One-Shot Time Management Analysis in EPOS

Giovani Gracioli Danillo Moura Santos Roberto de Matos Lucas Francisco Wanner
Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 88040900 - Florianópolis - SC - Brazil
{giovani,danillo,roberto,lucas,guto}@lisha.ufsc.br

Abstract

One of the tasks of an Operating System is to handle time events. Traditionally, time management is based on periodic interrupts from one of the system's hardware timers (ticks). However, this approach has some limitations, as lack of precision, large overhead, and large power consumption. These limitations have motivated the use of non-periodic timers (e.g. one-shot timers), specially in specific-purpose operating systems with timing restrictions, such as embedded, real-time, and multimedia systems. This work presents a comparison between one-shot and periodic time implementations in the time management abstractions in EPOS (Embedded Parallel Operating System). We compare both implementations in terms of memory footprint, number of context switches, number of interrupt handler executions and run time in different execution scenarios.

1 Introduction

Traditionally, general purpose Operating Systems implement time management based on periodic interrupts triggered by one of the system's hardware timers. However, this approach to time management has shown several problems such as lack of precision, high overhead and high energy consumption [10, 2, 1]. The use of non-periodic timers, such as the one-shot timer approach, promises to solve these issues, by allowing fine-grain control of timers [10].

Periodic timers are implemented using hardware clock interrupts (*ticks*). At every *tick*, the kernel performs time management tasks such as pooling the alarms queue, trigger ready alarms, task runtime accounting, scheduling analysis, preemption and possibly other system services. This kind of approach has some problems, such as waste of energy in the case of mobile and embedded systems and security problems [10]. Furthermore, *real-time* tasks are subject to the limited accuracy of the system clock.

In addition to these problems, the implementation of periodic timers can generate rough errors in the timing services provided by the OS (Operating System) [2]. For example, in the Linux operating system, the timer resolution is 10ms. Assume that an application requests a time interval of 15ms right after the completion of a timer interrupt. In this scenario, time accounting for this request will only start in the next timer interrupt and, due to limitations in timer resolution, the requested period will be rounded up to 20ms. This way, a request for a 15ms interval will translate into a period of roughly 30ms, which is unacceptable for many applications.

An alternative to solve these problems is the use of non-periodic time management, usually through the use of a one-shot timer. This approach guarantees that the timer interruption happens only when it was scheduled. Whereas a periodic timer is triggered at a predefined frequency, with timer requests logic performed in software, a one-shot timer is triggered according to the period of its next request. However, as one-shot timers are scheduled in hardware, the maximum resolution supported is the resolution of the hardware timer.

This work is an implementation analysis of two time management approaches in the EPOS (Embedded Parallel Operating System) [3], one based on one-shot timer and another based on periodic timer. For each approach, we present system code size (footprint), number of context switches, and total computation time. This analysis aims to discuss the positive and negative aspects of one-shot timer implementation compared to periodic timer implementation.

This paper is organized as follows: section 2 shows the related works and the different approaches in OS time management. Section 3 describes the design and implementation of periodic and one-shot time management strategies for EPOS. Section 4 describe test results and its analysis. Finally, section 5 closes the paper with final considerations and future work.

2 Related Works

Tsarfrir et. Al. in [10] show some problems of periodic time management, as lack of precision, and power consumption. This work proposes a solution based on *Smart Timers*, which have three basic properties: 1) Accurate timing with configurable maximum latency; 2) Reduced management overhead by triggering combined nearby events, and (3) Reduced overhead by avoiding unnecessary periodic events.

Kohout presents a strategy to efficiently support real-time OS, using core components implemented in hardware [6]. The main objective is to reduce the impact caused by the real-time OS in the application. This impact is measured in terms of response time and CPU use. This work shows the Real-Time Task Manager (RTM), which is a kind of tasks smart cache memory, implemented as a peripheral on the same processor chip (thus the processor must be in a FPGA). The RTM-core supports time management functions, causing a 10% reduction of CPU time (with 24 tasks), earlier used for periodic ticks handling.

