Programa Oficial de Postgrado en Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

TESIS DE MÁSTER

INSTRUMENTACIÓN DE CÓDIGO PARA EL CÁLCULO DE TIEMPOS DE EJECUCIÓN Y RESPUESTA EN SISTEMAS DE TIEMPO REAL

Álvaro García Cuesta
gcuestaa@unican.es

Director:
J. Javier Gutiérrez García
Grupo de Computadores y Tiempo Real
Departamento de Electrónica y Computadores

Curso 2007 / 2008
Santander, Octubre de 2008

A mis amigos
repartidos por el mundo

La presente Tesis de Máster ha sido desarrollada en el marco del siguiente proyecto de investigación:

"THREAD: Soporte integral para sistemas empotrados de tiempo real distribuidos y abiertos"

Proyecto del Plan Nacional de I+D+I subvencionado por el Ministerio de Educación y Ciencia del Gobierno de España (ref. TIC2005-08665-C03-02).

Índice de contenidos

Índice de figuras

1 - Introducción

Podemos definir un sistema de tiempo real como todo aquel sistema de procesado de información que tenga que responder a un estímulo de entrada generado externamente en un periodo de tiempo finito y concreto, siendo necesario para su correcto funcionamiento no sólo la obtención de un resultado lógico sino que este sea entregado dentro del tiempo especificado [BUR97].

Los sistemas de tiempo real tienen una presencia significativa en sectores tan relevantes como, por ejemplo, la aeronáutica, el control del tráfico aéreo, la domótica, las telecomunicaciones o la industria energética. Debido a la importancia del correcto funcionamiento de estos sistemas es necesario garantizar que se cumplen los requisitos temporales impuestos, siendo por ello indispensable un exhaustivo análisis de dichos sistemas.

A la hora de analizar un sistema de tiempo real, uno de los parámetros principales a tener en cuenta es el tiempo de ejecución de peor caso (o *WCET,* Worst Case Execution Time) de los bloques de software que lo componen. Es por ello que el desarrollo y utilización de métodos y herramientas de cálculo de tiempos de peor caso que ofrezcan resultados lo más fiables posibles es extremadamente importante.

1.1 - Antecedentes

Como hemos dicho, el estudio e implementación de sistemas de tiempo real es un campo importante en el área de la computación sobre el que se lleva años investigando. El presente trabajo no sería posible sin muchos de los conocimientos esenciales generados a lo largo de estos años y que comentaremos brevemente a continuación.

1.1.1 - Análisis de planificabilidad de sistemas de tiempo real

La planificación de un sistema de tiempo real consiste en la definición de las reglas de uso de cada uno de los recursos disponibles: procesadores, memoria, redes... etc. Un sistema de tiempo real se considera planificable si, en función de la política de planificación elegida, es capaz de satisfacer todos los requisitos temporales impuestos. [PAL99]. La elección, para su aplicación en un sistema de tiempo real, de una o varias políticas de planificación que aseguren el cumplimiento de los plazos debe basarse en la realización a priori de un test de planificabilidad, siendo este el único mecanismo fiable, dado que la simulación no garantiza la comprobación de todas las posibles situaciones. Todos estos tests de planificabilidad se basan en cálculos de utilización, lo que conlleva la necesidad de poseer datos fiables sobre los tiempos de ejecución de peor caso.

Un test de planificabilidad debe ser, como mínimo, suficiente. Un test es suficiente cuando en el caso de que el test afirme que el sistema cumple los requisitos temporales, estos se cumplan bajo cualquier situación. Un posible test de planificabilidad suficiente pasa por el cálculo de cotas superiores de los tiempos de respuesta de las tareas, mediante los cuales se puede garantizar el correcto funcionamiento del sistema [PAL99]. Si los tiempos de peor caso son estimados por exceso, puede darse el caso de que un test suficiente sea pesimista, considerando como no planificable un sistema que en realidad sí lo es. Esto hace que la obtención de cotas superiores lo más fiables posible para los tiempos de ejecución de peor caso sea un punto clave para el análisis de planificabilidad.

Para sistemas monoprocesadores existen tests de planificación exactos, no así para sistemas multiprocesadores o distribuidos, donde sólo existen tests exactos para ciertos casos. Para los sistemas distribuidos gobernados por eventos no existen tests de planificabilidad exactos conocidos y todos los test aplicables a ellos son tests suficientes [PAL99].

