Abstract—Harmless is a hardware architecture description language targeted to the simulation of embedded and real-time software. It allows to describe the instruction set and the micro-architecture of a processor. From this description, the Harmless compiler generates an Instruction Set Simulator and a Cycle Accurate Simulator. Both simulators are useful to test and validate embedded software and the latter is essential for Real-Time software. Their use is cheaper and more comfortable than the execution on the actual hardware. Moreover, with simulation, it is easy and unobtrusive to trace the execution and to report useful informations. However, tracing mechanisms may be difficult or even impossible to integrate without ad-hoc support in the simulator and, in our case, in the description of the processor. This paper presents how Harmless is modified and used to add tracing support to simulators. This mechanism called action is used to extract high level information such as the task scheduling observation and stack safety analysis from the low level simulation. It also highlights how the Harmless description of a processor should be updated to support these features and applies it on three processors models.

I. INTRODUCTION

Simulation is a widely used technique for embedded software design. Simulation may be conducted at many levels from high level simulation of a model of the software to low level simulation of the targeted hardware to run the actual binary of the software. The former may be done at the beginning of the development cycle while the latter may be done at the end of the development cycle, just before the test on the actual hardware.

Low level simulators include Instruction Set Simulators (ISS) and Cycle Accurate Simulators (CAS). An ISS simulates the behavior of the instructions of a processor. A CAS computes the execution time of each instruction and takes into account the additional delays coming from conflicts and dependencies between instructions by using a timing accurate model of the hardware. ISS and CAS may be combined to provide a simulator that fills both usages.

Both ISS and CAS are useful and have numerous advantages compared to execution on the real hardware: the analysis is not intrusive; there is no constraints due to the hardware; the time may be stopped and the state of the simulator may be saved to replay scenarios; the environment controlled by the application (sensors and actuators) may be simulated too. An ISS is faster than a CAS but a CAS fits the needs of the simulation of real-time system softwares because such systems need an accurate model of the hardware to provide simulated computing timings as close as possible to the actual hardware timings.

However the development of a low level simulator is time consuming and debugging may be difficult. So a Hardware Architecture Description Language (HADL) may be used to simplify this task [1][2]. A HADL is a Domain Specific Language dedicated to the modeling of a computer hardware. It provides high level abstractions that help the designer to focus on the hardware described. It eases the development, speeds it up and reduces the risks of errors. From such a description, many tools may be generated like simulators, an assembler or a compiler.

Low level simulation brings in a difficult issue: if it is easy to observe a low level event like a memory read or register access, a higher level behavior of the simulated software, like task scheduling of an operating system, may be difficult to observe because the information needed is not explicitly present in the software, especially when the source code of the basic software (operating system and device drivers) is not available. Anyway, the observation of a set of low level events allows to reconstruct a high level event as we showed in [3].

The work presented in this paper is the integration of high level events observation mechanisms in HARMLESS Hardware Architecture Description Language. This integration requires few annotations related to the Application Binary Interface (ABI) of the target architecture. Modifications of the simulated software, the source code of the libraries and the operating system, if any, are not needed though.

The paper is organized as follow: section III presents rapidly the HARMLESS HADL and the actions mechanism; section IV and section V explains the way actions are used to extract the task scheduling and to analyze the stack; section VI presents how the actions mechanism is integrated in HARMLESS; section VII shows some experiments on several
processor models and section VIII concludes the paper.

II. RELATED WORKS

Real-time system simulators for system architecture analysis, such as scheduling simulators or RTOS simulators, use a high abstraction level. Scheduling simulators [4], [5], [6], [7] use a model of the scheduling algorithm. Some of them model the tasks as abstract time slots with probabilistic execution and/or arrival time. Others support powerful features like synchronization constraints for instance. RTOS simulators (for instance [8]) take into account the execution time of the system calls and offer a more accurate model than scheduling simulators. However, these simulators do not model the hardware and do not use the actual code of the tasks.

