Measurement, Analysis and Modeling of RTOS System Calls Timing

Carlo Brandolese and William Fornaciari
Politecnico di Milano - DEI
Piazza Leonardo da Vinci, 32, Milano, Italy
carlo.brandolese@polimi.it, william.fornaciari@polimi.it

Abstract

This paper presents a methodology for accurately characterizing the system calls of an operating system for embedded applications. Characterization consists of two phases: measurements and modeling. Measurements allow a coarse-grained quantitative comparison of different operating systems. Models, on the other hand, have been derived to gain a more detailed view of the behavior of a RTOS. Furthermore, they have been used within a source-level execution time estimation framework and their accuracy and usefulness proved through benchmarking. The measurement framework is based on two prototyping boards based on Xilinx and Altera devices and the timing characterization has been performed on two real-time operating systems: VxWorks and RTEMS.

1. Introduction

Nowadays, the increasing system size, despite the need of achieving significant optimizations, requires a relevant presence of software because of its flexibility and easy manufacturability, with a level of complexity commonly requiring the management support of a Real-Time Operating System (RTOS). As far the software is concerned, recent literature reports many proposals focusing on application of software performance/power estimation and optimization techniques, but only relatively few include also the analysis of the operating system (OS) in a sufficiently systematic manner to be actually integrated in a tool-chain. Among the possible approaches to characterize the operating system, direct simulation of its code using available instruction-level simulators (ISS) can be considered in theory a correct solution but it is practically unfeasible. Large scale, fine grain simulation of operating system code, in fact, is affected by several problems: first of all, performance is crucial since even simple synchronization primitives require the execution of a huge amount of code; secondly, parametric estimators should be accurate enough to enable subsequent optimization; finally, most OS companies are hardly willing to disclose source code, which would be an important aid for operating system characterization.

One of the first approaches towards including the OS within the analysis loop, is SimOS, which extended the idea of ISS with those services of the peripherals necessary to simulate the behavior of an entire operating system. A step ahead has been done by SoftWatt [1], EMSIM [2] and SimBed [4] which introduced information for the simulation related both to the instructions and the connected peripherals in order to enable full-system simulations. However, the problem of long run times when using full-featured configurations is still present, as well as the loss of per-cycle information. Modeling of timing/power at the operating system level has been carried out in [3] where, starting from a call-tree of some applications, a proper clustering of power and execution time is derived to provide guidelines for application optimization. Other operating system characterizations have been presented in [6], where data collected from simulation are grouped to extract statistical models useful for performance prediction; in [5] this type of data are used at run-time to manage and optimize the power consumption. Other valuable proposals affording the characterization of the operating systems by using complex dedicated hardware or measurement boards, or very time consuming procedures are presented in [8, 9, 10].

Such literature results confirm that execution of operating systems code, even if in some cases is only in background, is a relevant part of the overall timing/power budget of general purpose systems, and can even be more significant for embedded systems where only the essential hardware is present in the architecture. For this reasons, solutions working at a coarse grain can be valuable to derive macromodels to be used during the early stages of the design. Nevertheless, their accuracy and level of detail is generally unacceptable to actually optimize, with good confidence, application code exploiting process-level concurrency.

The goal of our work is to overcome the above problems, providing the designer with: a *methodology* and a *probing*

978-0-7695-3277-6/08 $25.00 © 2008 IEEE
DOI 10.1109/DSD.2008.86

tool to characterize the execution of operating system services; a set of *models* characterizing the OS functions with cycle-accurate timing, and an *analysis framework* enabling a better understanding of the impact on power and timing of IPC decisions and process-level organization of the application. More precisely, we aim at providing a set of figures and/or models that can be used *statically* within a source-level analysis framework [13]. At this level of abstraction the overall application timing/power can bee seen as composed of four components: the *algorithmic* portion, the *library* functions that do not involve system calls, the *explicit system calls* and the *kernel services*.

This paper focuses on a methodology to characterize real-time operating systems with cycle accuracy. To show the viability and the accuracy of the proposed approach we present the results obtained for VxWorks and RTEMS. The platforms used for the measurements are based on the Xilinx/PowerPC and Altera/ARM SoC platforms.