1.1.2 - Cálculo de tiempos de ejecución de peor caso

Como ya hemos visto, el principal problema en el desarrollo de un sistema de tiempo real es asegurar que siempre cumple su funcionalidad requerida dentro de los márgenes temporales especificados y, por ello, la obtención de información precisa sobre el máximo tiempo de ejecución del software es imprescindible para asegurar que los requisitos temporales se cumplen y que el sistema de tiempo real funciona correctamente. Obtener cotas precisas para el *WCET* es un componente principal para establecer la confianza necesaria en el comportamiento temporal de un sistema.

En microprocesadores avanzados es simplemente imposible asegurar durante el testeo que todas las funciones y lazos en un camino han exhibido su tiempo de peor caso simultáneamente, lo que significa que el testeo solamente no puede proporcionar el nivel de confianza en el comportamiento temporal del sistema necesario para una puesta en funcionamiento segura [PET07] [BER02].

1.1.2.1 - Métodos de obtención de *WCETs*

Podemos definir el *WCET* como el máximo tiempo que lleva ejecutar un programa, tarea o sección de código en un hardware concreto. Obtener valores fiables para el *WCET* es bastante difícil, dada la complejidad del hardware actual y la dificultad de generar tests que aseguren representar estos casos. Tradicionalmente los diferentes métodos para obtener un valor para el *WCET* se dividen en dos grupos: “basados en análisis estático” y “basados en medidas”, teniendo ambos grupos sus ventajas e inconvenientes. En la bibliografía se encuentra un tercer grupo de métodos basados en características de los dos anteriores, denominados “híbridos” [RAP08b].

1.1.2.1.1 - Métodos basados en análisis estático

La aproximación básica al cálculo del *WCET* mediante análisis estático conlleva identificar los segmentos que conforman posibles caminos a través del código. Usando información acerca de estos subcaminos y cómo pueden combinarse, junto a un modelo preciso del comportamiento del hardware, es en teoría posible predecir el tiempo de peor caso. Por tanto, esta clase de métodos no depende de ejecuciones del código en el hardware real o en un simulador, sino que analiza el código, lo combina con algún modelo del sistema y obtiene valores para el *WCET*. Los sistemas basados en análisis estático tratan de obtener, matemáticamente, un límite superior para el *WCET* cuando el programa recorre el camino más largo, en tiempo, dentro del código y dependen de un modelo temporal del hardware lo más preciso posible. Los procesadores actuales han alcanzado una complejidad que dificulta en gran manera la obtención de este modelo.

La complejidad de obtener un modelo del procesador depende fuertemente del tipo de procesador Para procesadores de 8 y 16 bits el modelo es relativamente simple, pero para procesadores más actuales y avanzados la obtención de un modelo se hace altamente compleja [WIL07]. Los procesadores típicos contienen varios componentes que hacen que el tiempo de ejecución dependa del contexto de ejecución, como memoria, caché, *pipelines*, predicción de salto... haciendo que la ejecución de una instrucción dependa de la historia de la ejecución del programa. Para muchos procesadores avanzados de 32 bits, el único modelo realmente preciso del comportamiento temporal del hardware es el procesador en sí mismo, haciendo el análisis estático impracticable [BER03].

1.1.2.1.2 - Métodos basados en medidas

En los métodos basados en medidas el programa es sometido a las condiciones de test más exhaustivas posible, almacenando el tiempo de peor caso obtenido durante dichas medidas. Estos métodos tratan de obtener valores del *WCET* a partir de medidas producidas al ejecutar el programa en el hardware real (o en un simulador) para un conjunto de entradas y situaciones. Estas medidas suponen sólo un subconjunto de todas las posibles ejecuciones con lo que, en general, sólo es posible obtener estimaciones y distribuciones de probabilidad, no tiempos máximos absolutos. Si se supieran las condiciones de entrada necesarias para el peor caso, sólo se necesitaría una ejecución para obtener el *WCET*.