Low level simulation techniques for processors, computers and systems have been heavily studied in various research fields: hardware design, hardware-software co-design [9], computer architecture, system architecture, etc. Computer architects use simulation for performance evaluation of new architectural mechanisms, usually for mono-processor based computer; the simulators [10] are often CPU or memory hierarchy centric. Hardware or hardware-software designers focus on simulation accuracy and the ability to automatically generate the hardware (hardware synthesis) [11] [12]. HADL are often used to speed-up hardware-software co-design [13] [14] but the simulators that are generated by these tools lack support for the observation of the high level behavior of the simulated software.

To our knowledge, no other HADL combines the generation of a low level and accurate simulator being able to execute the actual code of an application with the ability to observe the high level behavior of the simulated software.

III. THE HARMLESS HADL

HARMLESS is a HADL dedicated to ISS and CAS simulator generation. A processor description contains at least a description of the instruction set and a description of the computing devices (registers, ALU, memory, . . . ). An ISS simulator is generated from these descriptions. Additionally, the generation of a CAS demands a description of the micro-architecture. This section focuses on the parts of the description related to low level events harvesting. It presents shortly how the instruction set is described and how the components are described. A more complete introduction to the language may be found in [2]. The micro-architecture view and informations about the generation of the CAS may be found in [15].

A. The Description of the Instruction Set

The description of the instruction set uses 3 views: the format view, the behavior view and the syntax view. The format view describes the binary formats of instructions, the behavior view describes the semantics of the instructions and the syntax view describes their textual representation. Each view is a set of trees where a node describes a piece of format, behavior or syntax. A node description conforms to the following syntax.

```
<kkind> <kname> <kkind_options> {
  <description>
}
```

<kkind> is the kind of the node: format, behavior or syntax. References to sub-nodes may take place in the <description>. By default, the references are sequentially aggregated. The keyword select changes the default and allows to choose among several sub-nodes according to a static condition. The example presented in listing 1 shows a part of the root format node of the ARMv5 instruction set where both sub-nodes aggregation and selection are presented.

```
format instBase
  condition
    select slice{27..25}
      case [m0] is dataProcessingOrMul
      case [m1] is LoadStoreWordOrByte
      case [b100] is LoadStoreMultiple
      case [b101] is branchInstruction
      case [b111] is coproAndSoftInterrupt
    end select
```

Listing 1. Part of the root node of the ARMv5 format description

Here, instBase is composed of the condition format node and, according to the value of bits 27 to 25, one of the format nodes appearing in the select. Using this model, in each view, an instruction is represented by a bough in a tree. Instructions sharing a common part in a view share nodes in the trunk of the tree, while specific parts are located in leaves.

A node may have one or more tags defined in the <description>. A tag is an identifier prefixed by a ‘#’. Since an instruction is a bough, each instruction gather a set of tags coming from the sub-nodes it uses. This set of tags is called the signature. In the example presented in listing 1, the branchInstruction node, used above, is described.

```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #nolink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```

Listing 2. Branch instruction format node

In this description, a branch instruction is tagged with a #branchInst and according to bit 24, the current address may be saved in the link register\(^1\) or not (0). Tags #nolink and #withLink are used to distinguish between the two types of branch instruction. So two instructions formats with two signatures are defined: #branchInst #nolink and #branchInst #withLink.

The signature is used to bind each view to the others. For instance, the behavior of the branch instruction is defined as

\(^1\)The link register is used to save the Program Counter value when branching. So the instruction bl (branch and link) saves the program counter and branch: this is used to call functions. However the instruction b (branch) does not save the program counter and may not be used to call functions.
shown in listing 3 and provides two behaviors according to the signature.

