HdSC: A Fast and Preemptive Modeling for on Host HdS Development

Bruno Prado[1,2], Edna Barros[2], Thiago Figueredo[2] and André Aziz[2,3]

[1] Departamento de Computação, Universidade Federal de Sergipe, Sergipe, Brazil
[2] Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brazil
[3] Departamento de Estatística e Informática, Universidade Federal Rural de Pernambuco, Pernambuco, Brazil
e-mail: bruno.prado@ufs.br

ABSTRACT

In modern embedded systems, the Hardware-dependent Software (HdS) plays a critical role due to its processor and platform dependency, such as device drivers and boot initialization. To support HdS development starting in an initial system design phase, fast and accurate preemptive processor models should be provided for simulating the software. These models should provide a register level interface to enable a compatible programming view on the host machine environment. This paper presents a strategy for processor modeling that enables HdS development, using the host machine tool chain. The proposed approach supports the specification of platform components, such as processor, software and devices accessed through the data bus and interruption interfaces. An adaptive technique for timing estimation is being proposed, which is very accurate and show a high simulation performance. Supporting the device driver development and interruption service routines, these two features can be implemented and simulated at an early system design phase and requiring no Instruction Set Simulator (ISS). This ISS model would be required only for performance and accuracy comparison purposes. Experimental results show that the virtual platform specified using this proposed approach can perform faster (up to 760x speed up) and high accurate (up to 12%) software simulation on native host environment.

Index Terms: HdS, Modeling, Preemptive, Simulation, Instruction Set Simulator.

1. INTRODUCTION

The increase in complexity of electronic system in consumer electronics devices, such as smart phones or tablets, due to new applications and human interfaces demands more Hardware-dependent Software (HdS) to handle these devices. This type of software corresponds about 80% of contents and functions [2] of the system. This challenging scenario cannot be dealt with traditional development approaches, such as Instruction Set Simulator (ISS), running at functional or cycle-accurate levels. The very fine granularity of ISS provides a precise execution timing report, however with the cost of a very low simulation performance. This longer simulation time prevents an efficient design space exploration for complex systems. Actual and future designs will require even more features integrated, leading to much more complex systems which must handle various hardware devices as well as multiprocessor platforms in order to explore thread parallelism. Thus a higher level of abstraction will be mandatory to achieve successful system designs.

In Figure 1, a modern consumer device is illustrated in terms of its more common interfaces. These multiple ways of interaction include touch sensitive screen, storage using different standards and wireless interfaces for short (Bluetooth) and long (802.11) range. The role of HdS in this example is to enable the system initialization, running boot procedures and hardware configuration. When the system is up, the HdS provides an Application Programming Interface (API) for an abstract device access and control. Also known as Hardware Abstraction Layer (HAL), this software layer interacts closely with the hardware, providing a generic way toaccess it.

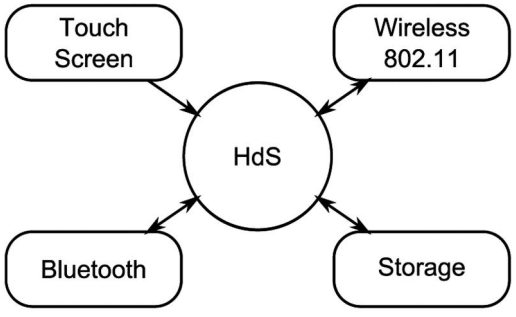

Figure 1. HdS example

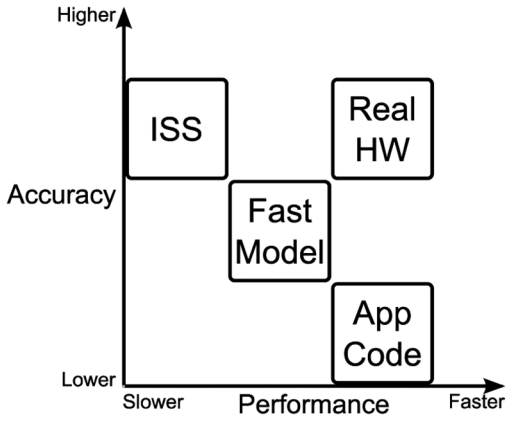

Figure 2. Simulation trade-off

As shown in Figure 2, software execution environments must deal with the trade off between simulation performance and accuracy. When more accurate simulation is chosen (for example using ISS), a slower execution is obtained, resulting in longer simulation time. The choice of an untimed model (App Code) yields to very inaccurate results, but delivers a faster execution. The best choice would be to use the real hardware (Real HW) for software development, but this option has some serious drawbacks: software development and debugging on chip can be hard due to lack or limitation of tools. Moreover, software development normally is started before the hardware is available and its specification cannot be even complete at this design phase. To cope with this scenario, a better trade off between performance and simulation precision can be achieved using Transaction Level Modeling (TLM Model), a widely accepted approach for communication abstraction. The TLM approach increases simulation performance, keeping accuracy due to its well-defined interfaces and timing approximation.

The problem addressed by this work consists in abstracting the HdS development environment, providing an estimation for application timed execution, a mechanism for process scheduling and external communication to platform components via register based interface. This work aim is to provide a development framework that enables an equivalent software execution in the proposed processing model and in the traditional ISS model. All software components require minimal code ports and no extra design skills for modeling.

This paper is organized as follows: in section 2, we review the related work and provide a brief description and link to the proposed work. In section 3, the proposed modeling approach is described in detail, including development framework and architecture model. In section 4, the HdSC kernel and language are described, including the techniques adopted and a short example. Section 5 and 6 shows the multiprocessor paradigm features and the results obtained through validation experiments performed. Finally, section 7 presents the work conclusion with future works.

2. RELATED WORK

To improve the related work analysis, the considered approaches are organized in three main areas: software generation (software is generated from system models using a set of tools), hybrid model simulation (a combination of native and ISS simulation) and native simulation (all components execute in native environment without system emulation).

