ISBN 978-987-4417-90-9

Administración del Tiempo Ocioso Mediante Slack Stealing en FreeRTOS

Francisco E. Páez[1], José M. Urriza[1], and Javier D. Orozco[2,3]

[1] Depto. de Informática, Facultad de Ingeniería, Universidad Nacional de la Patagonia San Juan Bosco, Sede Puerto Madryn, Argentina
[2] Depto. de Ingeniería Eléctrica y Computadoras, Universidad Nacional del Sur, Bahía Blanca, Argentina
[3] CONICET
fpaez@unpata.edu.ar, josemurriza@gmail.com

Resumen La planificación eficiente de conjuntos de tareas con requerimientos heterogéneos, sin perder la predictibilidad de ejecución de las tareas de tiempo real, es un área que esta tomando relevancia en la actualidad. Para lograr este objetivo, se requiere una correcta administración del tiempo ocioso disponible. En este trabajo se presenta una implementación que permite planificar estos conjuntos en FreeRTOS utilizando Slack Stealing.

Palabras claves: RTS · SS · RM · Planificación · RTOS

1. Introducción

En la actualidad existe una creciente necesidad de integrar, en un mismo sistema embebido, tareas con requerimientos de tiempo real *críticos* (para las que se debe garantizar el cumplimiento de sus restricciones temporales) junto con tareas sin requerimientos temporales estrictos, que deben cumplir con algún tipo de *calidad de servicio* (atención prioritaria, robustez, tolerancia a los fallos, etc.). El sistema debe poder planificar estos dos conjuntos de manera eficiente, brindando una calidad de servicio aceptable sin comprometer los requerimientos de tiempo real. Esto requiere utilizar el tiempo ocioso que dejan las tareas críticas para planificar el resto de las tareas. Diversos métodos han sido propuestos a tal fin y la planificación de estos conjuntos heterogéneos es una importante área de investigación [1].

Este trabajo presenta una implementación de la técnica de *Slack Stealing* (*SS*) para administrar el tiempo ocioso en el *Sistema Operativo de Tiempo Real* (*SOTR*) FreeRTOS[4]. Este es un *SOTR* de código abierto para dispositivos embebidos, con soporte para *Sistema de Tiempo Real* (*STR*) duros. Está desarrollado en lenguaje C, es de pequeño tamaño, modular y con bajos requerimientos de recursos. A la fecha, soporta más de 33 arquitecturas y es patrocinado por Amazon para su uso en sistemas IoT (*Internet of Things*).

[4] https://www.freertos.org

El desarrollo presentado se basa en [2], introduciendo mejoras en su diseño y aprovechando nuevas funcionalidades provistar por FreeRTOS en sus versiones más recientes.

A continuación se presenta una breve introducción a los *STR* y a la técnica de *SS*. En la sección 2 se detalla el diseño e implementación. La sección 3 presenta las pruebas realizadas. Las conclusiones y trabajos futuros se discuten en la sección 4.

1.1. Sistemas de Tiempo Real y Slack Stealing

En un *STR* los resultados, además de ser correctos aritmética y lógicamente, deben producirse antes de un determinado tiempo, denominado vencimiento [3]. Si no se admite la pérdida de ningún vencimiento, el *STR* es *duro* o *crítico* y si tolera la pérdida de algunos se lo denomina *blando*. Si existe una cota máxima de pérdidas se dice que es de tipo *firme*.

Como en un *STR crítico* la pérdida de un vencimiento puede tener consecuencias graves (pérdida de vidas, daños materiales, al medio ambiente, etc.), durante su diseño se garantiza que cada tarea cumpla con su vencimiento, mediante *tests de planificabilidad*. Un *STR* que cumple con estos tests se denomina *planificable*. Las primeras contribuciones al respecto fueron realizadas en [4], donde se demostró que cuando todas las tareas solicitan ejecución simultánea ocurre el *peor instante de carga* de un sistema monorecurso (*instante crítico*). Si el sistema es planificable en este instante, lo es en cualquier otro.

