Implementación de un Método de *Slack Stealing* en el *Kernel* de *MaRTE OS*

Luis A. Díaz[1], Francisco E. Páez[1,3], José M. Urriza[1], Javier D. Orozco[2,3], Ricardo Cayssials[2,3]

[1]Universidad Nacional de la Patagonia San Juan Bosco, Puerto Madryn, Argentina, [2]Universidad Nacional del Sur, [3]CONICET

josemurriza@unp.edu.ar

Resumen. Este trabajo presenta la implementación de un método *Slack Stealing* dentro del *kernel* del sistema operativo de tiempo real *MaRTE OS*. Existe en la actualidad, una necesidad de ampliar la funcionalidad de los sistemas operativos de tiempo real, para ofrecer la planificación de nuevos requerimientos heterogéneos, conservando la predictibilidad que deben poseer las tareas de tiempo real.

Palabras Claves: Sistemas de Tiempo Real, *Slack Stealing*, *MaRTE OS*, Planificabilidad.

1. Introducción

En la disciplina de los *Sistemas de Tiempo Real* (*STR*), la administración del tiempo ocioso es de vital importancia para lograr que un sistema pueda ser heterogéneo en el tipo de tareas de tiempo real o no a ejecutar. Este tipo de administración se logra implementando algún tipo de método o técnica que permita lograrlo. El método *Slack Stealing* (*SS*) es uno de estos, tal vez el mejor de ellos. Consecuentemente, numerosos sistemas informáticos industriales requieren de un sistema operativo de tiempo real (*SOTR*) como *MaRTE OS* y contar con métodos que permitan ejecutar tareas de no tiempo real, sin afectar al *STR*. En muchos casos, tareas como alarmas, ahorro de energía en dispositivos móviles, robustez en la ejecución de las tareas, tolerancia a los fallos mediante re-ejecución de tareas, etc., son modeladas de esta forma. Sin embargo, este tipo de tarea puede requerir de algún tipo de calidad de servicio (*QoS*) distinto, como puede ser una atención prioritaria. Este trabajo permite aportar una herramienta tangible para que los diseñadores de informática industrial puedan utilizarla en el diseño de sus sistemas.

En las últimas décadas los lenguajes de programación y los sistemas operativos (*SO*) han evolucionado a fin de poder desarrollar aplicaciones con características de Tiempo Real (*TR*). El lenguaje de programación *ADA* y las extensiones para *TR* del estándar *POSIX*, pueden tomarse como referencias en esta tendencia. Poco a poco, la

industria ha ido incorporando estos avances sin que los usuarios finales lo perciban, en por ejemplo *PLC*s, controladores industriales, etc.

Sin embargo, diversos aspectos del problema aún no han sido explorados profundamente y uno de estos, es la de conseguir niveles óptimos en el desempeño de la planificación de sistemas heterogéneos en termino de recursos físicos y lógicos. En lo que sigue, se denominará *Sistema de Tiempo Real Heterogéneo* (*STRH*) a un sistema compuesto por un subsistema de *TR* (*sTR*) y otro de no *TR* (*sNTR*). En particular, debido a que el *sTR* posee restricciones temporales que resultan insoslayables para que el funcionamiento global del sistema resulte aceptable, se asume que un sistema de este tipo es un es un *STR* compuesto por un *sTR* y un *sNTR*. Hoy en día, se utilizan diversos métodos específicos para tratar *STRH* en algunos *SOTR*. Entre los más comunes se encuentran los *Métodos Servidores* (*MS*) [1, 2, 3, 4, 5], *Slack Stealing* (*SS*) [6, 7, 8, 9], *Dual Priority* (*DP*) [10]. Sin embargo, algunos de estos métodos, no resultan eficientes a la hora de su implementación y consecuentemente no siempre pueden ser utilizados debido a restricciones de costo, escala y disponibilidad, entre otras.

La principal ventaja que poseen los métodos basados en *SS* frente a otros métodos, es que son los únicos que permiten detectar y aprovechar el tiempo ocioso del sistema en su totalidad. A pesar de ello, muchos de estos poseen un elevado *Costo Computacional* (*CC*) y consecuentemente no pueden ser utilizados en algunas aplicaciones reales. No obstante, en los últimos años se ha logrado una reducción en el *CC* de su implementación y existen diversos trabajos de investigación que están enfocados en ese sentido [11, 12].

En este trabajo se ha implementado un método *SS* en el *kernel* del *SOTR MaRTE OS*, más precisamente en el planificador de tareas, como una política más de planificación, dando soporte a la atención del *sNTR*.