The paper is organized as follow. Section 2 describes the framework that has been set up and used to obtain cycle accurate execution time measurements; Section 3 reports some representative sample measures that have been collected for both operating systems along with a dicussion on the observed behavior and the proposed modeling approach. Section 4 starts by reporting a more extensive set of measures and the result obtained by applying one of the proposed models to some sample function classes. To provide a broader view of the outcome of the work, the observed behavior of different classes of functions are briefly commented. Finally, Section 5 draws some conclusions and outlines further developments of the current investigation.

2. Analysis framework

2.1. Hardware/software setup

This section describes the *hardware* and *software* environments in which tests have been developed and run. The two SoC platforms used consist of an FPGA integrating a processor core (a Xilinx Virtex2Pro with IBM PowerPC and an Altera EPXA1 with ARM 922T) and platform-specific communication and debugging facilities. On top of the hardware platforms we run the ports of the two operating systems mentioned. In our framework the FPGA logic is used to build the essential peripherals for running the tests: an SDRAM controller a serial port, a timer, the bus arbiter, a bus bridge and an interrupt controller. A sketch of the complete system we have used is shown in Figure 1.

Although a few timers are made available by the processor cores we considered, they are used by the operating system so we could not use them without interfering with it. This motivated the adoption of an external timer built on the FPGA logic and running at the same clock frequency

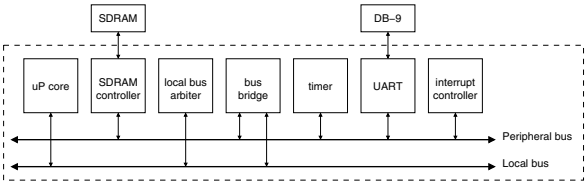

Figure 1. Measurement SoC on the FPGA

as the processor core. The timer has the control signals, interface registers and the drivers necessary to make it accessible via software thanks to a simple API consisting of the three functions `timerInit()`, `timerStart()` and `timerStop()`.

2.2. Measurement Framework

The main goal of this work is to build a general and easily portable measurement framework. This has been achieved by means of a timer and a communication channel (the serial port in the prototype board) only, on the hardware side, and by isolating the necessary software function calls in appropriate layers (interfacing and communication).

Measurements are based on the execution of *stubs*, i.e. suitable programs calling a given function for a large number of times under possibly varying conditions. Each stub generates five series of data and each series repeats the measure 10, 25, 50, 75, and 100 times. Measurements have been structured in series to account for different levels of operating system loads. For some functions (e.g. task creation and suspension, semaphore flush, etc.) the execution time might depend on the system load and in particular on the number of tasks, semaphores and message queues.

For simple functions, the core of a stub is the function invocation embedded in a timer start/stop pair. It is important noting that, due to the timing overhead implied in accessing a peripheral on an external bus, such as the timer, all measures are affected by a systematic error; this can be easily compensated by subtracting the time measured when invoking the timer start/stop pair without a function call in between. This timing overhead is actually measured only once at the beginning of the test suite. For more complex functions, such as task suspension and resuming, the timer cannot be started (or stopped) in the line preceding (or following) the invocation of the function under measurement since the execution flow is changed by the function call itself. In such cases the timer start/stop calls are embedded into the task itself and subject to suitable conditions.

It is worth noting that an unpredictable measurement error is introduced by the *system tick*: registered tasks are alerted of this event and are given the opportunity to react by executing some handler. This is a primary component of the operating system and its inhibition leads to system halt or