Otras aproximaciones, que algunos denominan métodos híbridos [RAP08a] [RAP08b] y otros engloban dentro de los métodos basados en medidas [WIL07], se basan en obtener medidas para trozos de código (bloques), siendo estas medidas después combinadas y analizadas para obtener una estimación del *WCET*. Así, al igual que en los métodos estáticos, se puede usar análisis de control de flujo para encontrar todos los posibles caminos. Esta solución puede incluir todos los caminos, pero todavía sería imprecisa si los valores medidos no son fiables (dado que los valores medidos sólo corresponden a un contexto de ejecución concreto del procesador).

1.1.2.2 - Comparación entre métodos basados en análisis y en medidas

Los métodos estáticos calculan cotas para los tiempos de ejecución. Usan análisis de control de flujo y cálculo de límites para cubrir todos los posibles caminos de ejecución en el código y abstracción para cubrir todas las posibles dependencias de contexto en el comportamiento del procesador. El precio que pagan es la necesidad de obtener modelos específicos del comportamiento del procesador, y posiblemente valores imprecisos al sobrestimar los límites del *WCET*. A favor de los métodos estáticos está el hecho de que el análisis se puede hacer sin necesidad de ejecutar el programa a analizar.

Los métodos basados en medidas sustituyen el análisis del comportamiento del procesador con las medidas. Por tanto, a no ser que todos los caminos de ejecución sean medidos o el procesador sea suficientemente simple para poder asegurar que las medidas se corresponden a las de peor caso, se pueden perder algunos contextos clave y obtenerse medidas imprecisas. Para la etapa de cálculo de estimaciones, estos métodos pueden usar análisis del control de flujo para incluir todos los posibles caminos de ejecución, o simplemente pueden usar los caminos observados (con lo que se obtendrían de nuevo medidas poco fiables). La ventaja proclamada por estos métodos es que son más simples de aplicar en procesadores modernos, porque no necesitan un modelo del hardware, produciendo estimaciones del *WCET* más cercanas al valor real que los métodos estáticos, especialmente en aplicaciones y/o procesadores complejos, ya que el principal problema de los métodos estáticos es modelar el comportamiento del hardware [WIL07] [HEI07] [RAP08a].

1.1.3 - Herramientas para el cálculo de tiempos de peor caso

Existe un número reducido de herramientas comerciales y prototipos de investigación para el cálculo de tiempos de peor caso [WIL07]. Algunas se basan completamente en métodos estáticos, como *aiT* [AIT08], *BoundT* [BOU08] y los prototipos programados en la Universidad Estatal de Florida [FSU08], la TU de Viena [TUV08], la Universidad Nacional de Singapur [NSU08] o *HEPTANE* [HEP08], existiendo otras basadas mayormente en métodos estáticos pero con aspectos muy concretos basados en medidas, como *SWEET* [SWE08] o el prototipo de la Universidad de Chalmers [CHA08], hasta llegar a *RapiTime* [RAP08a], que se apoya más que las anteriores en las medidas.

1.2 - Modelado de sistemas de tiempo real

Un sistema de tiempo real tipo puede estar basado en uno o más procesadores que interactúan con el entorno y que están conectados entre sí a través de un bus o una red de comunicaciones [PAL99]. El software que forma parte del sistema se compone de varios *threads* concurrentes localizados en los procesadores que se intercambian mensajes a través de las redes de comunicaciones, pudiendo también ocurrir que los *threads* de un mismo procesador intercambien datos mediante mecanismos de sincronización de memoria compartida [PER08].

Este tipo de sistemas se puede modelar como un conjunto de transacciones que se pueden ejecutar tanto en entornos monoprocesadores como multiprocesadores o distribuidos. Cada transacción se activa a partir de uno o más eventos externos y representa una serie de actividades que serán ejecutadas en el sistema como tareas en los procesadores o mensajes en las redes de comunicación. En el modelo de la transacción las actividades generan eventos internos que pueden por su parte activar otras actividades dentro de la transacción. Este modelo es capaz de representar sistemas gobernados por eventos y sobre él se han definido algoritmos de análisis de planificabilidad y de asignación de parámetros de planificación que permiten garantizar el cumplimiento de los requisitos de tiempo real.