El presente trabajo se organiza de la siguiente manera. A continuación se realiza una breve introducción a los *STR* y a la planificación heterogénea. En la sección 2 se selecciona un *SOTR*. En la sección 3, se presenta el algoritmo *SS* a utilizar. En la sección 4 se presenta la implementación efectuada. En la sección 5 se realiza el análisis de eficiencia. Finalmente, en la sección 6 se presentan las conclusiones y trabajos futuros.

1.1. Introducción a los Sistemas de Tiempo Real

La definición de *STR* más aceptada, es la realizada por Stankovic [13]: *"En los STR los resultados no sólo deben ser correctos aritmética y lógicamente sino que, además, deben producirse antes de un determinado tiempo, denominado vencimiento."*

La ejecución de una tarea por fuera de sus constricciones puede ser inaceptable y dicha inaceptabilidad puede ser local a la tarea o global. Esta consideración permite clasificar los *STR* en tres tipos:

- *Duros*: No admiten que ninguna tarea pierda su vencimiento. Esta condición es local y global.
- *Blandos*: Permiten que se pierdan algunos vencimientos. En este contexto la inaceptabilidad local puede no ser global.

- *Firmes*: Sólo permiten una cantidad acotada de pérdidas. Como en el caso previo, la inaceptabilidad local puede no ser global.

En los *STR duros*, la pérdida de un vencimiento en una tarea es inaceptable y en general puede tener consecuencias adversas sobre todo el funcionamiento del sistema. Por ello se debe garantizar que todas las tareas finalicen antes de su vencimiento. Para ello es necesario analizar, en tiempo de diseño, la planificabilidad del sistema mediante procedimientos analíticos denominados, tests de planificabilidad. Si el test es exitoso, a estos sistemas se los denomina planificables y se dice que han cumplido con todas sus constricciones de tiempo. Las primeras contribuciones en este sentido fueron realizadas por Liu y Layland [14] y han sido mejoradas en trabajos posteriores.

Un *algoritmo de planificación* es un conjunto de reglas que determinan qué tarea va a ser ejecutada en un instante determinado. Los algoritmos de planificación pueden ser clasificados en *estáticos* y *dinámicos* [15]. Los planificadores dinámicos, son los más utilizados y están basados en asignar una prioridad a cada tarea. Si la prioridad cambia en tiempo de ejecución, es *dinámica*; caso contrario, es *fija*.

En la planificación por prioridades fijas, las reglas más utilizadas son *Rate Monotonic* (*RM*) y *Deadline Monotonic* (*DM*). En prioridades dinámicas, las más utilizadas son *Earliest Deadline First* (*EDF*) y *Least Laxity First* (*LLF*). Además, existe un método de planificación que combina prioridades fijas y dinámicas, denominado *Dual Priority* (*DP*). Este trabajo se centra exclusivamente en las disciplinas de prioridades fijas y en *STR* de tipo *duro*.

1.2. Planificadores de Tiempo Real Heterogéneos Apropiativos

Generalmente, un planificador heterogéneo, debe garantizar las restricciones del *sTR* y proveer algún nivel en la *QoS* para el *sNTR*. Ello requiere una administración eficiente de los recursos del sistema con el objeto de no sobredimensionar innecesariamente la plataforma de ejecución con la intención de mejorar el desempeño del *sNTR* sin afectar al *sTR*. A fin de trabajar con un modelo restrictivo, se considerará en adelante que el *sTR* es del tipo *duro*. Consecuentemente, debe contarse con alguna técnica o servicio en tiempo de ejecución, que maximice la utilización de los recursos, maximizando el aprovechamiento de los tiempos ociosos que deja el *sTR duro*. A continuación se realiza una breve introducción a los métodos desarrollados para planificadores heterogéneos:

- *Servicio en Segundo Plano* (*Background Service - BS*): Su principio de funcionamiento es ejecutar el *sNTR* cuando las tareas del *sTR* no se están ejecutando. Su mayor desventaja es no poder garantizar ningún tipo de *QoS* para las tareas del *sNTR*.
- *Servidores:* Consiste en implantar en el *sTR*, una *tarea servidora*, con una determinada prioridad y utilizar el tiempo de ejecución de la misma para procesar el *sNTR*. Este tipo de método, no logra utilizar todo el tiempo ocioso, por lo cual también debe utilizar un *BS* y aguardar a que se instancie nuevamente la *tarea servidora*, para proseguir con las tareas del *sNTR*. Entre los *Métodos Servidores* clásicos para planificación de *sNTR*, se pueden mencionar: *Polling Server* [5],

Deferred Server [5], *Priority Exchange Server* [3], *Sporadic Server* [4] y *Total Bandwidth Server* [2].

