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 both POSIX or Xenomai interface. Running the application outputs an execution trace for each thread of the benchmark. It is also used to check the interruption latency and the preemption of 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 commercial embedded platforms, the need for a standard operating system has emerged. Linux distributions are more and more used in embedded systems. It becomes therefore important to test both the computing-power and real-time behaviour of this operating system. Benchmarks [1] are commonly used for that purpose. They allow testing the performance of a new platform as well as checking if a platform is powerful enough for the target software to be run. However, current benchmarks [2] [3] rather give a performance index than really testing complex real-time multi-threaded applications. For general parallel systems, a number of benchmark suites are available such as SPLASH-2 [7] and PARSEC [5]. However, to the best of our knowledge there is few open source benchmark suite that specifically targets parallel embedded systems.

Some recent works [4] have implemented parallelism in standard open-source benchmarks, so that the performance of multi-core platforms can be estimated. In the meantime, commercial benchmarks [1] have also implemented multi-core benchmark. Multibench assess the relative performance of multi-core platforms while [6] proposes a framework for writing parallel and real-time benchmarks in JAVA language. But testing new platforms is rarely done with a Java Virtual Machine. More often, only Linux with RT patch, or Xenomai are used for the first tests.

Another approach to evaluate real-time scheduling is to use simulators such as Cheddar [9] or Storm [10]. But in that case the main objective is to test the real platform.

The objective of this paper is to introduce a benchmark generator for evaluating the performance (in term of computing but also in term of interrupts latencies) of a multi-core platform using Linux and/or Xenomai. We also aim to rapidly build specific benchmarks similar to the target software to implement.

The section 2 presents the generator, the different components as well as examples of standard used functions. The section 3 presents results obtained using this methodology.

2. THE BENCHMARK GENERATOR

This methodology consists in a benchmarking software generator tool based on an application model created by the end user. The tool measures the tasks execution duration and is able to monitor the execution scheduling. Thanks to this, the user can then validate a software model and verify the performance of this model on the targeted hardware platform. So, the real-time constraints are analyzed ensuring that the tasks respect their execution time. Different software architecture models can be evaluated to explore the hardware architecture performance. As the generated code is POSIX compliant, it is possible to execute it on hardware platforms using this norm. Of course, the generated code can be used on multi-core platforms 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 a 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. In this example two inputs and one output are created for this component. In the following, all the parts of this component are described.

```
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 a 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 of the mac component previously defined (mac_behaviour). 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, a 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 given for a more complex

```
behaviour mac_behaviour 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.

behavior. Listing 3 describes a possible behavior of the phy_rx component. In this example, a variable is instantiated and then initialized (counter). A state machine is also created with only one state (SA).

```
behaviour phy_rx_behaviour of phy_rx {
   _var_ { int counter; };
   _init_ { 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 an 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 how timing can be defined on 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 required execution time expressed in millisecond (ms). For the signal upper_sap_2, 0.2ms are also required, but after sending the output signal (as seen before), the component has to compute again during 0.04ms. So, it indicates the processing time required after output signals activation.

```
characteristics( Timing_in_ms ) mac_characs
    of mac_behaviour {
  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 instantiated using the "component_instance" keyword. As an example, "main" is an instance of the "main_behaviour" with reference to its CPU processing time "main_timing_characs" previously defined.

```
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.

In the architecture file, it is also possible to implement timers for the application. Listing 6 shows an implementation of a timer having a 500.000ns period and beginning 500.000ns after receiving a "start" command. Each timer has an output named "tick" which can be connected to any other component. Timers are widely used for designing real time applications.

```
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.