Soft Timer [1] is an implementation of time management that is not triggered only at hardware timer interrupts. This approach test and trigger pending tasks in the return of system calls. This strategy decreases the number of context switches, and the number of timer interrupts, as long as the rate of system calls are higher than the timer interrupts. However, as the exact frequency of system calls is unpredictable, there is no guarantee of precision and periodic timer approach is used to satisfy minimum operating system requirements.

Firm Timer [5] combines three different approaches: One-shot Timer, *Soft Timer* [1], and periodic timer to provide an efficient high resolution and low overhead mechanism for time management. This combination decreases the number of timer interrupts, reducing the risk of excessive overhead.

The basis of these works is to avoid unnecessary interventions of the OS caused by the timer interrupts handlers, when the only action is to increase the ticks' counter of the system. In the One-Shot Timer management approach the interrupt happens only at the scheduled time, avoiding the overhead of periodic interrupts handling. Although One-Shot interrupt handler is more complex, because it has to program the hardware timer to trigger the next task in queue, the One-Shot time management approach has better performance than periodic timers in most cases. In a specific scenario, where the intervals between the tasks wake up time is close to one tick of the system, the periodic timer approach can be better than the One-shot, as will be shown in the next sections.

3 Time Management in EPOS

EPOS is an operating system for embedded systems developed following AOSD *(Application-Oriented System Design)* [3] concepts. AOSD uses Domain Engineering to define components that represent significant entities of a domain. The EPOS operating system combines concepts of FBD *(Family-Based Design)*, AOP *(Aspect-Oriented Programming)*, Object-Oriented Design (OOD) and static metaprogramming (SMP), which allow the organization of components in families which are scenario independent. Through a metaprogram that uses composition rules, EPOS creates a framework of components that allows the adapted systems generation to application. The system abstractions are adapted to execution scenarios using aspect orientation techniques, applied with Scenario Adapters [4]. Some rules coordinate the operations carried out by the metaprogram, specifying restrictions and dependencies to the system abstractions composition. Each aspect can be applied individually to each system abstraction, for instance, the application of shared aspect in a component will preserve its integrity. The use of metaprogramming to compose the system abstractions does not add run time overhead.

The use of hardware mediators [8] allows the support of different architectures by the system (e.g H8, AVR, POWERPC, SPARCV8, IA32) keeping the same interface with the application. Basically, the hardware mediators are constructions that encapsulate architecturals dependencies in systems design following AOSD, giving to hardware components as CPU, FPU and buses, common operating system interfaces. For example, in time management, the hardware timers present several distinct functions and can be configured in several different ways. A timer can act as a pulse width modulator controlling an analog circuit, like *Watchdog*, timer with programmable interval or a simple timer with fixed interval. Each of these timers have their configuration peculiarities.

In the highest level of EPOS system, the time is handled by ***Timepiece*** abstraction family, composed by ***Clock***, ***Alarm*** and ***Chronometer*** abstractions. Each abstraction has a system specific function. The ***Clock*** abstraction is responsible to store the current time and is only available in systems that have real-time clock (RTC). The ***Chronometer*** abstraction is used to perform high precision time measurements and the ***Alarm*** abstraction can be used to generate events, to wake up a Task/Thread or call a function. This family is supported by ***Timer***, ***TimeStamp Counters (TSC)*** and ***Real-Time Clocks*** hardware mediators [7].

The system periodicity is characterized in the ***Alarm*** abstraction, that is supported by ***Timer*** mediator that configures the hardware timer to generate interrupts in a determined frequency. In the timer handler these *ticks* are counted and a routine to verify and to release the sched-