El conjunto de reglas que determina cuál tarea ejecutar en un instante dado, se denomina *algoritmo de planificación*, que puede ser estático o dinámico [5]. En los algoritmos dinámicos la prioridad de cada tarea puede modificarse en tiempo de ejecución (*prioridades dinámicas*) o permanecer invariante (*prioridades fijas*). Los algoritmos de planificación dinámicos por prioridades fijas más utilizados son *Rate Monotonic* (*RM*) [4] y *Deadline Monotonic* (*DM*) [6].

En un *STR heterogéneo*, el conjunto de tareas del sistema se divide en dos conjuntos bien definidos:

- *Tareas de Tiempo Real* (*TTRs*): tareas periódicas con requerimientos de tiempo real duro.
- *Tareas de No-Tiempo Real* (*TNTRs*): tareas del sistema que no cuentan con requerimientos de tiempo real estrictos, pero sí deben cumplir con algún otro requerimiento de calidad de servicio.

En sistemas no-saturados, los métodos de *SS* permiten en un instante dado identificar y adelantar parte del tiempo ocioso [7], al que se denomina *Slack Disponible* (*SD*) *del sistema en el instante t* ($SD(t)$). Este tiempo se puede aprovechar para ejecutar las *TNTRs*, retrasando la ejecución de las *TTRs*, sin comprometer su planificabilidad. Esta técnica permite un mejor aprovechamiento del tiempo ocioso que el uso de servidores [8], aunque su complejidad de implementación es mayor.

Varias implementaciones de *SS* han sido propuestas en [9]-[14], las cuales realizan el cálculo del $SD(t)$ en tiempo de ejecución o en tiempo de inicialización, y

de manera exacta o aproximada. Por otro lado, existen trabajos previos de implementación de estos métodos en un *SOTR*. En [15] se implementa una variante del algoritmo de *SS* aproximado [13] en MaRTE OS[5] [16]. En [17] se presenta una implementación del método de cálculo exacto [14] también sobre MaRTE OS. Una implementación de un método de cálculo aproximado en LejosRT se desarrolla en [18].

2. Implementación del Soporte de SS

Se buscó reducir las modificaciones necesarias al núcleo de FreeRTOS en [2], moviendo la implementación de todas las funciones que no requieran utilizar de manera directa variables internas del núcleo a una librería externa.

A partir de la versión 10, FreeRTOS provee una macro[6] para agregar funcionalidad definida por el usuario en el núcleo. Si esta macro es definida con el valor 1, entonces un archivo de nombre `freertos_task_c_additions.h` es incluido al compilar el *kernel*. Las funciones auxiliares que requieren acceder a estructuras de datos internas del núcleo son implementadas de esta manera. Las versiones modificadas de algunas funciones del núcleo también se incluyen en este archivo y se utiliza la opción `wrap` del compilador GCC para reemplazar las originales al enlazar (*linking*) la aplicación con el *kernel*.

La figura 1 presenta como se relacionan los nuevos componentes. El archivo fuente `main.c` contiene el código de la aplicación (creación de las tareas y la lógica de cada una), que invoca tanto funciones de FreeRTOS (archivo `tasks.c`) como de la librería de *SS* (archivos `slack.c` y `slack.h`). Las funciones del núcleo modificadas se encuentran en el archivo `freertos_tasks_c_additions.h`.

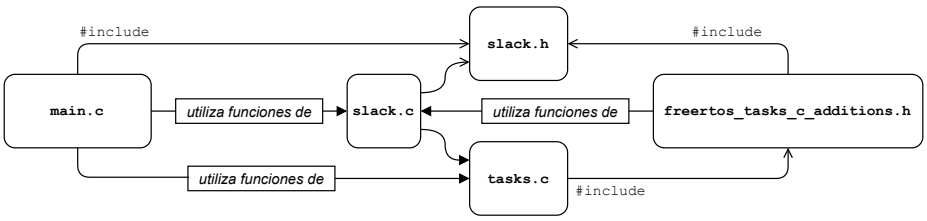

Figura 1. Esquema de la implementación en FreeRTOS.

