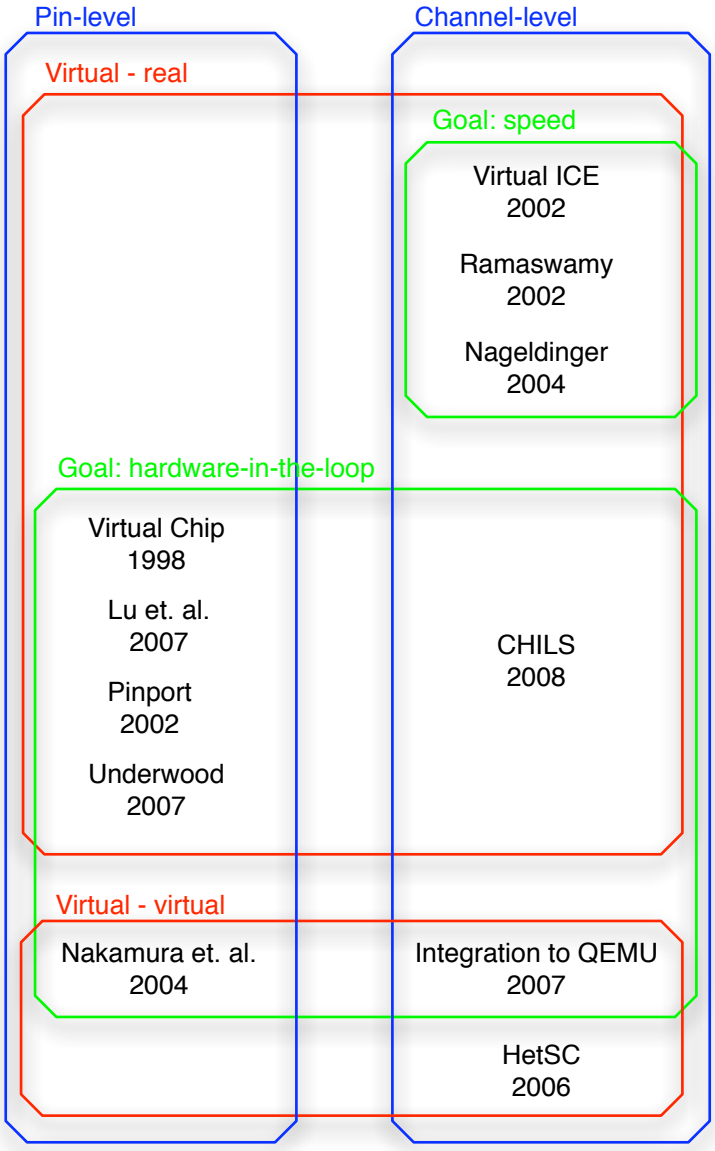


Figure 3.1. Overview of some of the related work for integration of different environments

HetSC [16] accomplishes to integrate multiple models of computation in a SystemC model together, which allows more accurate modeling of complete systems. This study only deals with integration of parts of a SystemC model and not integration of a SystemC model with external environments.

The work in [17] aims to integrate QEMU emulation environment and SystemC. It employs a SystemC module instead of a SystemC channel for representing the com-

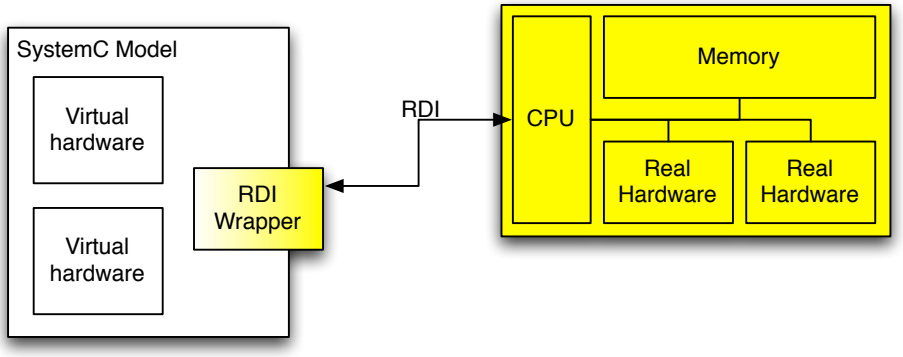


Figure 3.2. Architecture of Virtual ICE [2]

munication, which could cause design difficulties as SystemC foresees use of channels for communication purposes. In [17], an additional channel is necessary to connect the integration module to the rest of the model. However, this is a double effort. The integration can be directly implemented with an hierarchical SystemC channel.

3.2.2. Integrating Real and Virtual Environments

A major decision point in the integration of real and virtual subsystems is the level of abstraction. The communication between real and virtual subsystems can range from pin-level to transaction-level. The works in [3, 18, 19, 20, 21] are examples of the pin-level communication, and the works in [2, 22, 17, 23, 24] are examples of the transaction-level communication. It should be noted that not all approaches address the hardware-in-the-loop method; [18, 2, 24] propose connecting models to real subsystems only to increase the overall execution performance by using the real subsystems as coprocessors.

Each work has interesting properties that show the variety of possible answers to the question of which level of abstraction to use for modeling the communication. The hardware-in-the-loop framework proposed by Underwood [19] does the integration at the analog signal level via A/D, D/A converters, while Virtual Chip [3] and PinPort [21] use pins directly. Work in [18] employs register-level transfers, which can be considered as pin-level. Pin-level approaches offer great flexibility, as any communication protocol

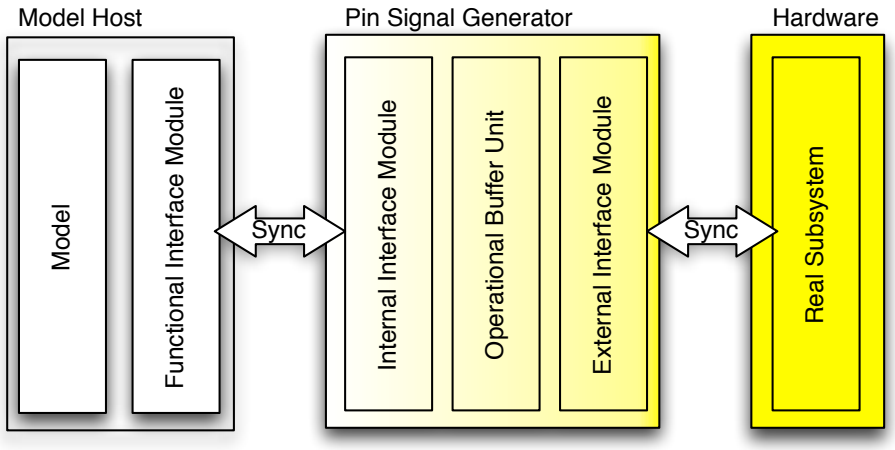


Figure 3.3. Architecture of Virtual Chip [3]

can be modeled on top of pin-level when necessary. However, when the subsystem using the communication protocol is in focus rather than the protocol itself, modeling the protocol will be costly, it will also degrade accuracy and the execution speed of the whole model unnecessarily.

Approaches that prefer transaction-level communication between real and virtual subsystems allow to skip the modeling of the details of the communication and focus on other interesting parts. Virtual In-Circuit Emulator [2] and Chip Hardware-in-the-Loop Simulation [22] use the remote debugging interface of a microprocessor to integrate it to the simulation model. This approach works well for microprocessors, however it does not address other hardware-in-the-loop configurations. The approach in [17], is more flexible in that aspect, it is already able to model two bus systems, namely Advanced Microcontroller Bus Architecture (AMBA) and Peripheral Component Interconnect (PCI).

3.3. Timing Concerns

Virtual Chip has a contribution in the timing management between real and virtual subsystems. The authors devise a three-level device integrating the model and the real system. The device consists of the Internal Interface Module, the Operational Buffer Unit and the External Interface Module. With respect to the timing, behaviors

of the Internal and the External Interface Module are synchronized with the model and the real subsystem, respectively. Operational Buffer Unit connecting both interface modules handles the timing difference via buffering methods [3].

Realtimify is an approach to real-time execution of SystemC models. Basically, a module is added to the model which synchronizes the simulation's execution to real-time with the objective to monitor the execution in real-time [25]. The approach is very lean and satisfactory for observing the model's execution in real-time. However, it does not address the issue of determinism as interaction with real subsystems besides human interaction is not targeted. Additionally the approach is intrusive as it requires changes in the model. Finally, it relies on the non-deterministic SystemC scheduler to execute the synchronization, which results in uncertainty about when the model will be synchronized.

3.4. Deterministic Behavior

Operating system plays a critical role in the issue of determinism. Real-time operating systems (RTOS) specialize in providing deterministic behavior, but they lack the variety of applications, I/O interfaces and functionality provided by a general purpose operating system (GPOS). Linux with real-time improvements seems as a promising tradeoff. Real-time Application Interface (RTAI) [11] which is used in [20] is built on top of Adaptive Domain Environment for Operating Systems (ADEOS) [26] and does time sharing with a Linux kernel. It provides real-time behavior by itself while leaving the resources to the Linux kernel for non-critical tasks. On the other hand RT_PREEMPT [10] employs a more direct method in which latency is decreased by increasing preemptibility throughout the Linux kernel. RT_PREEMPT offers the advantage of providing the same API (Application Programming Interface) as the standard Linux kernel.

4. HARDWARE-IN-THE-LOOP FOR HARDWARE/SOFTWARE CO-DESIGN

Figure 4.1 shows the architecture of our solution. Virtual subsystems, i.e. models, run on top of a simulation kernel, which runs on a GPOS. GPOS runs on a computer hardware. The triplet formed by the simulation kernel, the operating system and the computer hardware is called as the *modeling platform*. Our solution addresses two aspects of the problem: [a] achieving real-time behavior and [b] integrating real and virtual worlds.

4.1. Achieving Real-Time Behavior

The simulation kernel is patched at the point where the time advance is done (Figure 4.2). At this point, simulation clock is still t_S and is about to be advanced to t_{Snew}, while wall clock has advanced from t_W to $t_{Wactual}$ due to the delta cycle processing time. In order to satisfy Equation (2.1), our patch delays the execution of the simulation by an amount of t_{Wdelay}, which advances the wall clock to t_{Wnew}.

To assure determinism, the underlying operating system's preemptibility is improved with patches further sources of latency such as swap memory and power management are disabled. Additionally, the priority of threads executing the model is increased to guarantee availability of resources. Single-threaded execution model of SystemC guarantees that the contention among threads can only happen in hybrid channels, which can employ additional threads. So the hybrid channels are designed carefully to avoid excessive contention.

4.2. Hybrid Channels: Integrating Real and Virtual Subsystems

The channel concept of SystemC is extended so as to realize the hybrid channel functionality, i.e. realizing the communication between the real and virtual subsystems. Furthermore, channel types are categorized and the class hierarchy shown in Figure 4.3

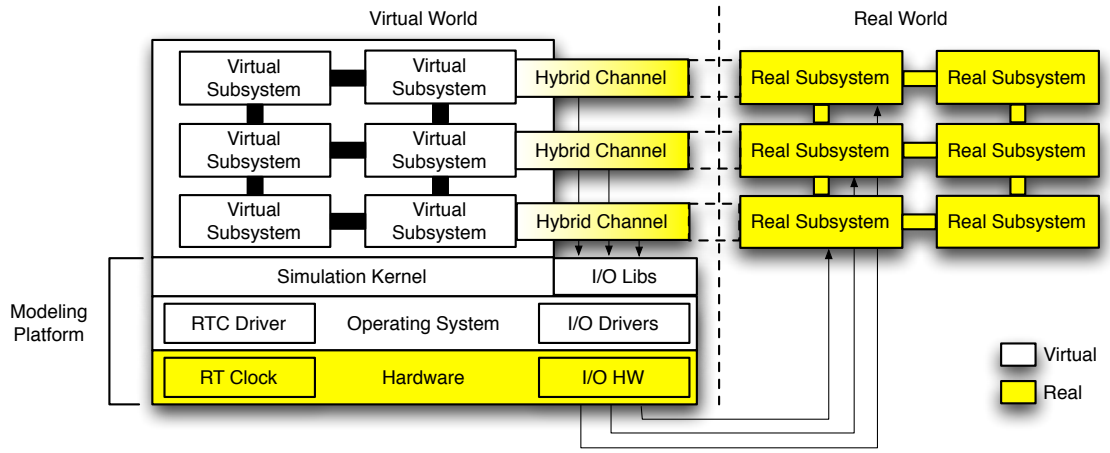


Figure 4.1. Architecture of our solution

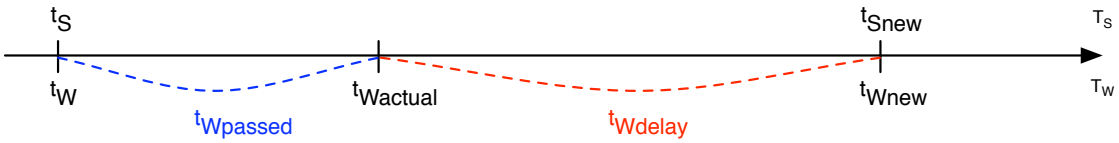


Figure 4.2. Real-time patch to simulation kernel

is devised. *dsc_hybrid_channel* class at the base of the hierarchy serves for distinguishing hybrid channels from other channels and implements the *update real* functionality, which will be explained in the next paragraphs. In SystemC, a channel can inherit from *sc_prim_channel* (primitive channel) or *sc_module* (hierarchical channel). As hierarchical channels offer a superset of primitive channels' capabilities [1], *sc_module* is chosen as the base of *dsc_hybrid_channel*. A hybrid channel can carry digital or analog data, which can be specified by choosing the appropriate subclass *dsc_digital_hybrid_channel* or *dsc_analog_hybrid_channel*. Digital channels can transfer data in a parallel way, e.g. the parallel port, or in a serial way, e.g. Universal Serial Bus (USB). Hence two further subclasses are provided for specifying this characteristic.