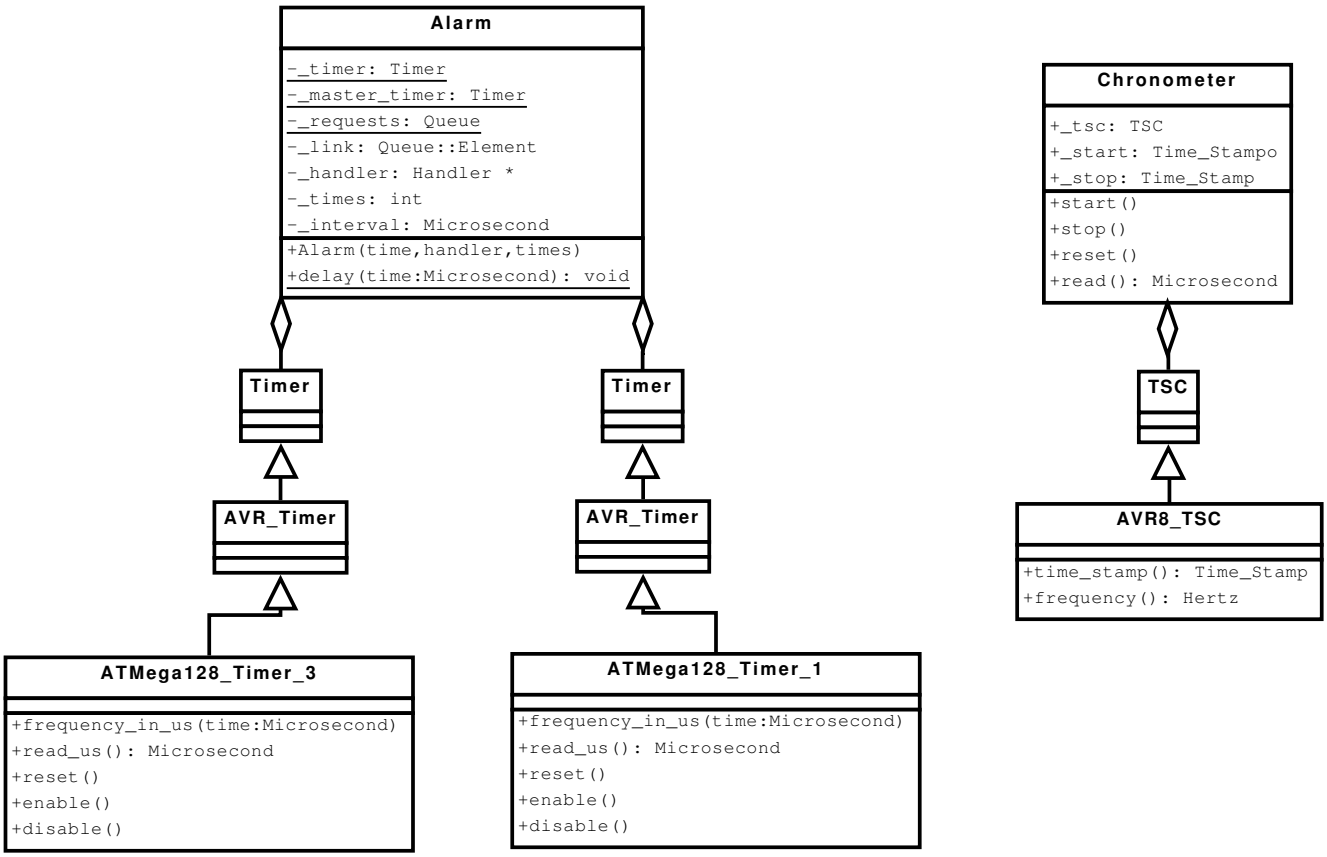


Figure 1. Classes diagram of EPOS abstractions and hardware mediators (Timers) with one-shot

uled events is executed. In addition to overhead caused by the periodic interrupt handler, if an event is scheduled in a shorter time or not multiple of system period, the invocation of scheduled task can be delayed by rounding errors due to *ticks* granularity. Figure 1 presents the classes diagram of the two main abstractions that compose the EPOS time management in the *AVR* architecture. The **Alarm** abstraction uses two hardware *Timers* available in *AVR* architecture. One of these *Timers*, **ATMega128_Timer_3**, is used to control the **Alarm** requests queue, this is a 16 bits *Timer*, which allows the alarms scheduling of approximately 9 seconds with the system Clock in $\sim$ 7.2MHz and *PRESCALE* of 1024 in *Timer* Clock. Another *Timer*, **ATMega128_Timer_1**, generates periodic interrupts that trigger the system schedule, these interrupts only occur when the schedule must execute and their time is configured based on the system *QUANTUM*.

