Métodos de *Slack Stealing* en *FreeRTOS*

Francisco E. Páez [1,3], José M. Urriza [1], Ricardo Cayssials [2], 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@unp.edu.ar

Resumen. En la actualidad, los Sistemas Operativos de Tiempo Real necesitan ampliar sus funcionalidades, para dar soporte a la planificación mixta con requerimientos heterogéneos, sin perder la predictibilidad de ejecución del subsistema crítico. Este trabajo presenta un *framework* para la implementación de métodos de *Slack Stealing*, en el *kernel* de *FreeRTOS*.

Palabras claves: Sistemas de Tiempo Real, Sistemas Operativos de Tiempo Real, Slack Stealing, Planificación, Sistemas Críticos Mixtos, mbed

1 Introducción

Existe en la actualidad una creciente necesidad de flexibilizar los *Sistemas Operativos de Tiempo Real* (*SOTR*) mediante la incorporación de soporte para tareas no críticas. Consecuentemente, dentro del conjunto de tareas del *SOTR*, deben coexistir dos subconjuntos bien definidos. El primero es el subconjunto de *tareas de tiempo real críticas* (*TTR*), para las cuales es necesario garantizar el cumplimiento de sus restricciones temporales. El segundo subconjunto está compuesto por tareas que pueden poseer diversos requerimientos no críticos, a las cuales se definirá como *tareas de no tiempo real* (*TNTR*). El *SOTR* debe ser capaz de planificar estos dos subconjuntos con diferentes características y requerimientos entre sí.

En general, el subconjunto *TNTR* necesita de algún tipo de calidad de servicio (*QoS, Quality of Service*), que puede comprender atención prioritaria, robustez, tolerancia a los fallos, etc. Esto requiere de técnicas que permitan aprovechar el tiempo ocioso que dejan las *TTR*, para ser utilizado en la planificación del subconjunto de *TNTR*. Diversos métodos han sido propuestos para tal fin, y la planificación de estos conjuntos heterogéneos de tareas es una importante área de investigación.

En este trabajo se presenta un *framework* para la implementación de métodos de *Slack Stealing* (*SS*), en el *kernel* del *SOTR FreeRTOS*, que permiten aprovechar el tiempo ocioso del sistema. El *framework* facilita implementar distintas técnicas de *SS*, lo que posibilita realizar evaluaciones de factibilidad y comparativas de desempeño, entre los distintos métodos.

A continuación se presenta una introducción a los *STR,* métodos de *SS* y el porqué de la elección de *FreeRTOS*. En la sección 2 se presenta el modelo de tareas utilizado. La sección 3 lista las modificaciones realizadas para el soporte de *SS*, y la sección 4 describe su ejecución. En la sección 5 se presentan las evaluaciones realizadas. Las conclusiones y trabajos futuros se discuten en la sección 6.

1.1 Introducción a los Sistemas de Tiempo Real y a los Métodos *Slack Stealing*

En un *Sistema de Tiempo Real* (*STR*) los resultados, además de ser correctos aritmética y lógicamente, deben producirse antes de un determinado tiempo, que se denomina *vencimiento* ([1]). Si el *STR* no puede admitir la perdida de ningún vencimiento, se dice que es *duro* o *crítico*. Si se tolera la pérdida de algunos vencimientos, se lo clasifica como *blando*. Finalmente, si acepta una cierta cantidad acotada de pérdidas de vencimientos se lo denomina *firme*.

En un *STR crítico* la pérdida de un vencimiento puede tener consecuencias graves, sobre el sistema y su entorno (pérdida de vidas, daños materiales, al medio ambiente, económicos, etc.). Por esta razón, durante la etapa de diseño se garantiza que cada tarea cumpla con su vencimiento, mediante procedimientos denominados *tests de planificabilidad*. Un *STR* que cumple con estos *tests* se denomina *planificable*, y se garantiza que cumple con todas sus restricciones temporales. Las primeras contribuciones al respecto han sido realizadas en [2], donde también se probó que el *peor instante de carga* de un sistema monorecurso (*instante crítico*), ocurre cuando todas las tareas solicitan ejecución simultánea. Si este instante es planificable, también lo será cualquier otro instante.

El conjunto de reglas que determina que tarea será ejecutada en un instante dado, se denomina *algoritmo de planificación*. Éstos pueden ser *estáticos* o *dinámicos* ([3]). Los algoritmos *dinámicos* asignan una prioridad a cada tarea, que puede modificarse en tiempo de ejecución (prioridades dinámicas) o permanecer invariante una vez asignada (prioridades fijas). Los algoritmos de planificación dinámicos por prioridades fijas más utilizados son *Rate Monotonic* (*RM*) y *Deadline Monotonic* (*DM*), descriptos en [2, 4]. En este trabajo se utilizarán estos algoritmos de planificación para la ejecución de las *TTR*.