```
behavior branchInst #branchInst
    field s24 offset
    ul condPassed
    cond_inst(condPassed)

    select
        case #withLink
            do
                if condPassed then
                    SRU.write32(14, PC)
                    PC : = PC + 4
                    PC_addOffset(offset)
                end if
            end do
        case #noLink
            do
                if condPassed then
                    SRU_PC_addOffset(offset)
                end if
            end do
        end select

end behavior
```

Listing 3. Branch instruction behavior

Of course, tags are not restricted to views binding and may be used to identify an instruction for any other purpose. For instance, instructions having a behavior related to the low level event which is observed may be tagged to be retrieved during the simulation as explained in section VI.

B. Components and Memory Description

Components are used to model the building blocks of the processors like the register file, the Arithmetic and Logic Unit (ALU), the memory, and so on. They are used by the instruction behavior. For instance, in listing 3, the System Register Unit (SRU) component is accessed at lines 10, 11 and 17. Components may be compared to objects of object-oriented languages and may contain variables and methods. Listing 4 shows a part of the ARMv5 SRU with the methods used by the Branch instruction behavior.

```
component SRU {

    program counter ul32 PC

    memory GPR {
        width := 32
        address := 0..143
        stride := 4
        type := register
        ...
    }

    u32 PC_read32() {
        return (u32)(PC + 4)
    }

    void PC_write32(u32 value) {
```

Listing 4. Part of the System Register Unit description

The memory datatype is another feature exhibited in the SRU description. A memory is characterized by its type (register, RAM or ROM), its width that defined the maximum word size that can be accessed, the address range and the stride: a memory uses byte addressing but takes the stride into account to compute the index using the following formula: index = address \times stride – start_address. Access method are implicitly defined according to the width parameter. The GPR memory shown in listing 4 (line 5 to 12) defines a 144 bytes 32 bits width memory. Access methods to read and write are implicitly defined according to the width parameter. The GPR memory shown in listing 4 (line 5 to 12) defines a 144 bytes 32 bits width memory. Access methods to read and write are implicitly defined according to the width parameter.

Memory access methods calls and, more generally, components methods calls are obviously low level events that may be easily harvested during simulation.

Components methods are also used in the micro-architecture view to specify the access concurrency that is allowed.

C. The Action Mechanism

Actions are a mechanism designed to capture low level events which occur during the simulation. An action is implemented by an object (a data structure and a piece of code) associated to a specific event, for example a read or write access in a memory location or a call to a component method. During the simulation, every occurrence of the event triggers the action. An action may store the last occurrence of the event, a subset of the occurrences or all of them. For each event occurrence, an action stores the current value of the simulation clock to date the event. More information, like processor or device states, may also be stored according to the kind of action. In the current implementation of HARMLESS, actions may be associated automatically, to each memory location. Actions may be automatically associated to memory locations by other triggered actions according to the dependancies required by the high level analysis. Actions are used in various ways to extract the simulated software high level behavior from low level events. The analysis presented in this paper use three kinds of actions:

- the read access to a memory location action;
- the write access to a memory location action;
- the execution of an instruction action.

Finally, actions are easy to customize and are not tied to a specific processor or system architecture. So this mechanism is very general.

IV. TASK SCHEDULING DETECTION

The scheduling of the tasks is one of the important informations a embedded systems designer would want to obtain during the simulation of an embedded real-time system. This is a useful high-level information needed to verify the proper
sequencing of tasks and also to check that each task meets its temporal constraints.

Our algorithm requires only that the instruction set is used correctly: i.e. a function call should use the instruction that was designed for it.

It is important to notice that the task scheduling can be obtained even if the source code of the operating system is not available, as no knowledge on the internal OS behavior is required. Moreover, this algorithm does not require a stack safety hypothesis. The fact that there is no hypothesis on the validity of the stack pointer allows to perform a stack analysis (see section V).

A. The Main Algorithm

The algorithm is based on following the execution path and detecting stack modifications both at the beginning of a task and at the end of functions. At the beginning of a task, the current task is detected (and its initial stack value is saved). If two or more tasks share the same initial function, the correct task is retrieved using the stack value. When there is a function call (using the suitable assembly instruction), the current stack value is saved. This value is compared with the value of the stack when the function returns:

- if the stack value is the same before and after the function, the task under execution is still the same. When a function returns, the stack should be in the same state than it was before;
- if the stack value is not the same, a context switch has been performed. Another task is under execution.

This approach based on the comparison of the stack value at the beginning and at the end of a function does not require a safe stack. A stack overflow may occur inside the function.

B. Implementation Using Actions

Let’s consider the following example in figure 1. System calls are based on the OSEK/VDX Operating System specification. This simple example is based on 2 tasks. At the beginning, TaskA starts. Then, it calls the system service

ActivateTask, which activates task TaskB (the task becomes ready) and call the scheduler. As TaskB has a higher priority, TaskA is preempted and TaskB runs. At the end of its execution, TaskB calls the TerminateTask system service. Eventually, TaskA resumes its execution.

Task detection should be performed:

- at the beginning of the task execution, i.e. when the first instruction of the task main function is executed (marks 1 and 3);
- when there was a preemption, where the execution path of the instruction stopped (mark 5);

The first case is the simplest. An action is set at the first instruction of the task’s main function. When the task is activated, the action is triggered and reports to the task controller which is the current running task.

The second case is a little bit more complicated, as the preemption can occur at any point in the execution flow. It is not possible to insert an action that will be triggered for each instruction (such an approach would lead to a computation overhead not compatible with the application validation purpose). The detection of the current running task is done using a stack analysis during function calls. The simulator should be modified in order to inform the task controller when a function call assembly instruction is executed. The detection of the current running task is done in 2 steps:

1) when a call instruction is found, an action is inserted at the address just following this call instruction: this is the first instruction executed when the function returns. The current value of the stack pointer is saved with the id of the current task (just before the function call simulation);

2) when the function returns, the action previously set is triggered. The current value of the stack pointer is compared with the one saved just before calling the function. If the two values are identical, the running task is the one with the saved id. The action is then deleted;

In step 2, the two stacks may be different. Let’s consider another example in figure 2 with three tasks. There are 2 preemptions at marks 3 and 4, because of the activation of higher priority tasks. At marks 3 and 4, the same code is executed (system code). Thus, two actions are set at the same address, one by task TaskA and another one by task TaskB.