A. Software Generation

Embedded software generation from a system model [6] is an effort to reduce the costs of platform-based hardware and software co-design. The set of tools can perform verification, partitioning and simulation, and code generation from the system level specification. The support for software generation from a system model is also provided by the proposed approach and allows abstract and faster modeling for both hardware and software.

System Level Design Language (SLDL) for embedded software generation [14][11][12] is an approach to address the challenging design complexity with system models in a higher level of abstraction. The methods presented automatically generate software directly from system specification written in SLDL, allowing several refinement steps and intermediate models in development flow. This flow speeds up the development and reduces the number of errors in system design, since higher abstraction levels are used. The proposed work aims to achieve higher abstraction models in order to deliver a fast and robust framework. Through a SLDL specification, the proposed approach enables software execution in a high-level processor model.

The HdS is one of the most challenging area in software development and the work DevC [9] proposes a Domain Specific Language (DSL) for device driver specification. This work aims to abstract from the engineer some device driver implementation details, defining the structures for hardware and software views. This specification is synthesized to a hardware controller and device driver code that are validated in a virtual platform environment. The main contribution of this paper to the proposed work is the DSL creation to model HdS behavior, dealing with hardware and software aspects.

B. Hybrid Model Simulation

A hybrid simulation strategy [7] is proposed in order to speed up software simulation performance to accomplish the task of efficiently develop complex software. In this framework, all software code that is platform-independent is executed on the host machine, while

platform-dependent code is executed in an ISS. This work is quite similar to the proposed work, as can be seen in detail in the next section, except by the fact that all HdS code is executed, debugged and time estimated on the host machine, instead of using an ISS for platform-dependent code.

Software execution speed up using abstract models [8] is a strategy to reduce the amount of software that runs on the slow ISS. This technique uses a platform-independent portion of code, such as a RTOS kernel, and emulates its behavior using a system model specification. The proposed work uses the same strategy to model the whole software as a system model component, instead of selecting a platform-independent part of the code. This proposed strategy keeps fast and accurate simulations, avoiding the ISS usage.

C. Native Simulation

System TLM for HdS design [3][4] is a native execution approach to hardware and software development supporting hardware interface to software and interruption service routines specification. This work provides an API to HdS access hardware resources and enables the software preemption through an interruption handler built in the processor model. The TLM approach and interruption handling are the major contributions to proposed work, improving register software interface modeling and interruption handling. These register and interruption functions are implemented through code instrumentation, providing register level access instead of function based hardware access.

Software performance estimation [13] is an important feature in embedded software development and ISS became an essential part in this work. However, the high increasing complexity of embedded software turns the ISS a slow solution and time consuming approach. This work provides a native execution strategy based on host native code instruction annotation with cross compiler information, thus allowing precise estimations of execution. The key contribution to the proposed work is the concept of code instrumentation to retrieve detailed execution information in host machine, while keeping the same code that runs on target.

The cycle-counting modeling accuracy [10] provides a timing accurate approach that eliminates excessive functional details, in order to achieve high simulation speeds. This strategy is implemented using a cycle-count-accurate (CCA) processor model that abstracts internal module pipeline and cache into abstract models. These models implement functional and timing behavior in the processor interface perspective. This high speed and accurate modeling demonstrate that software simulation time can be improved keeping high accuracy levels and the proposed work aims to accomplish the complex task of HdS development abstracting excessive ISS behavior details.

3. THE PROPOSED MODELING APPROACH

The aim of this work is to provide a high level modeling approach for processor based platforms, called HdSC, supporting embedded software development and performance assessment at an earlier design phase. The proposed modeling mechanism provides external communication through the specification of data bus and interruption operations using SystemC SLDL [5]. Additionally, strategies for timing approximation and hardware/software scheduling are proposed to achieve accurate simulation results. The proposed modeling approach supports uniprocessor as well as multiprocessors-based platforms.

A. Development framework

This proposed work is conceived to allow native embedded software execution, speeding up simulation performance and keeping high accuracy levels when compared to ISS models. Another important supported feature is a data bus and interruption interfaces, enabling TLM communication and device driver development using Interruption Service Routine (ISR). In other words, the HdS designer can execute the software on the native environment, improving simulation performance, keeping high timing accuracy and supporting preemption for complex HdS development.

The proposed development flow can be visualized in Figure 3, which consists of HdSC modeling for processor (Processor Module) and software (Software Module). These components support application requirements and native compilation for executable platform simulation.

In the processor modeling branch, a high-level processor model is implemented with both hardware and software views. The hardware view generally is necessary to accomplish data bus operations, such as external com-

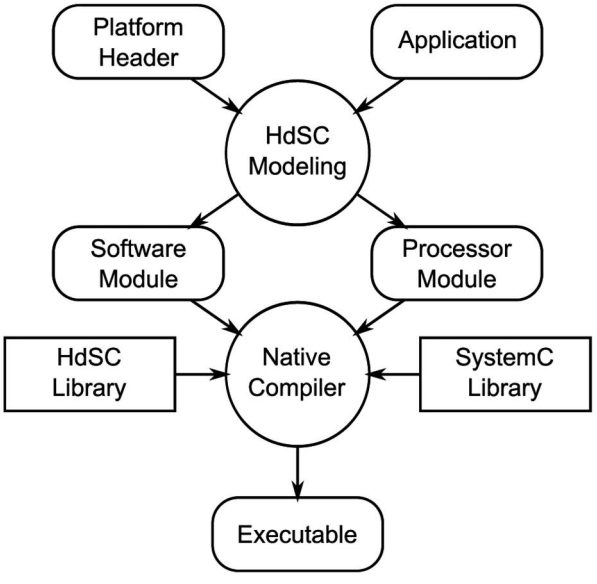

Figure 3. Development flow

munication and interruption handling. However, the system designer can specify any other specific processor characteristic, such as internal registers or extra interfaces.

In the software modeling branch, a low-level software module is implemented, enabling the application access to the platform. This module works as a basic layer for application storage and execution. The main idea is to emulate a program memory storing the application running on the target processor, in a higher abstraction level, encapsulating the application source and enabling multiple private instances.