Notar que la librería de *SS* y las modificaciones al núcleo son independientes del *hardware*. Por lo tanto, el soporte de *SS* puede utilizarse en cualquier plataforma para la que FreeRTOS este disponible.

A continuación se describen las modificaciones, nuevas funciones y estructuras agregadas al *kernel*.

[5] https://marte.unican.es

[6] `configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H` en el archivo `FreeRTOSConfig.h`

2.1. Atributos de las tareas

Se considera que las tareas que conforman el sistema se dividen en dos conjuntos:

- *TTRs*: modeladas a partir del modelo propuesto en [4]. Cada tarea genera genera una serie infinita de instancias, siendo $j_{k,i}$ la k-ésima instancia de la *TTR* i. Cada *TTR* se caracteriza mediante su periodo (T_i), su vencimiento relativo ($D_i \leq T_i$), su *peor caso de tiempo de ejecución* (C_i), su *peor caso de tiempo de respuesta* (R_i), el *SD* en el instante crítico ($SD_i(0)$), el *SD* en el instante actual ($SD_i(t_c)$), el tiempo ejecutado por la instancia actual ($c_i(t_c)$) y un contador de instanciaciones. Notar que $SD(t_c) = \text{mín}\, SD_i(t_c)$ con $i = 1, 2, \ldots, n$.
- *TNTRs*: Debido a su heterogeneidad no cuentan con un modelo específico asociado.

Los atributos del modelo de tareas son agrupados en una estructura de datos denominada `SsTCB_t`, que se agrega al *Task Control Block* (*TCB*) de la tarea mediante la funcionalidad *Thread Local Storage* (*TLS*) de FreeRTOS. Esta estructura y otras definiciones se encuentran en el archivo `slack.h`.

2.2. Colas de tareas adicionales

Se agregaron tres nuevas listas de tareas, implementadas mediante el tipo de datos `List_t` de FreeRTOS. Cada elemento de la lista contiene un atributo `xItemValue`, utilizado para ordenar la lista, y un puntero `pvOwner` que hace referencia al *TCB* de la tarea:

- `xSsTaskList`: Reúne todas las *TTR*, sin importar el estado, para simplificar el el cálculo del *SD*.
- `xDeadlineTaskList`: Registra los vencimientos absolutos de todas las *TTR*, y se mantiene ordenada por el vencimiento más próximo.
- `xSlackDelayedTaskList`: Contiene las *TNTR* suspendidas por falta de *SD*. Estas tareas continuarán su ejecución cuando $SD(t_c) > SD_{min}$.

2.3. Modificaciones al núcleo

El cálculo del *SD* se realiza antes del inicio del planificador de FreeRTOS y al finalizar la ejecución de cada instancia de una *TTR*. Además, los contadores de *SD* deben ser actualizados en cada *tick* de reloj. Para lograr esto, se modificó la implementación de las siguientes funciones del núcleo de FreeRTOS:

- `vTaskDelayUntil()`: Bloquea la ejecución de una tarea hasta un instante absoluto, permitiendo implementar tareas periódicas. Se modificó la función para que calcule el *SD* y actualice los atributos correspondientes de la tarea.

- `xTaskIncrementTick()`: Procesa la interrupción de reloj (*tick*). Desde esta función FreeRTOS realiza la activación de las tareas periódicas, controla *timeouts*, etc., y genera de ser necesario un cambio de contexto. Se agregó el control de vencimientos, la actualización de contadores de tiempo de ejecución ($c_i(t_c)$) y de SD. También se agregó la suspensión de las $TNTR$ cuando el SD sea menor al límite indicado por la aplicación ($SD(t_c) < SD_{min}$).