Los métodos de *SS* permiten para un instante dado, en sistemas no-saturados, identificar y adelantar parte del tiempo ocioso para la ejecución de las *TNTR* ([5, 6]). A este tiempo ocioso se lo denomina *slack disponible del sistema en el instante t* (*SD*(*t*)), y es el tiempo que a futuro las *TTR* no utilizaran, si cumplieran su peor caso de tiempo de ejecución. Al utilizar el *SD*(*t*), se retrasa la ejecución del subconjunto de *TTR*, pudiendo éste llegar al límite de su planificabilidad, pero sin comprometer la misma.

El *slack disponible* (*SD*) puede ser utilizado por ejemplo para mejorar la *QoS* del subconjunto *TNTR*. Caso contrario, se tendría que esperar a los intervalos ociosos para ejecutar las *TNTR*, método que se denomina *Servidor por Background*. Sin embargo, es posible que las *TNTR* requieran una cierta *QoS*, y esperar estos intervalos para su ejecución no permitiría satisfacer estos requerimientos.

Varias implementaciones de *SS* han sido propuestas en [7, 8, 9, 10, 11], 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 [12] se implementa una variante del algoritmo aproximado *DASS* ([11]), sobre *MaRTE OS*. En [13] se presenta una implementación del método de cálculo exacto [9], con menor *Costo Computacional* (*CC*) que *DASS*, también sobre *MaRTE OS*. En [14, 15, 16] se implementa un método de cálculo aproximado, sobre *RTSJ* (*Real-Time Specification for Java*).

1.2 Acerca de *FreeRTOS*

Existen en la actualidad múltiples *SOTR*, como *VxWorks*, *QNX*, *eCos*, *µs-OS*, *Windows CE*, *RT-Linux*, etc. Se escogió *FreeRTOS* por su soporte para *STR duros*, código abierto, tamaño reducido y disponibilidad de documentación.

FreeRTOS está desarrollado en lenguaje C, con mínimas secciones dependientes del *hardware* escritas en ensamblador. Sus principales características son el pequeño tamaño del *kernel*, y bajo requerimiento de recursos de memoria y computo. Dependiendo del número de tareas, puede llegar a necesitar solo 10KB de ROM, y una cantidad similar de memoria RAM. Se encuentra portado, a la fecha, a más de 33 arquitecturas.

Cada tarea en *FreeRTOS* cuenta con su propia pila, de tamaño configurable, y si la plataforma la provee, puede emplear una unidad *MPU* (*Memory Protection Unit*). Dispone de varios perfiles de administración de memoria, desde un esquema estático, a esquemas dinámicos que permiten operaciones *malloc* y *free*.

El planificador de *FreeRTOS* emplea una política apropiativa *FIFO* con prioridades. Garantiza que para cualquier instante dado, ejecutará la tarea de mayor prioridad en la cola de tareas listas. Si existen múltiples tareas listas para ejecutar, de idéntica prioridad, se aplica una política *Round Robin* (*RR*) entre las mismas, ejecutando cada una durante un *tick* de reloj.

2 Modelo de Tareas

Para el subconjunto de *TTR*, se utiliza un modelo basado en el propuesto en [2]. Se considera que este subconjunto está compuesto por n tareas periódicas ($\tau_1, \tau_2, \ldots, \tau_n$). Cada τ_i genera una serie infinita de instancias, siendo $j_{k,i}$ la k-ésima instancia de τ_i. El valor actual del reloj se indica como t_c. Una τ_i es caracterizada por su periodo (T_i), vencimiento relativo (D_i) y peor caso de tiempo de ejecución (C_i). Además, se registra su peor caso de tiempo de respuesta ($WCRT_i$, requerido por algunos métodos de *SS*), su *SD* en t_c ($SD_i(t_c)$), en el *instante crítico* ($SD_i(0)$), el tiempo ejecutado por la instancia actual de la tarea ($c_i(t_c)$) y un contador de instanciaciones (J_i). Se hace notar que $SD(t_c)=min\ SD_i(t_c)$ con $i = 1,2,\ldots,n$.

El subconjunto de *TNTR*, debido a su heterogeneidad, no cuenta con un modelo especifico asociado. Sin embargo, pueden contar con algún requerimiento de *QoS* para su atención. De no ser así, se consideran como tareas en *background* ([17]).

3 Implementación del Soporte para Métodos de *SS*

A continuación se describen las modificaciones, nuevas funciones y estructuras agregadas al *kernel* para implementar el *framework* que da soporte al método *SS*.

