Waveperf: A Benchmark Generator for Performance Evaluation

Joffrey Kriegel, Florian Broekaert
Thales Communications and Security
Paris, France
{joffrey.kriegel, florian.broekaert}@thalesgroup.com

Alain Pegatoquet, Michel Auguin
University of Nice Sophia-Antipolis
Sophia-Antipolis, France
{alain.pegatoquet, michel.auguin}@unice.fr

ABSTRACT
Multi-core processors are more and more present in the embedded and real-time world. This paper introduces a code generator software applied to the benchmarking of embedded platforms. This solution creates an application runnable on embedded multicore platform and compliant with POSIX or Xenomai interface. The execution outputs a trace of the execution of each thread of the benchmark. It is also used to verify the latency of interruption and the preemption for real-time platforms.

Categories and Subject Descriptors
D.4.8 [Performance]: Measurements

General Terms
Evaluation, Performance, Multi-core

Keywords
Multi-core, performance evaluation, code generator, Benchmark

1. INTRODUCTION

With the increasing number of embedded platforms on the market, the need of a standard operating system has increased. Linux distributions are more and more present in embedded systems, so it becomes important to test both the computing and real-time performances. Benchmarks [1] are commonly used to test the performance of a new platform and check if the platform is powerful enough for the target software to be run. But current benchmarks [2] [3] give a performance index and do not test complex and real-time multi-threaded applications. For general parallel systems, there exists a number of benchmark suites, e.g., SPLASH-2 [7] and PARSEC [5]. However, to the best of our knowledge few open source benchmark suite exists that specifically targets parallel embedded systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EWiLi’12 Lorient, FRANCE

Recent works [4] implements parallelism in standard open-source benchmarks to allow the performance estimation of multi-core platforms. In the same time, commercial benchmark [1] are also implementing multi-core benchmark. Multi-bench assess the relative performance of multi-core platforms. [6] proposes a framework for writing parallel and real-time benchmark in JAVA language. But the test of new platforms is rarely done with a Java Virtual Machine on it. More often, only Linux with RT patch, or Xenomai are used for the first tests.

The goal of this paper is to introduce a benchmark generator for evaluate the performance (in term of computing but also in term of interrupts) of a multicores platform using Linux and/or Xenomai. Another goal is to build quickly specific benchmark similar to the software the user wants to implement.

The section 2 will present the generator, the different components and examples of which standard functions are used. The section 3 speaks about results using this methodology.

2. THE BENCHMARK GENERATOR

This methodology is a benchmarking software generator tool based on an application model. The user has to create a model of its application. The tool measures the tasks execution duration and is able to monitor the execution scheduling. Thanks to this, the user can validate a software model and verify the performance of this model on the targeted hardware platform. So, the real-time constraints are analysed ensuring that the tasks respect their execution time. Different software architecture models can be evaluated to explore the hardware architecture performance. The generated code is POSIX compliant, it is hence possible to execute it on every hardware platform using this norm. Of course, the generated code can be used on multi-core platform to execute tasks in parallel. The CPU-affinity can be either static or dynamic.

2.1 Description

An executable C++ code can be generated from a specification using configuration text files. A configuration file is required for each software block and also for the architecture top level description. Configurations files are split into three distinct parts:

- Component: describes the external view of the block through input and output signals definition. As an example, Listing 1 shows a definition for the mac component of a radio benchmark application. Provides
and uses keywords respectively define input and output signals for that component. The example create 4 input and 3 output for this component. (The following example will describe all the parts of this component)

```
component mac
{
    provides Runnable upper_sap_1;
    provides Runnable upper_sap_2;
    uses Runnable lower_sap_0;
};
```

Listing 1: An example illustrating the Component definition for the radio benchmark.