The software component in the simulator that records tasks switches.

The simulated software remains unchanged.
At mark 7, both actions are triggered, but in only one case the two stack values are identical. This example shows the case where an action is triggered and the actual stack value is different than the saved one. If no task is found and all the actions have been triggered, the task controller cannot find the current task and set it to unknown. This latter case happens for instance when the operating system has its own stack.

There are as many actions for each task as there are nested functions.

**C. Stack Sharing**

The stack sharing method may be used for tasks that cannot be in a ready state at the same time (i.e., they cannot compete for the processor at the same time) [16]. This is useful for deeply embedded systems where RAM usage is a critical resource. That special case is supported by our algorithm without any modification. Tasks are detected at startup with the action set in the first instruction of the task. Actions that are set dynamically related to function calls are deleted after use. As tasks execute one after the other, there is no ambiguity.

**V. Stack Analysis**

The stack analysis is of primary importance for deeply embedded systems to reduce RAM requirements, but also to ensure that there is no stack corruption. Stack analysis is done in a multi-tasking hypothesis and relies on the task detection presented in the previous section. This analysis explains the strong requirement for the task’s scheduling analysis that should not require safe stacks.

This analysis does not require any timing information and is useful even on instruction set simulators. Two kinds of analysis are interesting: the actual stack usage (provided the designer set up an execution scenario leading to the maximization of the stack usage) and the stack overflow and underflow detection. The former happens when a stack is undersized and both may happen if a bug changes a stack pointer in a wrong way.

**A. Stack Usage**

To detect the actual stack usage of a task (for a scenario), an action is set to be triggered on memory write accesses. Each action has a flag initially set to false. During a stack write access by the application, the corresponding action is triggered and the flag is set to true. The real stack usage is detected by checking for the first flag set to true in the stack zone, starting from the bottom of the stack.

This approach does not make any hypothesis on the way the compiler accesses the stack. For instance, some compilers access the stack using an offset to the stack pointer, without changing the stack pointer itself, in leaf functions (function that do not call any other function). In this case, the stack pointer value is irrelevant to determine the actual stack usage.

**B. Stack Overflow/Underflow Detection**

The stack corruption analysis uses the detection of the current running task detailed in section IV. The stack corruption detection is based on memory protection, using two memory zones (before and after the stack) that should not be modified by the task. Each byte of these zones is associated to an action that signals the stack overflow if triggered.

This memory zone should be small enough to avoid interference with other memory zone used by the task, and should be large enough to detect effectively a memory overflow, even in the case of non consecutive accesses.

This memory protection depends on the current running task, the protected zone for one task may be the stack of another task. In few system function (typically the context switch function) that handles more than one stack, the stack controller may generate a false positive. This case happens only in very few system function. Symbols related to these functions may be given to the simulator to preclude false positives.

**VI. Integration in HARMLESS**

This section explain how the task scheduling detection and the stack analysis are integrated into the HARMLESS language to provide features presented before for automatically generated simulators.

**A. Implementation of the Action Mechanism**

As described in section III-C, tasks and stacks detection are based on 3 kinds of actions: read, write and execute actions. Because read and write actions may be triggered for each memory access, the implementation has a great impact on the performance of the simulator. Three implementations have been tested. The first one consists in adding a pointer to the action for each memory byte. The lookup of an action is fast but the memory needed by the implementation is eight (on 32 bits systems) or sixteen (on 64 bits systems) times
the simulated memory. The second one uses a lookup table where the address is used to search for the corresponding action. The lookup table uses a hashing storage technique. The memory overhead is very low but the computation of the hash function leads to slow down the simulation by 97%. The third implementation is comparable to the Supertrace algorithm\cite{17} and combines a bit vector and a lookup table. A bit indicates the existence of an action at the corresponding address and the action is retrieved from the lookup table if it exists. The memory overhead remains low (25% of the simulated memory) and the simulator is slowed down by 25% only.

The execute actions are not implemented in the same way and use the decorator design pattern, i.e. when an execute action is set, the corresponding instruction is replaced by an instrumented instruction that first triggers the action and then simulate the actual instruction. This implementation implies that there is no significant performance loss using this kind of actions.

B. Integration of Task and Stack Analysis

For the task scheduling and stack analysis, in addition to actions, we need:

- the address of the first instruction of the task. We have to set an action to detect that a task is started. If symbols are embedded in the executable file (elf format for instance), the symbol of the main function of the task gives this address;
- the stack bounds. This parameter depends on the underlying hardware and should be given at runtime. However, this information may be difficult to get for the user (and may change every time the application is linked again). So, the user has just to give the stack length to the simulator. The first time the task is started, the action triggered at the beginning of the task detects the initial value of the stack pointer. This way, the simulation script does not require an update if the application under test is rebuilt.
- to know which instruction performs a call. To detect the return of a function, we need to set an action at the instruction that just follows a call. This information is related to a processor and should be given into the HARMLESS description. It does not depend on the application under test;
- the size of the memory protection before and after the stack for the stack analysis only. This information should be given at runtime.

We have listed the information that should be given to the existing simulator. It can be split in 3 categories.

First, there is a part that do not depends on both the underlying hardware and the user application. It may be integrated into the generic code of the HARMLESS generated simulator. This is the case of the code of algorithms, using classes such as taskObserver, stackController, ...

Second, there is a part that depends only on the underlying hardware. It should be integrated in the HARMLESS simulation description. It is described once and usable for each user application. This is the case of the instructions that perform a call instruction and the stack pointer declaration (see section VI-B1).

Last, some parameters depend on the user application under test. The API of the simulator should be extended in order to give them in the scenario script (see section VI-B2).

1) Updating HARMLESS language for task detection:

As explained in section III, the HARMLESS description of the instruction set is based on 3 views (syntax, format and behavior). An instruction can be identified using its signature, which is the set of tags defined in the description (see section III). A tag is a word that begins with a ‘#’.

Our model of instructions implements some introspection. The simulator can ask if an instruction signature includes a particular tag. We use tags to identify instructions that may be a call. The tag used is #SP_Check. Here is the description of the branch instruction updated for task detection (ARM instruction set model):