3.1 Modificaciones al *API* del *Kernel*

Las siguientes funciones del *API* (*Application Programming Interface*) de *FreeRTOS* fueron modificadas:

- *vTaskStartScheduler*(): Inicia la ejecución del planificador. Se agregó el cálculo del *WCRT* y del $SD_i(0)$ para todas las *TTR*.
- *vTaskDelayUntil*(): Bloquea la ejecución de una tarea hasta un instante absoluto. Es utilizada para implementar la periodicidad de las *TTR*. Se agregó la invocación del método de cálculo de *SS* y la actualización de atributos del modelo de tareas.
- *xTaskResumeAll*(): Reinicia la ejecución del planificador. Se modificó para mover a la cola de tareas listas las *TNTR* suspendidas, si $SD(t_c) > 0$.

Se agregó la función *vTaskSetParams*() al *API*, para especificar los nuevos atributos del modelo para una *TTR*, tales como el período, vencimiento, etc.

Las siguientes funciones, de uso interno del *kernel*, también fueron modificadas:

- *xTaskGenericCreateTask*(): Crea una nueva *TTR*. Se modificó para agregar la tarea creada a una nueva lista global, empleada por el método de *SS*. Las funciones *xTaskCreate*() y *xTaskCreateRestricted*() son *macros* que invocan a esta función.
- *prvInitialiseTCBVariables*(): Esta función inicializa el *TCB* de una tarea. Se agregó la inicialización de los nuevos atributos del modelo de tareas.
- *prvInitialiseTaskLists*(): Esta función inicializa las listas de tareas de *FreeRTOS* (tareas listas para ejecutar, bloqueadas y suspendidas). Se agregó la inicialización de las nuevas listas de tareas agregadas al *kernel*.
- *xTaskIncrementTick*(): Procesa una interrupción de reloj, incrementado el valor del *tick* (t_c en el modelo). Mueve las tareas que se desbloqueen en t_c a la cola de tareas listas, y genera de ser necesario, un cambio de contexto. Se agregó un control de vencimientos, la actualización de los contadores $SD_i(t_c)$ y $c_i(t_c)$, y la suspensión de las *TNTR*, en caso de que $SD(t_c) = 0$.

3.2 Atributos agregados al *TCB* y nuevas listas de *TTR*

Los atributos del modelo de tareas son implementados mediante una estructura de datos denominada *xSsTCB*. Se agrega al *TCB* de *FreeRTOS* un puntero a una

instancia de la misma. En la función *prvInitialiseTCBVariables*() se reserva la memoria requerida para *xSsTCB* y se inicializan sus miembros.

En caso de que un método de *SS* requiera parámetros adicionales, un atributo *pvSsAux* permite hacer referencia a una estructura de datos adicional para tal fin.

Por defecto, *FreeRTOS* administra tres listas de tareas: listas para ejecutar (*pxReadyTaskLists*), bloqueadas (*xDelayedTaskList*) y suspendidas (*xSuspendedTaskList*). Para simplificar la implementación de los métodos de *SS*, la administración de las tareas *TNTR* y el control de vencimientos de las *TTR*, se agregaron tres nuevas listas de tareas al *kernel*:

- *xSsTaskList*. Esta lista reúne todas las *TTR*, sin importar su estado. Simplifica el acceso del método *SS* a todas las *TTR*.
- *xDeadlineTaskList*. Registra los vencimientos absolutos de todas las *TTR*, y se mantiene ordenada por el vencimiento más próximo.
- *xSlackDelayedTaskList*. Registra las *TNTR* suspendidas por falta de *SD*. Estas tareas continuarán su ejecución cuando $SD(t_c) > 0$.

Se agregó en la función *prvInitialiseTaskLists*(), la inicialización de las listas. Se modificó la función *xTaskGenericCreate*(), para agregar la *TTR* creada a *xSsTaskList*, y su vencimiento inicial (D_i) a *xDeadlineTaskList*. Además, se agregó al *TCB* las referencias a las nuevas listas.

Estas listas son implementadas con el tipo *List_t* de *FreeRTOS*. Este define listas doblemente enlazadas, compuestas por elementos de tipo *xLIST_ITEM*. Este tipo contiene un atributo *xItemValue*, que es utilizado para ordenar la lista, y un puntero *pvOwner* para referenciar al elemento de la lista (Figura 1).