4.2.1. Interactions from Virtual to Real Subsystems

Output values determined in the virtual subsystems can be transferred to the real subsystems in *evaluate*, *update* or *time advance* phases (Figure 2.2 and Table 4.1). The constraints of the channel model dictates the best phase for the transfer:

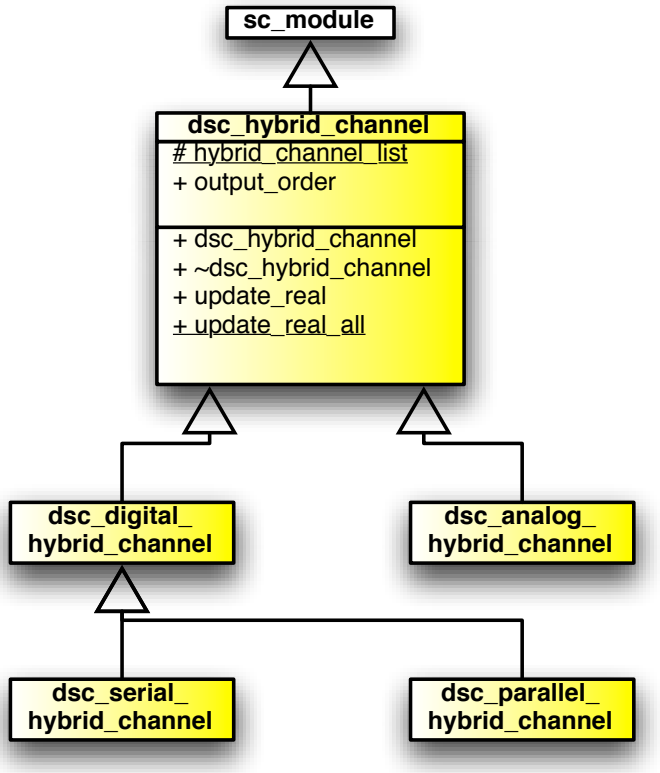


Figure 4.3. Class diagram of hybrid channels

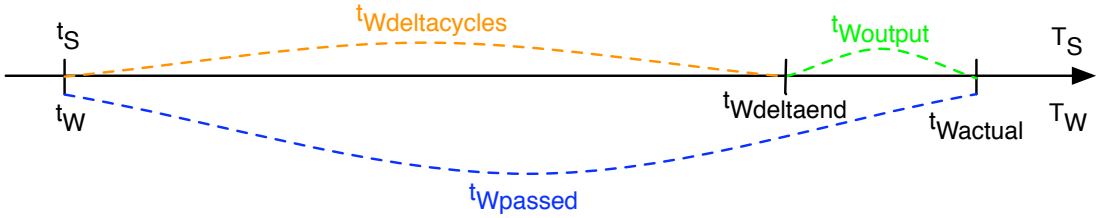


Figure 4.4. Detail of Figure 4.2

- Evaluate: The data may be transferred as soon as it is produced. (e.g. a fifo channel whose current value will not be affected by values in later delta-cycles.)
- Update: There might be several processes that affect the final value of an output variable. In that case, the data should not be written to the real subsystem until the final stable value is reached. If multiple successive delta-cycles change the data in the channel, real subsystems can observe this. (e.g. a signal channel whose actual value is established at the end of a delta-cycle)

Table 4.1. Comparison of timing alternatives for model outputs

Phase	Advantages	Disadvantages
Evaluate	Data do not wait at all. Glitches occurring at the end of delta-cycles can be relayed to real devices immediately.	Data must not change in subsequent cycles, e.g. sc_fifo. Delta-cycle processing time increases.
Update	The final data from concurrent processes are used. Glitches occurring at the end of delta-cycles can be relayed to real devices.	Data wait until the *update* phase. Delta-cycle processing time increases. Concurrent outputs are distributed over a larger window in real-time.
Time advance	The final data from concurrent processes are used. Fewer output values are used. Concurrent outputs calculated by delta-cycles are gathered together in real-time	Data wait until the *time advance* phase. Glitches occurring at the end of delta-cycles are not relayed to real devices

- Time advance: Due to the sequential operation of SystemC simulation kernel, concurrent outputs cannot be transferred to the real subsystems simultaneously. When output values are transferred in the *evaluate* or *update* phase, real subsystems observe them at time points distributed in $t_{Wdeltacycles}$ shown in Figure 4.4. Delaying the transfer until the *time advance* phase will gather the output points in time together in $t_{Woutput}$, which is much smaller than $t_{Wdeltacycles}$. So, outputs that are simultaneous regarding to the simulation clock are generated in a smaller time window regarding the wall clock. This option has another advantage of reducing simulation's execution effort, because the number of I/O operations are reduced. As a disadvantage, delta-delay changes are not observable by real

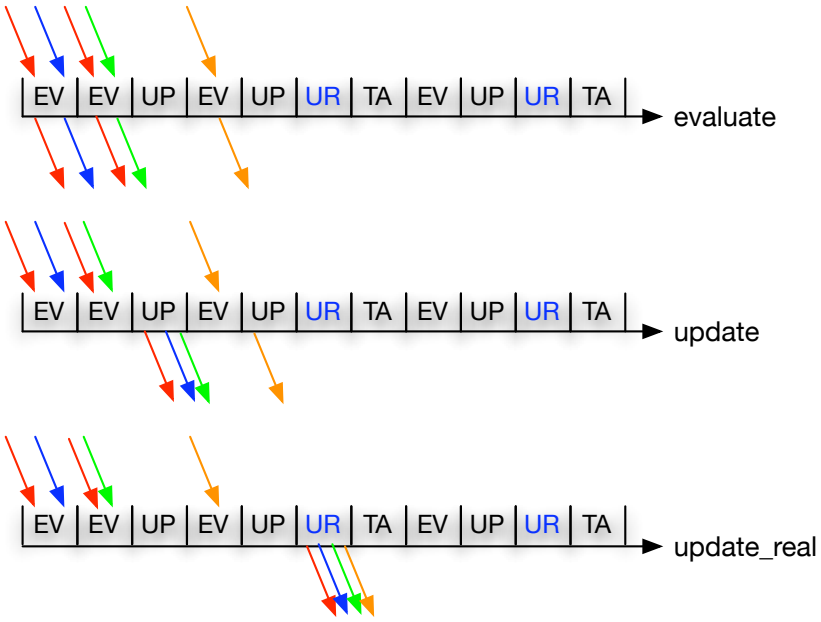


Figure 4.5. An example set of outputs in different output timings

subsystems in this scheme. However, delta-delay changes may not always need to be relayed to the real subsystems. In other words, a system that has to be insensitive to the transient behavior of the signals at its inputs benefits from this method.

Figure 4.5 shows an example set of outputs in different output timing settings. Here, output value computations are done only in *evaluate* (EV) phases. Then they can be timed to be executed in *evaluate* (EV), *update* (UP) or *update real* (UR) phases. *Time advance* (TA) phase always pairs with exactly one UR phase, and is solely used for advancing the time.

SystemC already offers methods for transferring the output values in *evaluate* or *update* phases. Our work further provides the method *update real* which is called at the *time advance* phase prior to the advance of the simulation clock. The developer of the hybrid channel can choose the appropriate output timing for the type of the channel.

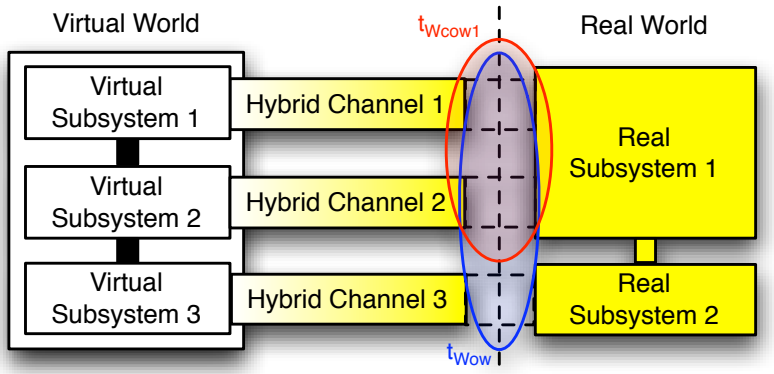


Figure 4.6. An example for concurrent output window constraint

If the hardware used by the hybrid channel is capable of executing multiple output executions simultaneously, e.g. a digital I/O device with multiple output channels, this capability can be leveraged by the hybrid channel developer. In this scheme, hybrid channel classes accumulate the values calculated during *evaluate* and *update* phases and save them in the hardware device. Finally, at *update real*, the hardware is triggered to realize the accumulated outputs. This will achieve 100% real-time concurrency for the group of outputs using the same hardware.

Some measures can be taken to improve the behavior of concurrent outputs also without special support from hardware. For example, if two concurrent outputs are done via different output hardware, e.g. an Ethernet channel and a digital I/O line. If the output time window constraint is stricter for a subset of output channels, generating outputs of this subset successively without other outputs inbetween improves the behaviour in terms of concurrent outputs. Such a subset can be defined as a *critical output subset* and there can be multiple critical output subsets of the set of all outputs. Figure 4.6 shows an example. Here, hybrid channels 1 and 2 belong to a criticial output subset. The stricter time constraint of this critical output subset is modeled with the *critical output window* parameter t_{Wcow_1}. It is the upper bound on the time delay between hybrid channels 1 and 2. The weaker constraint t_{Wow} as an upper bound on the time delay among all hybrid channels still exists. If the output execution of hybrid channel 3 comes between output generations of hybrid channels 1 and 2, t_{Wcow_1} can be violated. This can be prevented by specifying an ordering of all output channels

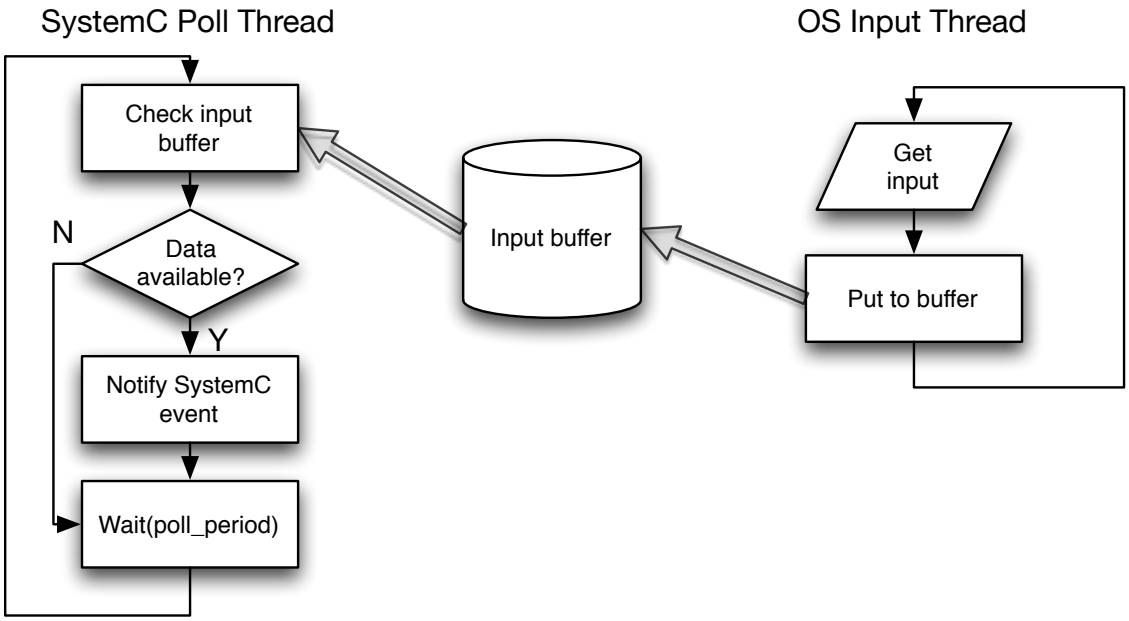


Figure 4.7. Polling mechanism to incorporate external events

in *update real*, so no undesired output generations can come between critical subsets of output generations. This is implemented by giving each hybrid channel a specific order number and sorting the pending outputs in *update real* just before starting the output generations at $t_{Wdeltaend}$.