```
behavior branchInst #branchInst
  field s24 offset
  ul condPassed
  cond_inst (condPassed)
  select
    case #withLink #SP_Check
      do
        if condPassed then
          SRU.writeR32 (14,PC)
          SRU.PC_addOffset (offset)
        end if
      end do
    case #noLink
      do
        if condPassed then
          SRU.PC_addOffset (offset)
        end if
      end do
    end select
  end behavior
```

In this description, 2 instructions are defined: #branchInst #noLink and #branchInst #withLink #SP_Check. Only the last one (that saves the current value of the instruction pointer in the link register) can perform a call instruction. Its signature should be updated in the two other views too.

Moreover, the set of tagged instructions may be larger than the real set of call instruction, as the simulator checks that the program counter does not points to the instruction just after (there is a branch). However it should not overlap with the set of branch (only) instructions.

This approach has the advantage of being very simple to implement in current processor descriptions: only 7 and 16 lines for the ARM and the PowerPC instruction sets have been updated to add the #SP_Check tag instructions that perform a call.
2) Required simulation runtime parameters: A stack spy controller provides an API to give runtime parameters that are application specific. The simulator scenario can be written either in C language and compiled with the simulator to offer a standalone application, or in Python, where the simulator is an external library.

The main functions for the task scheduling detection is addTaskToMonitor that advises the simulator that a new task should be monitored. Arguments are the internal name of the task in the simulator, the symbol of the function (that will give the address of the first instruction of the task) and the size of its task.

Other functions are related to the stack monitoring: setSizeOfStackProtectionArea allows to set the protected zone for the stack (see section V) and setExclusionOnSystFunction is used to avoid unwanted stack corruptions warnings in system functions.

Eventually, the generated simulator offers a new set of functions to get results as exposed in the next section.

VII. EXPERIMENTAL RESULTS

This section first gives an example of a task detection on a simple example, including the scenario scripts. Then it shows some simulation performances results.

A. A Simple Example

In this example, the application that runs onto the simulator uses Trampoline RTOS[18]. Trampoline is an open source RTOS compliant with the OSEK/VDX specification and the AUTOSAR 3.1 specification.

The application uses 3 tasks (T1 to T3). T1 is started automatically. It activates sequentially T2 and T3 which are higher priority tasks. They preempt T1, execute and terminate. Then T1 resumes and restarts. Most of the code is system code and there are many task scheduling.

Here is a minimal Python script that runs one billion instructions and detect the task scheduling of this application. The simulator used for this test is an AVR AT90CAN128 (8 bits RISC micro-controller from ATML).