In order to test non-deterministic behaviors (Interruptions not timed), an *ethernet* port can be defined to activate a thread. This thread listens to the ethernet connection and wakes up when something from the LAN is coming (for example a ping to the platform IP address). Listing 7 depicts how to an *ethernet* component, an ethernet sensor "Raw_ip_interface" and a connection between them. Note that the number of bytes received is outputted at the end of the benchmark. A timer can therefore be used to create random behavior, since a thread can be activated at any time.

```
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 must be used 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 estimations 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 processing unit (CPU) where the thread will be executed. So far, the CPU allocation is performed statically, i.e. a thread is assigned to a CPU and cannot migrate. If the CPU affinity is not set, the thread can migrate (thanks to the OS scheduler) on any CPU. However, the CPU activity will be unknown in the execution trace. 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. The next section introduces the generated code and useful functions.

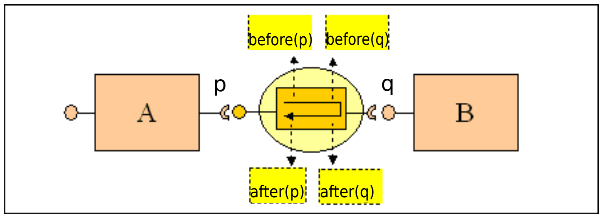

Figure 1: Synchronous connexion between thread A and B.

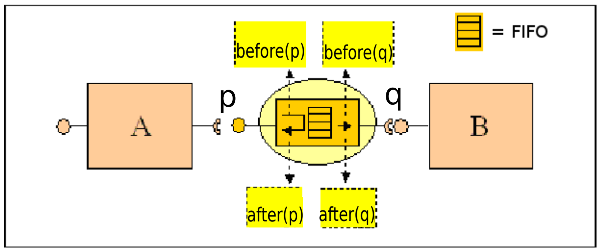

Figure 2: Asynchronous connexion between thread A and B.

2.2 What is generated ?

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

- For asynchronous connection, at startup of the benchmark, the generator uses "rt_task_create()" then "rt_task_start()" and finally creates a semaphore to suspend the thread with "rt_sem_create()". When the connection is activated by another thread, the semaphore sends "rt_sem_v()" and the thread is unblocked.

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

- For "execution time" Characteristics, the method consists in making a huge amount of loops during the benchmark initialization, and then to check the time required for each loop. Thus, 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 number of loops duration.

Similarly, the POSIX implementation is done as described below:

- The asynchronous connections are created with “pthread_create()” and “pthread_attr_setschedparam()” for setting the priority. A “sem_post()” is used to unlock the thread when needed.
- “timer_settime()” and “timer_create()” are respectively used to set the period and create a timer.
- The benchmark generator uses “gettimeofday()” in order to get the number of loops duration.

As a conclusion, only a few number of functions are needed (about 6) to implement the generator for a new OS (VxWorks, RTAI, ...) or standard (such as POSIX).

3. RESULTS

3.1 Interruption analysis

One of the main objectives of the benchmark generator is to test platform real-time performance in case of interruptions or preemptions. The framework is able to generate an application for the following configurations:

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

In order to test interruptions impacts on (real-time) embedded Linux, an application model of a radio communication has been created. Three tasks are implemented with different priorities. The task having the highest priority corresponds to the simulation of the physical layer (PHY). A timer has been implemented to simulate an activation of the PHY task at a 2000Hz frequency. Then, a second task simulates the medium access layer (MAC) with a 100Hz frequency. The third task, having the lowest priority, is the input buffer from the Ethernet stack (Fig.3). To ensure a correct behavior of the future application, the PHY thread must not be interrupted by another thread and must be perfectly periodic (regular).