4.2.2. Interactions from Real to Virtual Subsystems

SystemC kernel does not have a mechanism for receiving external events. Time is always advanced according to internal events and operations. A mechanism is needed to incorporate external events to the simulator.

There are three levels of remedy for this kind of situation: [a] implement polling for inputs, [b] make polling rate adaptive and [c] improve the SystemC patch to incorporate handling of external events.

4.2.2.1. Polling. SystemC thread processes are used to poll the external interfaces and relay this information to internal *sc_events*. In this way, the SystemC kernel becomes aware of the external events and the rest of the model can use this *sc_event* to check

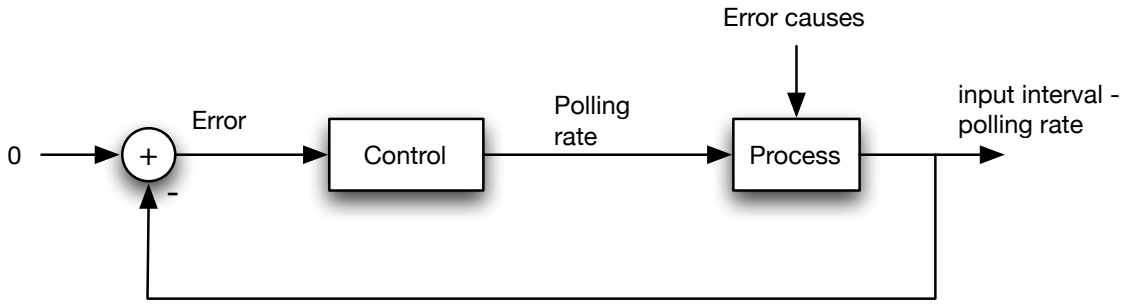


Figure 4.8. Control loop for adaptive polling

for the input. Figure 4.7 shows an example of this mechanism. Here the actual input operation is done asynchronously by an operating system thread, whereas it could also be done by the SystemC poll thread synchronously.

The *polling_rate* is a fixed parameter in this scheme. The input latency introduced by polling will range in $[0; polling\_rate]$ and on the average, it will be $polling\_rate/2$. So, increasing *polling_rate* will increase the input latency, and decreasing *polling_rate* will increase the demand of the simulation for computation power. So, a tradeoff is necessary here.

4.2.2.2. Adaptive Polling. If the interval between external events is changing during the model execution, the *polling_rate* can be changed dynamically to adapt to these changes. For instance, the model of a network device may need to adapt its *polling_rate* accordingly when the network traffic increases or decreases. Adaptive polling also helps in the case where the model developer does not have an a priori knowledge on the environment of the model, so the correct *polling_rate* is unknown at the time of the development.

A control loop (Figure 4.8) employing a PID (Proportional, Integral, Derivative) controller is used to accommodate for these changes. Here; *polling_rate* is the control variable, *input_interval* is measured independently by the unit handling the inputs. The process output is defined as the difference $input\_interval - polling\_rate$, which the control loop tries to converge to the set value of 0. The coefficients of the PID controller

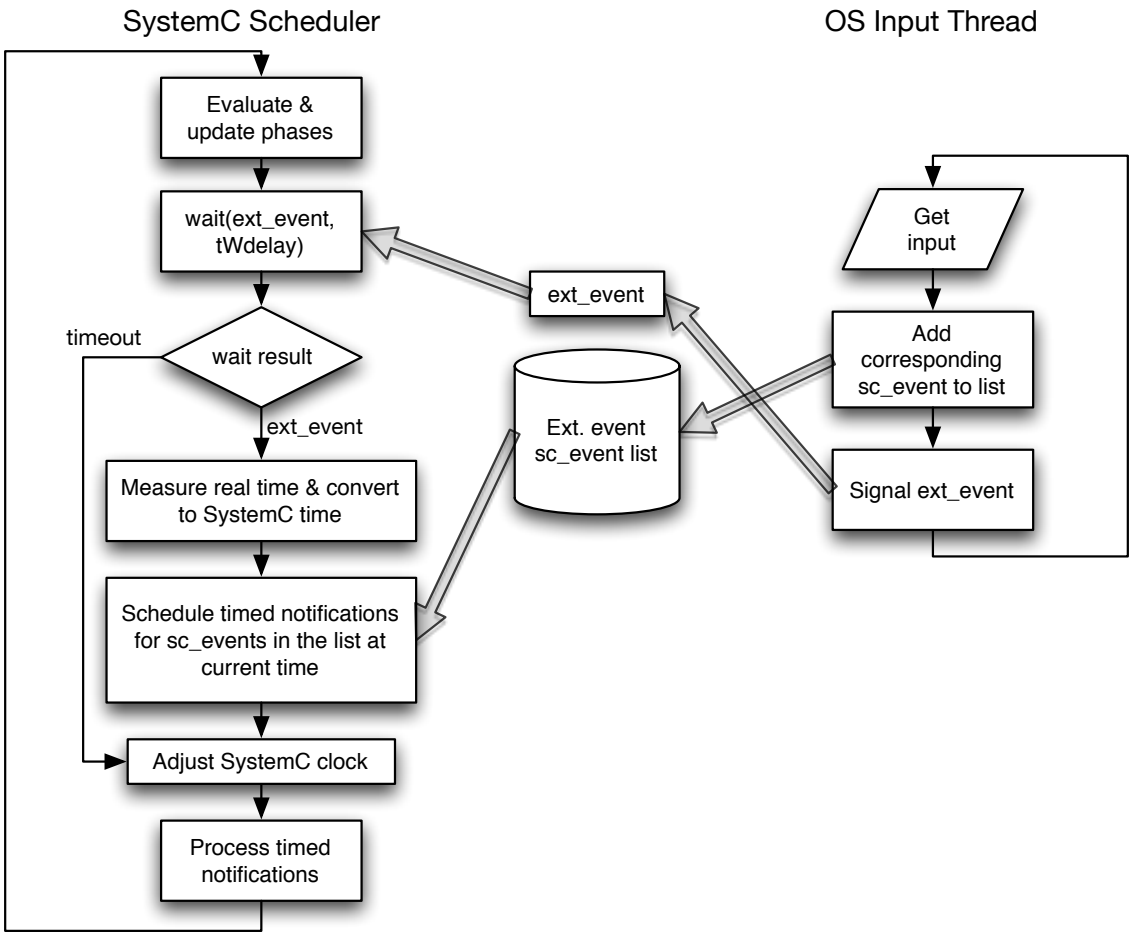


Figure 4.9. Event-driven solution for incorporating inputs

(proportional K_P, integral K_I, derivative K_D) are parameterizable. The assignment of these coefficients is a standard PID controller tuning task and same techniques can be applied here.

4.2.2.3. Fully Event-driven Simulation. The real-time patch to the SystemC kernel (Section 4.1) can be extended to eliminate the need for polling to incorporate external events.

In this scheme shown in Figure 4.9, the unconditional wait of t_{Wdelay} in Figure 4.2 is replaced by a conditional one waiting on *ext_event*. An *sc_event* is used to represent each external event. This *sc_event* is added to the external event list and the *ext_event* is signaled by the input handler. The SystemC scheduler waiting on *ext_event* wakes,

Table 4.2. Comparison of methods for incorporating external events

	Advantages	Disadvantages
Polling	Simple implementation, the fastest execution. No complex OS constructs.	Tuning necessary for *polling_rate*, tradeoff: I/O latency vs. simulation performance
Adaptive polling	No complex OS constructs	Tuning necessary for PID coefficients
Fully event-driven	No tuning necessary	Intricate implementation, uses complex OS constructs

adjusts the SystemC clock according to the current wall clock time and schedules timed notifications for the *sc_events* in the external event list. If *ext_event* is not signaled, the *wait* will end due to timeout and the execution will proceed to the next delta-cycle.

Table 4.2 shows a comparison of methods for incorporating external events to the simulation model.

4.3. Mathematical Model of Execution Performance

Following variables can be defined for modeling the simulation's execution according to Figure 2.2:

- A: Total number of *time advance* phases in the simulation.
- U_i: Total number of *update* phases before i'th *time advance* phase.
- E_{ij}: Total number of *evaluate* phases before j'th *update* phase, which comes before i'th *time advance* phase.
- O_i: Total number of outputs to real subsystems before i'th *time advance* phase
- H: Total number of hybrid channels
- P: Total number of thread processes

The execution of the simulation should satisfy following constraints to achieve real-time behavior:

(i) The relationship in Eq. (2.1) between the simulation clock T_S and the wall clock T_W has to be maintained.

(ii) Concurrent output generations with regard to T_S, have to occur within a maximum time window of t_{Wow} with regard to T_W, so that the real subsystems connected to the model should observe them as concurrent. This constraint can be further hardened by adding narrower time windows of t_{Wcow_s} for specific subsets of output channels (Figure 4.6)

4.3.1. Real-Time Simulation for Hardware/Software Co-design

The first constraint means that t_{Wdelay_i} must remain nonnegative, as a negative delay is not implementable. This implies the inequality Eq. (4.1) to be held (Figure 4.2).

$$t_{Wpassed_i} \leq t_{Snew_i} - t_{S_i} \tag{4.1}$$

Using Figure 4.4, Eq. (4.1) can be further detailed into Eq. (4.2).

$$t_{Wdeltacycles_i} + t_{Woutput_i} \leq t_{Snew_i} - t_{S_i} \tag{4.2}$$

Figure 2.2 shows that $t_{Wdeltacycles}$ consists of multiple delta-cycles and a delta-cycle consists of multiple *evaluate* phases and exactly one *update* phase. Equation (4.3) shows this.

$$t_{Wdeltacycles_i} = \sum_{j=1}^{U_i} \left(\sum_{k=1}^{E_{ij}} t_{Wevaluate_{ijk}} + t_{Wupdate_{ij}} \right) \tag{4.3}$$

$t_{Wevaluate_{ijk}}$ is the sum of run-times of processes that are active in the *evaluate* phase k before the *update* phase j which comes before the *time advance* phase i. $t_{Wupdate_{ij}}$ is the sum of channels' update times that were written during the previous *evaluate* phases. If an upper bound can be guaranteed for $t_{Wevaluate_{ijk}}$, $t_{Wupdate_{ij}}$ and U_i, then the satisfying of the inequality Eq. (4.1) can be guaranteed statically, because the total number of thread processes in the model, i.e. P, is always an upper bound for the number of processes executed in an *evaluate* phase, i.e. E_{ij}. However, SystemC is based on C++, so it supports all programming structures of this language, including very complex ones. Furthermore, the input from real subsystems arriving via hybrid channels can change the execution path of threads. Therefore it is nearly impossible to determine the run-times statically or guarantee upper bounds for these variables. Common industry practice of profiling, i.e. run-time analysis, should be chosen here. As an addition to profiling for the special case of SystemC, measuring the ratio $t_{Wpassed_i}/(t_{Snew_i} - t_{S_i})$ at each *time advance* phase is proposed to monitor if the inequality Eq. (4.1) is held.

4.3.2. Concurrent Outputs

The production of an output from the virtual subsystems is done in three steps. For the discussion, these steps are defined as follows:

(i) Output value computation: Computation of the data to be output
(ii) Output trigger: The trigger for starting the output generation
(iii) Output generation: The actual realization of the output so that the real subsystems can get it

$t_{Woutput_i}$ is the sum of output production run-times of hybrid channels at the i'th *time advance* phase. This can vary a lot for different types of output hardware. Following assumptions are introduced for simplification:

- All output generations are delegated to asynchronous output threads which employ sufficiently large output buffers to guarantee decoupling from the simulation's generation so that the simulation thread does not wait on output threads. These threads are created in the *initialization* phase of the simulation in order to avoid latencies caused by the thread creation.
- The triggers of output threads from the simulation thread is implemented via similar OS mechanisms, e.g. notifying a condition variable.
- The number of concurrent output productions at a specific simulation time does not exceed the number of processing cores in the modeling platform.[2]
- All hybrid channels use the *update real* method for output timing.

Under these assumptions, it becomes safe to claim that all output triggers take the same amount of time, t_{Wo}, in the simulation thread. The total number of hybrid channels H in the model can be used as an upper bound for the number of output productions at a *time advance* phase. If there are multiple output productions pending on a hybrid channel at a *time advance* phase, they can be still regarded as one, because [a] the trigger to output generation is still single, and [b] multiple output value computations imply a sequential relationship, so the concurrency is not actually required for the output productions of one hybrid channel. From here, Eq. (4.4) can be derived.

$$t_{Woutput_i} = \sum_{u=1}^{O_i-1} t_{Wo} = (O_i - 1).t_{Wo} \leq (H - 1).t_{Wo} \tag{4.4}$$

[2]Quad-core computation platforms are quite common in the industry. It can also be assumed that high-end platforms will be employed in a dedicated way for the execution of the model. So availability of platforms with eight cores is safe to assume. If one core is used for the simulation, seven cores remain for execution of concurrent outputs. This enables seven hybrid channels doing an output at the same time. Seven is a large number for interfaces of a system. And this can be multiplied if not every channel is executing an output concurrently at each cycle. Further possibilities will arise with the advent of many-core platforms.