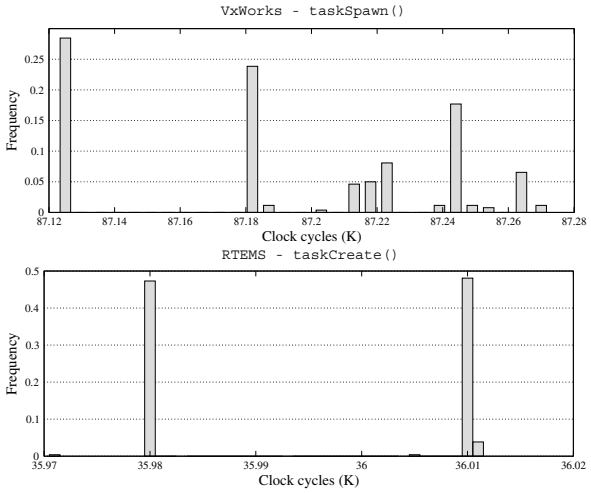

Figure 2. Task creation

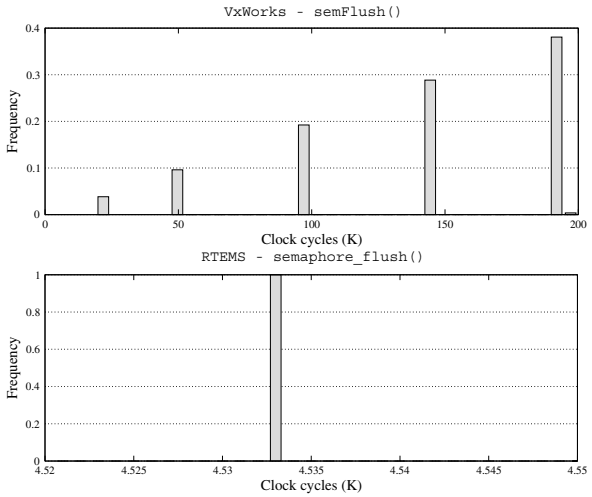

Figure 3. Semaphore flushing

Library	# Functions	
	VxWorks	RTEMS
OS-specific Library	94	58
Standard C Library	66	66
POSIX Library	41	45

Table 1. Number of measurements performed

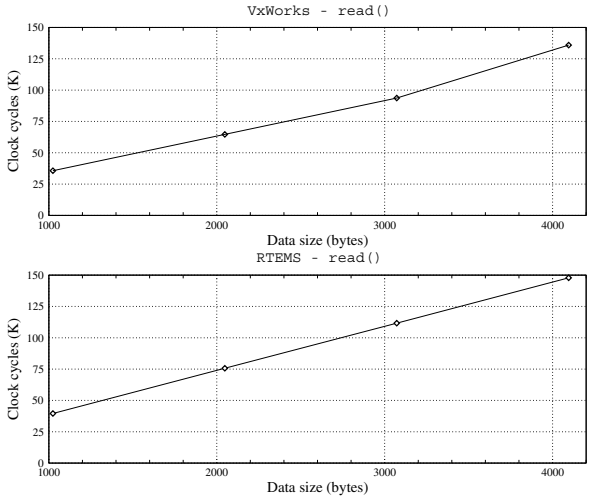

Figure 4. Raw read from file

malfunctioning. Although ad-hoc techniques can be studied for each operating system, the most secure and portable solution consists in slowing down the tick frequency as much as possible and running tests between two consecutive ticks.

3. Measures and Modelling

In this section we present a selection of the collected data. For the most relevant functions we present average execution time and standard deviation together with a plot of the data.

For the measured values to be significant, all data is preanalyzed and only the suites in which *all* the series show a sufficiently low standard deviation have been considered. Series with higher standard deviations are further investigated to identify a possible dependence on "hidden" parameters. The measurement campaign studied a large number of system calls (in different operating conditions) belonging to the standard C library, the POSIX libraries and the OS-specific libraries. Table 1 reports the number of measurements performed, classified according to the library they belong to (a measure refers to a pair system call/operating condition).

As an example, Figure 2 reports the execution time density for the functions for task creation on the PowerPC processor. In both cases we see that the behavior is characterized by two or three clear peaks and a few measures falling nearby these peaks. Such deviations from the average value affect almost all measurements and can be justified, at a first approximation, when effects such as bus activity of the platform debugging infrastructure, or background operating system operations are taken into account.