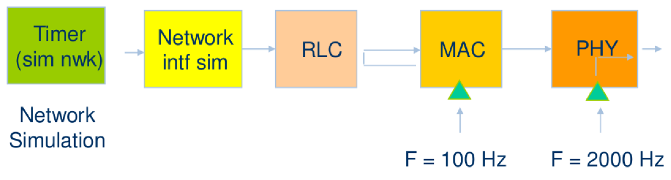

Figure 3: Radio benchmark simplified description.

This example of application model has been done for a mono-core platform to make sure that all threads are executed on the same processor.

As a consequence, this benchmark has been first generated with a Posix interface and used on an embedded Linux 2.6.26. The kernel is configured with high resolution timers.

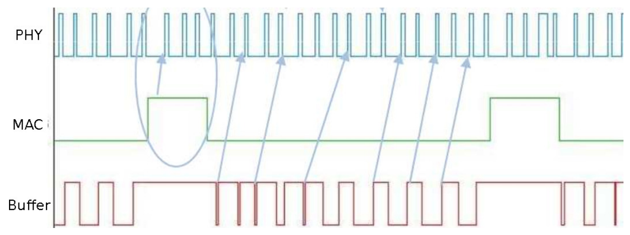

Figure 4: Posix implementation.

The Fig. 4 depicts an execution trace of the radio benchmark tasks. The PHY task (on top of the Fig. 4) has the highest priority level and should tick at a 2000Hz frequency. However, it can be observed that the PHY task execution is not regular. This is due to the other threads execution. A first analysis of that problem could be that the standard Linux is not good enough for our real-time needs. Actually, this kind of problem can also be observed using a POSIX interface with the xenomai patched kernel. Therefore, a xenomai native application has then been generated to check if native interfaces have a better behavior.

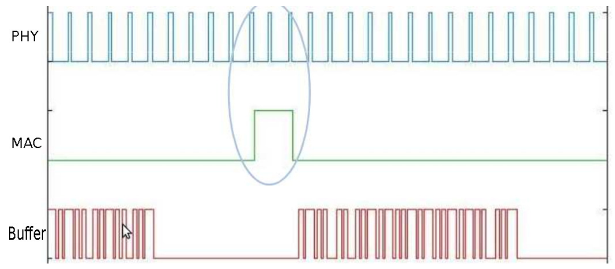

Figure 5: Xenomai native implementation.

Fig. 5 represents the behavior of the generated application using a xenomai native implementation. As it can be seen, right now the PHY thread has a fully regular behavior as it is not perturbed by the execution of other threads. Neither the MAC layer timer nor the inputs from the ethernet port modify the behavior.

This framework allows the user to model different application test cases very easily and to generate the related code for a set of standard interface (Linux Posix, Xenomai Posix, Xenomai Native). The generated application is able to detect interruption issues (either due to the OS or the interface) in the embedded platform.

3.2 Performance analysis

Another objective of our benchmark generator is to evaluate the performance of a CPU. Each component can execute one of the three different “characteristics” :

- A number of Dhrystone instructions: the processor executes an amount of Dhrystone instructions.
- An active wait: the processor executes instructions during an amount of time.
- A passive wait: The processor sleeps during an amount of time.

The number of instructions is mainly used for estimating the performance of a platform. It can also be used to determine if a processor is able to respect the execution constraints of an application. In [8] for instance, authors are able to determine the number of instructions of an application without any profiling. Authors show that they can extract the number of instructions of a software radio application as well as its complexity and its data rate. The active wait is used if the actual execution time of a component is known a priori, or if only the interruptions are tested. The passive wait (sleep) is used to model, for example, the latency between the data transmission on the radio interface and the reception of input data.

As output, each benchmark provides an execution trace exhibiting the CPU load for each processor as well as each block activity (Fig.6). Performance can thus be measured and problems, such as a real-time constraints violations or CPU overload, can be easily identified.

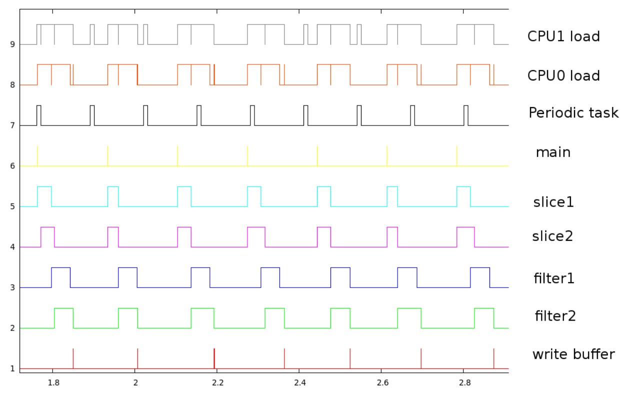

Figure 6: Output of H.264 decoder model on a dual-core.

Platform name	Real Application	Benchmark Model Application	error (%)