```python
from AT90CAN128 import arch
simu = arch()
simu.readCodeFile("./TrampolineBasicTest.elf")
stackCtrl = simu.getStackSpyController()

stackCtrl.setSizeOfStackProtectionArea(16);

stackCtrl.addTaskToMonitor("T2",
    "secondTask_function",256);
stackCtrl.addTaskToMonitor("T3",
    "thirdTask_function",256);

simu.execInst(1000000000) #run 1 billion insts
# print tasks information
stackCtrl.printTaskList();

# print results (tasks switching dates)
stackCtrl.printControllerSwitchList()
stackCtrl.writeTraceT3("trace.txt");
```

The function printTaskList gives information about the task:

```text
Task 3 (T3) :
   Fct@124 Stack (size= 256)
   StackRealUse=22
```

It gives the address of the entry point of the task (0x0124), the detected initial value of the stack (detected the first time the task is started) (0x5ac) and the real stack usage on that scenario (22 bytes). This last information is the result of the stack usage detection.

Functions printControllerSwitchList and writeTraceT3 are used to retrieve the tasks switching dates. It is available either in a text form (first function), or using T3 file format (second function). T3 is a Java tool to draw Gantt diagrams. An example of a trace generated using this scenario is on figure 4. At the beginning of the simulation, the controller does not know which is the current running task and uses the unknown task (id=3).

If the stack size specified is not large enough, an error is generated at runtime:

```text
ERROR in task #3 (T3) @17ea STACK OVERFLOW
```

The displayed address is the value of the program counter when the stack corruption occurred.

B. Generated Simulators Performances

We show in this section the impact of the analysis tools on the simulation performances. The simulators use the first implementation of the read and write actions mechanism explained in VI-A. Five examples (called ex1 to ex5) are used for the benchmark. They are all based on the Trampoline RTOS and highlight characteristics of the task and stack analysis: basic test, test with unknown tasks, tasks with the same entry point, tasks that are preempted using event mechanism (OSEK synchronization service) and tasks that corrupt the stack. All the examples are periodic and most of the code executed is system code.

Simulations are made on an Intel Core2Duo@2.4Ghz computer (the simulator is single threaded and uses only one core), for 1 billion instructions. This simulation is long enough to generate many task switches (more than 1.2 million in the first
example for the AVR target). Four versions of the simulator are provided:

- the first is the original simulator, with no support for read/write actions (named no R/W action in figures);
- the second has support for actions, and implements the task detection analysis (named with task monitor in figures). This detection uses only execute actions;
- the third adds support for read/write actions, but without any action based analysis (named actions only in figures). No action is triggered during simulation. This configuration gives the performances hit of actions support;
- the last simulator adds the stack corruption detection in addition to the second simulator (named with stack monitor in figures). It is based on R/W actions.

We updated 3 processor descriptions to support the task scheduling analysis: The Atmel AVR (8 bits RISC architecture, designed for deeply embedded systems) (figure 5), the ARM (figure 6) and the powerPC (figure 7) instruction sets. Even if the analysis presented here requires a cycle-accurate model of the processor, the accuracy of the models is not discussed in this paper. We used a simple model with no pipeline for each model, but more complicated models can be modeled in Harmless (see [15]).

Simulators execution times for these examples are slightly the same, instruction throughput is about 26 MIPS with the AVR, 20 MIPS for the ARM and 27 MIPS for the PowerPC simulator with all optimizations turned on.

The task detection algorithm, based on the execute actions, slows down the simulation by nearly 30%. The detection occurs only when there is a call instruction, which is rarely executed. For instance, there are 9.25 call instructions every 1000 instructions in the first example with the AVR.

However, as soon as R/W actions are added on the simulator the instruction throughput is reduced between 24% (ARM) and 46% (AVR). Even if the computation overhead is limited to a simple pointer comparison, this test is done for each memory byte access: a word access (4 bytes) triggers 4 actions.

The stack monitoring does not impact significantly performances as soon as both task detections and R/W actions are available. Adding the 2 detection algorithms with the actions support only divides performances by nearly 2.

**VIII. CONCLUSION**

This paper has presented the integration of two tracing methods into the HARMLESS Hardware Architecture De-
scription Language. These tracing methods allow to get high-level information from the low-level simulated behavior. They are all based on a mechanism called action that can extract low-level information from the simulation. We focused on the task detection and the stack analysis which are very important during the validation of real-time (deeply) embedded systems.

Generated simulators can now benefit from these improvements at very limited cost: only 7 lines (out of more than 5000) have been updated for the ARM description to support task detection and stack analysis. A script interface is embedded in the simulators and offers a mean to automate the execution and to configure the high level observers.

Simulation performances are good. 3 simulators of RISC processors (8 bits AVR, 32-bits ARM and PowerPC) have been used. They all show that the computation overhead is approximately a factor $2$ and is limited to 30% if only task detection is performed. These results show that such mechanisms are well suited for real time systems validation.

Presently, we focus on the optimizations of actions to reduce the computation overhead. We also work on the extension of Harmless to model multi-core processors, and the adaptation of actions to a model of the memory hierarchy. Harmless is free-software and is available at http://harmless.rts-software.org.

**References**