The interrupt handler in the periodic time management, in addition of counting *ticks* and release the scheduled events, also invokes the scheduler time handler (*master_handler*). This routine decreases the scheduler *ticks* counter and verifies if this counter is shorter or equal to zero (QUANTUM ended), calling the thread reschedule routine when necessary.

The modifications to eliminate the timer periodic interrupts (counting of *ticks*) and to create the timer structure based on one-shot timer focused in the **Alarm** abstraction. By changing the concept from periodic interrupt to variable interval of the next trigger, the ***Alarm*** interrupt handler was implemented as the sequence diagram in Figure 2. Unlike of periodic approach, where every *tick* the interrupt handler called *master_handler*, the new implementation uses two independent hardware timers. One related to alarms requests (*Queue _requests*), which is programmed to trigger at the precise moment that was scheduled to wake up a task and another timer related to system schedule, programmed to trigger every schedule *QUANTUM*.

In this new implementation, the timer related to scheduling is set in the system initialization and when it is shot simply calls the scheduling routine. This treatment is simple and fast, because it does not need to do the control of the *ticks* number in every interrupt and also decreases the schedule influence in the system. The timer related to alarms only is initialized in the **Alarm** creation. When it is created, the **Alarm** is added in an ordered and relative queue, then the added value is exactly the value required to program the