With filter OMAP3 @ 600MHz	8.9 FPS	9.73 FPS	9.7
Without filter OMAP3 @ 600MHz	19.3 FPS	21.2 FPS	9.8
Man's month to develop application	6	0.05 (1 day)	

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

Different kinds of benchmark have been modeled using our generator. As an example, a H.264 video decoder (Table1), a Software Define Radio Physical layer and a GSM sensing application. The generator is able to generate benchmarks for different operating systems such as Linux, LynxOS or Xenomai, and for all the platforms supporting these OSes. For example, we have generated benchmarks that can be executed on an ARM Cortex-A8 (monocore), ARM Cortex-A9, Intel x86 and Freescale QorIQ (multi-core).

4. CONCLUSION

This paper have presented a benchmark generator that can rapidly evaluate the performance of a new platform. Moreover, this generator can be used to compare different platforms performance (even if the real software is not available yet), or to determine if a platform is able to respect the application performance requirements (thanks to the number of Dhrystone instruction characteristic). It can also model a new software architecture that needs to be tested on a platform. For example, different settings of task priorities can be tested as well as new computing models in the generated threads (to add for instance some cache miss). Finally, an interesting feature is that our tool can easily generate an application model with different operating system implementations (standard Posix, Native Xenomai).

Future works will be focused on generating benchmark for more types of OS (VxWorks for example) as well as adding new execution characteristics for the components.

5. REFERENCES

[1] J.A. Poovey, T.M. Conte, M. Levy, S. Gal-On - "A Benchmark Characterization of the EEMBC Benchmark Suite" *IEEE Micro*, Volume : 29, Issue:5, Sept.-Oct. 2009

[2] S. Cho and Y. Kim - "Linux BYTEmark Benchmarks: A Performance Comparison of Embedded Mobile Processors", *IEEE The 9th International Conference on Advanced Communication Technology*, Feb. 2007

[3] M. R. Guthaus, J. S. Pingenberg, D. Emst, T. M. Austin, T. Mudge, R. B. Brown - MiBench: A free, commercially representative embedded benchmark suite, *WWC-4. IEEE International Workshop on Workload Characterization*, 2001

[4] S.M.Z. Iqbal, Y. Liang, and H. Grahn - "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems", *IEEE Computer Architecture Letters, VOL. 9, NO. 2*, July-December 2010

[5] C. Bienia, S. Kumar, J.P. Singh, and K. Li - "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 72-81, 2008.

[6] V. Olaru, A. Hangan, Gh. Sebestyen RTSJMcBench - "a Framework for Writing Parallel Benchmarks for Real-Time Java on Multi-Core Architectures", *IEEE International Conference on Automation Quality and Testing Robotics (AQTR)*, 2010

[7] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta - "The SPLASH-2 programs: characterization and methodological considerations", *Proc. of the 22nd Int'l Symp. on Computer Architecture*, pp. 24-36, 1995.

[8] J. Neel, P. Robert, J.H. Reed - "a Formal Methodology for Estimating the Feasible Processor Solution Space for a Software Radio", *Proceeding of the SDR 05 Technical Conference and Product Exposition*, 2005

[9] K. Pradheep Kumar,A.P. Shanthi - "Multicore Real Time Scheduling Using Fuzzified Priority and Non-uniform Laxity", *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2010

[10] R. Urunuela, A. Deplanche, Y. Trinquet - "STORM a simulation tool for real-time multiprocessor scheduling evaluation ", *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2010