Desde `vTaskDelayUntil()` se calcula el SD de la tarea invocando la función `vTaskCalculateSlack()` (implementada en el `slack.c`). Como la tarea ya consumió parte del tiempo de computo del *time slice*, se calcula el SD en el instante $t_c + 1$ ($SD_i(t_c + 1)$). Luego, si $C_i - c_i(t_c) > 0$, agrega el tiempo ganado a los contadores $SD_k(t_c)$ de todas las tareas k de menor prioridad y actualiza $SD(t_c)$. Si $SD(t_c) > SD_{min}$ las $TNTR$ bloqueadas son puestas en la cola de tareas listas para ejecutar.

Cuando `xTaskIncrementTick()` mueve una TTR a la cola de tareas listas, si esta fue bloqueada por una invocación a `vTaskDelayUntil()` se incrementa su contador de instancias en uno.

Al procesar cada *tick*, `xTaskIncrementTick()` incrementa el contador $c_i(t_c)$ de la tarea en ejecución. Notar que la ejecución de una instancia en general emplea una fracción de su último *time slice*. Por lo tanto el valor de $c_i(t_c)$ es, en el peor caso, mayor en un *tick* que el tiempo efectivo de ejecución. Cuando se invoca la función `vTaskDelayUntil()`, se asigna cero a $c_i(t_c)$. Por defecto, en FreeRTOS la interrupción de reloj es cada 1 ms.

Además, `xTaskIncrementTick()` resta un *tick* a todos los contadores $SD_i(t_c)$ si se estuviera ejecutando una $TNTR$ o la tarea inactiva (*idle*). Si se estuviera ejecutando una TTR, resta los contadores de todas las TTR de mayor prioridad. Una vez que los contadores están actualizados, recalcula $SD(t_c)$. Luego, si $SD(t_c) < SD_{min}$ cualquier $TNTR$ que estuviera ejecutando o lista para ejecutar, es suspendida y puesta en la cola `xSlackDelayedTaskList`.

Para actualizar los contadores de SD se utilizan las siguientes funciones auxiliares (implementadas en `slack.c`):

- `vSlackDecrementTasksSlack(i, n)`. Resta n *ticks* a los contadores de SD de las TTR con prioridad mayor a i.
- `vSlackDecrementAllTasksSlack(n)`. Resta n *ticks* a los contadores de SD de *todas* las TTR.
- `vSlackGainSlack(i, n)`. Suma n *ticks* a los contadores de SD de las TTR con prioridad menor a i.
- `vSlackUpdateAvailableSlack()`. Actualiza $SD(t_c)$.

El cada *tick* de reloj se verifica si el vencimiento de la primer TTR en `xDeadlineTaskList` es mayor al tiempo actual. Por cada tarea que pierda su vencimiento se invoca la función `vApplicationDeadlineMissedHook()`. La función `vTaskDelayUntil()` actualiza el vencimiento absoluto de la tarea y reordena la lista de vencimientos.

Las siguientes funciones se agregan al núcleo, ya que requieren interactuar directamente con la cola de tareas listas para ejecutar:

- `vTaskSlackSuspend()`. Mueve todas las *TNTR* de la cola de tareas listas para ejecutar a la lista de tareas `xSlackDelayedTaskList` (ver sección 2.2).
- `vTaskSlackResume()`. Mueve todas las *TNTR* que se encuentren en la lista `xSlackDelayedTaskList` a la cola de tareas listas para ejecutar.

2.4. Interfaz para el desarrollador

Las siguientes funciones forman la interfaz para el desarrollador:

- `vSlackSystemSetup()`: inicializa las colas de tareas adicionales (sección 2.2). Debe invocarse antes de especificar los parámetros de las tareas.
- `vTaskSetParams()`: permite indicar los atributos adicionales de una tarea, tal como su tipo (*TTR* o *TNTR*), período, vencimiento, etc. Si es una *TTR*, la agrega a `xSsTaskList` y su vencimiento inicial (D_i) a `xDeadlineTaskList`.
- `vSlackSchedulerSetup()`: realiza la verificación de planificabilidad, el cálculo del peor caso de tiempo de respuesta de las tareas y el cálculo del *SD* en el instante crítico. Debe invocarse antes de iniciar el planificador de FreeRTOS.