- *Slack Stealing:* Consiste en adelantar una fracción del tiempo ocioso, de manera tal que se pueda ejecutar el *sNTR*, retrasando las ejecuciones de las tareas del *sTR* al límite de su planificabilidad. En la actualidad existen diversos métodos que aplican esta técnica, algunos lo hacen en tiempo de ejecución (*on-line*), otros lo pre-calculan en el tiempo de inicialización (*off-line*); algunos lo realizan de manera exacta y otros de manera aproximada [6, 7, 8, 9, 11, 12].

2. Selección de un *SOTR*

Actualmente existe una amplia variedad de *SOTR*. Se puede mencionar a *VxWorks*, *FreeRTOS*, *Windows CE*, *RT-Linux*, *Lynx*, *MaRTE OS*, *QNX Neutrino*, entre otros. La selección del *SOTR* a utilizar en este trabajo, tuvo como objetivo encontrar el primero que ofreciera soporte para *STR duros*, un entorno de desarrollo práctico y accesible, y un *kernel* factible de ser modificado o extendido en cuanto a sus capacidades de planificación de tareas.

Inicialmente se consideró el *SOTR ERIKA* en su versión Educativa, ya que es pequeño, está escrito en lenguaje *C* y posee documentación accesible en la web. Sin embargo, la versión Educativa ha sido discontinuada y sólo está disponible para la arquitectura *Hitachi H8/300*. Estas cuestiones determinaron no utilizarlo para la parte experimental del presente trabajo.

La siguiente elección recayó en *VxWorks*. Si bien es *POSIX* compatible y soporta numerosas plataformas de *Hardware* (*HW*), tiene la desventaja de no ser de código abierto, lo cual es indispensable para desarrollar la implementación propuesta. Por ello, *VxWorks* también fue descartado.

El tercer *SOTR* estudiado fue *MaRTE OS* v1.1, desarrollado por M. A. Rivas, ([16]) el cual es de código abierto, y cuenta con documentación en castellano. Brinda soporte para una arquitectura de *HW* muy accesible (PC *x86*), está programado principalmente en lenguaje *Ada* y es compatible con *POSIX*. Posee un entorno de desarrollo cruzado que puede montarse fácilmente teniendo una PC con *Linux* como *SO* anfitrión y otra PC *x86*, en la cual pueden ejecutarse las aplicaciones desarrolladas. Por sus características, *MaRTE OS* ha satisfecho todos los requerimientos solicitados.

3. Selección del Algoritmo de *SS* a Implementar

Implementar un método de *SS* exacto con bajo *CC*, permitiría administrar eficientemente el tiempo ocioso disponible en un determinado instante en tiempo de ejecución. De esta manera, es posible obtener un mejor desempeño en el *sNTR*. Por ejemplo, en términos de tiempo de respuesta, pueden obtenerse tiempos menores que con cualquier otro método existente. Es por ello que uno de los principales objetivos, es realizar la implementación y evaluar su factibilidad. La gran mayoría de los métodos de *SS* previamente mencionados se basan en el cálculo de los períodos de

actividad e inactividad del microprocesador para calcular el *Slack Disponible* (*SD*). Esto conlleva a que posean un alto *CC* temporal o espacial o ambos.

Como trabajos previos, se puede mencionar que en el año 2003, Espinosa Minguet en [17] implementó en *MaRTE OS*, una versión ligeramente modificada del algoritmo aproximado *DASS* presentado por Davis en [9], donde el *SD* se calcula de forma dinámica. Su principal inconveniente reside en el alto *CC* del algoritmo aun siendo aproximado.

El método de cálculo de *SS* que se implementó es el presentado en [11]. Dicho método ha sido seleccionado por haber sido diseñado para sistemas embebidos y por ser un método de cálculo exacto de bajo *CC*, desarrollado para ser aplicado en tiempo de ejecución. A diferencia del resto de los métodos de *SS*, este método reduce el *CC* mediante la reducción de la cantidad de puntos a inspeccionar y el intervalo de búsqueda.