- Behavior: defines the behavior of the block when an input signal is received. For that purpose, a state number (if a state machine is defined), the output signal and its corresponding number of activation must be specified. As an example, Listing 2 describes the behavior definition (mac_behavior) of the mac component previously defined. This definition indicates that each time the upper_sap_1 signal is received, no output signal will be generated, but each time upper_sap_2 signal is received, lower_sap_0 signal is generated once. For this block behavioral description, no state machine is required. This is expressed by the (1) statement which means that only one state is possible. The following parameters (equal to 1 for both output signals) indicate the number of times output signals will be generated. This feature allows defining multi-rate system. A second example is shown with a more complex behavior. Listing 3 describes a possible behavior of the phy_rx component. In this example, a variable is instanciated and then initialized (counter). A state machine is also created with only one state (SA).

```
behaviour mac_behavior of mac
{
    upper_sap_1.run
    {
        (1) 0 { none . none }
    }
    upper_sap_2.run
    {
        (1) 1 { lower_sap_0 . run }
    }
};
```

Listing 2: An example illustrating part of the Behaviour definition for the mac component.

```
behaviour phy_rx_behavior of phy_rx
{
    _var_ { int counter ; }
    _initial_ { counter = 0 ; }
    _state_ { SA ; }
    _initial_state_ { SA ; }
    tick_samples_in.run {
        (1) SA - [ ( this . counter < 10 ) ] >
        SA 0 { none . none } ! { this . counter ++ ; }
        (2) SA - [ ( this . counter == 10 ) ] >
        SA 1 { frame_out . run } !
            { this . counter = 0 ; }
    }
};
```

Listing 3: An example illustrating part of the Behaviour definition for the phy_rx component.

- Characteristics: defines the CPU processing time or the number of operations to execute when a input signal is received. For instance, Listing 4 depicts the processing time characteristics that corresponds to the behavior of the component. This is achieved by indicating “Timing_in_ms” after the characteristics keyword. This listing shows the way timing can be defined onto input signal reception and before output signal activation. The (1) statement means that only one state is possible. Then, for upper_sap_1, 0.2 indicates the execution time in ms required. For the signal upper_sap_2, 0.2 ms are required too, but after sending the output signal (as seen before), the component has to compute again during 0.04 ms, it indicates the processing time after output signals activation.

```
characteristics ( Timing_in_ms ) mac_chars_of_mac_behavior
{
    upper_sap_1.run
    {
        (1) 0 . 2
    }
    upper_sap_2.run
    {
        (1) 0 . 2 0 . 04
    }
};
```

Listing 4: An example illustrating the Characteristics definition of the mac component.

Architecture files define the way blocks are connected. After having included blocks configuration files, blocks must be instantiated in order to declare a behavior and characteristics for each block. Then, connections between blocks must be specified through input/output signals. The Listing 5 depicts the top level architecture file for the H.264 application. As shown, all required blocks are instanciated using the “component_instance” keyword. An example, “main” is an instance of the “main behaviour” with reference to its CPU processing time “main_timing_chars” previously defined. In the architecture file, it is also possible to implement timers for the application. Listing 6 shows an implementation of a timer beginning 500.000 ns after receiving a “start” command with a period of 500.000 ns. Each timer have an output named “tick” which can be connected to any other component. Timers are very useful component when designing real time application.

To add the possibility to test non-detereminstic behaviour (Interruptions not timed), the ethernet port can be used
include rlc_manager.txt;
include mac.txt;
include phy_tx.txt;

component_instance rlc_manager_behaviour
   rlc_manager rlc_manager_characs;
component_instance mac_behaviour mac
   mac_characs;
component_instance phy_tx_behaviour phy_tx
   phy_tx_characs;

Listing 5: An example illustrating the system software architecture definition for a H.264 application.

component_instance Timer_impl phy_timer timer
configuration phy_timer->
   configure_timerspec_and_sched_fifo( 0, 500000, 0, 500000, true, 10 );
connection( synchronous ) phy_timer_to_phy_tx
   phy_timer.tick phy_tx.tick;

Listing 6: An example illustrating the instantiation of a timer.