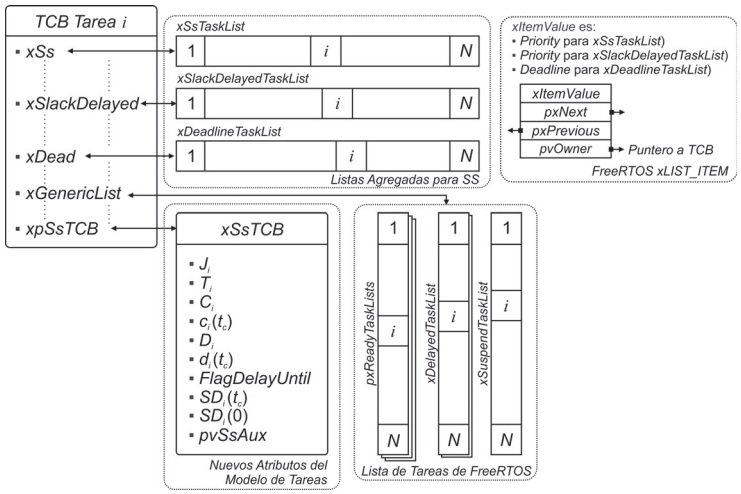

Fig. 1. Modificaciones al *TCB* y nuevas listas de tareas agregadas al *Kernel* de *FreeRTOS.*

3.3 Administración de *TTR*

Las siguientes modificaciones permiten mantener actualizados los parámetros de las *TTR*, y permiten realizar verificaciones sobre las restricciones temporales.

- **Contador de Instancias.** La instancia de una *TTR* finaliza al invocar la función *vTaskDelayUntil*(), que bloquea su ejecución hasta el inicio del próximo periodo. Luego, cuando una *TTR* es transferida a la cola de tareas listas, si hubiera estado bloqueada por una invocación previa a *vTaskDelayUntil*(), se incrementa su contador de instancias (J_i). Esta modificación se agregó a la función *xTaskIncrementTick*().
- **Cálculo del *WCRT*.** El *WCRT* puede ser especificado con la función *vTaskSetParams*(), o calculado al iniciar el planificador, en *vTaskStartScheduler*(). Si $WCRT_i \leq D_i$ para toda τ_i, se considera que el *STR* es planificable. Caso contrario, se invoca la función *vApplicationNotSchedulable*(), cuya implementación depende de la aplicación, y por lo tanto se delega al desarrollador. Esta evaluación de planificabilidad no tiene en cuenta el costo del *cambio de contexto* (*CS*) del *SOTR*, y realiza sólo una validación del modelo de tareas.
- **Registro de Uso de Tiempo de *CPU*.** El tiempo de uso de *CPU* para la instancia actual de τ_i, $c_i(t_c)$, se mide en *ticks* de reloj, siendo ésta la granularidad mínima. Al procesar cada *tick* de reloj en la función *xTaskIncrementTick*(), se incrementa el contador $c_i(t_c)$ de la tarea en ejecución. Cuando se invoca la función *vTaskDelayUntil*(), $c_i(t_c)$ es puesto en cero. Se hace notar que la ejecución de una instancia puede no durar un número entero de *ticks*, empleando sólo una fracción del último. Luego el valor de $c_i(t_c)$, en el peor de los casos, es mayor en un *tick* con respecto a su tiempo efectivo de ejecución.
- **Control de Vencimientos.** Se realiza en cada *tick* de reloj, desde la función *xTaskIncrementTick*(). Para comprobar si alguna *TTR* perdió su vencimiento, se verifica si el primer ítem de *xDeadlineTaskList* es mayor a t_c. Por cada tarea que pierda su vencimiento, se invoca la función *vApplicationDeadlineMissedHook*(). Esta función realiza las acciones requeridas según el diseño del *STR* y su implementación se delega al desarrollador. Al invocar la función *vTaskDelayUntil*(), el valor *xItemValue* del ítem correspondiente a la *TTR*, se actualiza con el vencimiento absoluto de su próxima instancia.

La estructura general del *framework* se presenta en la Figura 2, donde se indica las relaciones entre los distintos componentes del mismo.

4 Método de *Slack Stealing*

4.1 Ejecución del método de *Slack Stealing*

Para el cálculo de $SD_i(t_c)$ se invoca la función *prvTaskCalculateSlack*(), con τ_i y t_c como parámetros. Esta función implementa el cálculo de *SD*, del método de *SS* utilizado, el cual se mide en *ticks*. El cálculo del *SD* se realiza al inicio de la ejecución del planificador de *FreeRTOS* (*instante crítico*) y en la finalización de cada instancia

de una *TTR*. Además, los contadores $SD_i(t_c)$ de las *TTR* son actualizados al procesar cada *tick* de reloj.

Para el cálculo del $SD(0)$, se agregó a la función *vTaskStartScheduler*(), la invocación a *prvTaskCalculateSlack*(). Esta se realiza antes de ejecutar la función *xPortStartScheduler*(), que configura la máscara de interrupciones, la *ISR*, la frecuencia de *tick* de reloj y pasa el control a la *TTR* de mayor prioridad.