Figure 3 shows the behavior of the semaphore flush primitive. When multiple tasks are all suspended waiting for a semaphore, this call atomically makes all of them ready to run. In this case a number of tasks equal to the number of iterations in the series has been created, and

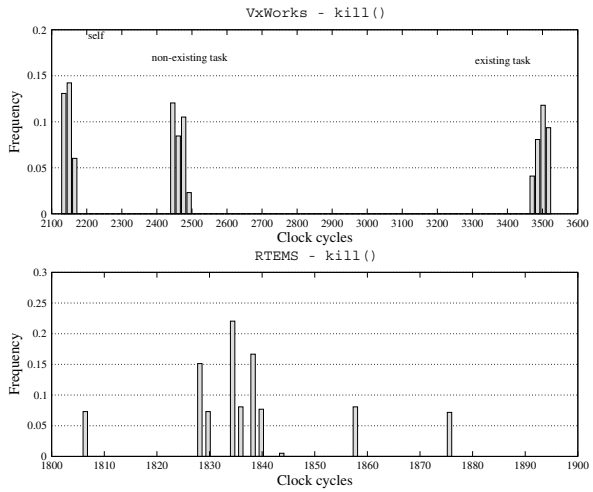

Figure 5. Signal sending

forced to block on a semaphore. This, again, was done to identify possible load dependency. The hypothesis proved correct as shown by the 5 clusters in the histogram for VxWorks. Of course, anytime a 5-column histogram is discovered further data-dependent analysis should be performed. In addition to the low-level functions, we also measured most of the functions from the Standard C Library. Figure 4 shows the timings for the `read()` call under VxWorks. This function takes a block-size as a parameter and, as expected, shows a linear dependency of the execution time on this parameter. The same behavior has emerged for all other operating systems studied. As a last example, consider the `kill()` function. Measures have shown no dependency on the type of signal being sent but only on the process being signaled, namely whether it is the sender itself, a different task or a non-existent task (see Figure 5). Collecting the results of all measurements, the following behaviors emerged:

- *Constant*. Always the same behavior.
- *State dependent*. The behavior depends on some notion of the state of the system or a part of it, such as the average load, the state of a process, etc.
- *Enumerated parameter dependent*. The behavior differs for a small number of cases, such as the state of a process, buffered/unbuffered I/O, etc. In such cases the dependence can be made explicit by enumeration of the different cases.
- *Numerical parameter dependent.* The behavior depends on the numerical value of a parameter, such as buffer size, memory block size, etc.

To classify such cases with the goal of identifying a simple mathematical model, we adopted the following criteria:

1. A dependency on an enumerated parameter is eliminated by associating a different model to each case.
2. The model of a function in a specific operating condition can only be constant, linear or piecewise-linear.

According to these criteria, for example, the `kill()` function will be represented by 3 different models (`kill-self`, `kill-other` and `kill-nonexistent`), one for each possible condition; all models will be, in this case, constant.

4. Results

4.1. Measurements

In this section we present a selection of the collected data. For the most relevant functions we present average execution time and standard deviation. To build density histograms for a function, the interval between the minimum and maximum times has been split in 30 bins and the occurrences of the different measures have been counted for each bin. This type of plot is useful to show the distribution of the execution time for those functions with no explicit parameter dependency. For functions displaying a dependency from a quantitative parameter such as data size, number of elements, etc. one of the three possible models has been used. For such models to be significant, all data is preanalyzed and only the data-sets in which *all* the series show a sufficiently low standard deviation have been considered.

Tables 2 and 3 summarize the results of the characterization on a subset of the selected functions. Similar results have been obtained for the remaining functions of both operating systems, but are omitted here for the sake of conciseness.

For each function, along with its name, average execution time, and standard deviation, a brief description of the operating conditions under which it has been run is also reported. Furthermore, the following operating conditions apply for all functions:

1. all task are identical;
2. semaphores use FIFO queues without priority for pending tasks;
3. tasks access messages according to a FIFO policy without priority.