to activate a thread. This feature sense the ethernet connection and wakes up when something is coming from the LAN (for example a ping to the IP address of the platform).
Listing 7 creates an ethernet component, an ethernet sensor "Raw_ip_interface" and a connexion between them. Moreover, the end of the benchmark outputs the number of byte received.

include eth.txt;
component_instance ethernet_behaviour eth_inst ethernet_timing_characs;
component_instance Raw_ip_interface eth_device ip_interface;
configuration eth_device->
   configure_priority_and_sched_fifo( 5, true );
connection( synchronous ) timer_connection_1
   eth_device.data_out eth_inst.rx_from_io;

Listing 7: An example illustrating the instantiation of an IP interface.

Another feature that can be described in this architecture file is the connection (dependency) between threads. These connections can be either synchronous or asynchronous. A synchronous connection is blocking for a thread (A in our example) that starts the execution of another thread (e.g. thread B). As a consequence both threads are executed on the same CPU. The behavior of a synchronous connection is therefore similar to a function call. A thread inherits the priority from its father thread. In another hand, an asynchronous connection allows the parallel execution of threads. Figure 2 illustrates this parallel execution. As it can be seen, a FIFO is used for that between two threads (A and B in our example). The thread A will first copy data for thread B into the FIFO and then continue its own execution. The thread B can then take the data and process them in parallel. When an asynchronous connection is set, it is possible to configure the new threads with priority.

In order to get estimation for multi-core based platforms, a new parameter has been added. This parameter ("configure_affinity") allows the designer (or a future automated tool) to choose the CPU where the thread will be executed. So far, the CPU allocation is performed statically, i.e. a thread is assign to a CPU and cannot migrate. The C++ code generated from this specification can be executed on any platform respecting the POSIX standard, in order to verify for instance the right scheduling of tasks or that real-time constraints are respected. For that the application model has to be annotated with dynamic information such as the number of instructions to execute, the number of memory load/store and the size of memory print. Next section describes how the dynamic information is collected from an existing application.

2.2 What is generated?

As seen previously, waveperf is able to generate standard code using Posix or native Xenomai. C++ objects are created for each component in the system. The generated component are the same for Posix or Xenomai standard. The main difference is in the thread and timer creation. A library is created for Posix or Xenomai use. In these libraries can be found the implementation of Characteristics parts, interaction between components, and timers.

First, let’s see the Xenomai native implementation for the different parts:

- For asynchronous connexion, at startup of the benchmark, the generator use “rt_task_create()” then “rt_task_start()” and finally create a semaphore to pause
the thread with "rt_sem_create()". When the connection is activated by another thread, the semaphore is sent "rt_sem_v()" and the thread is unblocked.

- For timer instantiation, the "rt_task_set_periodic()" function is used to create a timer with X-nanoseconds of period.

- For "execution time" Characteristics, the used method is to make a huge amount of loops during the initialization of the benchmark, and to check the time it takes for each loop. Then when a call is done for an "execution time", the right number of loop is performed. The calibration is done with "rt_timer_read()" to get the time before and after the huge number of loops.

In the same way, the POSIX implementation is done like described below:

- The asynchronous connections are created with "pthread_create()" and "pthread_attr_setschedparam()" for the priority. A "sem_post()" is used to unlock it when needed.

- "timer_settime()" and "timer_create()" are used to set the period and create a timer.

- To get the time of the loops, the generator use "gettimeofday()".

To conclude, to implement the generator for a new OS (VxWorks, RTAI, ...) or standard, a few number of function are needed (about 6).

3. RESULTS

3.1 Interruption analysis

One of the main goal of the benchmark generator is to test the performance of the real-time interruption and pre-emption on a platform. The framework is able to generate an application for either:

- Linux Posix standard
- Linux Posix with Xenomai
- Xenomai native driver

An application model has been created to test the real-time interruption on an embedded Linux. This application is a model of a radio communication. Three tasks are implemented with different priorities. The higher priority task is the simulation of a PHY stack, with a timer at 2000Hz. The second task simulate the MAC layer and have a timer at 100Hz. And the lower priority task is the input buffer from the ethernet stack. To ensure the good behavior of the future application, the PHY thread must not be interrupted by other thread and must be regular. This example has been done on a monochrome platform to be sure that all threads are executed on the same processor.