When all of hardware and software aspects are properly implemented, the processor and software modules are instantiated and connected. This HdSC virtual platform may contain whichever component is required by the application, such as data bus, timer, serial interface or interruption controller. All of these components will be compiled by a native C++ compiler, with HdSC and SystemC libraries, to generate an executable specification of the platform. The native application of the designed platform allows both hardware and software simulation and debugging, using all available tools in the host environment.

B. Model Architecture

This section details the processor and software modules, explaining how their conception allows the abstraction of the ISS and the binary memory components, respectively. The key idea is to model each aspect of the processor and the binary memory in native programming elements, applying SystemC and C++ constructors. This strategy delivers higher level processing models, avoiding the excessive details from ISS and enhancing model construction.

Another key point, in the proposed work, is to enable the same target software to be executed on the HdSC model. In other words, the same software that can be compiled for real hardware or ISS model can be executed on the HdSC processor. Aside from minimal implementation changes, the source code remains compatible with cross compilers, without affecting application behavior or performance at binary code level. The native execution of embedded software enables the use of a rich set of tools (compilers, debuggers, etc) and provides much higher execution performance due to the lack of virtualization.

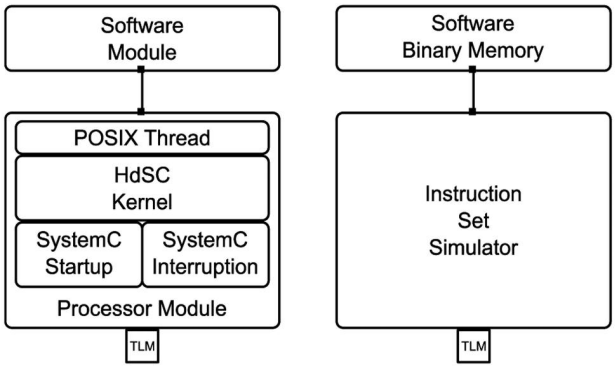

Figure 4. The proposed model (left hand side) and ISS model (right hand side)

The proposed model architecture (left hand side) is illustrated in Figure 4, side by side with the classic ISS architecture (right hand side). These two architectures are black box equivalent, allowing their straightforward use in a virtual platform. In other words, the system designer can replace the processor and software modules by the ISS module and its software binary memory achieving the same system behavior.

The subcomponents of the Processor Module, responsible for software execution and interaction with the platform, are described in more detail as follows:

- POSIX Thread: in most operating systems (Unix, Windows, Embedded OSes, etc), the POSIX standard is wide adopted and its thread programming is available. The HdSC processor implements the application main function with its own thread, supporting all features provided, such as synchronization and scheduling. The POSIX thread also naturally supports the concurrent software execution, in other words, more than one processor executing the same software instance;
- HdSC Kernel: applying the notion of time in native software execution, this kernel performs timing estimation essential for accuracy and performance evaluation. Also, this part of the HdSC processor model schedules the software and platform tasks, controlling how hardware and software communication should be performed. This control is critical to avoid inconsistencies and incorrect behavior in the complete system simulation;
- SystemC Startup: this SLDL task starts the HdSC kernel, working as a checking point. There is a direct dependency on the SLDL simulator, thus the HdSC components are initialized by the SystemC kernel to ensure the correct behavior. When SystemC executes this process, the POSIX Thread is started and manages the HdSC kernel in order to execute the attached software;
- SystemC Interruption: portion of the HdSC processor responsible for receiving external interruption requests. All external requests come from SystemC modules, such as hardware peripherals, thus the management of them is also performed by a SLDL task. When an interrupt is generated, the HdSC Kernel receives the requisition to preempt the software execution and call the assigned ISR;
- TLM: defines how the HdSC processor module accesses external devices. In this particular example, the TLM paradigm is employed due to

the fact that it supports a high level communication among processes. However, the system designer can use a more detailed approach, such as pin accurate, because system architect freely defines this interface.

Detailing Software Module, it is conceived to be connected and executed by the Processor Module. As explained earlier, this component mimics a binary memory function, behaving as an application container. The C++ class definition ensures that each instance of same application has its own memory space for variables, enabling multiple and private instances in the same platform. This software encapsulation is essential for system multiprocessing, which uses multiples processors, due to concurrency and data sharing.

4. HDSC NATIVE SIMULATION

This section provides more detailed information about the HdSC native simulation strategy and how it improves HdS development. A fully native execution is naturally faster than virtualized execution in ISS, due to reduced amount of needed computation. The simulation performance is boosted when compared to the ISS model, however it is necessary to keep a high accurate timing behavior. This requirement is achieved by two essential proposed features: code instrumentation, to enable interrupt routines preemption at basic block level; and dynamic timing translation, for accurate cycle estimation from native time measurement.

A. Code Instrumentation

In HdS development, one of the most critical task is the design of ISR to handle platform interruption requests. It is highly desirable to avoid unexpected behavior due to sequence invocation order or timing variation. When ISS models are employed, the software execution preemption occurs whenever the interrupt flag request is enabled, halting the current execution flow. After the suspension, the execution flow is changed to the ISR and when it ends, the previous software execution flow is resumed.

In ISS, the software is a sequence of instructions executed step by step to implement the desired behavior. Before or after the execution of a single instruction, the interruption flags are checked for ISR activation. In other words, if interruption is enabled, the processor checks for new requests every each instruction in software execution. This fine granularity provides a precise execution flow, supported by dedicated hardware components.

Considering a native software execution, there is no implicit procedure to halt the current flow in order to execute another one. This behavior could be achieved by explicit mechanisms, such as code instrumentation. Code instrumentation automatically inserts checkpoints at basic block level to enable interruption checking, instead of instruction level as occurs in ISS. Once the platform generates an interruption request, before or after basic a block execution the interruption is handled and the software flow is halted. In other words, instead of checking interruptions at instruction level, the HdSC approach checks for interruptions every each basic block execution.

In Figure 5, it is illustrated which control flow statements are instrumented and how this automatic code instrumentation is performed. The control flow directives if, switch, while and for are automatically instrumented by HdSC to check for pending interruption requests. The basic block is executed atomically (atomic block) until the next checkpoint is reached. The automatic code instrumentation is performed by the native compiler preprocessor which inserts HdSC's interruption checking function and if necessary triggers the ISR execution.