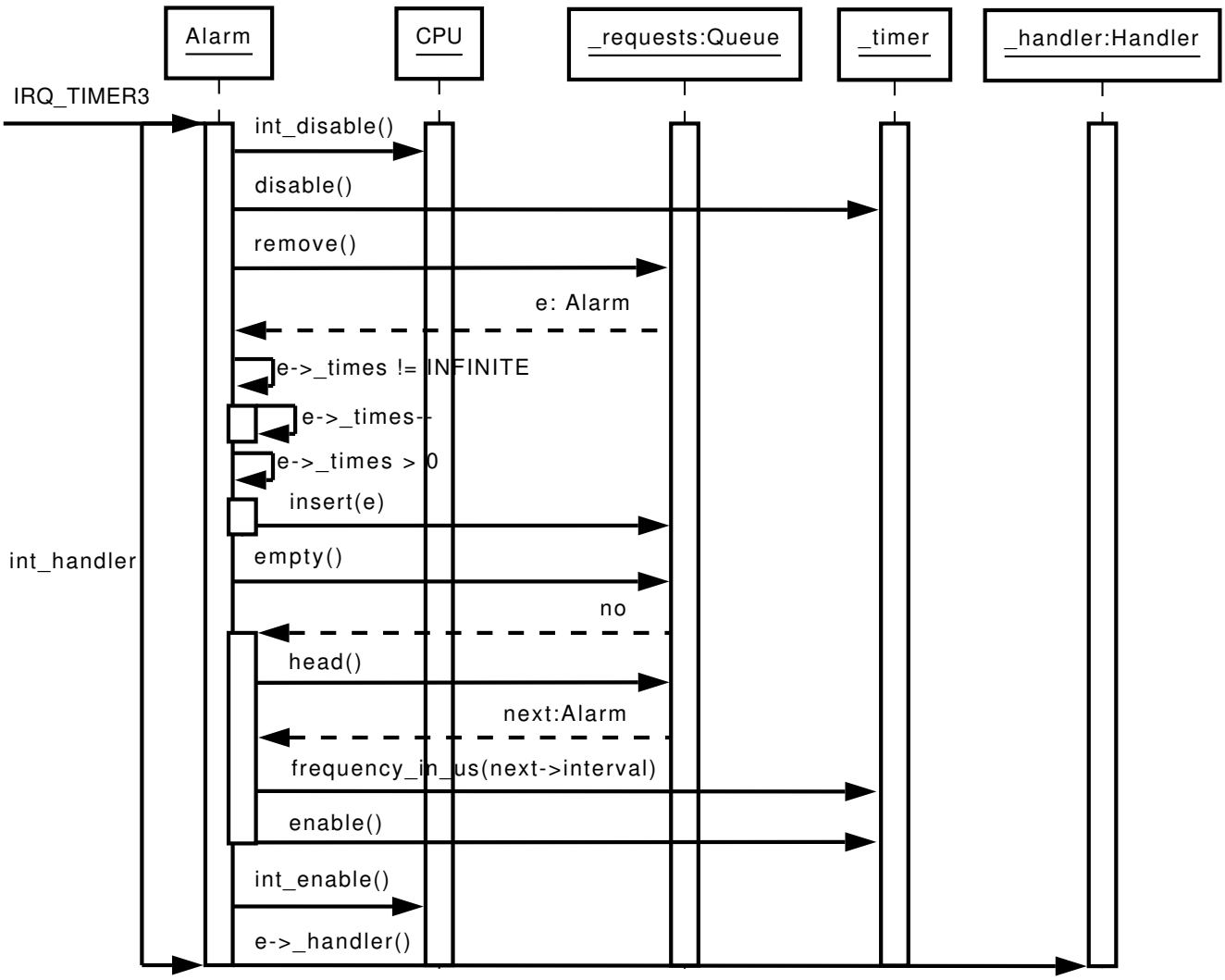


Figure 2. Sequence diagram of EPOS one-shot timer interrupt handler

timer. When an interrupt arrives, the handler removes the **Alarm** on the first position in the queue, verifies if there are **Alarms** in the queue and if exist, reprogram the timer with the next value. This interrupt handler is executed much less than periodic handler, because it is called only when a timer event needs to be handled. We can see the impacts on these numbers in the next section.

4 Results Analysis

To analyze the impact of our One-Shot timer implementation in EPOS was used the well-known Philosophers dinner application [9]. This implementation of the problem has five threads, that behave like the philosophers, changing their state between thinking and eating. The philosopher needs two forks to eat the Spaghetti and there are five forks on the table, therefore, two non-adjacent philosophers can eat at the same time. The thinking and eating behaviors are a delay (call to the **Alarm::delay method**) and to eat, the philosopher have to catch two forks that are Semaphores in our implementation (call to **semaphore.p()**). Each thread executes ten times.

This test application was compiled to Atmel Atmega128 microcontroler. This is an 8 bits microcontroler, with 4KB RAM memory, 128KB flash memory, two 8 bits and one 16 bits timers. As explained in section 3 one 8 bits timer was used to trigger the operating system scheduler and one 16 bits timer was used to manage the Operating System Alarm queue (delays). Four characteristics were analyzed in these tests:

1. *Memory footprint*: embedded systems have low resources available, processing, memory and power con-

Table 1. Memory footprint of the test application using both implementations: Periodic and One-Shot Timer

Periodic Timer (bytes)	*Single-shot (bytes)*	*Difference (bytes)*
32000	**32504**	**504**

Table 2. Number of executions of the interrupt handler. Periodic Time Manager VS Single-Shot Time Manager

Scenario	*Handler*	*Periodic*	*Single-Shot*	*Difference*
Scenario 1	int_handler	**26005**	**100**	**25905**
Scenario 1	*master_handler*	**26415**	**3559**	**22856**
Scenario 2	int_handler	**9843**	**100**	**9743**
Scenario 2	*master_handler*	**10252**	**320**	**9932**
Scenario 3	int_handler	**3079**	**100**	**2979**
Scenario 3	*master_handler*	**3239**	**68**	**3171**

sumption must be closely adjusted. An embedded operating system must be able to save resources usage. In this context, this metric evaluates the impact of the One-Shot implementation in the final footprint of the system (OS plus application) compared to the traditional implementation with periodic time management.

2. *Threads processing time*: the thread processing time is the amount of time between the thread start time and its finish time. The processing time of a thread may be increased if other thread, with higher priority, is triggered during the thread execution, so the running thread is preempted, and the higher priority thread gets the processor. Therefore, in periodic time management, the running thread is also interfered by the time interrupt handler, as the interrupt handlers have higher priority in the system. In non-periodic time management, the interrupt handler executes only when a thread is ready to execute.

3. *Context switches number*: a context switch happens when a thread that is running is changed for other thread, is possible that the running thread has not finished yet. It is important that when the previous context is restored, the thread resumes by the same execution point it was when preempted.