Si $R_i \leq D_i$ para todas las *TTR* el *STR* es planificable. Caso contrario, se invoca la función `vApplicationNotSchedulable()`[7]. La evaluación de planificabilidad no tiene en cuenta el costo del cambio de contexto y se realiza cómo validación del modelo de tareas.

2.5. Planificación mediante Slack Stealing

El planificador de FreeRTOS emplea una política apropiativa *First In, First Out* (*FIFO*) con prioridades [19] y garantiza que siempre ejecutará la tarea de mayor prioridad lista para ejecutar. La cola de tareas listas se implementa como un arreglo de listas. Dadas M prioridades, la posición cero del arreglo contiene la cola de tareas de menor prioridad y la posición $M-1$ la de máxima. Las tareas dentro de cada lista se planifican mediante *FIFO* o *Round Robin* (*RR*) según la configuración.

El rango de prioridades $(M, M-n]$ se reserva para las *TNTR* y sólo estarán en la cola de tareas listas si $SD(t_c) > SD_{min}$. De esta manera, FreeRTOS les da preferencia hasta que las mismas finalicen, se bloqueen o agoten el *SD*. El rango de prioridades $(M-n, 1]$ es para las *TTRs*, con sólo una *TTR* por nivel. Las tareas con prioridad cero son ejecutadas cuando no existan *TTRs* o *TNTRs* listas para ejecutar y se excluyen del cálculo de *SD*. Notar que un $SD(t_c) < 1$ no es detectado. Por lo tanto, las *TNTR* pendientes de ejecución no son puestas inmediatamente en la cola de tareas listas y una *TTR* u otra tarea pueden ejecutarse hasta el siguiente *tick*.

La Figura 2 presenta un ejemplo de organización de la cola de tareas listas. Si $SD(t_c) = 0$ las *TNTRs* no estarían presentes.

[7] Que debe implementar el desarrollador.

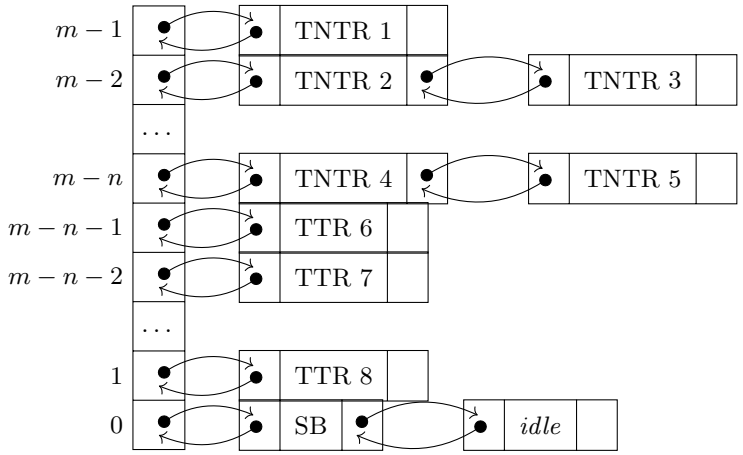

Figura 2. Cola de tareas listas organizada para para planificación mediante SS.

3. Resultados Experimentales

3.1. Ejemplo de Ejecución

Se presenta una traza de ejemplo sobre en una placa *mbed* LPC1768 y FreeRTOS v10.1.3. El sistema consta de 4 *TTR*, $\{(1,3,3),(1,4,4),(1,6,6),(1,12,12)\}$ con tiempos expresados en segundos, y dos *TNTR*, TA1 (de máxima prioridad) y TA2, con un tiempo de ejecución aleatorio no mayor a los 2000 ms.