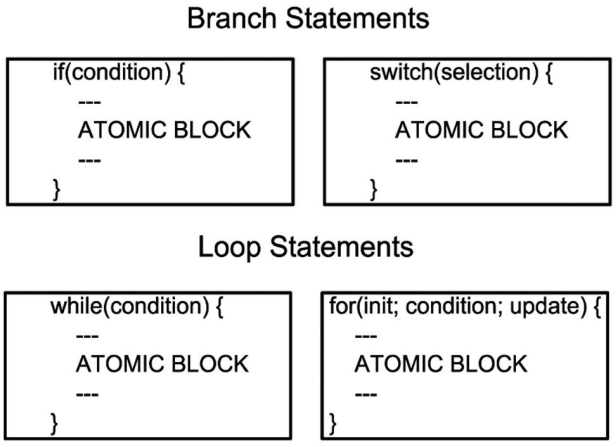

Figure 5. Code Instrumentation

Instead of instruction by instruction emulation, the native software execution runs a sequence of basic blocks, providing a coarser-grained simulation when compared to ISS. This coarser granularity speeds up execution due to the reduction of checking procedures and their overhead. However, this flow abstraction implies a lower interruption handling resolution due to the fact of the execution of basic blocks as the minimal step. The logical consequence is the loss of accuracy, but as discussed earlier, there is a trade off between performance and accuracy. Keeping accuracy at acceptable levels is the key to boost simulation performance.

B. Run-time Dynamic Timing Estimator

The native execution of a HdS application, without any extra tool or library in the host machine, lacks timing information, such as number of clock cycles and executed instructions. However, the timing information is mandatory in HdS development due to external communication and peripherals timing requirements. In other words, the simulation time is an essential part of HdS behavior and must be supported by the native tool chain.

To accomplish this timing requirement, three related work approaches were considered: software generation from high level models (see subsection 0) where the time is a model parameter; hybrid model (see subsection 0) in which both high level model and ISS are integrated; and native simulation (see subsection 0) that executes embedded software on the host machine with timing annotation. This novel work proposes a dynamic timing translation, instead of static timing annotation acquired from previous ISS simulations. The key concept relies in the dynamic mapping between host machine time and virtual platform time, providing the equivalent of software execution delay in virtual platform approach. The dynamic timing mapping allows the synchronization between the native application execution and the SLDL environment, keeping consistency and high accuracy of HdS behavior.

$$\begin{aligned} &BB\ (native\ time) \leftrightarrow BB\ (simulation\ time) \\ &VP\ (native\ time) \leftrightarrow VP\ (simulation\ time) \end{aligned} \quad (1)$$

Once the application is started, each basic block has its execution time measured, using the host's high precision timer. The native execution time of the virtual platform is also measured using this same high precision timer, as well as the simulated time. In Relation 1, the basic block (BB) and virtual platform (VP) have their native times measured plus the availability of the VP simulation time. With this information it is possible to estimate BB equivalent simulation time in VP.

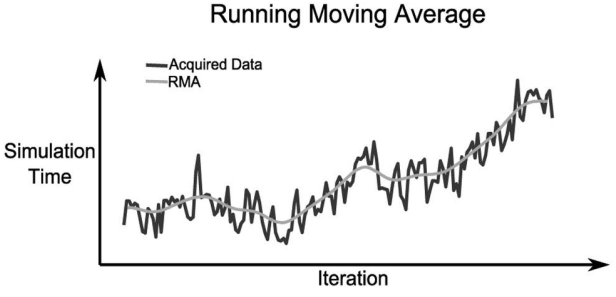

Figure 6. RMA behavior over iterations

The obtained estimation samples acquired throughout the software iterations are noisy due to the interference of the host unrelated application and services sharing the same resources. These estimations are smoothed by Running Moving Average (RMA) filtering, as can be seen in Figure 6. This filtering greatly reduces out of context points and provides a much more reliable basic block timing estimation.

$$Average_t = (1 - CPI) \times Average_{t-1} + CPI \times Current_t \quad (2)$$

$$Estimation = CLK \times \left(\frac{Current_t}{Average_t} \right) \quad (3)$$

In order to estimate real hardware performance, the processor clock cycle period (CLK) and cycles per instruction (CPI) are needed as parameters to RMA filtering. The Formula 2, keeping the range $0 < CPI \leq 1$ constraint, defines the amount of stored samples considered in the filtering process. This accurate average estimation ($Average_t$) of each basic block is continuously updated and the current basic block sampling ($Current_t$) delivering a dimensionless value. Multiplying this obtained value and platform's CLK (see Formula 3), plus custom defined model parameters, the current basic block simulation time is estimated. In other words, the equivalent platform time of basic block is calculated and approximated to enable an accurate timing behavior of embedded software in this high level model.

C. Modeling Example

This subsection will provide an example of the proposed approach, comprising the hardware modeling

```
1   /**
2    * My Software class declaration
3    */
4   HSC_SOFTWARE_TEMPLATE(my_address, my_data);
5   HSC_SOFTWARE(my_software) {
6      // My Software constructor
7      HSC_SOFTWARE_CTOR(my_software) {
8         // Binding registers
9         bind(DATA, 0x80000000);
10        bind(CONTROL, CONTROLbits, 0x80000004);
11     }
12     ...
13     // Software main implementation
14     void hsc_software_main() {
15        // Calling main
16        if(hsc_name().compare("my_software.elf") == 0)
              main_my_software(hsc_argc(), hsc_argv());
17     }
18     // Software interruption implementation
19     void hsc_software_interruption(const
   hsc_device_code_t& d) {
20        // Executing IRQ bind handler
21        (this->*irq_handler_table[d])();
22     }
23     ...
24     // CONTROL bits definition
25     HSC_SOFTWARE_REGISTER(CONTROLbits) {
26        // CONTROL fields
27        HSC_SOFTWARE_REGISTER_FIELD(WR);
28        HSC_SOFTWARE_REGISTER_FIELD(RD);
29        // CONTROL mapping
30        HSC_SOFTWARE_REGISTER_MAP
31        (
32           // Setting ranges
33           HSC_SOFTWARE_REGISTER_RANGE(WR, 1,
   1);
34           HSC_SOFTWARE_REGISTER_RANGE(RD, 0,
   1);
35        )
36     };
37     // DATA register
38     HSC_SOFTWARE_REGISTER_DECLARATION(DATA)
   ;
39     // CONTROL register
40     HSC_SOFTWARE_REGISTER_DECLARATION(CONT
   ROL);
41     // CONTROL bits register
42     HSC_SOFTWARE_REGISTER_CUSTOM_DECLARAT
   ION(CONTROLbits, CONTROLbits);
43  };
```