So it has first be generated with Posix interface and be used on an embedded Linux 2.6.26. The kernel is configured with high resolution timers.

Fig. 4 shows a problem with the first task. It is not regular because of the interruption of the other threads. The first assumption is to say that the standard Linux is not good enough for our real-time needs. But using the POSIX interface with the xenomai patched kernel also shows this kind of problem.

The xenomai native application has then been generated to verify if the native interface are better for our needs.

Fig. 5 shows a better behaviour of the generated application. On this figure, the first thread has a totally regular behaviour and is not perturbed by the execution of another thread.

The framework allows the user to model different application testcases very easily and to generate the associated code for a set of standard interface (Linux Posix, Xenomai Posix, Xenomai Native). The generated application are able to detect interruption issues in the embedded platform due to the OS or the interface used.

3.2 Performance analysis

The second purpose of the generator is to evaluate the performance of a CPU. Each component can execute one of the three different "characteristics":

- A number of Dhystone instructions: The processor executes an amount of Dhystone instructions
- An active wait: The processor executes instructions

Figure 3: Radio benchmark simplified description.

Figure 4: Posix implementation.

Figure 5: Xenomai native implementation.
during an amount of time

- A passive wait: The processor sleeps during an amount of time.

The number of instructions is mainly used for the performance estimation of the platform. The active wait is used if the real execution time of a component is known or if only the interruption are tested. And the passive wait (sleep) is used to model, for example, the latency between sending data on radio interface and waiting the input data.

Each benchmark gives as an output its execution trace showing the CPU load of each processor and the activation of each block. Performance can then be measured and problems such as a scheduling issues or CPU overload can be identified.

![Figure 6: Output of H.264 decoder model on a dual-core.](image)

Table 1: Comparison between Auto-generated benchmark and real H.264 decoder application.

<table>
<thead>
<tr>
<th>Platform name</th>
<th>Real Application</th>
<th>Benchmark Model Application</th>
<th>error ( % )</th>
</tr>
</thead>
<tbody>
<tr>
<td>With filter OMAP3 @ 600MHz</td>
<td>8.9 FPS</td>
<td>9.73 FPS</td>
<td>9.7</td>
</tr>
<tr>
<td>Without filter OMAP3 @ 600MHz</td>
<td>19.3 FPS</td>
<td>21.2 FPS</td>
<td>9.8</td>
</tr>
<tr>
<td>Man's month to develop application</td>
<td>6</td>
<td>0.05 (1 day)</td>
<td></td>
</tr>
</tbody>
</table>

Some different kind of benchmarks have been modeled thanks to the generator. For example, a H.264 video decoder (Tab.1), a Software Define Radio Phy layer and a GSM sensing application. The generator is able to create benchmarks for many OSes like Linux, LynxOS or Xenomai and all the platforms which can run these OSes. For example, some generated benchmarks have run on a ARM Cortex-A8 (monocore), ARM Cortex-A9, Intel x86 and Freescale QorIQ (multicore).

4. CONCLUSION

The presented generator allows evaluating the performance of a new platform and compare it to another platform even if the real-software is not available yet. It can also model a new software architecture that needs to be tested on a platform.

It allows trying and testing different task priorities and the capability of a platform to run the application (thanks to the number of Dhrystone instruction characteristic). An important feature is the possibility to easily generate an application model with different implementations (Standard Posix, Xenomai Posix, Native Xenomai). Another advantage of this work is the possibility to add new computing models in the generated models (for example add some cache miss).

Future works can be done on generation for other kind of OS (VxWorks for example) and adding new execution characteristics for the components.

5. REFERENCES