4. Implementación del Método de *SS* en *MaRTE OS*

En primer lugar, se implementó el algoritmo *SS* en lenguaje *Ada* y se modificó el planificador de *MaRTE OS*, para que invoque al algoritmo cada vez que la instancia de una tarea de *TR* finalice. Para ello se creó un nuevo paquete *Ada* que contiene el procedimiento `calculateSlack` y otras funciones y procedimientos auxiliares. Este, recibe como parámetros un identificador de tarea, una estructura que almacena los contadores de *SD* y otros de atributos temporales de las tareas del sistema (`Tasks_List`), la cantidad de tareas periódicas en el sistema (`TasksCant`) y el tiempo actual en el cual fue invocado el método (`T_Current`).

Además, se implementaron funciones y procedimientos auxiliares que son utilizados por el algoritmo de cálculo: la función `workload`, que calcula la carga de trabajo del subsistema (`0 .. ptask`) en el instante `tc` y el procedimiento `SlackCalc`, que calcula el *SD* de la tarea `ptask` en el intervalo (t, t_c), y lo retorna en un parámetro de salida `k`.

4.1. Modificaciones al *kernel* de *MaRTE OS*

Para lograr que el *SOTR* sea capaz de planificar sistemas heterogéneos utilizando el *SD del Sistema* (*SDS*), se realizaron modificaciones al paquete `marte-kernel-scheduler` y a otros que implementan servicios que controlan las operaciones de las tareas dentro del *SOTR*. Además, se agregaron nuevos archivos fuente en los cuales se especifican las estructuras y procedimientos necesarios para calcular y almacenar el *SD* en cualquier instante de tiempo, presentados en el apartado anterior.

4.1.1. Nuevos Archivos Fuente

En primer lugar, se agregó al *kernel,* el paquete `Slack_Utilities`, que contiene el cuerpo y la especificación del algoritmo de cálculo de *SD* para una tarea periódica (T_i) en un tiempo dado (t_c), y de los demás procedimientos y funciones auxiliares presentadas en el apartado anterior.

Además, se agregó el paquete `Tasks_Attributes`, que contiene la especificación de las estructuras necesarias para almacenar en memoria los contadores de *SD* de cada tarea periódica T_i, como así también los atributos temporales requeridos por el algoritmo de cálculo que deben especificarse durante la inicialización (*WCET*, *Período*, *Vencimiento*, etc.) y una variable en la cual se almacena el *SDS*. Dicha variable es visible para el planificador del sistema, el cual puede consultar su valor en cualquier instante (Fig.1).

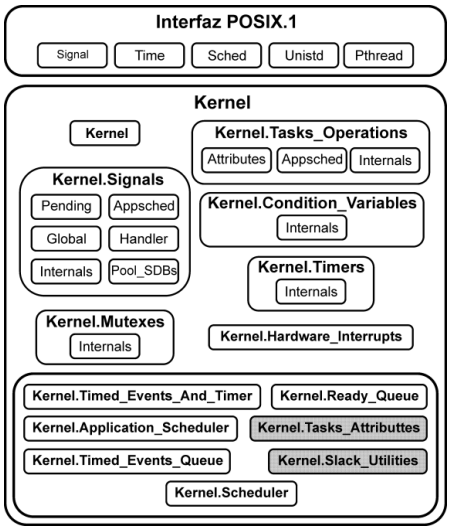

Fig. 1. Paquetes añadidos al *kernel* de *MaRTE OS*.

4.1.2. Archivos Fuente Modificados

A continuación se presentan las modificaciones a los archivos fuente de *MaRTE OS*:

- **Incorporación de una nueva política de planificación** (`SCHED_FIFO_WS`)**:** Se agregó al subtipo `Scheduling_Policies` definido en `marte-kernel.ads` la nueva política que permite planificar las tareas haciendo uso del *SDS*.
- **Incorporación de nuevos parámetros de configuración:** Se agregó al archivo de configuración un conjunto de parámetros, que son utilizados por el planificador cuando se utiliza la política `SCHED_FIFO_WS`. Mediante los mismos se podrá identificar (en base a la cantidad de tareas que posee la aplicación), cuál es el grupo de tareas que debe considerarse críticas y cuál es el grupo de tareas que debe considerarse como de no *TR*.
- **Incorporación de nuevos atributos al Bloque de Control de cada tarea:** El *Task Control Block* (*TCB*) de cada tarea está definido dentro del archivo de especificación `marte-kernel.ads`, y en él se han añadido un conjunto de atributos que permiten identificar dentro del *kernel* si pertenece al *sNTR* o no, contabiliza el número de instancias de las tareas del sistema, almacena el tiempo de uso de *CPU* de las mismas, indica si la instancia actual de la tarea ha finalizado o aún está

pendiente de ejecución, o especifica si una tarea del *sNTR* ha sido suspendida por haberse agotado el *SDS*.