La figura 3 muestra la traza generada por Tracealyzer[8] v3.1.2. Se observa cómo la ejecución de la primer instancia de TA2 es desalojada por TA1 y luego al agotarse el *SD* (en este caso $SD_{min} = 1$). FreeRTOS ejecuta entonces la *TTR* T1, que como ejecuta un tiempo menor a su peor caso de ejecución genera *SD*. Esto permite ejecutar nuevamente a TA2. Se puede ver también cómo en el instante $t = 12$ todas las *TTR* son desplazadas por la ejecución de otra instancia de TA2. En la salida a través del puerto serial cada línea indica la tarea en ejecución, si inicia (`S`) o finaliza (`E`), el valor del reloj, el *SD* del sistema, el *SD* de las *TTR* T1 a T4 y el tiempo ejecutado (en *ticks*) por la tarea.

3.2. Costo del Cambio de Contexto

A continuación se evalúa el *Costo Computacional* (*CC*) temporal del cambio de contexto, al ejecutar `vTaskDelayUntil()`. Se contabilizó el número de ciclos de CPU para las primeras 30 ejecuciones de cada *TTR*. Se empleó FreeRTOS v10.3.1, sobre una placa *mbed* LPC1768, con el método de *SS* exacto [14]. La duración del *tick* se configuró en 1 ms y se empleó el mecanismo de cambio de contexto optimizado para Cortex-M3 de FreeRTOS. El número de ciclos de *CPU* se obtuvo con el contador *Clock Cycle Counter* (*CYCCNT*).

[8] https://percepio.com/tracealyzer/

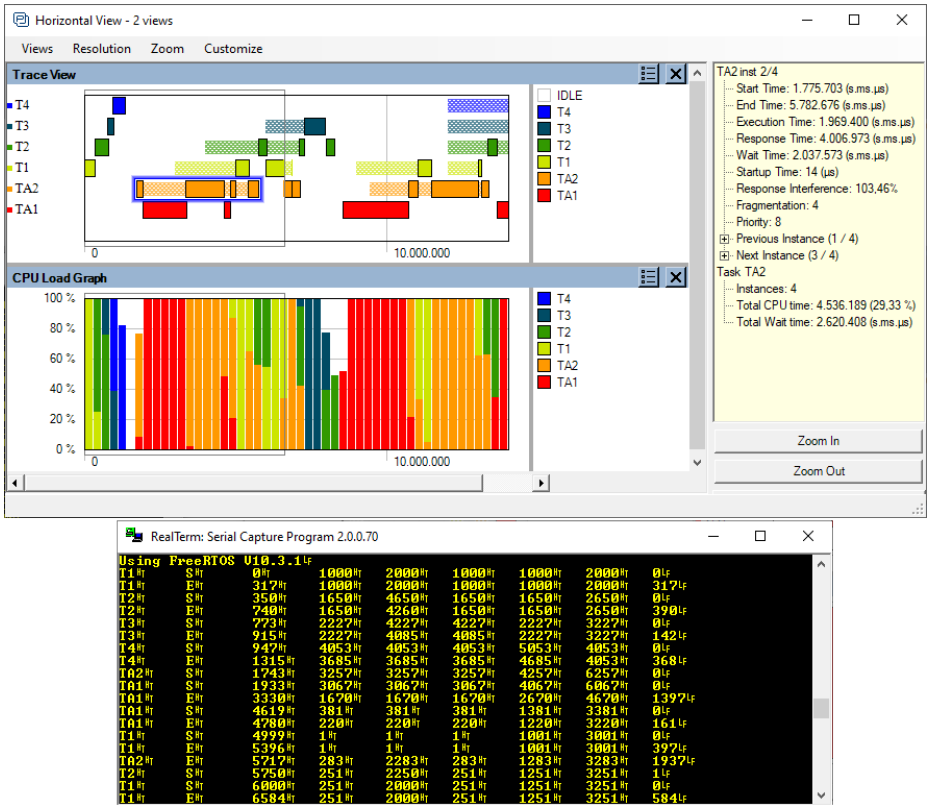

Figura 3. Traza de ejecución mediante Tracealyzer y salida serial.