La aplicación de las técnicas de análisis normalmente se hace mediante herramientas que implementan los algoritmos basadas en modelos del sistema. En el grupo de *Computadores y Tiempo Real* en el que realiza el presente trabajo, se ha propuesto *MAST* [GON01][MAS08] como modelo abierto sobre el que implementar las herramientas de análisis de sistemas de tiempo real. El conjunto de herramientas de *MAST* se ofrece como código abierto y de libre distribución. Existen otras herramientas comerciales que también implementan algunas de las técnicas integradas en *MAST*. Así por ejemplo, *SymTA/S* [SYM08] integra herramientas de análisis y optimización de unidades de control electrónico (ECUs), buses/redes y sistemas de tiempo real embebidos, que incluye además el tratamiento de sistemas gobernados por tiempo (*time-triggered*). En cualquiera de los casos estos modelos y herramientas se fundamentan en el conocimiento de los *WCETs* de las actividades que se modelan.

1.3 - Software de tiempo real

La arquitectura software de un programa o sistema de computación es la estructura o conjunto de estructuras del sistema, que abarca los componentes software, la propiedades de esos componentes visibles desde el exterior y las relaciones entre ellos. En la mayoría de los sistemas actuales esta arquitectura se compone de dos grandes capas: el Sistema Operativo que ofrece los servicios necesarios para la gestión del hardware a la vez que independiza a las capas superiores del mismo, y la Aplicación o Aplicaciones desarrolladas en algún lenguaje de programación concreto. Desde el punto de vista de tiempo real, estas dos capas requieren un tratamiento claramente diferenciado a la hora de aplicar las técnicas de obtención de *WCETs*.

Así, el software perteneciente al nivel de Aplicación se crea a través de lenguajes de programación como, por ejemplo, ADA, Java o C, y el acceso a sus fuentes es inmediato para su programador, siendo por ello relativamente fácil la instrumentación de su código para su caracterización temporal, y la integración con herramientas capaces de extraer los *WCETs* de esta instrumentación. Sin embargo, los sistemas operativos de uso común no siempre ponen a disposición del usuario su código fuente, dificultando seriamente la caracterización temporal de sus servicios para el posterior análisis del sistema. Es posible que incluso el cálculo de *WCETs* en aplicaciones multitarea pueda

requerir el acceso al sistema operativo. En esta Tesis se va a trabajar con *MaRTE OS* [MAR08], sistema operativo desarrollado en el grupo de investigación de *Computadores y Tiempo Real* en el que se realiza este trabajo y distribuido bajo licencia *GPL*, lo que permite, por tanto, el acceso a sus fuentes.

MaRTE OS es un sistema operativo de tiempo real para aplicaciones empotradas que sigue el perfil "Sistema de Tiempo Real Mínimo" definido en el estándar *POSIX.13* [POS03]. Proporciona el soporte necesario para ejecutar aplicaciones concurrentes sobre una máquina desnuda, proporcionando, además de soporte multitarea, facilidades para la inicialización del sistema, el lanzamiento de la aplicación y la gestión de dispositivos. La funcionalidad descrita en el perfil *POSIX* antes mencionado consiste básicamente en el soporte para la concurrencia a nivel de *threads*, no incluyendo soporte para sistema de ficheros ni la existencia de múltiples procesos. Los principales servicios proporcionados por *MaRTE OS,* incluidos algunos que no forman parte del perfil *POSIX.13*, son [ALO05]:

- Gestión de *threads*: creación, finalización, atributos, ...
- Planificación basada en prioridades: se soportan las políticas *FIFO*, *round-robin* y servidor esporádico.
- *Threads* independientes y sincronizados.
- *Mutexes* y variables condicionales.
- Semáforos.
- Señales de tiempo real.
- Reloj monotónico y de tiempo real. Temporizadores.
- Relojes y temporizadores de tiempo de ejecución.
- Servicios de temporización de *threads*: suspensión absoluta y relativa.
- Dispositivos de Entrada/Salida.
- Gestión de memoria dinámica.
- Planificación definida por la aplicación
- Gestión de interrupciones a nivel de aplicación.
- Marco para la instalación de manejadores de dispositivo.

1.4 - Objetivos

El presente trabajo se centra en la instrumentación de código para el cálculo de tiempos de ejecución de peor caso en sistemas de tiempo real, concretamente en el estudio de la aplicación de herramientas de cálculo de tiempos de ejecución de peor caso a la instrumentación de código de tiempo real tanto a nivel de aplicación como de sistema operativo.