- **Incorporación de nuevos procedimientos para registrar y remover *sNTR*** (`marte-kernel-scheduler.ads y .adb`): Al utilizar la nueva política de planificación `SCHED_FIFO_WS` es posible que inicialmente no haya una tarea del *sNTR* para ejecutar, pero sí en un instante posterior. Si hubiera tareas en el *sNTR* pendientes de ejecución y existe *SD*, el planificador debe seleccionar una de ellas para su ejecución y posponer la ejecución de las tareas críticas. Caso contrario, la existencia (o no) de *SD,* no altera la ejecución de las tareas críticas del sistema. Consecuentemente, se implementó un procedimiento que se utiliza para registrar a las tareas del *sNTR* en el sistema ni bien son instanciadas, denominado `Register_sNTR_Task()`, que le permite a *MaRTE OS* determinar si hay tareas del *sNTR* pendientes. Del mismo modo, el procedimiento `Unregister_sNTR_Task()` permite determinar que no hay tareas en el *sNTR* pendientes, con lo cual el planificador no ejecuta ninguna acción específica al detectar que hay *SD*.
- **Inicialización de la estructura que almacena los atributos temporales de las tareas:** La estructura definida en el nuevo paquete `Tasks_Attributes` es inicializada en el procedimiento de inicialización del planificador. Los atributos temporales de las tareas deberán ser especificados por el programador de aplicaciones antes de compilar la aplicación, y junto con el archivo `configuration_parameters.ads` son los únicos dos archivos en los cuales se deberán introducir valores específicos que dependen exclusivamente del *STR* que ejecutará la aplicación.

4.1.3. Modificaciones al planificador

Se agregó un nuevo procedimiento `Update_System_Slack()` al paquete que implementa el planificador. El mismo se encarga de invocar al procedimiento que implementa el algoritmo de cálculo de *SD*, y además actualiza los contadores de *SD* y el tiempo de uso de *CPU* de cada instancia de las tareas que corren en el sistema. (`marte-kernel-scheduler.ads` y `.adb`)

- **Procedimiento** `Update_System_Slack()`: Es invocado desde el procedimiento `CPU_Time_Accounting()` y es el encargado de invocar al algoritmo de cálculo de *SD,* y de realizar posteriormente la actualización de los contadores de *SD* y de los contadores de tiempo ejecutado por cada instancia.
- **Modificación a** `CPU_Time_Accounting()`: Se agregó al procedimiento `CPU_Time_Accounting()` la invocación al procedimiento que calcula el *SD* de las tareas, actualiza los contadores y el *SDS*. `CPU_Time_Accounting()` es invocado cada vez que se realiza un cambio de contexto (*CS*), y luego de que el manejador de interrupciones ha atendido todas las interrupciones de hardware. (`marte-kernel-scheduler.ads` y `.adb`)
- **Modificación al procedimiento** `Task_Gets_Ready()`: *MaRTE OS* utiliza distintos tipos de eventos temporales dentro del *kernel*, los cuales se encolan en la cola global de eventos temporales del sistema. Además de éstos, existe un tipo de evento particular, denominado Evento de Tiempo de Ejecución (`Scheduler Timed`

`Event`), que permite representar acciones programadas para ser ejecutadas cuando el tiempo de ejecución de una tarea alcance un determinado valor. Estos eventos se agregan en una cola que es propia de cada tarea. Dentro del procedimiento `Task_Gets_Ready()`, si la política de planificación de la tarea que lo invoca es `SCHED_FIFO_WS` y el atributo `Is_sNTR_Task` de la tarea tiene un valor booleano verdadero, se procede a generar un evento de tiempo de ejecución, con un tiempo de expiración igual al *SDS* en ese instante, y posteriormente se encola dicho evento en la cola de eventos temporales de la tarea que ha invocado a `Task_Gets_Ready()`. Cuando vence el evento que limita la ejecución de una tarea en el *sNTR*, se produce una interrupción del temporizador de *HW* que activa al manejador de interrupciones, el cual procede a suspender la tarea que generó la interrupción. Posteriormente se invoca al planificador para que seleccione la siguiente tarea que deberá tomar el uso del procesador. De esta manera, la tarea ejecutará hasta agotar el *SDS*, o finalizará la ejecución de su instancia actual en un instante de tiempo menor. En el primer caso, será suspendida y desalojada por el planificador hasta que se genere nuevo *SD* en el sistema. En el segundo caso, la tarea se suspenderá hasta la próxima activación por haber alcanzado la sentencia `delay_until`, con lo cual deberá desecharse el evento de tiempo de ejecución que genera la interrupción por agotamiento de *SD*, ya que bajo estas circunstancias es irrelevante. La activación de una tarea en *sNTR* que se encuentra suspendida por no contar con *SD* en el sistema para su ejecución, se realiza al final del procedimiento `Update_System_Slack()`. Dentro de este procedimiento, si luego de actualizar el *SDS*, este no es nulo y es suficiente como para permitir la ejecución de trabajo que no es de *TR*, se chequea la existencia de alguna tarea en el *sNTR* suspendida y se procede a la activación de la misma invocando al procedimiento `Task_Gets_Ready()`.