Listing 1. Software Modeling

(processor), the software modeling (application container) and the source code application in C. All details of the proposed language will be explained from syntax through the implementation of a hypothetical platform. This system is composed by a peripheral with two registers for data manipulation (DATA) and operation request (CONTROL).

In the software modeling, shown in Listing 1, the designer must specify the platform address and data types (line 4). These types could be defined in any desired abstraction level, such as TLM or pin accurate level. The chosen type must match the processor model interface, once it is responsible for data bus operations. In the constructor (lines 6 to 11), all modeled registers must be bound to its platform addresses and specific initialization should be performed. Finally, the main loader (lines 13 to 17) and the interruption handler (lines 18 to 22) are implemented, followed by register definitions (lines 24 to 36) and declarations (lines 37 to 42).

```
1   /**
2    * My Processor class declaration
3    */
4   HSC_PROCESSOR_TEMPLATE(my_address, my_data);
5   HSC_PROCESSOR(my_processor) {
6       // My Processor constructor
7       HSC_PROCESSOR_CTOR(my_processor) {
8           // Enabling traps
9           enable_traps();
10          // Starting up interruption handler
11          SC_THREAD(my_interruption_handler);
12      }
13      ...
14      // TLM Bus port
15      my_bus_port BUS;
16      ...
17      // Enable Trap flag
18      bool ET;
19      ...
20      // Enable Traps method
21      void enable_traps() {
22          // Enabling Traps
23          ET = 1;
24      }
25      // Disable Traps method
26      void disable_traps() {
27          // Disabling Traps
28          ET = 0;
29      }
30      // My interruption handler SystemC process
31      void my_interruption_handler() {
32          // Loop forever
33          while(1) {
34          // Waiting interruption event
35          wait(BUS->get_interrupt_event());
36          // Retrieving device code
37          hsc_device_code_t device_code = BUS-
    >get_interrupt_device_code();
38          // Waiting for ET (Enable Traps) if not enabled
39          while(!ET) wait(1, SC_NS);
40          // Calling my software interruption handler
41          hsc_processor_interruption(device_code);
42          // Bus interruption acknowledgment
43          BUS->interruption_acknowledge();
44          }
45      }
46      // Processor read call back implementation
47      void hsc_processor_read(const my_address& a,
    my_data& d) {
48          // Calling bus read
49          BUS->read(a, d);
50      }
51      // Processor write call back implementation
52      void hsc_processor_write(const my_address& a,  const
    my_data& d) {
53          // Calling bus write
54          BUS->write(a, d);
55      }
56  };
```

Listing 2. Processor Modeling

The processor or hardware modeling, described in Listing 2, requires platform address and data types (line 4) for the data bus interface. This interface must be compatible with the software model definitions. The default constructor (lines 6 to 12) initializes all default processor register fields and values (line 9) and starts the SystemC interruption thread handler (line 11). The TLM interface (lines 14 to 16) and the processor specific variables are declared (lines 17 to 19), including interruption handler behavior (lines 30 to 45) and data bus access callbacks (lines 46 to 55).

```
1   /**
2    * Real Hardware header
3    */
4   // Real Hardware header
5   #ifdef __REAL_HW__
6   // Standard I/O include
7   #include <stdio.h>
8
9   // Register address definitions
10  #define DEVICE_DATA_ADDRESS (0x80000000)
11  #define DEVICE_CONTROL_ADDRESS (0x80000004)
12
13  // Register structures definitions
14  typedef struct CONTROLbits {
15      unsigned int   : 30;
16      unsigned int WR : 1;
17      unsigned int RD : 1;
18  }
19  CONTROLbits;
20
21  // Register aliases definitions
22  #define DATA (*((volatile unsigned
    int*)(DEVICE_DATA_ADDRESS)))
23  #define CONTROL (*((volatile unsigned
    int*)(DEVICE_CONTROL_ADDRESS)))
24  #define CONTROLbits (*((volatile CONTROL-
    bits*)(DEVICE_CONTROL_ADDRESS)))
25
26  // HdSC Model header
27  #else
28  // HdSC My Software include
29  #include <my_software.hpp>
30  // HdSC wrapping software header for code instrumentation
31  #include <hsc_wrapper.hpp>
32  #endif
33
34  // Main function
35  int main(int argc, char* argv[]) {
36      // Loops for 256 iterations
37      while(DATA < 256) {
38          // Data write request
39          CONTROLbits.WR = 1;
40          // Data increment
41          DATA = DATA + 1;
42          // Clearing control register
43          CONTROL = 0;
44          // Data read request
45          CONTROLbits.RD = 1;
46          // Outputing data value
47          printf("DATA = %d\n", DATA);
48          // Clearing control register
```

```
49          CONTROL = 0;
50      }
51      // Returning success
52      return 0;
53  }
```

Listing 3. Application in C

This source code application in C, detailed in Listing 3, has in the first part a dual platform header definition (lines 5 to 32). This dual platform defines which model will be used: HdSC or ISS. The main function (lines 34 to 53) iterates for 256 times writing the incremented data in the peripheral register and reading it after this operation (lines 38 to 49). This application was implemented just to demonstrate HdSC modeling features, executing the same ISS software and requiring almost no code porting.