4.2. Modeling

Due to the large amount of data derived from the measurements, an automated procedure for data fitting and

Function	Operating conditions	μ_T	σ_T
taskSpawn[1]		87262	1073
taskDelete[1]	Task suspended	29856	224
taskDelete[1]	Task Blocked	30642	224
taskSuspend[1]	Spawned Task	2361	24
taskResume[1]	Spawned Task	2442	13
semBCreate[1]	Binary, full	12535	22
semDelete[1]	Binary, full	12833	53
semTake[1]	Binary, full	1211	8
semGive[1]	Binary, N blocked tasks	3881	11
msgQCreate[1]		101716	29198
msgQDelete[1]	N messages	99118	31340
msgQSend[1]	Normal,	4002	11
msgQSend[1]	Normal, full, no wait	2883	10
msgQSend[1]	Normal, N blocked tasks	6562	19
msgQReceive[1]	N messages	3865	20
msgQReceive[1]	No messages, no wait	2746	13
raise[2,3]		2021	14
kill[2,3]	Existing task	1847	19
open[2]		354803	4279
close[2]		18411	30
read[2]	1024 bytes	35672	75
write[2]	1024 bytes	161779	1492
getc[2]	Fully buffered	3504	7416
putc[2]	Fully buffered	16298	36117
fopen[2]	File in root directory	380526	149
fclose[2]		28955	25
fread[2]	1024 bytes	58493	595
fwrite[2]	1024 bytes	164135	4622
fgetc[2]	Fully buffered	3529	7219
fputc[2]	Fully buffered	16372	36070
fdopen[3]		8908	22
fileno[3]		231	7
setjmp[2]		992	8
longjmp[2]		3549	316
malloc[2]	1024 bytes	6699	20
calloc[2]	128 items, 8 byte/item	9726	17
realloc[2]	1024 to 2048 bytes	44501	80
free[2]	1024 bytes	8578	25
memcpy[2]	1024 bytes	21122	81
memmove[2]	1024 bytes	29933	15
memset[2]	1024 bytes	3134	12

[1] Functions from VxWorks Base Library.
[2] Functions from Standard C Library.
[3] Functions from POSIX Library.

Table 2. VxWorks characterization results

Function	Operating conditions	μ_T	σ_T
task_create[1]		35998	18
task_delete[1]	Task suspended	3499	347
task_delete[1]	Task Blocked	3499	348
task_suspend[1]	Spawned Task	3402	346
task_resume[1]	Spawned Task	3324	344
semaphore_create[1]	Binary, full	6034	16
semaphore_delete[1]	Binary, full	3021	275
semaphore_obtain[1]	Binary, full	2714	204
semaphore_release[1]	Binary, N blocked tasks	5521	705
queue_create[1]		16226	3393
queue_delete[1]	N messages	8161	25
queue_send[1]	Normal,	8650	20
queue_send[1]	Normal, full, no wait	4885	15
queue_send[1]	Normal, N blocked tasks	4902	235
queue_receive[1]	N messages	7148	25
queue_receive[1]	No messages, no wait	4501	11
raise[2,3]		N/A	N/A
kill[2,3]	Existing task	1837	15
open[2]		49609	18
close[2]		29105	103
read[2]	1024 bytes	39270	5
write[2]	1024 bytes	63854	832
getc[2]	Fully buffered	1696	3224
putc[2]	Fully buffered	2491	5001
fopen[2]	File in root directory	56959	23
fclose[2]		31085	29
fread[2]	1024 bytes	60896	25
fwrite[2]	1024 bytes	175693	6768
fgetc[2]	Fully buffered	1963	3133
fputc[2]	Fully buffered	2878	5022
fdopen[3]		40876	18319
fileno[3]		606	9
setjmp[2]		813	8
longjmp[2]		1045	8
malloc[2]	1024 bytes	10679	8
calloc[2]	128 items, 8 byte/item	26503	29
realloc[2]	1024 to 2048 bytes	55075	26
free[2]	1024 bytes	10235	11
memcpy[2]	1024 bytes	22802	12
memmove[2]	1024 bytes	137670	9
memset[2]	1024 bytes	13412	16

[1] Functions from RTEMS Base Library.
[2] Functions from Standard C Library.
[3] Functions from POSIX Library.