4. *Interrupt handler execution times*: this characteristic is important to compare how many times the interrupt handler is executed in each implementation. This is an interesting comparison parameter that shows how many times the periodic timer implementation calls the interrupt handler and how many times the One-Shot does, together with the response time, we can examine the trade-off of having a light interrupt handler called many times (periodic timer implementation) VS an interrupt handler that does some computation but is called less times (One-Shot timer).

Table 1 shows the memory footprint of the two versions of EPOS, one with periodic time management and other with One-Shot time management. The One-Shot time manager was 504 bytes larger than the periodic time manager. This is expected, as long as the One-Shot time manager uses two different hardware timers, one 8 bits timer to the scheduler and one 16 bits timer to control the Alarm queue, thus two hardware mediators [8] were need.

The following tests were done in three scenarios. These scenarios are described bellow, showing the time each philosopher spend thinking and then eating in each execution.

- Scenario 1: Thinks for 1000 ms and eats for 5000 ms
- Scenario 2: Thinks for 100 ms and eats for 500 ms
- Scenario 3: Thinks for 25 ms and eats for 125 ms

These Scenarios were tested to analyze the impact of the One-Shot time manager in applications that spend most of their time waiting timer events, as expressed in the following tables and graphs.

Table 2 shows the number of executions of each interrupt handler. Obviously, in the periodic time management, the timer interrupt handler executes every *Tick* ($\sim$ 1,38 ms), even if no timer event exists. Differently, the interrupt timer handler of the One-Shot time manager, only will be executed when there is an event in the queue to be fired, this results in approximately 25905 less executions (Scenario 1). In each *tick*, the periodic time handler calls the *master_handler* (scheduler), this is the way found to give the

Table 3. Number of Context Switches of the threads (all summed) in the three Scenarios

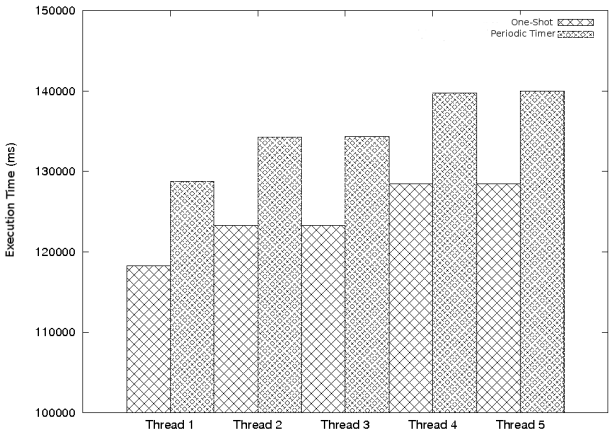
Scenario	*Periodic*	*Single-shot*	*Difference*
Scenario 1	**616**	**344**	**272**
Scenario 2	**615**	**360**	**255**
Scenario 3	**617**	**317**	**300**

scheduler time event the highest priority in the time events, so it runs even if other time event is triggered in the same *tick*. In the One-Shot implementation, the *master_handler* will only be triggered when the Scheduler *QUANTUM* is met.

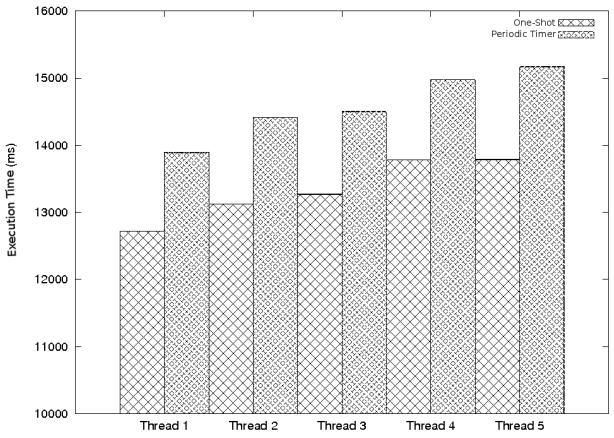
Table 3 shows the number of context switches in both versions of EPOS running the test application. As expected the One-Shot version had up to three hundred less context switches (Scenario 3), this is in someway an effect of less interrupt handler calls, that reduces the effective *QUANTUM* of the *Thread*. Reducing the operating system *overhead*, the time *QUANTUM* to each thread execution is better used, resulting in these preemptions.