Otro de los objetivos buscados, una vez completados los arriba citados, es el estudio del proceso de integración de herramientas de cálculo de tiempos de peor caso con herramientas de modelado y análisis de planificabilidad.

Las herramientas elegidas que se emplearán en la realización de esta Tesis de Máster serán:

- *MAST* como herramienta de análisis de planificabilidad, debido a su disponibilidad al ser una herramienta de código abierto y libre distribución y al conocimiento que se tiene sobre su funcionalidad al haber sido desarrollada en el grupo de investigación de *Computadores y Tiempo Real* en el que se realiza esta Tesis.

- *RapiTime* como herramienta de cálculo de tiempos de peor caso. Esta herramienta ha sido elegida dada su disponibilidad en el grupo y la ausencia de alternativas de código abierto que fueran fiables.

- El sistema operativo de tiempo real seleccionado ha sido *MaRTE OS*, también de libre distribución y desarrollado en este grupo, con lo que se posee un mayor conocimiento y accesibilidad a sus fuentes.

Nuestros objetivos específicos serán, por tanto:

- Estudiar las capacidades para el cálculo de tiempos de peor caso de la herramienta *RapiTime.*

- Integrar *RapiTime* en una plataforma de tiempo real basada en el sistema operativo *MaRTE OS* y desarrollar una metodología para la instrumentación de código de tiempo real a nivel de aplicación para los lenguajes de programación ADA y C.

- Desarrollar una metodología de uso de la herramienta *RapiTime* para la instrumentación del propio código fuente del sistema operativo *MaRTE OS* de forma que se puedan obtener valores para las características temporales de los servicios que ofrece.

- Estudiar la viabilidad de la integración de los datos generados con *RapiTime* en el conjunto de herramientas de análisis de planificabilidad *MAST*.

2 - Estudio de la herramienta de cálculo de tiempos de ejecución de peor caso *RapiTime*

RapiTime es una herramienta comercial para el cálculo de tiempos de ejecución de peor caso que trabaja sobre una representación en forma de árbol del programa. Esta estructura puede ser generada a partir del código fuente o directamente del análisis de los ejecutables [WIL07]. Los datos temporales de los bloques individuales se generan a partir de medidas exhaustivas sobre el sistema real. *RapiTime* no sólo calcula una estimación del valor del *WCET* como un valor numérico, sino también la distribución de probabilidad de camino más largo del programa (y de otras subunidades). Esta distribución tiene un dominio acotado, siendo la cota superior absoluta el *WCET* estimado.

Aunque no entra dentro de los objetivos de este trabajo profundizar en el funcionamiento teórico de la herramienta, cabe mencionar que *RapiTime* hace uso de la *teoría de cópulas*, una teoría general para construir distribuciones conjuntas y describir la estructura de las dependencias entre variables aleatorias [QUE03].

Los datos de entrada que necesita *RapiTime* son los ficheros fuentes en lenguajes ADA o C, o los ejecutables para las plataformas objetivo, y los valores obtenidos en la fase de toma de medidas. La información temporal es capturada sobre el sistema en ejecución bien usando una librería de instrumentación software y almacenando los datos en memoria (que es el método utilizado en este trabajo) o mediante hardware externo. El proceso de instrumentación del código consiste en la introducción, en puntos relevantes, de funciones que nos permitan obtener información sobre el instante temporal en el que se ha pasado por una posición durante la ejecución del programa.

2.1 - Herramientas

El conjunto de herramientas que integra *RapiTime* está formado por los siguientes programas de línea de comandos (versión 2.0rc1):

- **cins/adains:** Instrumentación automática de código fuente C/ADA y extracción de información estructural.
- **xstutils:** Análisis estructural.
- **traceutils:** Preprocesado de las trazas temporales.
- **traceparser:** Generación de datos de tiempo de ejecución medidos a partir de la información estructural y de trazas.
- **wcalc:** Cálculo de tiempo de ejecución de peor caso y generación de datos de tiempo de ejecución computados.

El resultado final tras usar las herramientas de línea de comandos, en la versión actual de *RapiTime,* es una base de datos *SQLite* con la información sobre los tiempos de ejecución a la que se puede acceder usando un *plug-in* para el entorno de desarrollo *Eclipse* (en anteriores versiones el informe final se generaba en formato *HTML*).