5. SUPPORTING MULTIPROCESSOR PLATFORMS

The great majority of embedded systems are including multiprocessing as an approach to keep system performance growing. It is an alternative approach to single core processing, which is limited by conceptual and architectural bottlenecks, such as the Instruction Level Parallelism (ILP) and power consumption limitation. These limitations were the main causes for this new processing paradigm that brings challenges for development and validation of complex systems.

A. Concurrent Software Execution

To take full advantage of a multiprocessor platform, the embedded software must be implemented to properly handle the shared resources, avoiding the concurrency issues. The compiled application is loaded in a memory and executed by two or more processors. There are two main strategies in multiprocessing: Symmetric Multiprocessor System (SMP) where all processors share the same centralized memory; and Asymmetric Multiprocessing (AMP) where the application is stored and executed in separated memories.

Whatever strategy employed, the Figure 7 summarizes the N processors to 1 software relation. It is important to notice that the same sequential or pseudo parallel behavior obtained in the single processor scenario must be achieved in a multiprocessor platform.

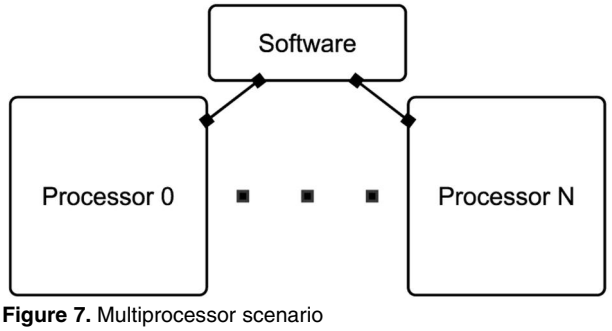

Figure 7. Multiprocessor scenario

The system designer can use as many processors as possible, but the overall system performance is limited by its sequential code. The Amdahl's Law (defined by Formula 4) defines this maximum theoretical speed up that could be achieved by a multiprocessor platform.

$$Maximum\ Speed\ up = \frac{1}{(1-P)+\frac{P}{N}} \tag{4}$$

$$\lim_{N\to\infty} \frac{1}{(1-0.95)+\frac{0.95}{N}} = 20 \tag{5}$$

For example, if a program has 95% of parallel (P) and 5% of sequential (1 - P) code, the maximum speed up is limited by 20 times (see Formula 5). No matter how many cores are used, this single task could not be improved indefinitely. However, modern complex platforms are composed by dozens of applications to deliver the complete system features. That is why multiprocessing can truly boost system performance, despite the size of a sequential application code.

B. Supported Concurrent Programming Model

There are several parallel programming techniques, but we will focus on the POSIX Thread API, which the HdSC kernel is based on, to provide concurrent software execution. Every processor instance, as shown in Figure 4, is powered by its own thread running on the local machine. All the available functions that can provide thread creation, termination, synchronization, scheduling and debugging are supported and are available in most Operating Systems, including the ones for embedded platforms.

```
1   /**
2    * Programming Model Example
3    */
4   // Real Hardware header
5   #ifdef __REAL_HW__
6   // Standard I/O include
7   #include <stdio.h>
8   // Standard Library include
9   #include <stdlib.h>
10  // POSIX Threads include
11  #include <pthread.h>
12
13  // HdSC Model header
14  #else
15  // HdSC My Software include
16  #include <my_software.hpp>
17  // HdSC wrapping software header for code instrumentation
18  #include <hsc_wrapper.hpp>
19  #endif
20
21  // POSIX Thread Mutex
22  pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
23  // Counter variable
24  int counter = 0;
25
```

```
26  // Main function
27  int main(int argc, char* argv[]) {
28      // Loops for 256 iterations
29      while(counter < 256) {
30          // Printing Processor ID
31          printf("Processor #%d\n", pthread_self());
32          // Mutex lock
33          pthread_mutex_lock(&mutex);
34          // Incrementing counter
35          counter++;
36          // Mutex unlock
37          pthread_mutex_unlock(&mutex);
38      }
39      // Returning success
40      return 0;
41  }
```

Listing 4. Parallel Software Example

The example in Listing 4 describes a simple concurrent counter executed by multiple processors. The race condition in the counter variable increment is properly handled by the use of mutex (declaration in lines 21 to 22) synchronization functions to lock and unlock the critical region (lines 32 to 37) shared by all processors. This example of concurrent synchronization avoids an inconsistent behavior and delivers the same behavior obtained in a single core platform. The processor instance identification (line 25) is also possible, for information purposes or task scheduling, for example.

Since the target platform supports POSIX Thread, this same source code could be directly used in HdSC models and real hardware. However, if it is not available in the target platform, the system designer can port or emulate the target functions in the software module. This seamless parallel programming approach, employing HdSC, boosts the multiprocessor software development by native and straightforward concurrency support. This efficient processing approach keeps the focus on software development and how the parallel code should be implemented.

6. EXPERIMENTAL RESULTS

To validate the proposed approach for high level hardware and software modeling for HdS development, some benchmarks and custom applications were executed to compare and analyze the performance and timing accuracy of the proposed modeling approach.

For comparison purposes, a virtual platform was specified using the SystemC SLDL and the ArchC Architecture Description Language (ADL) [1] infrastructure, powered by a SPARC-V8 ISS.

A. Accuracy and Performance Metrics

The accuracy and performance measurements are acquired from two metrics of simulation report: the simulation time and the execution time of simulation. The simulation time is the total amount of clock cycles simulated in virtual platform. The execution time of simulation is the run-time of the platform executable application in native host environment, in other words, it is the real time spent in simulation.

In related work [12], these two metrics are used to provide the number of cycles simulated in a certain time in seconds (Cycles/s). Another metric is adopted by [13] is MIPS, using the instruction timing information collected from ISS to report the number of executed instructions. This work uses the MIPS metric obtained from timing estimation plus processor specifications, such as the CLK and the CPI.

$$MIPS = \frac{\left(\frac{Simulation\ Time(ns)}{10^6 \times CLK(ns) \times CPI} \right)}{Execution\ Time(\text{sec})} \tag{6}$$

Instead of estimate the time of a single or few instructions, the whole basic block time is estimated, reducing the computational complexity of simulation. Since the clock cycle and the CPI of processor are known, it is possible to determine how many instructions can be executed in certain amount of time.