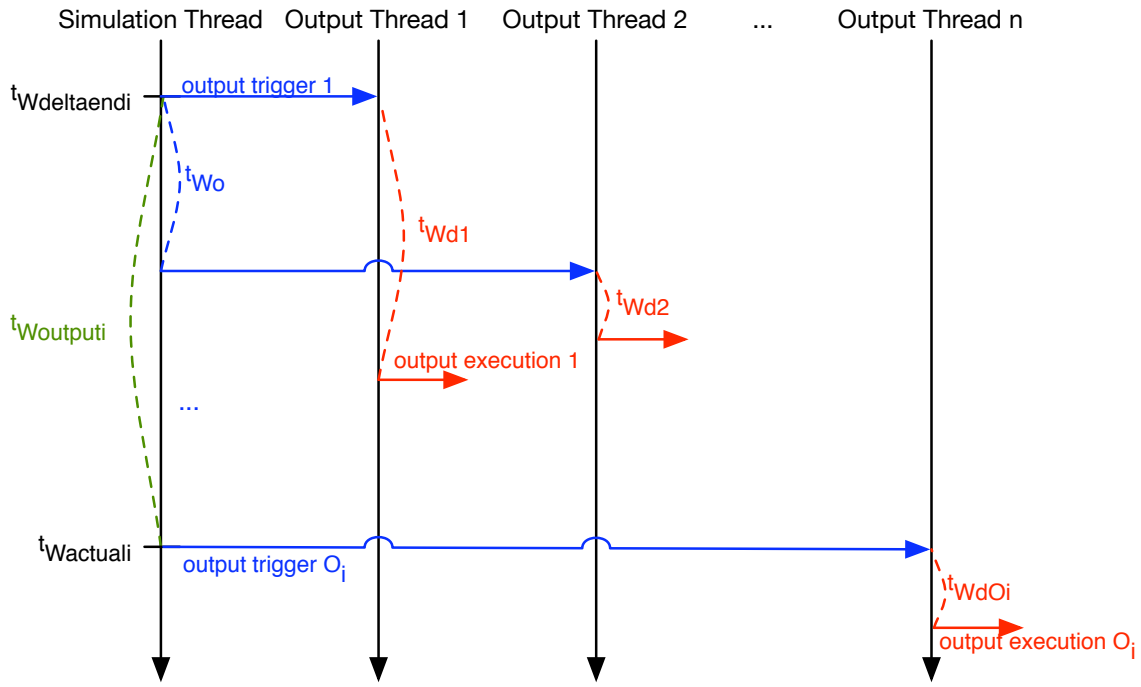


Figure 4.10. Concurrent output productions

Figure 4.10 outlines the production of outputs clearly. Output value computation is done in *evaluate* and *update* phases, i.e. before $t_{Wdeltaend_i}$. Output trigger is done in the simulation thread, which takes always t_{Wo}. Output generation is done in asynchronous output threads and may take various times, denoted as t_{Wd_u}.

The satisfaction of the second constraint can be discussed on this basis. As Figure 4.10 shows, concurrent output triggers are distributed over $t_{Woutput_i}$, which should be lower than the output window t_{Wow}. So the constraint can be formulated as Eq. (4.5).

$$t_{Woutput_i} \leq t_{Wow} \tag{4.5}$$

Eq. (4.4) can be used to get an upper bound for $t_{Woutput_i}$, so Eq.(4.6) can be used as an estimator for Eq. (4.5). Here, H can be derived statically from the model and t_{Wo} can be measured as a characteristic of the modeling platform.

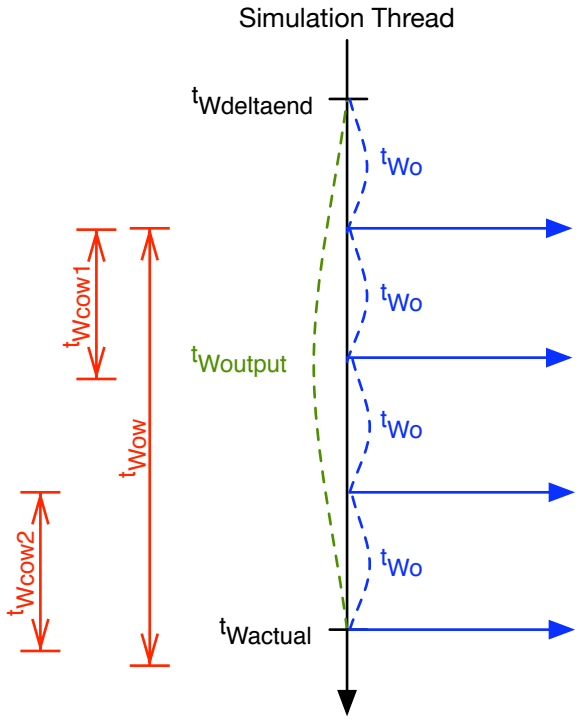


Figure 4.11. Concurrent output windows

$$(H-1).t_{Wo} \leq t_{Wow} \tag{4.6}$$

Satisfying a narrower time window of t_{Wcow_s} for a subset s of output channels (see Figure 4.11 for an example) is achieved by specifying a fixed order of output generations, where output channels in the subset are ordered successively. This requires that the subsets are exclusive, i.e. do not share an output channel. Thereby the time between two output generations in the subset is set to t_{Wo}. For a subset s consisting of m_s channels, the constraint can be formulated as in Eq.(4.7).

$$(m_s-1).t_{Wo} \leq t_{Wcow_s} \tag{4.7}$$

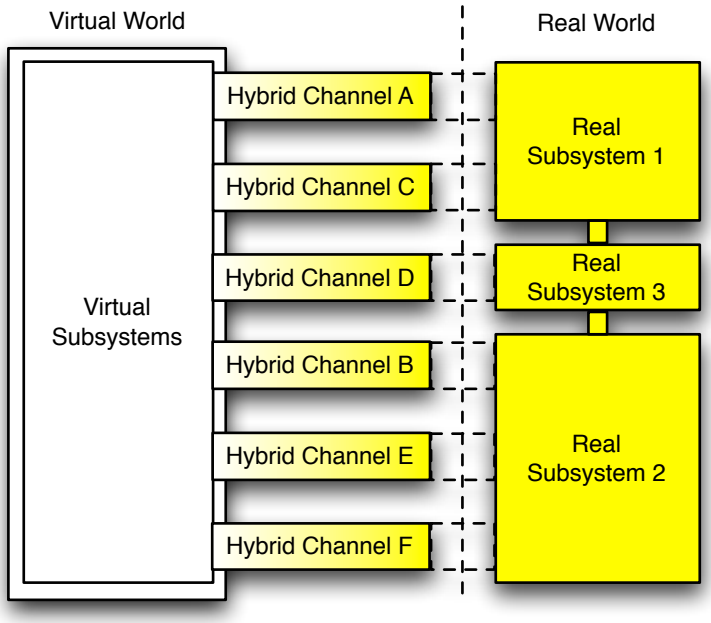


Figure 4.12. Setup of illustrative example

As seen in Figure 4.10, the real point in time where the output is observable by real subsystems depends also on the run-time of the output thread t_{Wd_u}. However this does not imply an inaccuracy of the model from the real system, because several forms of delay are also present in a real system as propagation delay, signal setup time etc. The same fact is also true for input channels.

Another point regarding output timing is the hold time for channels with parallel semantics (dsc_parallel_hybrid_channel). As new values overwrite old values, there should be a minimum time window for the value to remain unchanged, so that the real subsystems can get the value. However, this is strongly dependent on the model, and the model developer should and can specify this hold time by inducing a *wait* in SystemC. Our method will guarantee that the value will be held for at least this duration.

4.3.3. Illustrative Example

Let there be an embedded system under development where the model is connected to the real subsystems via 6 hybrid channels: A, B, C, D, E, F. A and C are connected to a device and B, E and F are connected to another device. Figure 4.12 shows the setup.

As concurrent outputs from the model to single devices should be produced within a narrow time window, two critical output subsets of the set of all hybrid channels are defined. Hybrid channels A and C belong to the critical output subset 1 ($m_1 = 2$) and B, E and F belong to the critical output subset 2 ($m_2 = 3$). Critical output subsets 1 and 2 have the critical output windows $t_{Wcow_1} = 10\mu s$ and $t_{Wcow_2} = 15\mu s$, respectively. There is also the general output window $t_{Wow} = 50\mu s$ for all hybrid channels.

To satisfy the critical output window constraints, a strict ordering of all hybrid channels is forced by assigning them integers for ordering as follows: A: 10, C: 10, B: 20, E: 20, F: 20, D: 30. All outputs are triggered during *update real* phase and the output generation is delegated to asynchronous output threads. Let the time of one output trigger be $t_{Wo} = 5\mu s$.

Figure 4.13 shows an excerpt of the execution of the example system. In all cycles, $t_{Wpassed_i}$ has remained below $t_{Snew_i} - t_{S_i}$, i.e. in the first cycle $t_{Wpassed_1} = 36\mu s \leq t_{Snew_1} - t_{S_1} = 75\mu s$; in the second cycle $t_{Wpassed_2} = 51\mu s \leq t_{Snew_2} - t_{S_2} = 100\mu s$; and in the third cycle $t_{Wpassed_2} = 63\mu s \leq t_{Snew_3} - t_{S_3} = 75\mu s$. As a result, the *time advance* phase can always be executed.

Following observations can be made about the behavior of concurrent outputs:

(i) In the first cycle, only one channel from each critical output subset has an output, so critical output windows need not be checked. Only general output window can be checked, and it is satisfied: $(no_1 - 1).t_{Wo} = 5\mu s \leq t_{Wow} = 50\mu s$.

(ii) In the second cycle, two channels from the critical output subset 2 have new

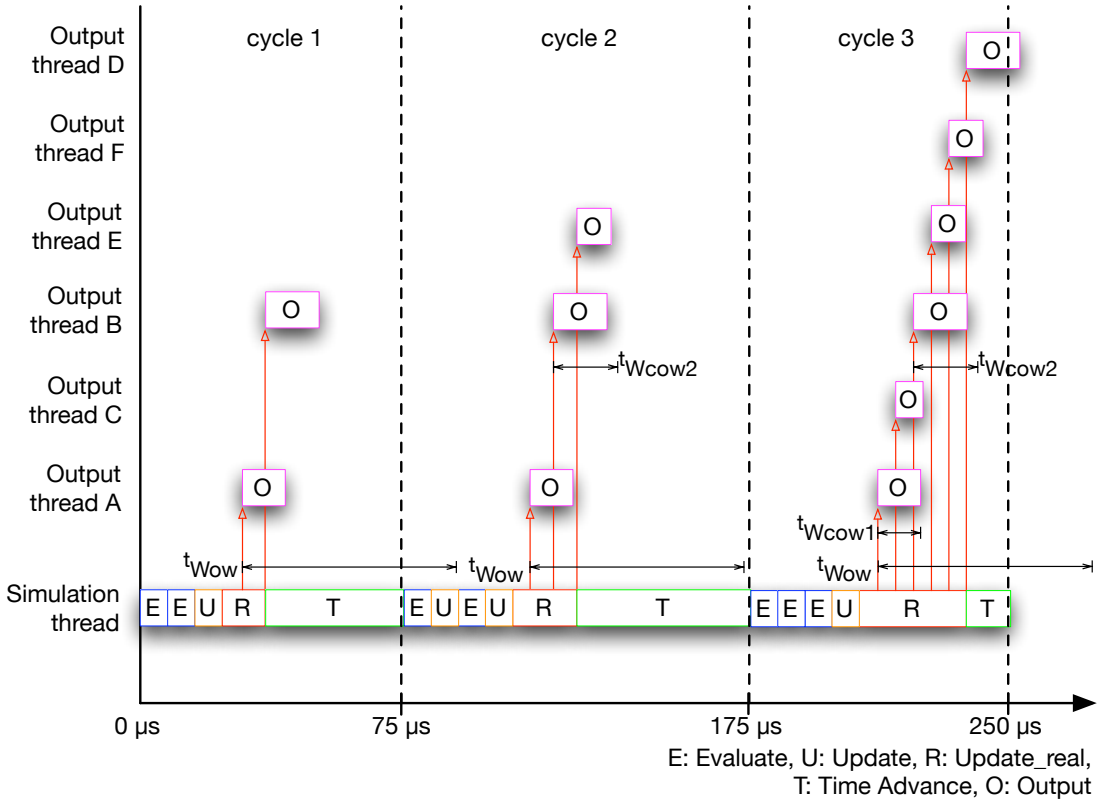


Figure 4.13. Execution excerpt of illustrative example

output values, so the critical output window 2 is checked and it is satisfied, too: $(m_2 - 1).t_{Wo} = 10\mu s \leq t_{Wcow_2} = 15\mu s$. Additionally, general output window is also satisfied: $(no_2 - 1).t_{Wo} = 5\mu s \leq t_{Wow} = 50\mu s$.