Se generaron 1000 *STR* de 10 tareas, para cada *Factor de Utilización* (FU) del 10 % al 90 %, en intervalos de 10 %. Los parámetros T y C de las *TTR* se distribuyeron uniformemente, entre 25 y 1000 *ticks*. Se evaluó el *CC* temporal del cambio de contexto de FreeRTOS sin modificaciones y se comparo con las siguientes configuraciones:

- Sólo actualizando los contadores de *SD* de cada tarea con el valor calculado en el instante crítico, para determinar el costo introducido independientemente del método de *SS*.
- Realizando el cálculo del *SD* en tiempo de ejecución.

La figura 4 presenta los resultados. El costo del cambio de contexto sin el cálculo del *SD* se mantiene constante y dentro del mismo orden que FreeRTOS sin modificaciones. Por otro lado, al realizar el cálculo del *SD* el costo crece con el *FU* del sistema, de manera similar al presentado en [14]. Dado que la interrupción de reloj ocurre cada 1 ms, el *time slice* abarca $\approx$ 96000 ciclos de CPU. Por lo tanto el *CC* temporal del cambio de contexto es, en el peor caso promedio, $\approx$ 2,5 % del *time slice*.

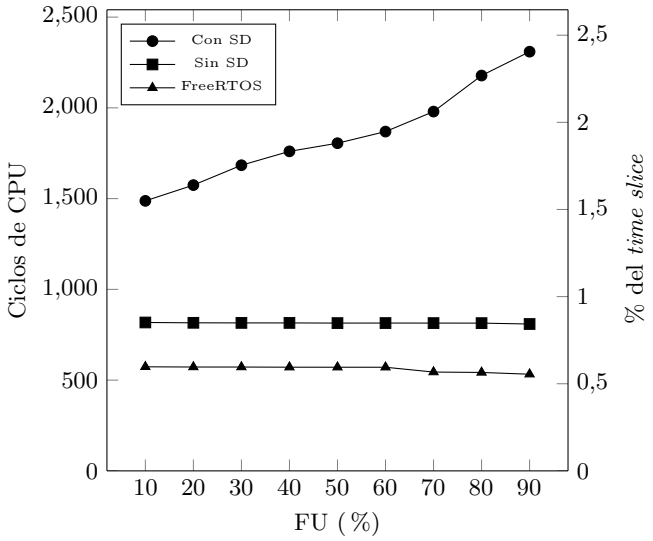

Figura 4. Costo promedio en ciclos de CPU del cambio de contexto al finalizar la instancia de una TTR.

4. Conclusiones

Una administración eficiente del tiempo ocioso puede facilitar el diseño e implementación de soluciones que satisfagan los requerimientos funcionales y temporales de sistemas con conjuntos heterogéneos de tareas. Aunque el uso de un *SOTR* facilita cumplir con los requerimientos temporales en mucho casos, no siempre posibilita planificar conjuntos heterogéneos de tareas de manera eficiente, dadas las pocas opciones de políticas de planificación que ofrecen. En este trabajo se presentó una implementación de administración del tiempo ocioso exacto, en un *SOTR* de amplio uso. El diseño propuesto favorece la modularidad y requiere de mínimas modificaciones al *SOTR*.

El código fuente esta disponible para su descarga en GitHub[9], con ejemplos de ejecución en las placas *mbed* LPC1768, FRDM-K64F y EDU-CIAA-NXP.

En trabajos futuros se evaluará el costo computacional real, tanto temporal como espacial, de diversos métodos de *SS*, y el uso del *SD* para cumplir requerimientos como tolerancia a fallas, ahorro de energía, etc.