Figure 3 shows the processing time of the *Threads* (Philosophers) in the three scenarios. Scenarios 1 and 2 show that the processing time of the threads are smaller in the One-Shot implementation. Scenario 3, where the thinking and eating time of each philosopher (thread) is smaller, is showed that the periodic time manager implementation (*ticks* based) causes less impact in the threads processing time. This happens because the tick size processed by the operating system is quite close to the delay intervals requested by each philosopher (thread), resulting in almost an event triggered in each tick. In conclusion, the processing time would be very similar in both implementations for applications with very small delays times (less than 10000 μs).

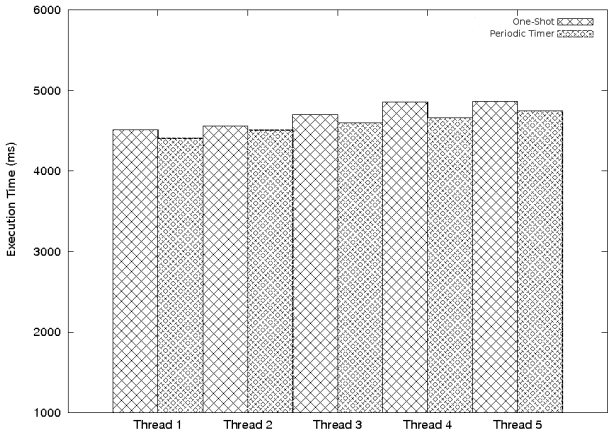
Figure 4 shows a new test series executed in the One-Shot and the periodic implementations varying the delay time (eat and think time) of each thread (philosopher). This figures shows that when the application spends most of the execution time waiting time events, the overhead of the periodic time manager is bigger, resulting in bigger total execution time. in the other hand, when the total waiting time is small, there is little difference in performance of both implementations, resulting in execution times close to each other.



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Figure 3. Processing time of the threads in the three scenarios

5 Conclusions and Future Works

This paper presented a comparison between One-Shot timers and Periodic timers in EPOS time management. Although the one-shot approach is not the most efficient be-