(iii) In the third and last cycle, all hybrid channels have an output, so both critical output windows are checked and they are satisfied: $(m_1 - 1).t_{Wo} = 5\mu s \leq t_{Wcow_1} = 10\mu s$ and $(m_2 - 1).t_{Wo} = 10\mu s \leq t_{Wcow_2} = 15\mu s$. Additionally, general output window is satisfied, too: $(no_3 - 1).t_{Wo} = 5\mu s \leq t_{Wow} = 50\mu s$. In the last cycle, output generations continue after the end of the cycle. This is expected and acceptable, because outputs are executed in asynchronous output threads.

5. EXPERIMENTAL RESULTS

The proposed method has been evaluated in four experiments. The first and second experiments help to measure the performance of our method in terms of real-time behavior and I/O performance, respectively. The third experiment aims to analyze the behaviour of concurrent outputs. The last experiment is a case of design of a new industrial embedded system, which is tested with real subsystems at transaction-level model TLM phase.

5.1. Implementation Details

SystemC simulation kernel is executed on Linux operating system kernel. The real-time patch has been applied to SystemC in *sc_simcontext::simulate*, where time is advanced. Additionally, the code for calling *update real* methods of all channels of type *dsc_hybrid_channel* has been inserted at the same place.

Figure 5.1 shows a simple example of the proposed hybrid channel. It has a SystemC interface *sc_signal_inout_if* on one side, and a handle to the I/O driver on the other side. The pin is a parallel communication, so inherits from the class *dsc_parallel_hybrid_channel*.

Two more complex examples realize the communication over Ethernet:

- sc_hybrid_eth_in for data from Ethernet to SystemC model
- sc_hybrid_eth_out for the reverse direction as shown in Figure 5.2.

In order to minimize the time during simulation's execution ($t_{Wpassed}$ in Figure 4.2), I/O operations have been delegated to operating system threads. For instance in *sc_hybrid_eth_in*, *recv_thread* does the actual reception from Ethernet, then a SystemC thread gets the data to the model. Similarly, actual transmission is done in the operating system thread *send_thread*, which is triggered by a SystemC thread.

Queues hold the incoming/outgoing data for the transfer between threads. Ethernet is a kind of serial communication, so these classes inherit from *dsc_serial_hybrid_channel*. On the same principles as the Ethernet hybrid channel, a UDP/IP hybrid channel *sc_hybrid_udp* capable of both input and output has been built, too.

Management of delta cycles has been done differently in pin channel and Ethernet channels. Pin channel implements a SystemC signal, so the value can change in successive delta cycles and it makes sense to delay the output until the final value is established for the current simulation time. On the other hand Ethernet channels are first in first out (FIFO) channels, and each written value should be transferred to the output regardless of later operations so that the output device can start processing the data as soon as it receives it [1]. Thus, *sc_hybrid_pin* generates the actual output in *update real*, i.e. at the beginning of *time advance* phase, while *sc_hybrid_eth_out* sends the data right away to the operating system thread at the *evaluate* phase. This makes the transmission in parallel to the simulation's execution.

To improve determinism, Linux kernel's preemptibility has been increased via the RT_PREEMPT patch [10]. SystemC thread and operating system threads doing the I/O operations have been set to real-time scheduling and their priorities have been set to a priority directly below the interrupt handling threads. Since a computer with swap memory has been used in these experiments, all memory pages belonging to the simulation process have been locked in memory to avoid latencies due to page faults. Finally, the thread stack has been extended beyond the maximum point used, in order to avoid page faults due to stack growth.

Software platform of modeling consists of Linux 2.6.31.6-rt19 with RT_PREEMPT mode turned on and SystemC version 2.2.0 with our real-time patch. CPU load is generated via multiple instances of an infinitely spinning shell script. For the PWM and RTT experiments a PC with dual Intel Quad-Core Xeon processors running at 3.4 GHz is used and for the BBMD experiment a PC with Intel Pentium 4 3.2 GHz HT is used.

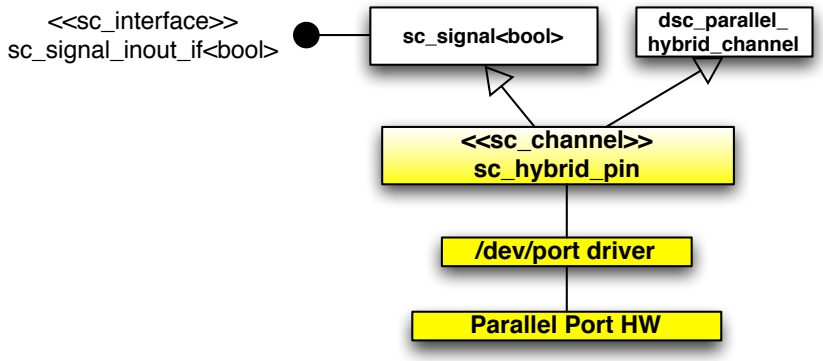


Figure 5.1. Hybrid pin channel

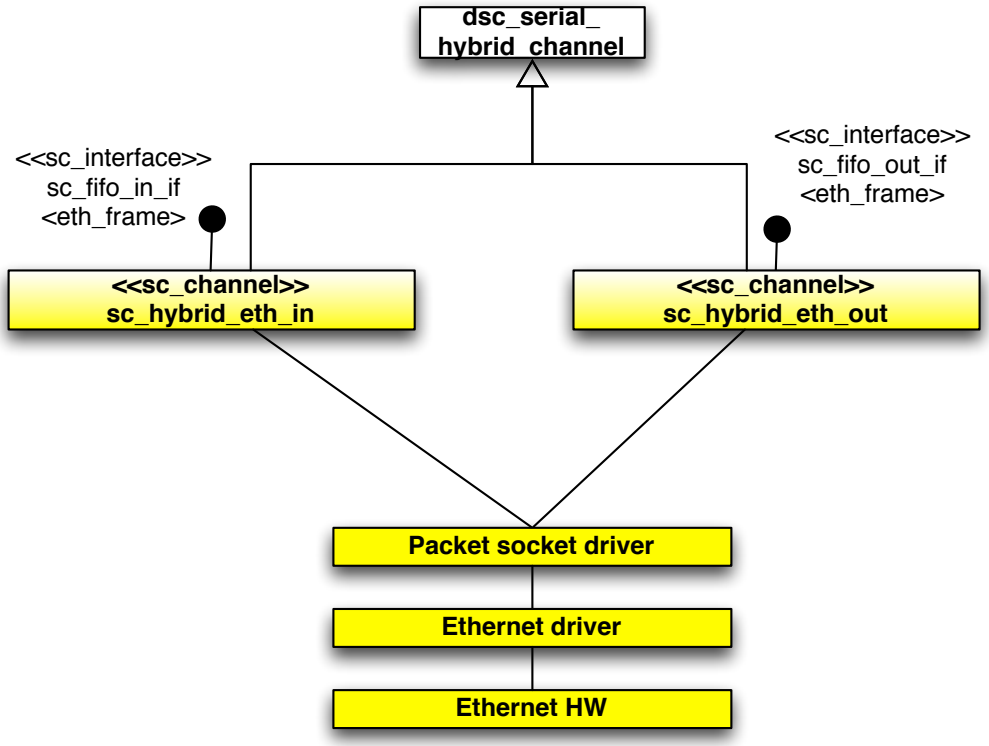


Figure 5.2. Hybrid Ethernet channels

5.2. Simulation Speed & Determinism

To measure maximum reachable simulation speed and the level of determinism the pulse width modulation (PWM) experiment is conducted. This experiment (Figure 5.3) consists of generating a square wave and examining the jitter in the output signal. Square wave is generated by a SystemC module *sc_pwm* with 50% PWM duty cycle

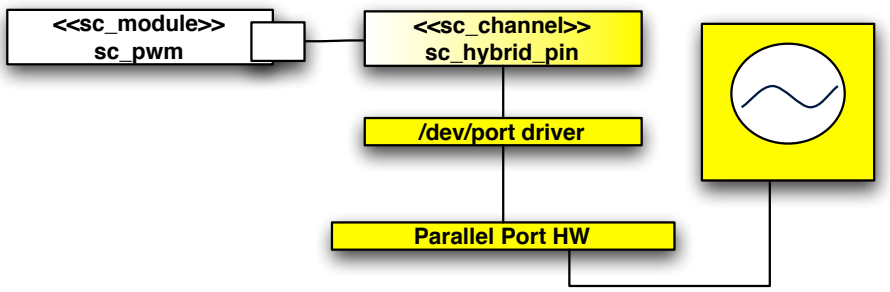


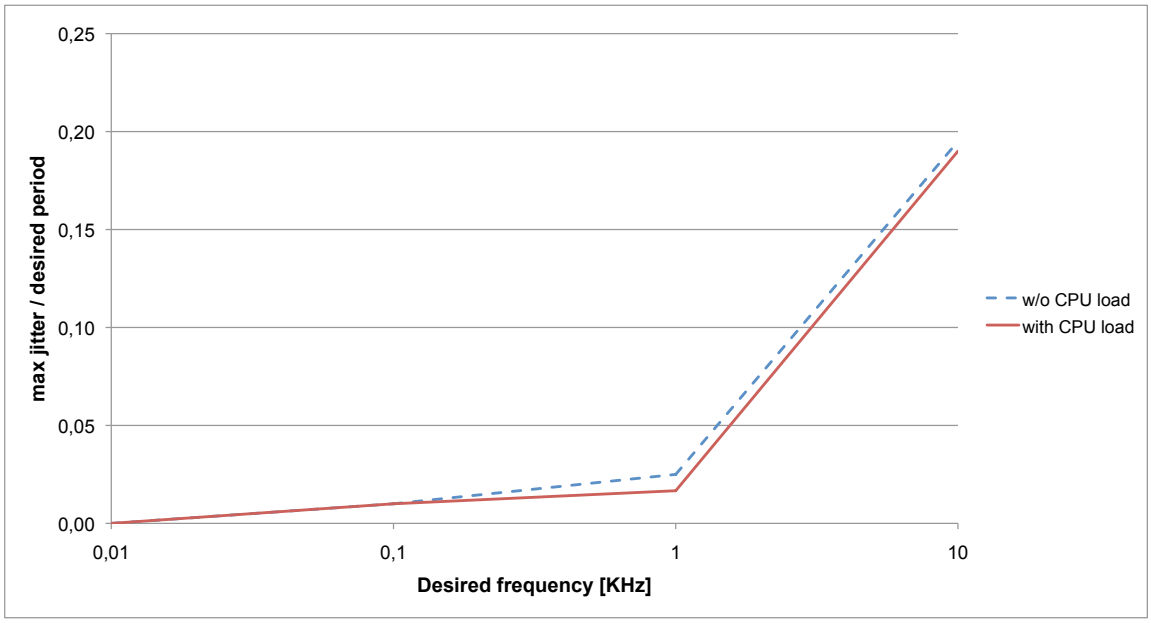
Figure 5.3. Experimental setup of PWM

Table 5.1. Parameters of PWM experiment

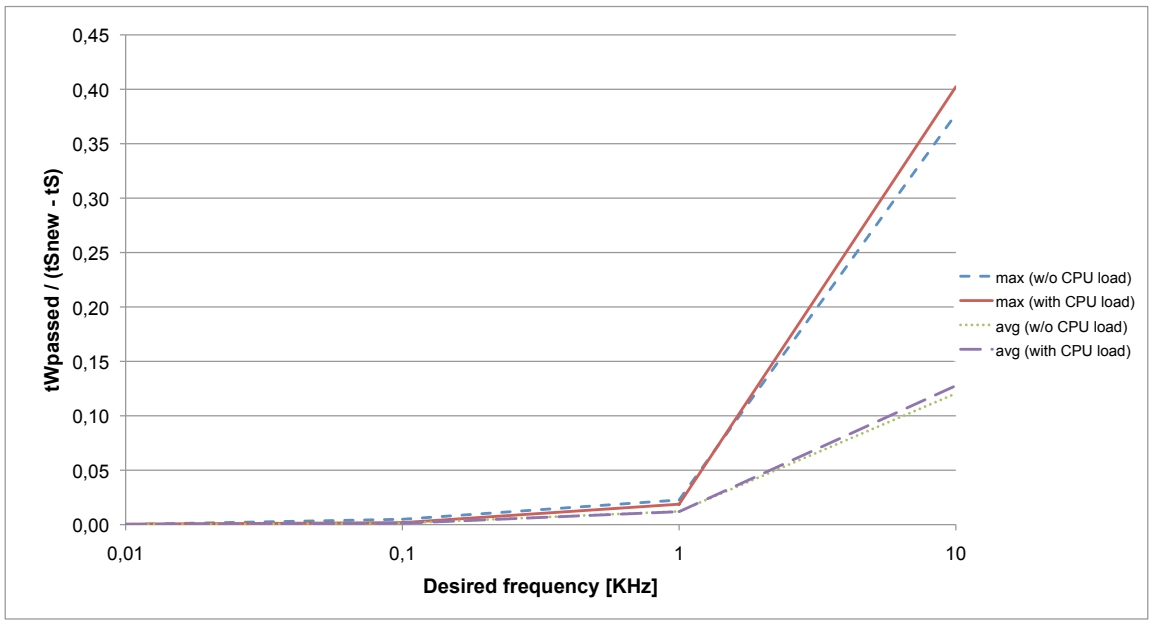
Parameter	Value(s)
Desired frequency	10 Hz, 100 Hz, 1 KHz, 10 KHz, 100 KHz
CPU load present?	no, yes

and is relayed to the parallel port of the computer via the hybrid channel *sc_hybrid_pin*. An oscilloscope is used to examine the quality of the generated signal. The persistence parameter of the oscilloscope is set to infinite, in order to keep all of past waveforms and observe the maximum jitter. Parameters of the PWM experiment are desired frequency, and presence of additional CPU load, as shown in Table 5.1. The evaluation criterion is the ratio of maximum jitter to the desired period. Additionally, the ratio $t_{Wpassed}/(t_{Snew} - t_S)$ (Figure 4.2) is also measured at each *time advance* to separately see the simulation's performance apart from the I/O performance.