La instancia de una τ_i finaliza con la invocación de *vTaskDelayUntil*(). A esta función, se agregó la invocación a *prvTaskCalculateSlack*(), antes de que la tarea se bloquee. Como la τ_i ya consumió parte del tiempo de computo del *tick*, se calcula el *SD* en el instante $t_c + 1$ ($SD_i(t_c + 1)$). Luego, si $C_i - c_i(t_c) > 0$, se agrega el *tiempo ganado* a los contadores $SD_k(t_c)$ de todas las τ_k de *menor prioridad* ($i < k$).

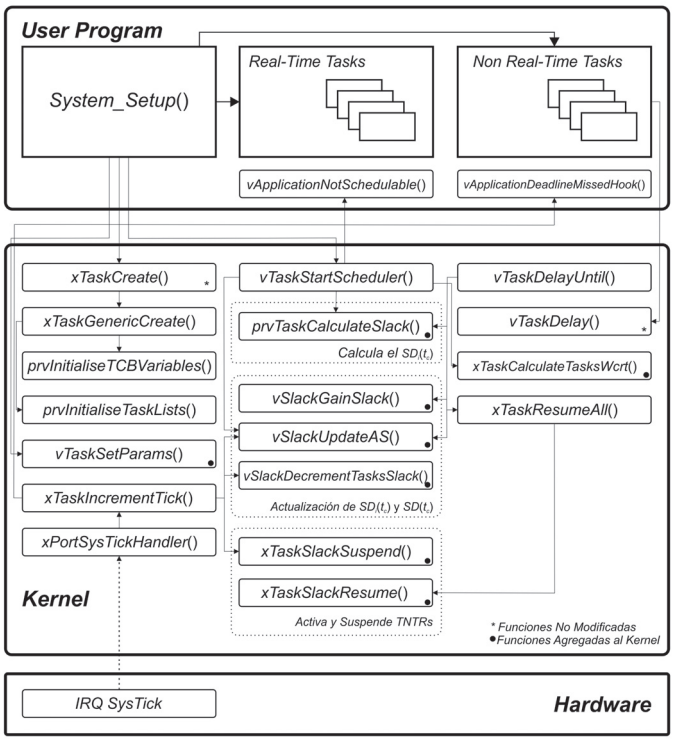

Fig. 2. Modificaciones y Agregados al *Kernel* del *FreeRTOS*.

Una vez actualizados los contadores, se calcula nuevamente $SD(t_c)$. Si $SD(t_c) > 0$, las *TNTR* bloqueadas en *xSlackDelayedTaskList* son puestas nuevamente en la cola de tareas listas, para continuar con su ejecución. Este modelo de ejecución, corresponde a una heurística del tipo golosa de asignación de *SD* al conjunto *TNTR*.

Al procesar cada *tick* de reloj en la función *xTaskIncrementTick*(), se restará un *tick* a los contadores $SD_i(t_c)$ de la siguiente manera. Si se está ejecutando una *TNTR* o la

tarea *idle*, se restan todos los $SD_i(t_c)$. Si se estuviera ejecutando una τ_i, se restan sólo los contadores de las tareas de mayor prioridad que τ_i. Una vez que los contadores $SD_i(t_c)$ correspondientes han sido actualizados, se calcula nuevamente $SD(t_c)$. Luego, si $SD(t_c) = 0$, y hubiera *TNTR* listas para ejecutar, estas son suspendidas y puestas en la cola *xSlackDelayedTaskList*.

Para actualizar los contadores $SD_i(t_c)$ y agregar y remover *TNTR* de la cola de tareas listas, las siguientes funciones auxiliares se incluyen en el *kernel*:

- *vSlackDecrementTasksSlack*(). Dada una τ_i, resta a los contadores $SD_j(t_c)$ de todas las tareas τ_j de *mayor* prioridad ($j < i$) una cierta cantidad de *ticks*.
- *vSlackDecrementAllTasksSlack*(). Resta a los contadores $SD_i(t_c)$ de *todas* las *TTR* la cantidad de tiempo especificada.
- *vSlackGainSlack*(). Dada una τ_i, suma a los contadores $SD_k(t_c)$ de todas las tareas τ_k de *menor* prioridad ($i < k$) una cierta cantidad de *ticks*.
- *vSlackUpdateAvailableSlack*(). Realiza la actualización del $SD(t_c)$.
- *vTaskSlackSuspend*(). Mueve las *TNTR* de la cola de tareas listas a *xSlackDelayedTaskList*.
- *vTaskSlackResume*(). Pasa las *TNTR* bloqueadas por falta de *SD*, desde *xSlackDelayedTaskList* a la cola de tareas listas.

4.2 Planificación Mediante *SS*

FreeRTOS implementa la cola de tareas listas mediante un arreglo de colas, administrando cada nivel de prioridad con una cola de tareas (Fig. 3). Estas pueden contener una o más tareas con el mismo nivel de prioridad, e implementar una política *FIFO* o *RR*. Para una configuración con *M* prioridades, el nivel 0 corresponde a la prioridad más baja, y *M* - 1 a la más alta.