5. Análisis de Eficiencia

Para realizar el análisis de eficiencia de la implementación, se tomaron como referencia los datos obtenidos en [16] en el análisis de las prestaciones de *MaRTE OS*, los cuales fueron recabados mediante la ejecución del *SOTR* en un Pentium III a 1.1Ghz.

Descripción de la medida	PIII a 1.1Ghz	PIII a 750Mhz
CS después de una operación de cesión voluntaria de CPU	2.4 µs	3.52 µs
Liberación y posterior toma de un mutex de herencia de prioridad	3.5 µs	5.13 µs
CS provocado por una interrupción	5.5 µs	8.06 µs

Tabla 1. Tiempos de *CS* estimados en un PIII a 750Mhz.

La PC que fue utilizada como *Target* en este trabajo, para depurar la implementación en el entorno de desarrollo, posee un procesador Pentium III a 750Mhz (Fig.2), y sobre dicho *HW,* se ejecutaron también las aplicaciones de prueba, con lo cual los tiempos que se debían obtener, fueron estimados a partir de los anteriores. Teniendo en cuenta la menor velocidad de procesamiento del *Target* con respecto al utilizado en [16], los tiempos serían los siguientes:

5.1. Impacto de las modificaciones

Para poder determinar la sobrecarga introducida al planificador de *MaRTE OS*, se analizaron particularmente las secciones de código que fueron agregadas al paquete que implementa las funciones de planificación del *SOTR* y aquellas agregadas a los procedimientos que se encargan de gestionar los distintos estados de las tareas, ya que éstas son las que producen un incremento en el tiempo que ha de emplearse para realizar los *CS*.

A fin de reducir el número de invocaciones a procedimientos dentro del *kernel*, se ha tratado de centralizar todo el código fuente asociado al cálculo del *SDS* en un único procedimiento, el cual realiza las siguientes acciones:

- Actualización de atributos temporales de cada tarea.
- Cálculo del *SD* para la tarea *i* en un instante t_c.
- Actualización del *SDS*.
- Activación de tareas de *sNTR* en espera de ejecución.

La actualización de los atributos temporales de cada tarea se realiza cada vez que se produce alguna interrupción en el sistema, ya que luego de la ejecución del manejador de interrupciones, siempre se ejecuta el procedimiento `CPU_TimeAccounting()`.

Este procedimiento se encarga de contabilizar el tiempo de ejecución de las tareas después de un *CS*: contabiliza el tiempo consumido en su última activación por la tarea que deja la *CPU* y registra el instante de activación de la tarea que toma la *CPU*. Además, para la tarea que toma la *CPU*, actualiza los tiempos de sus eventos de tiempo de ejecución. Dentro de `CPU_TimeAccounting()` ocurre la invocación al procedimiento que realiza la actualización del *SDS*: el procedimiento `Update_System_Slack()`.

En el caso en que la tarea que estaba en posesión de la *CPU* no haya finalizado su instancia actual, `Update_System_Slack()` se encarga de actualizar los contadores de *SD* de las tareas. Se actualiza luego el *SDS* tomando el mínimo valor de *SD* del conjunto de contadores y posteriormente chequea si es suficiente como para ejecutar tareas del *sNTR* pendientes. Sólo los *CS* producidos por la finalización de una instancia de una tarea *i* provocarán que el procedimiento `Update_System_Slack()` calcule el *SD* para dicha tarea. El cálculo se realizará antes de la actualización de los atributos temporales de todas las tareas y del *SDS*. De esta manera, se puede ver que los *CS* que no implican cálculo de *SD* para una tarea *i* tendrán una sobrecarga menor que aquellos que sí lo requieran.