The results of the experiment are given in Figure 5.4. The signal is stable up to 10 KHz. At this rate jitter becomes significantly high, however it is still below half of the period, so the square wave is still generated at the desired frequency, hence it may be still considered acceptable. At 100 KHz no meaningful jitter can be measured, as the waveform is largely corrupted (Figure 5.5). At stable frequencies, load has only a minor effect although the CPU is loaded 100%. This shows that the real-time operating system scheduler is working fine. Moreover, the internal measurement of the maximum value of $t_{Wpassed}/(t_{Snew} - t_S)$ matches the external measurement, so it can be concluded that the simulation performance is the main factor affecting the performance rather



(a)



(b)

Figure 5.4. PWM: For different desired frequencies, (a) Max jitter / desired period (b) $t_{Wpassed}/(t_{Snew} - t_S)$

than the I/O performance. Finally, the average value of $t_{Wpassed}/(t_{Snew} - t_S)$ shows that there is still unused capacity for higher frequencies in terms of computation power, however jitter prevents this capacity from being utilized.

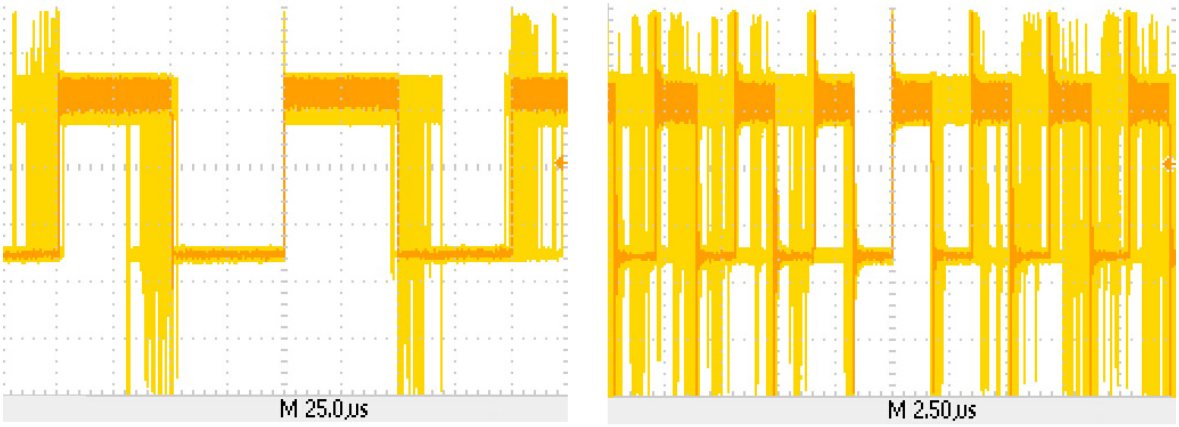


Figure 5.5. Waveforms of (a) 10 KHz and (b) 100 KHz desired frequencies (persistence = infinite)

5.3. I/O Performance

Round-trip time (RTT) experiment is used the measure the I/O performance of our proposed system in isolation. Figure 5.6 shows the setup of the experiment. Here, the SystemC model functions as a frame replier, which sends the incoming frames back. The SystemC module *sc_eth_mirror* is responsible for sending the received frames back. *sc_hybrid_eth_in* and *sc_hybrid_eth_out* relay the frames between the Ethernet interface and the SystemC model. The measurement is then done on the initial sender's side. Evaluation criteria of the RTT experiment are maximum, minimum and average values for RTT. One hundred samples are taken in each run. Three proposed methods for incorporating external events to the simulation are evaluated in three experiments: polling, adaptive polling and event-driven solution. All three experiments have frame length as a parameter. Polling period is another parameter in the polling experiment. Parameters for adaptive polling include a maximum and minimum value for the polling period instead of a single value and the coefficients K_P, K_I, K_D of the PID controller. Tables 5.2, 5.3 and 5.4 give detailed information about experiment parameters.

In RTT experiment with polling, Figure 5.7 shows that the polling time has a more significant effect than the frame size. It has also been observed that polling time results in high jitter of RTT, because the response time is directly dependent

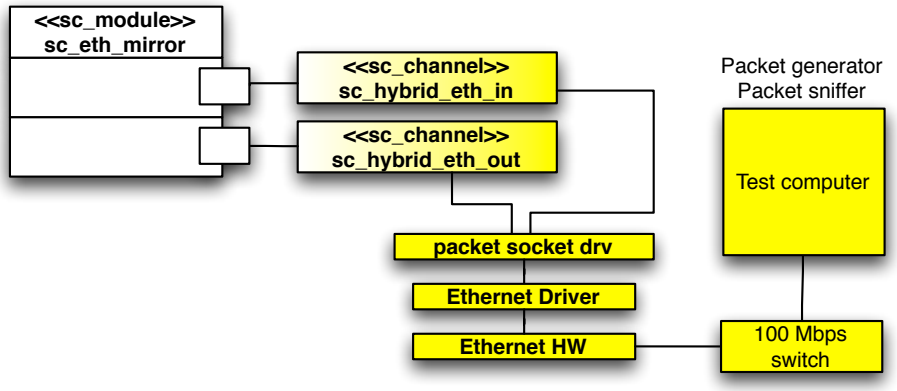


Figure 5.6. Experiment setup of RTT measurement

Table 5.2. Parameters of RTT experiment with polling

Parameter	Value(s)
Frame size	64 bytes, 780 bytes, 1514 bytes
Polling period	100 μs, 1000 μs

Table 5.3. Parameters of RTT experiment with adaptive polling

Parameter	Value(s)
Frame size	64 bytes, 780 bytes, 1514 bytes
K_P	-0.1
K_I	-0.05
K_D	0
Max. *polling_period*	10 ms
Min. *polling_period*	10 μs

Table 5.4. Parameters of RTT experiment with event-driven solution

Parameter	Value(s)
Frame size	64 bytes, 780 bytes, 1514 bytes

on the current phase of the polling cycle. Furthermore, frame size has a linear effect on RTT, as most operations - copy, Ethernet propagation - are affected linearly. For cyclic communication, cycle times of 1 ms may need low polling cycles, 100 μs in this

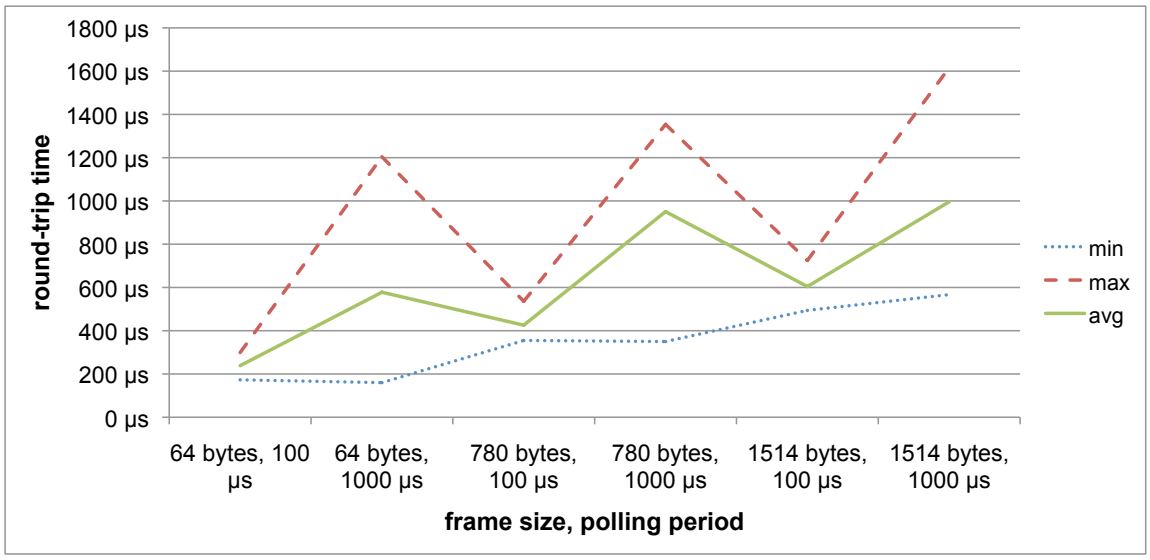


Figure 5.7. RTT's max/average/min measurement with polling

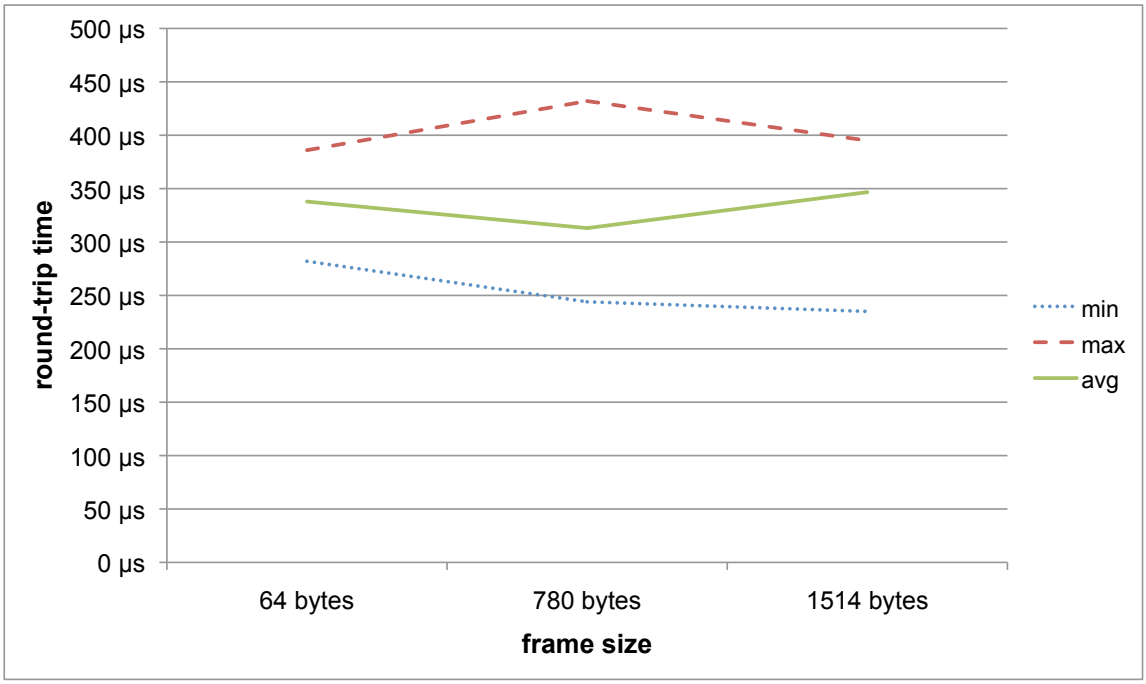


Figure 5.8. RTT's max/average/min measurement with adaptive polling

experiment, since measured values exceed this value in other cases. However, at periods of 10 ms and higher, no further tuning is necessary for satisfactory performance.