Table 3. RTEMS characterization results

model parameters calculation has been developed and implemented. The methodology adopted for analysis and modeling operates on all data sets corresponding to a specific function and is based on the following steps:

1. Initial clustering. As an hypothesis, all data is considered as belonging to a single cluster, assuming thus that the function has either a constant, linear or piecewise-linear behavior.

2. Statistical analysis. For each cluster (initially one) the standard deviation is calculated and compared to a fixed threshold. If standard deviation is below the threshold, the clustering is finished, otherwise a new cluster in introduced. This process is repeated until the standard deviation within each cluster falls below the threshold.

3. Data in each cluster is then modeled both with a constant value or with a linear relation with the least square method. Of the two models, the one exhibiting the minimum average error is selected.

This procedure is repeated for each operating system. Table 4 reports some sample models obtained for a few functions (file access and memory management). In such cases the measured data could be fitted by simple linear funcions of the form $t = mx + q$, where t is the estimated execution time, x is the value of the independent parameter chosen to model the function, while m and q are the coefficients derived with least square fitting.

4.3. Qualitative comparison

This section discusses the differences between the two operating systems, observing function classes only.

			VxWorks		RTEMS	
Function	Condition	x	m	q	m	q
`fopen`		depth	12984.35	380526	2984.44	56959
`fread`		size	59.05	15551	58.06	1414
`fputs`	full	size	3.16	19470	4.05	10952
`fputs`	unbuf	size	1.17	39556	0.99	28870
`fseek`	set			12675		6789
`memcpy`		size	0.84	21122	0.91	22802
`memset`		size	0.79	3134	1.22	13412
`malloc`				6699		10679
`realloc`		Δsize	0.66	44501	0.86	55075
`calloc`		size	0.13	9726	0.28	26503

Table 4. Examples of models

File access. Every function in this class has two variants, one that refers to files via *file descriptors* and the via *streams*. We observe that functions based on file descriptors are generally faster. The form of the models for the functions in this category (constant or linear) is the same for the two operating systems, except for the `fseek()` and `fdopen()` functions. The former, in particular, always executes in constant time except on VxWorks when seeking to the end of file: in this case the execution time grows linearly with the file size.

Memory. All the functions that copy, search, compare or assign characters to blocks of memory are linear in the block size, while allocation and deallocation functions are constant. Two exceptions are the `calloc()` function, that not only allocates a block of memory but also clears it to 0, and the `realloc()` function, that needs to move the block whenever it cannot be expanded inplace. There are no difference in the form of the models for the two operating systems for this class of functions.

Tasks. All functions show a constant behavior. It might be expected that the functions managing tasks to depend linearly on the number of tasks in the systems, due to the need to scan the list of active tasks. This is, indeed, not true for neither of the two operating systems. Probably, they make direct use of the task id to access a vector containing all the tasks information. The only exception is the `taskNameToId()` function on VxWorks: since it returns a task's id given the task name as a string, the function cannot avoid scanning the entire list to find the named task.

Semaphores. Most of the semaphore management functions show a constant behavior. We noticed that, on VxWorks, mutex semaphores are managed differently from other types of semaphores during obtain and release procedures in presence of tasks pending on the semaphore. Furthermore, the `semFlush()` function on VxWorks executes in a time that linearly depends on the number of tasks pending on the flushed semaphore, showing that the list of the tasks is scanned to awake them. In RTEMS all the semaphores functions display a constant behavior.

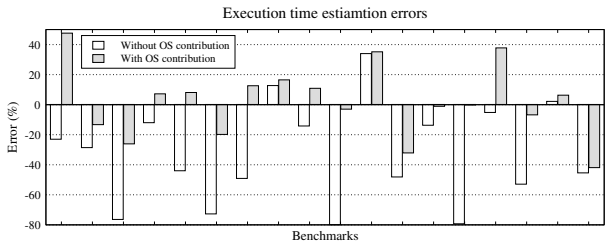

Figure 6. Source-level estimation errors

Message queues. Management of message queues is similar on both operating systems and the models obtained confirm this observation. Semaphore creation is, on both operating systems, linear in the maximum number of messages to hold. The delete function, although actually linearly on both operating systems, has such a small slope on RTEMS that can be considered constant. The time for sending a message only depends on the status of the queue, with the difference that, on RTEMS, sending when there are blocked tasks take the same time that when the queue is full. The message receive function shows the same behavior, but in this case RTEMS shows two different execution times related on the type of queue, priority-based being slower.