2.2 - Proceso detallado para la obtención de *WCETs*

El proceso de análisis usando *RapiTime*, mostrado en la Figura 2.1, comprende seis fases [RAP08b]:

1. *Construcción*
2. *Análisis estructural*
3. *Testeo y generación de trazas*
4. *Procesado de las trazas*
5. *Cálculo del WCET*
6. *Visionado del informe*

A continuación se describen estas fases en mayor detalle para el caso de una aplicación escrita en C y una versión de *RapiTime* igual o superior a la 2.0, siendo el caso ADA muy similar (usando *adains* en vez de *cins* para instrumentar automáticamente el código).

2.2.1 - Construcción

El primer paso es crear una versión instrumentada de la aplicación. Esto se puede hacer de forma manual o automática, añadiendo puntos de instrumentación al código fuente C. Para instrumentar el código automáticamente, este debe ser previamente preprocesado (código C puro con todas las *macros* e *includes* expandidos por el preprocesador C). El código preprocesado se pasa a la herramienta *cins*, que añadirá automáticamente los puntos de instrumentación necesarios de acuerdo a las indicaciones prefijadas por el usuario mediante un fichero de configuración o *pragmas* insertados en el código. El código instrumentado, junto a la librería de *RapiTime* (*rpt.c)* se compila y enlaza usando un compilador estándar.

Como parte del proceso de construcción, la herramienta *cins* extrae información estructural a partir de cada fichero fuente C, almacenando esta información en un archivo *.xsc* (*extended syntax tree components*).