5.2. Diseño de las Pruebas

Los tiempos medidos durante la ejecución de las pruebas son los siguientes:

- ***Tiempo Acumulado de CS:*** Comprende el tiempo que demora el *SOTR* en realizar un *CS*, ya sea por una sesión voluntaria de *CPU* por parte de una tarea, o por una interrupción que provoca el desalojo de la tarea que estaba en ejecución.
- ***Tiempo de actualización de SDS:*** Comprende únicamente el tiempo que demora el *SOTR* en actualizar los contadores de *SD* de todas las tareas del sistema, actualizar el tiempo de ejecución de la instancia de la tarea que hasta ese momento

estaba en poder de la *CPU*, calcular el *SD* de dicha tarea (sólo si ha finalizado su instancia actual) y actualizar el *SDS*.

- ***Tiempo de cálculo de SD para cada tarea:*** Comprende únicamente el tiempo que demora el *SOTR* en calcular el *SD* de la tarea que hasta ese momento estaba en poder de la *CPU* (sólo se realiza si la tarea ha finalizado su instancia actual).

Los tiempos de *CS* se han agrupado, siendo categorizados como de *nivel i* aquellos *CS* en los cuales la tarea *i* es la que cede la *CPU* a otra tarea del sistema. Se separaron los *CS* que incluyen cálculo de *SD* de los que no lo hacen, a fin de poder distinguir la sobrecarga introducida. Esto se debe a que los *CS* que incluyen cálculo de *SD* sólo se producen cuando la instancia de una tarea ha finalizado.

Para poder almacenar los tiempos de *CS* durante la ejecución de la aplicación de prueba, se ha introducido dentro de *MaRTE OS* una sección de código al final del procedimiento `Do_Scheduling()` del paquete `marte-kernel_scheduler`, en la cual se procede a almacenar en un arreglo los datos de cada *CS*, previo a invocar a la rutina `HAL.Context_Switch()`.

5.3. Ejecución de las Pruebas

Se desarrolló una aplicación sencilla en el lenguaje *Ada*, que consta de un programa principal, el cual instancia un *STR* de *n* tareas periódicas. La duración de las pruebas comprendió 30 instanciaciones consecutivas de la tarea de mayor período. Esto se debe a que sistemas con un hiperperíodo elevado, implicarían un tiempo de ejecución demasiado prolongado.

Sistema de 5 tareas	***MaRTE OS***	***MaRTE OS* con *SS***	
		Sin Cálculo de *SD*	**Con Cálculo de *SD***
Nº *CS*	425	144	281
Tiempo Acumulado de *CS*	2316,950µs	938,237µs	3861,684µs
Tiempo Máximo de *CS*	9,913µs	10,893µs	31,006µs
Tiempo Mínimo de *CS*	2,905µs	4,053µs	9,652µs
Tiempo Promedio de *CS*	5,452µs	6,516µs	13,743µs

Tabla 2. Comparativa de tiempos de *CS* para *STR* de 5 tareas.

Sistema de 10 tareas	***MaRTE OS***	***MaRTE OS* con *SS***	
		Sin Cálculo de *SD*	**Con Cálculo de *SD***
Nº *CS*	1401	563	838
Tiempo Acumulado de *CS*	8023,990µs	4532,594µs	15561,543µs
Tiempo Máximo de *CS*	8,983µs	14,601µs	76,172µs
Tiempo Mínimo de *CS*	2,792µs	4,040µs	9,822µs
Tiempo Promedio de *CS*	5,727µs	8,051µs	18,570µs

Tabla 3. Comparativa de tiempos de *CS* para *STR* de 10 tareas.

En las Tablas 2, 3 y 4 se presentan los resultados obtenidos al ejecutar en *MaRTE OS* original y con *SS*, sistemas de 5, 10 y 20 tareas respectivamente, con períodos entre 25 y 1000 distribuidos de manera uniforme, y todos ellos con un factor de utilización del 80%, ya que es donde el método de *SS* implementado alcanza su mayor costo computacional ([11]).

Sistema de 20 tareas	*MaRTE OS*	*MaRTE OS* con *SS*	
		Sin Cálculo de *SD*	Con Cálculo de *SD*
Nº *CS*	2670	1155	1515
Tiempo Acumulado de *CS*	15788,672µs	8973,878µs	41732,681µs
Tiempo Máximo de *CS*	13,374µs	15,042µs	195,726µs
Tiempo Mínimo de *CS*	3,180µs	4,626µs	9,885µs
Tiempo Promedio de *CS*	5,913µs	7,770µs	27,546µs

Tabla 4. Comparativa de tiempos de *CS* para *STR* de 20 tareas.