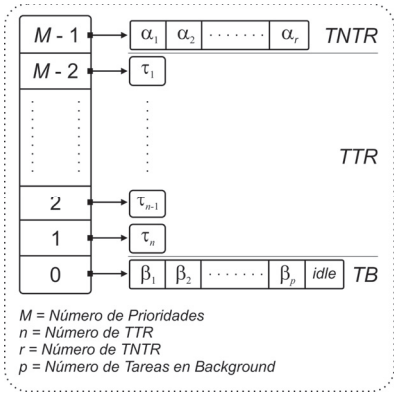

Fig. 3. Distribución de prioridades en la cola de tareas listas de *FreeRTOS*.

La prioridad *M* - 1 se reserva para las *TNTR*. Estas deben ser creadas asignándoles esta prioridad. Sin embargo, las *TNTR* sólo estarán presentes en la cola de tareas listas

si $SD(t_c) > 0$. De ser así, y al tener la máxima prioridad, el planificador de *FreeRTOS* desaloja cualquier *TTR* para ejecutar las *TNTR*, hasta que las mismas finalicen, se bloqueen o agoten el *SD*.

Las prioridades M - 2 a 1, están disponibles para las *TTR*. En el modelo, τ_1 es la *TTR* de mayor prioridad, y le corresponde la prioridad M - 2, a τ_2 M - 3, hasta llegar a τ_n, a la que se le asigna el nivel $M - n - 1$. La implementación sólo soporta una *TTR* por nivel de prioridad. Luego, para un sistema con n *TTR*, se debe configurar mínimamente a *FreeRTOS* con $M = n + 2$ niveles de prioridad.

Las tareas con prioridad 0 son ejecutadas en *background* ([17]), cuando no existan *TTR* o *TNTR* listas para ejecutar, y son excluidas de la planificación por *SS*. Se hace notar que puede existir $SD(t_c) < 1$, y como la granularidad es de un 1 *tick*, la implementación no lo detecta. Esto ocasiona, que de existir *TNTRs* pendientes de ejecución, no son puestas en la cola de listas y como consecuencia tareas en *background* o *TTRs* pueden ejecutarse en lo que resta del *tick*.

La Figura 3 presenta la cola de tareas listas de *FreeRTOS*, con n *TTR*, r *TNTR* y p tareas en *background* (*TB*), todas listas para ejecutar y con $SD(t_c) > 0$. Notar que si $SD(t_c) = 0$, la cola de tareas listas del nivel M - 1 se encontraría vacía. Por defecto en *FreeRTOS*, la tarea *idle* siempre se encuentra lista para ejecutar.

4.3 Ejemplo de Ejecución

Se presenta un ejemplo de un *STR* compuesto por las *TTR* τ_1 ($T_1 = 30$, $C_1 = 10$) y τ_2 ($T_2 = 40$, $C_2 = 10$), una *TNTR* α_1 ($C_\alpha = 15$) y una *TB* β_1. Se emplea una política *RM*, y la *TNTR* se activa mediante una *ISR*. En el inicio, se tiene $SD_1(0) = 20$ y $SD_2(0) = 10$.

La τ_1 finaliza su instancia en $t = 6$ (1), al invocar a *vTaskDelayUntil*(). Se calcula $SD_1(7) = 43$, y se actualiza $SD_2(6) = 14$ (*slack ganado*). Luego, τ_2 ejecuta hasta $t = 12$, donde una *ISR* activa α_1 (2). Como $SD(12) > 0$, α_1 desaloja a τ_2 y se ejecuta hasta el instante $t = 26$ (3), donde se agota el *slack* ($SD(26) = 0$). Consecuentemente, es removida de la cola de tareas listas y continúa τ_2 hasta $t = 28$, donde finaliza e invoca *vTaskDelayUntil*() (4). Luego, se calcula $SD_2(29) = 22$. Como $SD(29) > 0$, la tarea α_1 reinicia su ejecución, y finaliza en (5). Luego, la nueva instancia de τ_1, que fue postergada por la *TNTR*, inicia su ejecución. Al terminar τ_1, se ejecuta la nueva instancia de τ_2 (6). Finalmente, al no existir *TTR* o *TNTR*, β_1 es ejecutada (7).