When using adaptive polling, the behavior of RTT is complex (Figure 5.8). Longer frames increase the transmission time, but also decrease the polling period,

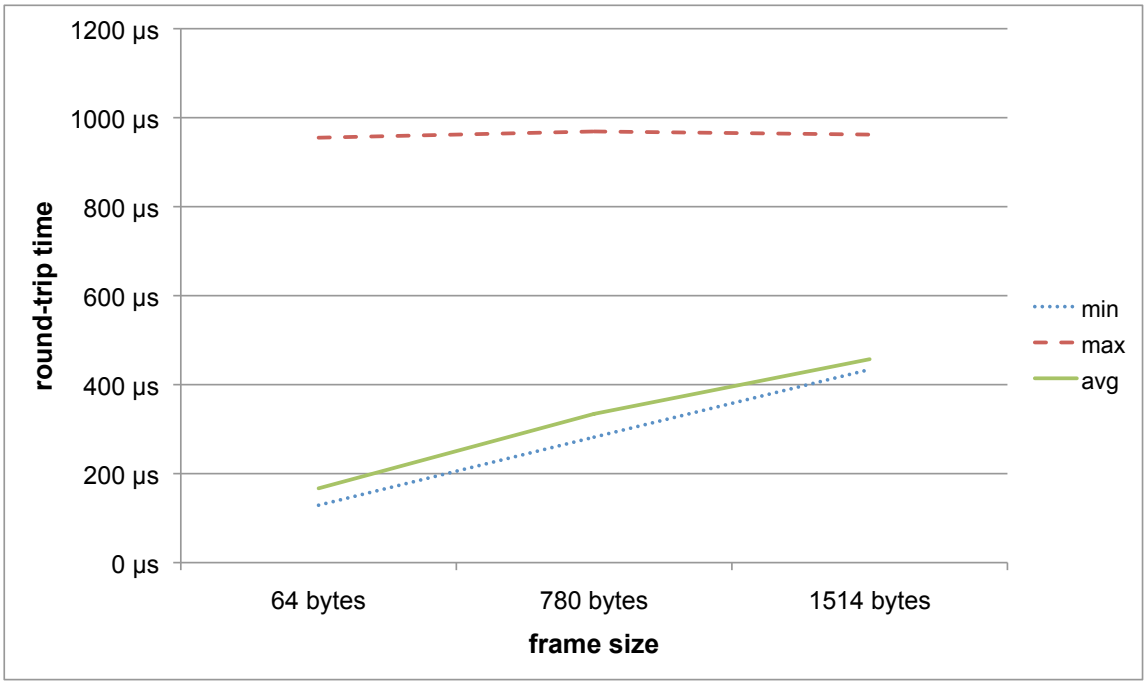


Figure 5.9. RTT's max/average/min measurement for with event-driven solution

so maximum length frames have a better RTT result than medium-size frames. In the current state, communication cycles of 1 ms are possible. The results depend strongly on the behavior of the PID controller, so fine tuning of PID coefficients may yield much better results, however this fine tuning is costly in terms of effort.

Event-driven solution performs worse than adaptive polling for large frames (Figure 5.9). Event-driven solution's average RTT value grows linearly with the frame size while adaptive polling performs better for maximum size frames. Maximum RTT is not effected by frame size, probably this occurs due to an inherent latency in the OS constructs, which are more complex in comparison to polling or adaptive polling. More complex OS constructs can also explain worse RTT values than adaptive polling. Event-driven solution has the big advantage of not requiring any tradeoff between I/O latency and simulation performance or any tuning.

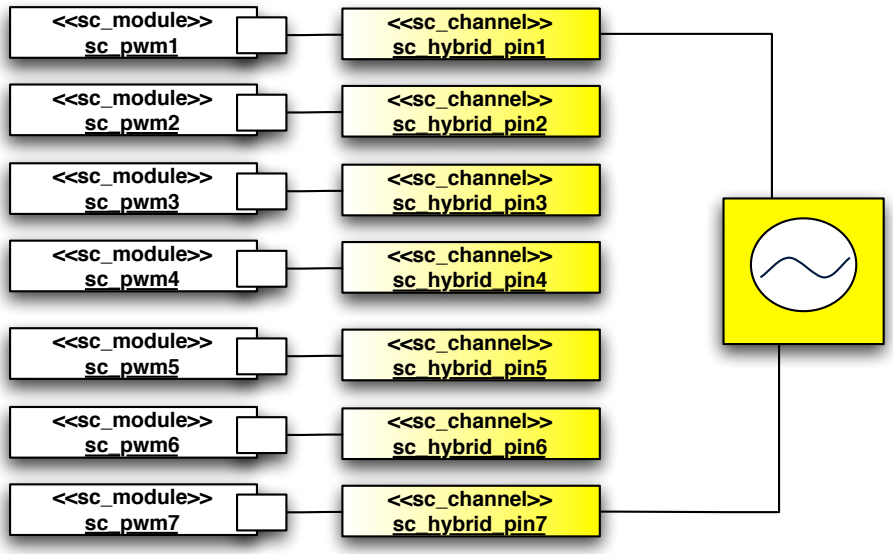


Figure 5.10. Experiment setup for concurrent outputs without hardware support

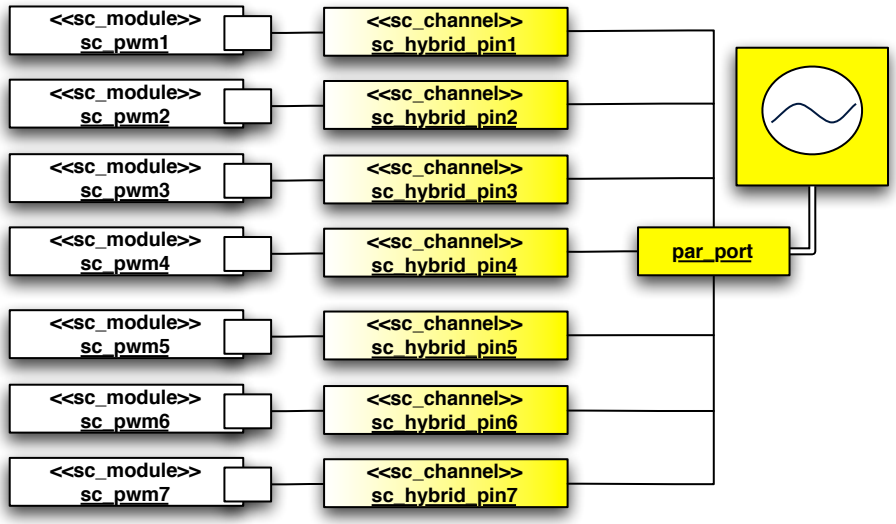


Figure 5.11. Experiment setup for concurrent outputs with hardware support

5.4. Concurrent Outputs

The behavior of concurrent outputs in real-time has been evaluated in this experiment. Two proposed measures, i.e. ordering outputs and using hardware support if available, have been studied in two experimental setups shown in Figure 5.10 and Figure 5.11, respectively. In both setups, seven digital output signals are written by the model to the parallel port. First and last signals are connected to an oscilloscope.

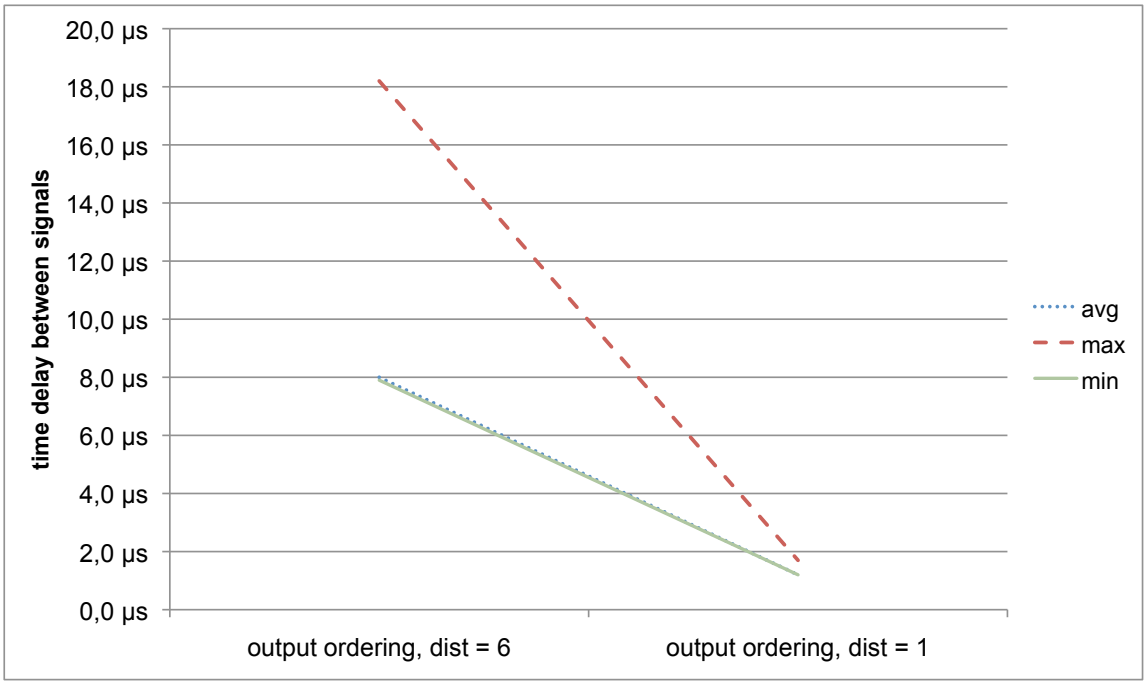


Figure 5.12. Real-time difference between concurrent signals

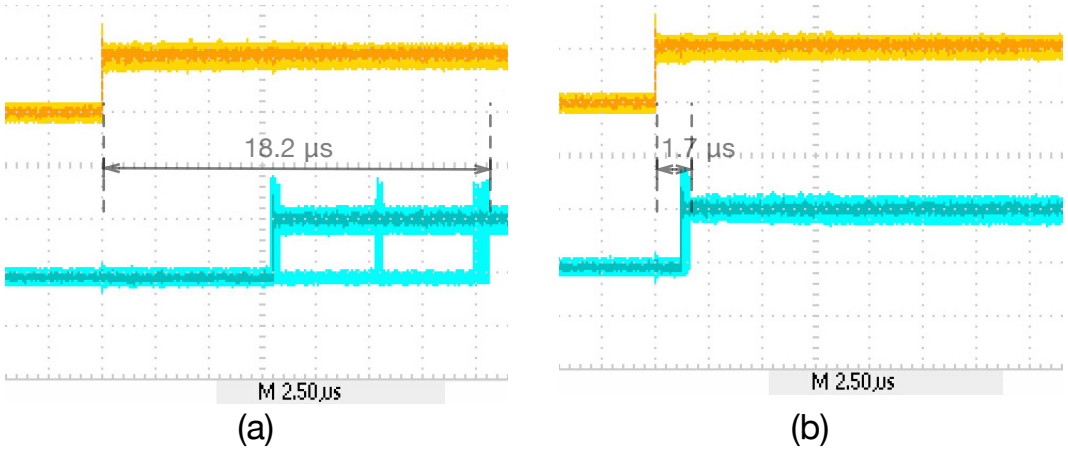


Figure 5.13. Waveforms of two concurrent signals in real-time (persistence = infinite) (a) with 5 channels inbetween, (b) successive

When using hardware support, all output channels send their data first to another class called *par_port*, which combines all values and writes to hardware at a single point in time. Without hardware support, all channels write their values to the parallel port without combining it with other channels.

100% concurrency is achieved when using hardware support. On using output

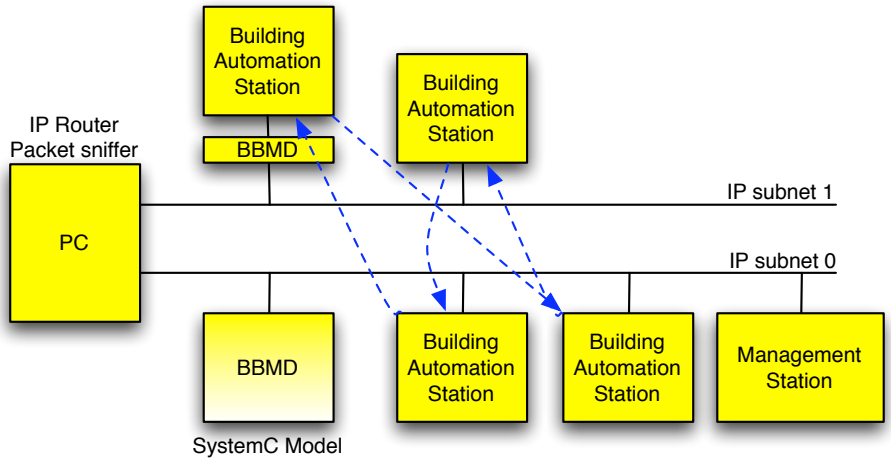


Figure 5.14. Experiment setup of BBMD

ordering without hardware support, the time delay between concurrent signals can be reduced 6.7 times on average (Figures 5.12 and 5.13). Maximum delay is reduced 10.7 times probably because longer delay is more likely to be caught by the maximum system latency.