In other words, if the simulation time, the processor clock cycle and the CPI are known, the number of executed instructions can be directly calculated dividing the simulation time by processor clock cycle times CPI (see Formula 6).

$$Error = 100 \times \frac{\left| HdSC_{\#\,Instr} - ISS_{\#\,Instr} \right|}{ISS_{\#\,Instr}} \tag{7}$$

For accuracy analysis, the ISS simulation report is the reference solution, offering 0% accuracy error due to its high detailed modeling. The error is calculated using the Formula 7, which computes the modulo of difference between ISS and HdSC timing, in order to obtain a percent error from ISS reference.

$$Speed\ up = \frac{HdSC_{MIPS}}{ISS_{MIPS}} \tag{8}$$

To obtain the simulation speed up, the achieved HdSC MIPS rate is calculated from estimated simulation time, applying Formula 5. This HdSC MIPS is divided by ISS MIPS, delivering the speed up ratio, as can be seen in Formula 8.

It is important to notice that the proposed HdSC modeling aims to provide the same architectural efficiency from ISS strategy, delivering a behavior as close as possible to the ISS approach. Also is desired the reduction of the computational complexity, and thus the increasing performance by the reduction of execution time on the host machine.

B. Uniprocessor Platform

This platform is composed by a SPARC-V8 ISS running at 1 GHz clock frequency and performing 500 instructions per clock cycle (CPI = 1 / 500).

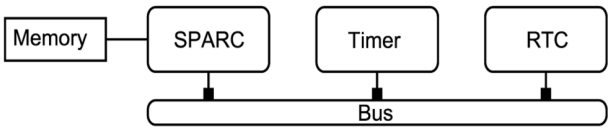

Figure 8. Reference virtual platform

Figure 8 shows how the processor is bound to other devices, such as application binary memory (Memory) and interconnection data bus (Bus). Also connected via data bus, the peripherals configurable timer (Timer) and real-time clock (RTC) were integrated to fulfill the benchmarks requirements.

The experiments were performed using two well known benchmarks (EEMBC Coremark and Dhrystone 2.1) and three custom applications for message printing (Hello World), mathematical intensive operations (Iterator) and multiple timer controller (Timer Control), detailed below:

- EEMBC Coremark: developed by the Embedded Microprocessor Benchmark Consortium, this industry class benchmark employs real-world algorithms to measure the embedded devices performance. Its implementation is focused in processor evaluation and it was executed using size 1000 performance parameters;
- Dhrystone 2.1: synthetic algorithms developed by Reinhold P. Weicker, this benchmark execute integer only operations and it is wide used for processor performance evaluation. The number of iterations is set to 100,000 to enable stable performance estimation;
- Hello World: considered as a base case study due to its minimalism and simplicity, illustrating that even the most basic examples can be successfully executed in this processing model;
- Iterator: an application that performs sequential mathematical operations, repeating for 1 billion times, over its local variables. This computer-intensive application is intended to match the worst case scenario in data processing, because all computation is dependent from previous calculation, preventing parallel execution and significant compiler optimization;
- Timer Control: this application was developed to perform I/O operations (external communication), exploring register model capabilities and validating HdS development behavior across the platforms. The start up code and ISR were implemented, enabling the control of up to 8 timers instances, showing the same behavior in ISS and HdSC platforms.

The simulation results when executing the mentioned applications are organized in Table 1, providing timing accuracy measurement. The applications are listed in the first column, in the second column the number of executed instructions in ISS is shown and in the third column the HdSC instruction counting estimating is shown. In the last column, the error between the precise ISS timing and the approximated HdSC timing is calculated.

Table 1. Instruction counting estimation.

Application	ISS	HdSC	Error
Coremark	2,369,025,691	2,379,923,209	0.46%
Dhrystone	102,099,192	102,761,469	0.65%
Hello World	1,011	1,118	10.58%
Iterator	1,700,008,234	1,516,477,037	12.10%
Timer Control	95,787,868	95,659,477	0.13%

Another important aspect, analyzed in application execution, is the simulation speed up achieved by the HdSC approach, detailed in Table 2. The performance achieved in ISS for each application are shown in the second column, with an average value of 0.32 MIPS. The proposed HdSC model performance is shown in the third column, providing an average value of 193.57 MIPS and speed up, shown in the last column, reaches an average improvement of 580.30 times.

Table 2. Simulation performance speed up.

Application	ISS	HdSC	Error
Coremark	0.36 MIPS	178.29 MIPS	495.26x
Dhrystone	0.37 MIPS	234.82 MIPS	634.65x
Hello World	0.15 MIPS	53.13 MIPS	354.20x
Iterator	0.37 MIPS	242.93 MIPS	656.57x
Timer Control	0.34 MIPS	258.68 MIPS	760.82x

Despite the simulation trade-off between performance and accuracy discussed earlier, this work demonstrated to provide a fast and accurate hardware and software modeling mechanism. When compared to precise and widespread ISS approach, the HdSC model provides up to 12% of timing error and up to 760.82 times of speed up, enabling a fully ISS compatible development view. Considering the related works: [12] with up to 3% of accuracy error and 45 times of speed up in HW/SW BFM level; [13] with up to 2% of accuracy error and 400 times speed up in native execution; and [10] with 0% of accuracy error and 60 times of speed up in CCA model, this work contributes to enhance the state of art of high level system simulation.

C. Multiprocessor Platform

For evaluation of multiprocessing platform features, in the previous uniprocessor platform (see Figure 8) were added more three processors. This addition is intended to analyze the HdSC four core multiprocessing behavior and collect data about simulation accuracy and performance. The same applications executed on the single core platform are executed on this multiprocessor environment in order to compare results.

The timing estimations shown in Table 3 demonstrate the amount of processing performed by each core. The accuracy achieved has a lower error rate (up to 10.58%) due to less operation performed in processors.

Table 3. Instruction estimation per processor.