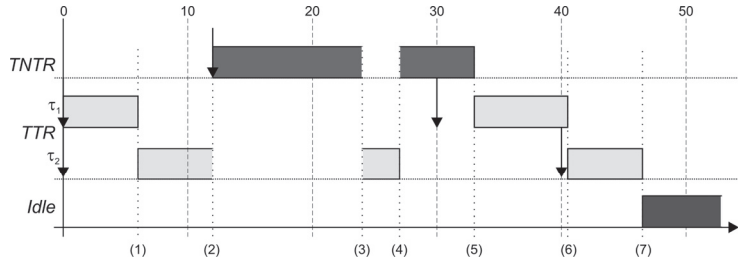

Fig. 4. Ejemplo de ejecución de un *STR mixto* mediante *SS*.

5 Resultados Experimentales y Análisis

Se generaron 1000 *STR* de 10 tareas, por cada *FU* del 10% al 90%, en intervalos de 10% mediante el generador de *STR* presentado en [18]. Los parámetros *T* y *C* de las *TTR* se distribuyeron uniformemente, entre 25 y 1000 *ticks*.

Se implementó el método de *SS* exacto descripto en [9]. En la evaluación se mide el número de ciclos de *CPU* requeridos desde la invocación de *vTaskDelayUntil*(), hasta la ejecución del *CS* a la siguiente tarea lista, en la función *vTaskSwitchContext*().

En las pruebas se utilizó *FreeRTOS* v8.1.2, ejecutado sobre una placa *mbed* LPC1768, basadas en *ARM Cortex-M3*. La duración del *tick* se configuro en 1*ms*. Se desactivo el control de desbordamiento de pila, y se empleó el mecanismo de *CS* optimizado, provisto por *FreeRTOS*, para *Cortex-M3/4*. La medición de los ciclos se realizó con el contador *CYCCNT* (*Clock Cycle Counter*), disponible como parte de la funcionalidad *DWT* (*Data Watchpoint and Trace*) del *Cortex-M3*.

El costo por *FU* que se presenta en la Figura 5, corresponde al promedio de todas las finalizaciones de todas las tareas de cada *STR* ejecutado. Se realizaron 3 distintas configuraciones. En la primera se utilizó *FreeRTOS* sin modificaciones. Para la segunda, se utilizó todas las modificaciones introducidas sin el cálculo del $SD_i(t_c)$ al finalizar cada instancia. De esta manera se determino el costo introducido al *CS*, independientemente del algoritmo de *SS*. La tercera configuración, es el peor caso, ya que realiza el cálculo del $SD_i(t_c)$ y la actualización de contadores. En todas las configuraciones, se promedió las primeras 20 finalizaciones de cada tarea (Fig. 5).

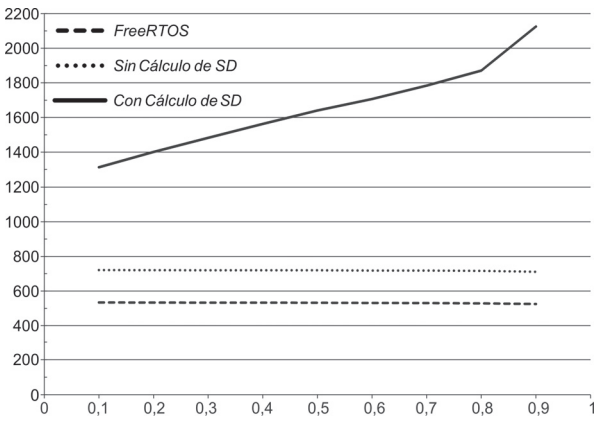

Fig. 5. Costo promedio en Ciclos vs *FU* del *STR*.

Los resultados obtenidos fueron los esperados. El costo introducido sin el cálculo de *SS*, se mantiene prácticamente constante, independientemente del *FU*. Además, el incremento con respecto a *FreeRTOS* sin modificaciones se mantiene en el mismo orden para cualquier *FU*. Por otro lado, el algoritmo utilizado introdujo un incremento al costo en ciclos de *CPU*, similar al costo presentado en [9]. Se hace notar que un

tick de 1*ms* equivale aproximadamente a 96000 ciclos. Consecuentemente el nuevo costo del *CS* es menor al 3% del *tick* en el peor de los casos.

6 Conclusiones y Trabajos Futuros

Actualmente los *SOTR* son utilizados en diversas áreas de aplicación. Cada una de estas áreas impone requerimientos funcionales particulares que deben ser satisfechos por la implementación final del sistema. La propuesta del modelo de tareas permite estructurar el diseño mediante la implementación de cada una de estos requerimientos funcionales por conjuntos de tareas. Las especificaciones temporales de estas tareas las categorizan en *TTR* y en *TNTR*.

Los *SOTR* están orientados para satisfacer los requerimientos temporales de las *TTR*. Sin embargo, la coexistencia con *TNTR* puede provocar una importante degradación de la eficiencia del sistema y en el desaprovechamiento de sus recursos. Existen diversas propuestas para flexibilizar la coexistencia de estos grupos de tareas, pero el desarrollo teórico de los mismos dista mucho de sus resultados reales, debido a la complejidad de su implementación.