4.4. Platform dependence

To complete the analysis we have studied the influence of the different hardware configurations of the platforms used during the test phase. The only significant difference in the two configurations is the position of the SDRAM controller, connected either on the local bus or on the peripheral bus. The values of the models parameters in the two configurations showed an almost constant ratio K. As Table 5 shows, K is not strictly constant but rather varies in a range between K_{min} and K_{max}. This is due to the fact that, for each function, the size of the portion of code that can take advantage of the RAM access speed-up varies. The very small standard deviation, though, allows considering the ratio K as constant to a first approximation.

Comparing the values of K for the two operating systems, we noted that the value for VxWorks is larger that that for RTEMS. This might suggest that RTEMS is structured in such a way to be less dependent than VxWorks on the memory access speed, but a deeper analysis is required to prove this hypothesis.

O.S.	K	K_{max}	K_{min}	σ	$\sigma(\%)$
VxWorks	1,83	2,42	1,49	0,11	6,09
RTEMS	1,07	1,73	0,92	0,10	9,58

Table 5. HW configuration performance ratios

4.5. Source-level estimation

Following the presented approach we have derived the models for all the functions under several significant operating conditions. Such models have then been integrated into a software estimation tool-chain that operates entirely at the source code level [13].

To prove the accuracy of the models and to show their usefulness in such a context several benchmarks from the MiBench [11] belonging to different domains: security, automotive, telecom, network, office and consumer. We have estimated the execution time for each benchmark first without considering operating system calls and then including their estimates derived from the presented models (see Table 6).

The target platform used for benchmarking is based on the ARM 922T core and the VxWorks operating system. The reference values of the execution times have been directly measured following a procedure similar to that employed for the measurement of the execution times for individual functions. As Figure 6 shows, there is a significant improvement of the estimation error when OS function calls modes are accounted for. The average error in absolute value, in fact, is reduced from 35.4% down to 17.2% as Table 6 shows.

	Execution times (ms)			Errors (%)	
Benchmark	T_{act}	$T_{w/o}$	$T_{w/}$	$\epsilon_{w/o}$	$\epsilon_{w/}$
adpcm-enc	191.0	95.5	282.0	12.3	47.6
adpcm-dec	171.0	89.3	148.2	-55.7	-13.3
bitcount	318.0	175.7	234.9	-78.8	-26.1
blowfish-enc	483.0	495.0	518.0	-24.9	7.2
blowfish-dec	479.0	494.0	518.0	-39.4	8.1
crc32	637.0	119.4	511.1	-51.9	-19.8
dijkstra	1766.0	1248.9	1989.1	-1.7	12.6
fft	3366.0	962.9	3919.9	-35.6	16.5
ispell	80.5	19.8	89.3	-76.1	10.9
jpeg	504.7	391.8	489.5	-22.2	-3.0
qsort	199.1	205.6	269.1	28.0	35.2
rijndael-enc	6874.4	1407.6	4664.5	-83.4	-32.1
rijndael-dec	245.0	183.8	242.3	-8.2	-1.1
sha	238.0	148.4	237.6	-5.2	-0.2
stringsearch	74.0	96.8	102.0	-57.0	37.8
susan	44.0	28.6	41.0	-17.7	-6.8
tiff2rgba	180.0	65.0	191.4	-19.6	6.3
tiffdither	16.6	4.2	9.7	-55.9	-41.9
Average				35.4	17.2

Table 6. Results on MiBench benchmarks

5. Conclusions

In this paper we presented a measurements and characterization methodology achieving cycle-true accuracy in the analysis of operating system calls. The approach is based on the opportunities offered by SoC hardware/software architectures (Xilinx/PowerPc and Altera/ARM), where reconfigurable hardware is also used as a probing tool.