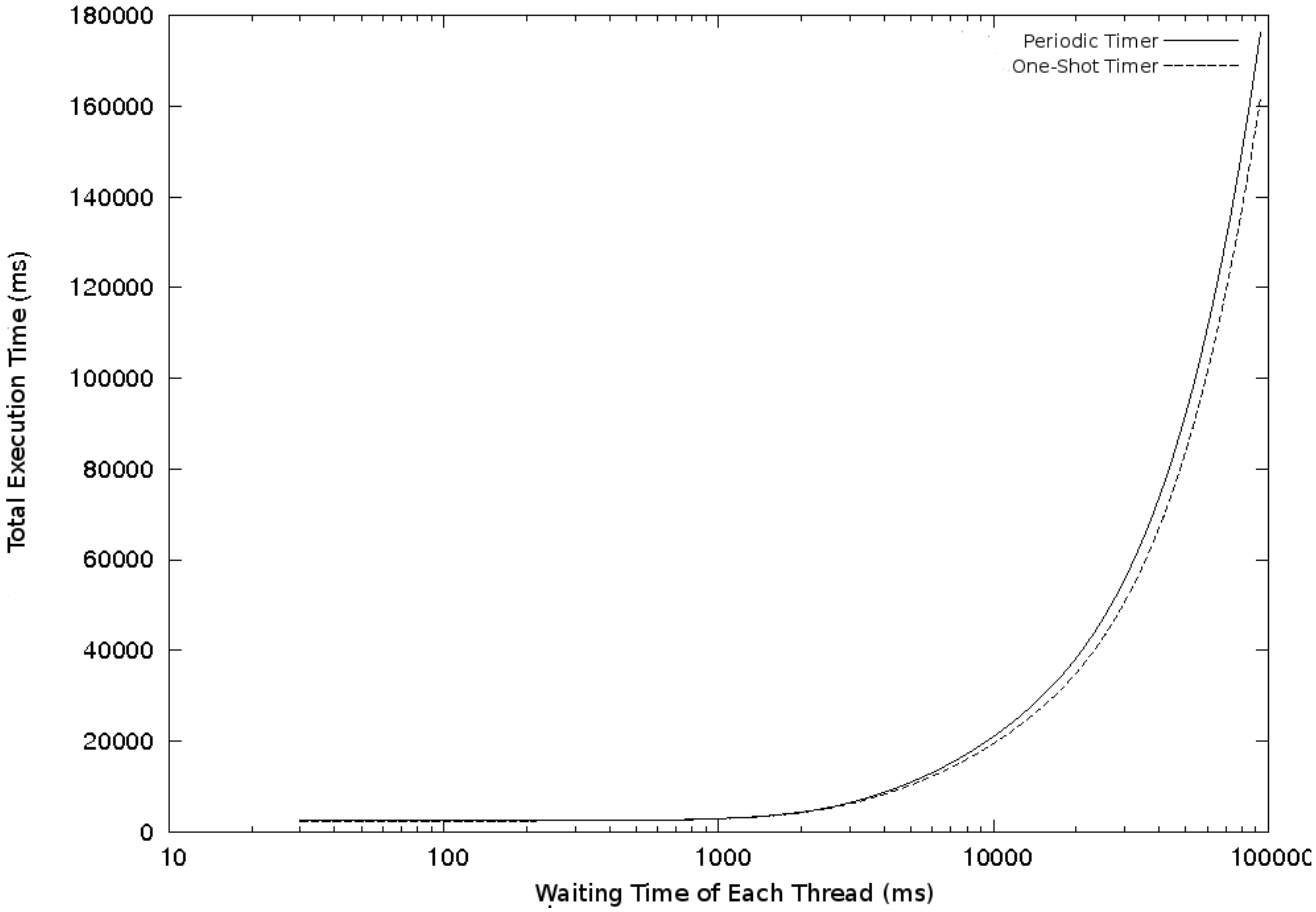


Figure 4. Total processing time varying the wait time

tween non-periodic time management algorithms, this implementation has allowed a first view of positive and negative impacts of non-periodic timers class in EPOS.

Four metrics were evaluated in order to measure the impacts of EPOS periodic and non-periodic time management implementations. The number of context switches and the number of interrupt handler executions obtained significant improvements in the system performance. And as expected, the memory consumption of non-periodic implementation was higher, due to use of two timers. The tasks computation time suffer less impact in the One-Shot time management when the application spends most of the time waiting for timer events.

Despite the improve of system performance in the medium case, there are situations where the non-periodic approach can be a bad alternative, for example, when the one-shot timer interrupt handler frequency tends to the frequency of the *ticks* number in the periodic timer. Therefore, the number of handler calls is similar.

The implementation of other more efficient non-periodic time management algorithms, like *smart timers*, is a future work.

References

[1] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, 2000.

[2] J.-M. Farines, J. da Silva Fraga, and R. S. de Oliveira. *Sistemas de Tempo Real*. Escola de Computação: IME-USP, S ao Paulo, SP, 2000.

[3] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[4] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[5] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity os. In *OSDI*, 2002.

[6] P. Kohout, B. Ganesh, and B. Jacob. Hardware support for real-time operating systems. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 45–51, New York, NY, USA, 2003. ACM.

[7] H. Marcondes, A. Hoeller, L. Wanner, and A. Frohlich. Operating systems portability: 8 bits and beyond. *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 124–130, 20-22 Sept. 2006.

[8] F. V. Polpeta and A. A. Fröhlich. Hardware mediators: A portability artifact for component-based systems. In L. T. Yang, M. Guo, G. R. Gao, and N. K. Jha, editors, *EUC*, volume 3207 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 2004.

[9] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[10] D. Tsafrir, Y. Etsion, and D. G. Feitelson. General purpose timing: the failure of periodic timers. Technical Report 2005-6, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, February 2005.