En este trabajo se implementó, en un *SOTR* ampliamente utilizado en aplicaciones embebidas, un mecanismo eficiente para el soporte de sistemas heterogéneos y flexibles por medio de métodos de *SS*. Mediante el *framework* propuesto, la implementación de un nuevo método de *SS* requiere, para la mayoría de los casos, únicamente de la programación de su algoritmo de cálculo de *SD*. El código fuente se puede obtener en el sitio web del *Real Time Systems Group* de la UNPSJB[1].

En trabajos futuros se implementarán otros métodos *SS* de menor costo, distintas heurísticas para un mejor aprovechamiento del *SD* y otros tipos de requerimientos como tolerancia a fallas, ahorro de energía, etc., mediante el uso del *SD*.

7 Agradecimientos

Los autores agradecen a Richard Barry por sus consejos para mejorar las evaluaciones realizadas, a los revisores por sus comentarios y al *ARM University Program*, por medio del cual se accedió a las placas micro controladoras *mbed* empleadas en las pruebas.

Referencias

[1] John A. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *Computer,* vol. 21, Nº 10, pp. 10-19, 1988.

[2] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM,* vol. 20, Nº 1, pp. 46-61, 1973.

[1] http://www.rtsg.unp.edu.ar/

[3] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal,* vol. 6, Nº 3, pp. 116-128, 1991.

[4] J. Y. T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real Time Tasks," *Perf. Eval. (Netherlands),* vol. 2, Nº pp. 237-250, 1982.

[5] Sandra Ramos-Thuel and John P. Lehoczky, "On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems," in *Real-Time Systems Symposium*, 1993, pp. 160-171.

[6] Too-Seng Tia, Jane W.-S. Liu and Mallikarjun Shankar, "Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed Priority Preemtive Systems," *The International Journal of Time-Critical Computing Systems,* vol. 10, Nº 1, pp. 23-43, January 1996.

[7] R. I. Davis, K. W. Tindell and A. Burns, "Scheduling Slack Time in Fixed-Priority Preemptive Systems," *Proceedings of the Real Time System Symposium,* Nº pp. 222-231, 1993.

[8] Rodrigo M. Santos, José M. Urriza, Jorge Santos and Javier D. Orozco, "New methods for redistributing slack time: applications and comparative evaluations," *The Journal of Systems & Software,* vol. 70-2, Nº pp. 115-128, 2004.

[9] José M. Urriza, Francisco E. Paez, Ricardo Cayssials, Javier D. Orozco and Lucas Schorb, "Low Cost Slack Stealing Method for RM/DM," *International Review in Computers and Software (IRECOS),* vol. 5, Nº 6, pp. 660-667, 2010.

[10] Lin Caixue and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, 2005, pp. 12 pp.-421.

[11] Robert I. Davis, "Approximate Slack Stealing Algorithms for Fixed Priority Pre-Emptive Systems," Real-Time Systems Research Group, University of York, York, England, Internal Report 1994.

[12] Agustín Rafael Espinosa Minguet, "Extensiones al Lenguaje Ada y a los Servicios POSIX para Planificación en Sistemas de Tiempo Real Estricto," Universidad Politecnica de Valencia, 2003.

[13] Luis A. Díaz, Francisco E. Páez, José M. Urriza, Javier D. Orozco and Ricardo Cayssials, "Implementación de un Método de Slack Stealing en el Kernel de MaRTE OS," *43 JAIIO (Jornadas Argentinas de Informática),* vol. 3º Simposio Argentino de Informatica Industrial (SII), Nº pp. 13-24, Septiembre 2014.

[14] S. Midonnet, D. Masson and R. Lassalle, "Slack-Time Computation for Temporal Robustness in Embedded Systems," *Embedded Systems Letters, IEEE,* vol. 2, Nº 4, pp. 119-122, 2010.

[15] Damien Masson and Serge Midonnet, "Userland Approximate Slack Stealer with Low Time Complexity," in *Proceedings of the 16th International Conference on Real-Time and Network Systems*, Rennes, France, France, 2008, pp. 29--38.

[16] Damien Masson and Serge Midonnet, "Slack time evaluation with RTSJ," presented at the Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, Ceara, Brazil, 2008.

[17] J. K. Strosnider, John P. Lehoczky and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Transactions on Computers,* vol. 44, Nº 1, pp. 73-91, 1995.

[18] Gabriela Olguín, Laura Biscayart and José M. Urriza, "Generación de tareas periódicas y aperiódicas para simulación de sistemas de tiempo real," *Journal of Industrial Engineering (IJIE),* vol. 3, Nº 6, pp. 53-69, 2011.