The main benefits of such approach lay in the simplification of the measurement infrastructure, the flexibility with respect to different operating systems and microprocessors, the reliability deriving from the use of data coming from in-the-field measures and, finally, the coverage of the entire set of system calls of full-featured commercial and open-source operating systems (VxWorks and RTEMS). The results we have obtained constitute a sound starting point for a more complete analysis of software timing/power characteristics, both for estimation and optimization purposes and allow covering the whole spectrum from source-level down to system calls. Work is in progress to extend the experiment considering other operating systems and to derive a general set of parametric models for standard systems calls to speedup, with good confidence, the typical software or hardware/software design flows.

References

[1] S. Gurumurthi, A. Sivasubramaniam, M.J. Irwin, N. Vijaykrishnan, M.T. Kandemir, T. Li, and L.K. John, "Using complete machine simulation for software power estimation: The SoftWatt approach," *In Proc. of Int. Symposium on High Performance Computer Architecture,* pp. 141–150, 2002.

[2] T. K. Tan, A. Raghunathan and N.K. Jha, "EMSIM: An energy simulation framework for an embedded operating system," *In International Symposium on Circuits and Systems,* pp. 464–467, 2002.

[3] R.P. Dick, G. Lakshminarayana and N.K. Jah, "Analysis of power dissipation in embedded systems using real-time operating systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 22, N. 5, pp. 615–627, May 2003.

[4] K. Baynes, C. Collins, E. Fitterman, B. Ganesh, P. Kohout, C. Smith, T. Zhang and B. Jacob, "The performance and energy consumption of embedded real-time operating systems," *IEEE Transactions on Computers,* Vol. 52, N. 11, pp. 1454–1469, November 2003.

[5] T. Li and L.K. John, “Run-time modeling and estimation of operating system power consumption,” *In SIGMETRICS’03,* San Diego (CA), US, June 2003.

[6] T.K. Tan, A. Raghunathan and N.K. Jha, “Embedded operating system energy analysis and macromodeling,” *In IEEE International Conference on Computer Design: VLSI in computers and Processors,* 2002.

[7] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” *In ISCA’00, Int. Symposium on Computer Architecture,* pp. 83–94, 2000.

[8] A. Acquaviva, L. Benini, B. Riccò, “Energy Characterization of Embedded Real-Time Operating Systems,” *In ACM SIGARCH Computer Architecture News,* Vol. 29, N. 5, December 2001, Special Issue PACT’01.

[9] J. Furunäs, “Benchmarking of a Real-Time Systems that Utilises a Booster,” *In Conf. on Parallel and Distributed Processing Techniques and Applications*, PDPTA’00, Las Vegas, US, June 2000.

[10] R. Oliver, W.P. McGregor and P.J. Teller, “Accurate measurement of system call service times for trace-driven simulation of memory hierarchy designs,” *Proceedings of the 1998 IEEE International Performance, Computing, and Communications Conference,* pp. 239–244, Tempe/Phoenix, AZ, February 1998.

[11] M.R. Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite,” *Proc. IEEE 4th Annual Workshop on Workload Characterization,* Austin, TX, USA, Dec. 2001.

[12] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, “Source-Level Execution Time Estimation of C Programs,” *Proc. of International Workshop on Hardware Software Co-Design,* Copenhagen, Denmark, April 2001.

[13] C. Brandolese, F. Curto, W. Fornaciari and F. Salice, “Analysis and Modeling of Energy Reducing Source Code Transformations,” *Proc. of IEEE/ACM Conference on Design Automation and Testing in Europe,* Paris, France, February 2004.

[14] C. Brandolese, W. Fornaciari, F. Salice and D.Sciuto, “Library Functions Timing Characterization for Source–Level Analysis,” *Proc. of IEEE/ACM Conference on Design Automation and Testing in Europe,* Munich, Germany, March 2003.