Analizando los tiempos obtenidos (Fig.3), se observa que la sobrecarga introducida es mínima para los *CS* que no realizan cálculo de *SD* y se mantiene en el mismo orden que en la versión original. Para el caso de los *CS* con cálculo de *SD*, se observa que los tiempos máximos se elevan un orden de magnitud con respecto a los anteriores. En el peor de los casos, para el *STR* de 20 tareas se obtiene un tiempo máximo de *CS* cercano a 200µs, y un tiempo promedio que no supera los 30µs. Sin embargo, dado que estos *CS* sólo ocurren cuando finaliza la instancia de una tarea, se podrían considerar como una sobrecarga propia de la ejecución de la tarea.

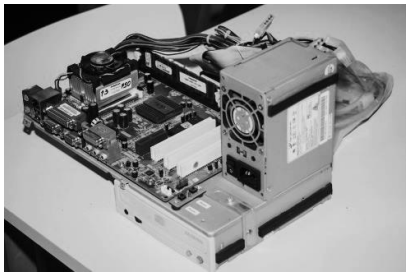

Fig. 2. Equipo utilizado como *Target*.

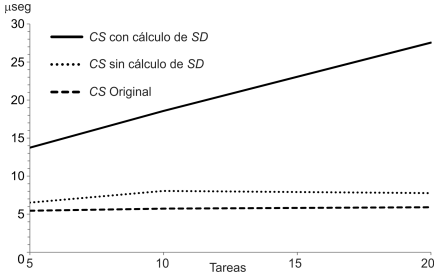

Fig. 3. Valores Promedio de *CS* (PIII-750Mhz)

6. Conclusiones y Trabajos Futuros

Se han realizado experimentos que muestran que el método *SS* implementado puede presentar una alternativa sólida y eficiente a la hora de planificar sistemas heterogéneos, brindando buenas prestaciones con un *CC* aceptable.

En trabajos posteriores se podrán realizar aplicaciones que hagan uso del *SDS* para implementar técnicas de Tolerancia a Fallos, o ahorro de energía, entre otras. Además, se implementarán otros algoritmos de *SS* exactos de menor *CC,* pero más complejos, pudiendo intercambiarlos de manera sencilla con el menor impacto posible sobre el código del *kernel* de *MaRTE OS*. También se verá la factibilidad de adaptar las modificaciones realizadas para que puedan incluirse en el estándar *POSIX*.

Referencias

[1] Brinkley Sprunt, John P. Lehoczky and Lui Sha, "Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm," in *IEEE Real-Time Systems Symposium*, Huntsville, Alabama, USA, 1988, pp. 251-258.

[2] Gerhard Fohler, Tomas Lennvall and Giorgio Buttazzo, "Inproved Handling of Soft Aperiodic Tasks in Offline Scheduled Real-Time Systems using Total Bandwidth Server," in *8th IEEE International Conference on Emerging Technologies & Factory Automation*, Nice France, 2001, pp. 151-157.

[3] John P. Lehoczky, Lui Sha and Jay K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Enviroments," in *IEEE Real-Time Systems Symposium*, 1987, pp. 261-270.

[4] B. Sprunt, L. Sha and John P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *The Journal of Real-Time Systems,* vol. 1, Nº 1, pp. 27-60, 1989.

[5] 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.

[6] T. S. Tia, J. W. S. Liu and M. 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 pp. 23-43, 1996.

[7] Sandra Ramos-Thuel and John P. Lehoczky, "Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priorys Systems using Slack Stealing," in *Real-Time Systems Symposium*, 1994, pp. 22-33.

[8] 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.

[9] 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.

[10] Robert I. Davis, "Dual Priority Scheduling: A Means of Providing Flexibility in Hard Real-Time Systems," Department of Computer Science, University of York, York, England, Internal Report1995.

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

[12] José Manuel Urriza, Javier Dario Orozco and Ricardo Cayssials, "Fast Slack Stealing methods for Embedded Real Time Systems," in *26th IEEE International Real-Time Systems Symposium (RTSS 2005) - Work In Progress Session*, Miami, EEUU, 2005, pp. 12-16.

[13] J. A. Stankovic, "Misconceptions About Real-Time Computing: A Serius Problem for Next-Generations Systems," *IEEE Computer,* Nº pp. pp. 10-19, Octubre 1988.

[14] 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.

[15] A. and Audsley Burns, N., "Scheduling hard real-time systems: A review. ," *Software Engineering Journal,* Nº pp. 116–128, May 1991.

[16] Mario Aldea Rivas, "Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas," Doctor, Facultad de Ciencias, Departamento de Electrónica y Computadores, Universidad de Cantabria, Santander, España, 2002.

[17] 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.