Application	ISS	HdSC	Error
Coremark	592,382,221	594,810,988	0.41%
Dhrystone	31,196,108	31,364,567	0.54%
Hello World	1,011	1,118	10.58%
Iterator	361,251,750	377,652,579	4.54%
Timer Control	20,558,102	20,327,639	1.13%

The Hello Word application could not be partitioned due to its small size, being replicated in processors and performing the same operations. The results obtained are the same due to low overhead of this application.

In Table 4, the ISS simulator has it performance decreased due to their multiples instances sharing the same simulator. The HdSC kernel runs also in lower speeds because of necessary synchronization to avoid race conditions. This overhead implies in lower speed up levels than expected (up to 735.76x).

The current development efforts are in synchronization mechanisms to reduce simulation overhead and boost overall performance.

Table 4. Simulation performance speed up per processor.

Application	ISS	HdSC	Error
Coremark	0.09 MIPS	49.29 MIPS	547.61x
Dhrystone	0.10 MIPS	67.31 MIPS	673.14x
Hello World	0.15 MIPS	53.13 MIPS	354.20x
Iterator	0.09 MIPS	61.05 MIPS	678.33x
Timer Control	0.10 MIPS	73.57 MIPS	735.76x

7. CONCLUSIONS

In this paper, the HdSC approach was presented, aiming to improve HdS development flow. Adopting the host native execution strategy, a higher-level processor and software modeling allow a more efficient system execution in a virtual platform environment. The requirement of executing the same software source code as a native application, instead of using SLDL models, creates an innovative abstraction model. The key concepts and the modeling examples showed how this approach could truly enable a more efficient HdS development. This efficiency comes from native execution that boosts simulation performance (up to 760.86 times), preemption support for ISR design and accurate timing estimations (up to 12% error) mechanisms.

Next steps consist of applying the proposed approach to a broader variety of industrial benchmarks and design more complex platforms and applications. Additionally, ongoing research aims to improve HdSC algorithms and techniques, in order to reduce the timing estimation error and enhance simulation performance. The goal is to offer a solid alternative to traditional ISS approach, keeping its high accuracy and HdS support. Hence this work reduces some critical bottlenecks in complex HdS system design, the contribution of this proposed work is relevant to considered state of art.

ACKNOWLEDGEMENTS

We would like to acknowledge the Ministry of Science, Technology and Innovation (MCTI) for the financial support provided by the funding agencies CNPq and CAPES.

REFERENCES

[1] Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., and Barros, E., "The ArchC Architecture Description Language and Tools". Int. J. Parallel Program. 33 (October 2005), 453-484.

[2] Dömer, R., Gerstlauer, A., and Müller, W., "Introduction to Hardware-dependent Software Design for Multi- and Many-core Embedded Systems". In Proceedings of the 2009 Asia and South Pacific Design Automation Conference (Piscataway, NJ, USA, 2009), ASP-DAC '09, IEEE Press, pp. 290-292.

[3] Ecker, W., Esen, V., Schwencker, R., Steininger, T., and Velten, M., "TLM+ Modeling of Embedded HW/SW Systems". In Proceedings of the Conference on Design, Automation and Test in Europe (3001 Leuven, Belgium, Belgium, 2010), DATE '10, European Design and Automation Association, pp. 75-80.

[4] Ecker, W., Heinen, S., and Velten, M., "Using a Dataflow Abstracted Virtual Prototype for HdS-design". In Proceedings of the 2009 Asia and South Pacific Design Automation Conference (Piscataway, NJ, USA, 2009), ASP-DAC '09, IEEE Press, pp. 293-300.

[5] Grotker, T., *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[6] Herrera, F., Posadas, H., Sanchez, P., and Villar, E., "Systemic Embedded Software Generation from SystemC". In Proceedings of the conference on Design, Automation and Test in Europe - Volume 1 (Washington, DC, USA, 2003), DATE '03, IEEE Computer Society, pp. 10142-.

[7] Kraemer, S., Gao, L., Weinstock, J., Leupers, R., Ascheid, G., and Meyr, H., "Hysim: A Fast Simulation Framework for Embedded Software Development". In Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (New York, NY, USA, 2007), CODES+ISSS '07, ACM, pp. 75-80.

[8] Krause, M., Englert, D., Bringmann, O., and Rosenstiel, W., "Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation". In Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (New York, NY, USA, 2008), CODES+ISSS '08, ACM, pp. 143-148.

[9] Lisboa, E., Silva, L., Chaves, I., Lima, T., and Barros, E., "A Design Flow based on a Domain Specific Language to Concurrent Development of Device Drivers and Device Controller Simulation Models". In Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems (New York, NY, USA, 2009), SCOPES '09, ACM, pp. 53-60.

[10] Lo, C.-K., Chen, L.-C., Wu, M.-H., and Tsay, R.-S., "Cycle-Count-Accurate Processor Modeling for Fast and Accurate System-level Simulation". In Proceedings of the Conference on Design, Automation and Test in Europe (2011), DATE '11.

[11] Schirner, G., Gerstlauer, A., and Dömer, R., "Automatic Generation of Hardware dependent Software for MPSoCs from Abstract System Specifications". In Proceedings of the 2008 Asia and South Pacific Design Automation Conference (Los Alamitos, CA, USA, 2008), ASP-DAC '08, IEEE Computer Society Press, pp. 271-276.

[12] Schirner, G., Gerstlauer, A., and Dömer, R., "Fast and Accurate Processor Models for Eficient MPSoC Design". ACM Trans. Des. Autom. Electron. Syst. 15 (March 2010), 10:1-10:26.

[13] Wang, Z., and Herkersdorf, A., "Software Performance Simulation Strategies for High-level Embedded System Design". Perform. Eval. 67 (August 2010), 717-739.

[14] Yu, H., Dömer, R., and Gajski, D., "Embedded Software Generation from System Level Design Languages". In Proceedings of the 2004 Asia and South Pacific Design Automation Conference (Piscataway, NJ, USA, 2004), ASP-DAC '04, IEEE Press, pp. 463-468.