5.5. Evaluation of the Mathematical Model

The proposed mathematical model can be quickly verified using the experiment results so far. The first part of the model dealing with the real-time execution can be verified using the PWM experiment. As seen in Figure 5.4, the ratio $t_{Wpassed}/(t_{Snew} - t_S)$ derived directly from Eq. (4.1) is reflected in the output signal. So the first part can be regarded as verified.

The second part of the mathematical model implying that the time delay between concurrent output signals should reduce proportionally by $(H-1)/(m_s-1)$ with output ordering is observed in the concurrent outputs experiment, as shown in Figure 5.12. In this experiment, $H = 7$ and $m_s = 2$, and the reduction ratio is 6.7, which can be deemed close to 6. The second part of the mathematical model is thereby verified.

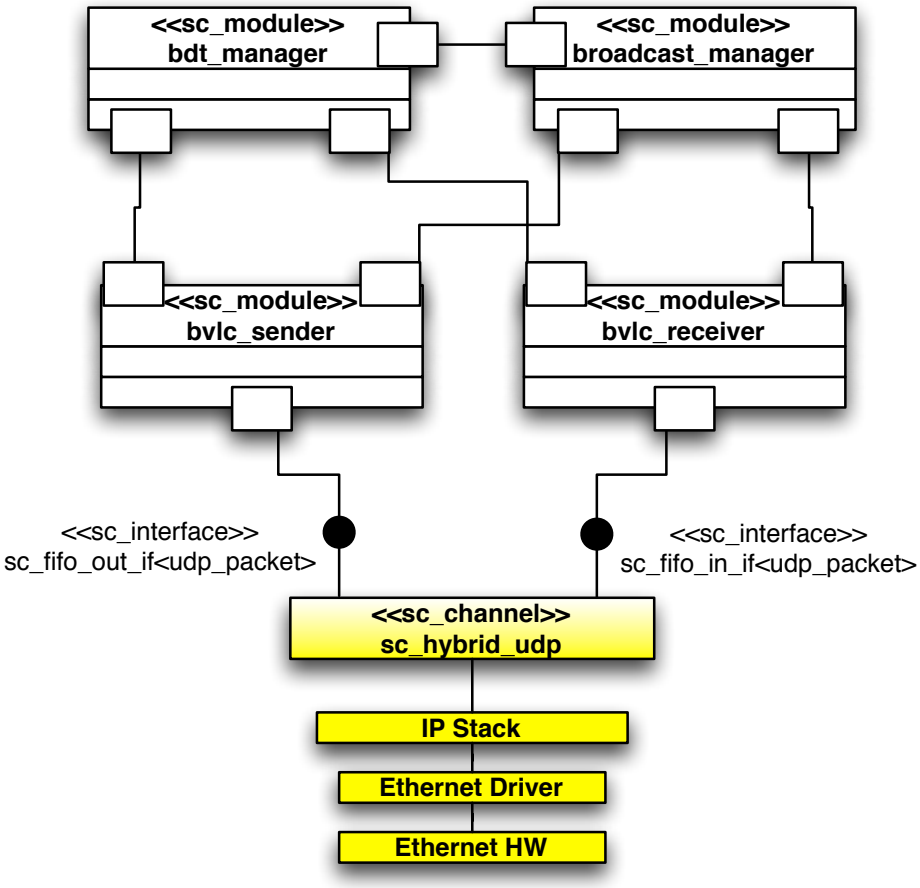


Figure 5.15. Model of BBMD

5.6. Real-life Experiment

Figure 5.14 shows the setup of BACnet Broadcast Management Device (BBMD) experiment. BACnet is a communication protocol used widely in building automation networks [27]. BBMD is a subset of the BACnet protocol and is used in BACnet networks running over the Internet Protocol (IP) to ensure that the broadcast packets used by BACnet are distributed correctly to all IP subnetworks constituting the BACnet network. In this experiment a non-timed TLM of a new BBMD has been created, which might be then extended to a fully BACnet-compliant device. Our method has allowed us to test this model with real partners already in TLM phase. The test network consists of two IP subnets comprising four Siemens S7-300 building automation stations, a management station for monitoring other stations and a PC acting as an IP router and a packet sniffer. Each IP subnet needs one BBMD. One of the build-

ing automation stations has been configured to act also as a BBMD for one subnet. For the other subnet the BBMD model has been connected to the network using our method. In addition to the traffic between the management station and the building automation stations, there has been peer to peer traffic between building automation stations shown with arrows in Figure 5.14.

Our transaction-level model of the new BBMD has performed very well in the experiment. It has outperformed the real BBMD in terms of response time and packet drop rates. Under a traffic burst of 2000 incoming packets per second, our model has not dropped any packet while the real BBMD has dropped 67% of packets. As the model was non-timed the accuracy of response time has not been an evaluation criterion and the model has outperformed the real BBMD in average response time up to 80 times.

6. CONCLUSIONS

In this study a hardware-in-the-loop concept for hardware/software co-design of real-time embedded systems has been developed, implemented with SystemC and evaluated experimentally.

The lack of a structured communication mechanism between the virtual, i.e. modeled, and the real subsystems has been addressed with the concept of *hybrid channel*. This concept allows minimal interference to SystemC model development and also provides a very clear interface via SystemC's native mechanisms. Hybrid channels are also very generic tools to implement every kind of communication between the real and virtual subsystems.

In order to let virtual subsystems behave according to the real-time in an accurate manner, a real-time patch to the SystemC kernel has been developed. This patch is non-intrusive to the SystemC model, it allows all models running on top of the patched simulation kernel to behave according to the real-time in a transparent manner. A deterministic behavior for industrial applications has been achieved even with inexpensive off-the-shelf components like the standard parallel port and Ethernet hardware.

Polling remained as an obstacle in the initial phase of our research, as it has posed a difficult tradeoff between the I/O performance and the simulation performance, both of which are important. Adaptive polling has been devised as a remedy, but a fully event-driven simulation kernel capable of incorporating external events could be realized, too. Polling is still an option for simple configurations. Adaptive polling can be used under circumstances where complex OS constructs have to be avoided to reach lower I/O latencies. Event-driven solution can be used even more widely after ensuring better behavior of complex OS constructs, e.g. by using a native RTOS instead of Linux with real-time improvements.

The mathematical model has also been developed and verified in the scope of this study. This model can foremost be used as a first test to see whether our method is applicable to a given SystemC specification. The empirical data needed by the model is easy to obtain. Additionally, the mathematical model can also be used in order to understand bottlenecks and improvement points in the simulation's execution. The mathematical model can also be adapted to improved SystemC kernels where parallelism is employed to increase simulation's performance.

The level of determinism gained via RT_PREEMPT solution of Linux is satisfactory for our purposes. API-transparency is a big advantage of this approach. However, this method remains largely manual. The modeling platform needs to be tested and tuned until a satisfactory result is obtained. Anyway, Real-time Linux [10] is constantly being improved, and new tools are being developed. So the manual approach is fading in favor of more structured ones, which favors our choice to use RT_PREEMPT.

Due to the inherent nature of our approach, only models able to run at least as fast as real-time can be targeted. Thus, industrial applications domain with transaction-level models is feasible with the current state-of-the-art. Nowadays there are a lot of studies in the fields of increasing computation power of processors and increasing simulations' execution speed via parallel execution or optimization. Further advances in these fields will belivably multiply the possible domains to use the proposed method.

The scalability of our method, i.e. the change in performance for SystemC models mainly with higher number of processes, but also with higher number of modules, channels, hybrid channels etc., remains as a future work. While the BBMD experiment is a realistic one, it is still small compared to specifications developed in the industry. Experiments should be done with specifications from the industry, although the mathematical model can still be used to estimate the performance with larger-scale SystemC specifications.

APPENDIX A: PUBLISHED WORK OF THIS THESIS

1. Fennibay, D., Yurdakul, A., Sen, A., "Introducing Hardware-in-Loop Concept to the Hardware/Software Co-design of Real-time Embedded Systems", *International Conference on Embedded Software and Systems, 7*, pp. 1902-1909, 2010.
2. Fennibay, D., Yurdakul, A., Sen, A., "Hardware-in-the-loop for hardware/software co-design of real-time embedded systems" (Poster), *DATE'10 Workshop Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications*, 2010.
3. Fennibay, D., Yurdakul, A., Şen, A., "Endüstriyel Uygulamalar için SystemC ile Döngü İçinde Donanım", *Ulusal Yazılım Mühendisliği Sempozyumu, 4*, pp. 219-226, 2009.

REFERENCES

1. Groetker, T., S. Liao, G. Martin, and S. Swan, *System design with SystemC*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

2. Benini, L., D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Virtual in-circuit emulation for timing accurate system prototyping", *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pp. 49–53, 2002.

3. Kim, N., H. Choi, S. Lee, S. Lee, I. Park, and C. Kyung, "Virtual chip: making functional models work on real target systems", *Proceedings of the 35th annual conference on Design automation*, pp. 170–173, ACM New York, NY, USA, 1998.

4. Bacic, M., "On hardware-in-the-loop simulation", *44th IEEE Conference on Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05*, pp. 3194–3198, 2005.

5. "IEEE Standard SystemC Language Reference Manual", 2005.

6. "Industrial communication networks - Fieldbus specifications", 2007.

7. "Industrial communication networks - Profiles", 2007.

8. Schriber, T. J. and D. T. Brunner, "Inside discrete-event simulation software: how it works and why it matters", *WSC '04: Proceedings of the 36th conference on Winter simulation*, pp. 142–152, Winter Simulation Conference, 2004.

9. Fujimoto, R., *Parallel and Distributed Simulation Systems*, Wiley-Interscience, New York, 2000.

10. "Real-Time Linux Wiki", `http://rt.wiki.kernel.org/index.php/ Main_Page`, 2009.

11. Mantegazza, P., E. Dozio, and S. Papacharalambous, "RTAI: Real time application interface", *Linux Journal*, Vol. 2000, No. 72es, 2000.

12. Rose, A., S. Swan, J. Pierce, and J.-M. Fernandez, "Transaction Level Modeling in SystemC", Technical report, OSCI TLM-WG, 2005.

13. Schirner, G. and R. Dmer, "Using Result Oriented Modeling for Fast yet Accurate TLMs", Technical report, CECS-05-05, 2005.

14. MathWorks, "xPC Target", `http://www.mathworks.com/products/xpctarget/`, 2010.

15. MathWorks, "Real-Time Windows Target", `http://www.mathworks.com/products/rtwt/`, 2010.

16. Herrera, F. and E. Villar, "A framework for embedded system specification under different models of computation in SystemC", *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pp. 911–914, ACM, New York, NY, USA, 2006.

17. Monton, M., A. Portero, M. Moreno, B. Martinez, and J. Carrabina, "Mixed SW/SystemC SoC Emulation Framework", *IEEE ISIE, 2007.*, pp. 2338–2341, 2007.

18. Nakamura, Y., K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura, "A fast HW/SW co-verification method for SoC by using ac/c++ sim. and FPGA emu. with shared register comm.", *Proc. DAC*, 2004.

19. Underwoood, R., "An Open Framework for Highly Concurrent Hardware-in-the-Loop Simulation", *Master's thesis, University of Misouri-Rolla*, 2007.

20. Lu, B., X. Wu, H. Figueroa, and A. Monti, "A low-cost real-time hardware-in-the-loop testing approach of power electronics controls", *IEEE Transactions on Industrial Electronics*, Vol. 54, No. 2, pp. 919–931, 2007.

21. SynaptiCAD, "PinPort", http://www.syncad.com/ pr_pinport_release.htm, 2002.

22. Koehler, C., A. Mayer, and A. Herkersdorf, "Chip Hardware-in-the-Loop Simulation (CHILS) - Embedding Microcontroller Hardware in Simulation", *Proceedings of the 19th IASTED International Conference on Modelling and Simulation*, Acta Press Inc,# 80, 4500-16 Avenue N. W, Calgary, AB, T 3 B 0 M 6, Canada, 2008.

23. Nageldinger, U., A. Pyttel, and H. Kleve, "System Simulation Speedup Combining SystemC Models and Reconfigurable Hardware", http://speac.fzi.de/WORKSHOP2/Speac_Paris_2004_01.pdf, 2004.

24. Ramaswamy, R. and R. Tessier, "The Integration of SystemC and Hardware-Assisted Verification", *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pp. 1007–1016, Springer-Verlag, London, UK, 2002.

25. Trams, M., "Realtimify - A small Tool for Real Time SystemC Simulations", http://www.digital-force.net/download.php?file=publications/systemc_realtimify.pdf, 2005.

26. Yaghmour, K., "Adaptive domain environment for operating systems", *Opersys Inc.*, 2001.

27. "BACnet - A Data Communication Protocol for Building Automation and Control Networks", 2004.