Referencias

[1] A. Burns y R. I. Davis, «A Survey of Research into Mixed Criticality Systems», *ACM Comput. Surv.*, vol. 50, n.º 6, 82:1-82:37, 2017.

[9] `https://github.com/unpsjb-rtsg/slack-freertos`

[2] F. Páez, J. M. Urriza, R. Cayssials y J. D. Orozco, «Métodos de Slack Stealing en FreeRTOS», en *44 Jornadas Argentinas de Informática (JAIIO)*, JAIIO, ed., SADIO, 2015.

[3] J. A. Stankovic, «Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems», *Computer*, vol. 21, n.º 10, págs. 10-19, 1988.

[4] C. L. Liu y J. W. Layland, «Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment», *J. ACM*, vol. 20, n.º 1, págs. 46-61, 1973.

[5] A. Burns, «Scheduling hard real-time systems: a review», *Software Engineering Journal*, vol. 6, n.º 3, págs. 116-128, 1991.

[6] J. Y. Leung y J. Whitehead, «On the complexity of fixed-priority scheduling of periodic, real-time tasks», *Perform. Eval.*, vol. 2, n.º 4, págs. 237-250, 1982.

[7] S. Ramos-Thuel y J. P. Lehoczky, «On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems», en *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, December 1993*, 1993, págs. 160-171.

[8] T.-S. Tia, J. W.-S. Liu y M. Shankar, «Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-priority Preemptive Systems», *Real-Time Syst.*, vol. 10, n.º 1, págs. 23-43, 1996.

[9] R. I. Davis, K. Tindell y A. Burns, «Scheduling slack time in fixed priority preemptive systems», en *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, December 1993*, 1993, págs. 222-231.

[10] R. M. Santos, J. M. Urriza, J. Santos y J. Orozco, «New methods for redistributing slack time in real-time systems: applications and comparative evaluations», *Journal of Systems and Software*, vol. 69, n.º 1-2, págs. 115-128, 2004.

[11] R. C. José Manuel Urriza Javier D. Orozco, «Fast Slack Stealing methods for Embedded Real Time Systems», 2005.

[12] C. Lin y S. A. Brandt, «Improving Soft Real-Time Performance through Better Slack Reclaiming», en *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA*, IEEE Computer Society, 2005, págs. 410-421.

[13] R. I. Davis, *Approximate slack stealing algorithms for fixed priority pre-emptive systems*. University of York, Department of Computer Science, 1993.

[14] J. Urriza, F. Paez, R. Cayssials, J. Orozco y L. Schorb, «Low cost slack stealing method for RM/DM», *International Review on Computers and Software*, vol. 5, n.º 6, págs. 660-667, 2010.

[15] A. R. E. Minguet, «.Extensiones al Lenguaje Ada y a los Servicios POSIX para Planificación en Sistemas de Tiempo Real Estricto"», Tesis doct., Universidad Politécnica de Valencia, 2003.

[16] M. A. Rivas y M. G. Harbour, «MaRTE OS: An Ada Kernel for Real-Time Embedded Applications», en *Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference Leuven, Belgium, May 14-18, 2001, Proceedings*, D. Craeynest y A. Strohmeier, eds., vol. 2043, Springer, 2001, págs. 305-316.

[17] L. A. Díaz, F. E. Páez, J. M. Urriza, J. D. Orozco y R. Cayssials, «Implementación de un Método de Slack Stealing en el Kernel de MaRTE OS», en *Proceeding XLIII Jornadas Argentinas de Informática e Investigación Operativa (43 JAIIO) - III Argentine Symposium on Industrial Informatics (SII)*, 2014, págs. 13-24.

[18] S. Midonnet, D. Masson y R. Lassalle, «Slack-Time Computation for Temporal Robustness in Embedded Systems», *Embedded Systems Letters*, vol. 2, n.º 4, págs. 119-122, 2010.

[19] C. Svec, «FreeRTOS», en *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*, lulu.com, 2012.