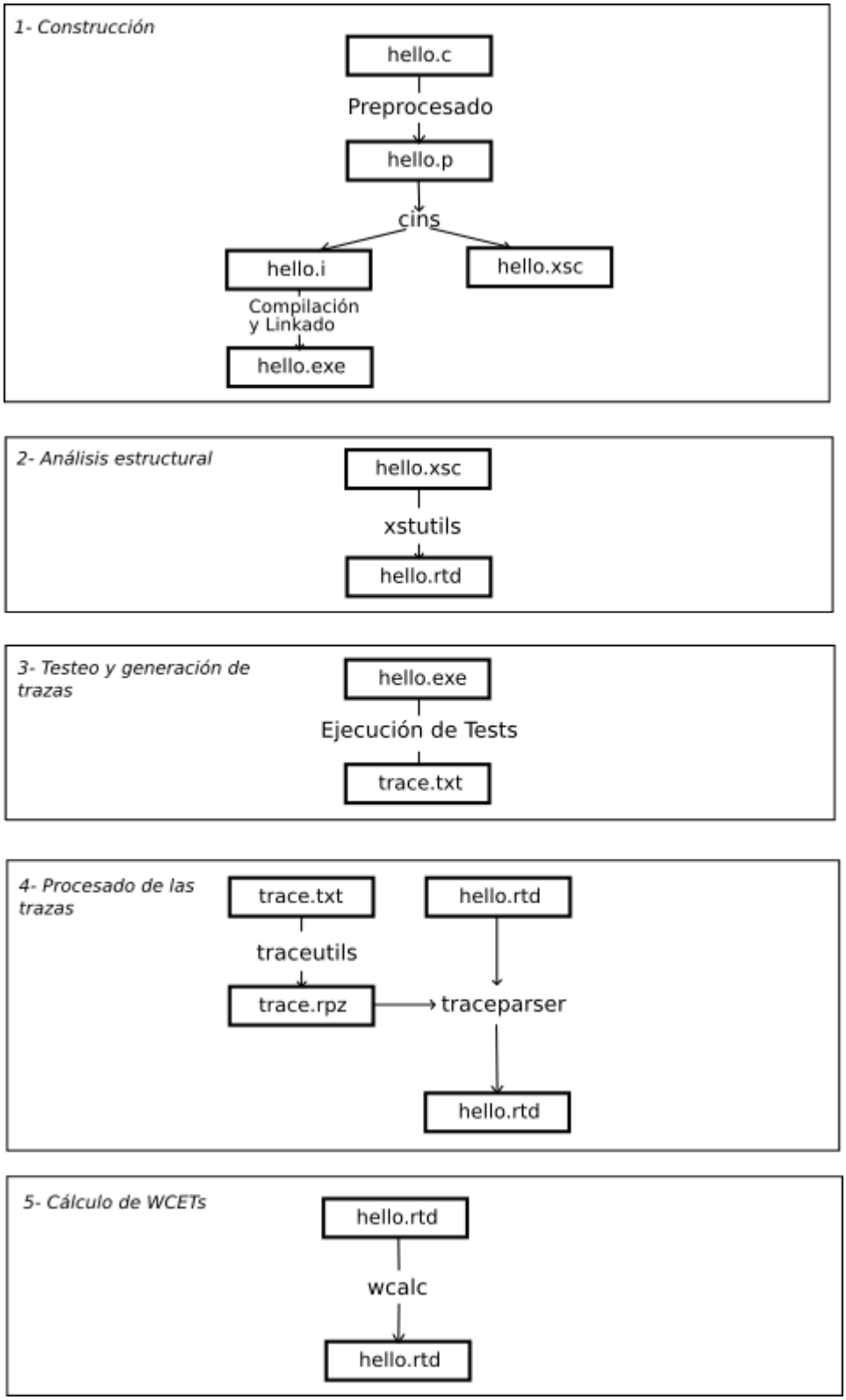

Figura 2.1. Proceso de obtención de WCETs.

2.2.2 - Análisis estructural

La herramienta *xstutils* se usa para analizar la información estructural contenida en los ficheros *.xsc* generados con *cins*. Así, *xstutils* genera una descripción de la estructura general del código llamada árbol de sintaxis extendida, que se almacena en una base de datos *SQLite* con extensión *.rtd* (en versiones anteriores se hacía en un fichero diferente) y que se irá actualizando en los siguientes pasos.

2.2.3 - Testeo y generación de trazas

El programa generado en el *paso 1* es ejecutado en la plataforma objetivo y sometido a una serie de tests, cuyo objetivo es ejercitar el mayor número de subcaminos a través del código bajo diferentes condiciones.

En una aproximación basada en instrumentación, como la que nos ocupa, la información de trazas es almacenada como el binomio *identificador de punto de instrumentación* más *timestamp (marca de tiempo)*. Esta información de traza es extraída al finalizar cada test y almacenada en un fichero para su posterior análisis. De forma alternativa, si el hardware lo permite, el punto de instrumentación puede ser simplemente escrito en un puerto de salida monitorizado por un dispositivo hardware de captura de trazas, como por ejemplo un analizador lógico.

2.2.4 - Procesado de las trazas

La herramienta *traceutils* preprocesa los ficheros que contienen las trazas de información temporal generadas durante el testeo, es decir, filtra los datos y corrige situaciones como *wrap-arounds* (situación que ocurre cuando el identificador de punto de interrupción alcanza su valor máximo y el siguiente vuelve a empezar por el valor 0). A continuación, *traceutils* genera ficheros de trazas comprimidos en formato binario (*.rpz*).

La herramienta *traceparser* parsea estas trazas binarias usando además la información estructural generada por *xstutils*. El resultado obtenido es una distribución de tiempos de ejecución observados para los elementos que conforman el programa (funciones, subfunciones, lazos, bloques y subcaminos únicos entre puntos de decisión en el código). Los datos de tiempo de ejecución se almacenan en la base de datos *.rtd*.

2.2.5 - Cálculo del *WCET*

La herramienta *wcalc* recibe como entrada la base de datos y tras realizar los cálculos de tiempos de peor caso la actualiza.

2.2.6 - Visionado del informe

La forma de visionar el informe de resultados cambia entre diferentes versiones de la herramienta. En la versión actual de *RapiTime*, 2.0rc1, se incluye un *plug-in* para *Eclipse* que se encarga de mostrar al usuario los datos almacenados en la base de datos (otra novedad de la nueva versión). Esta forma de visionado ha sustituido a la utilizada en anteriores versiones, que generaban un informe en formato *HTML* a partir de los archivos binarios que iban generando las herramientas.

2.2.6.1 - El *plug-in* para *Eclipse*

RapiTime, en sus versiones más recientes, genera la información de análisis en forma de una base de datos *SQLite* que se presenta al usuario mediante una interfaz gráfica de usuario (GUI) llamada *reportviewer* similar a la mostrada en la Figura 2.3. El informe creado por *RapiTime* contiene información sobre las propiedades del programa extraídas por *xstutils* y la información temporal obtenida por *traceparser.*

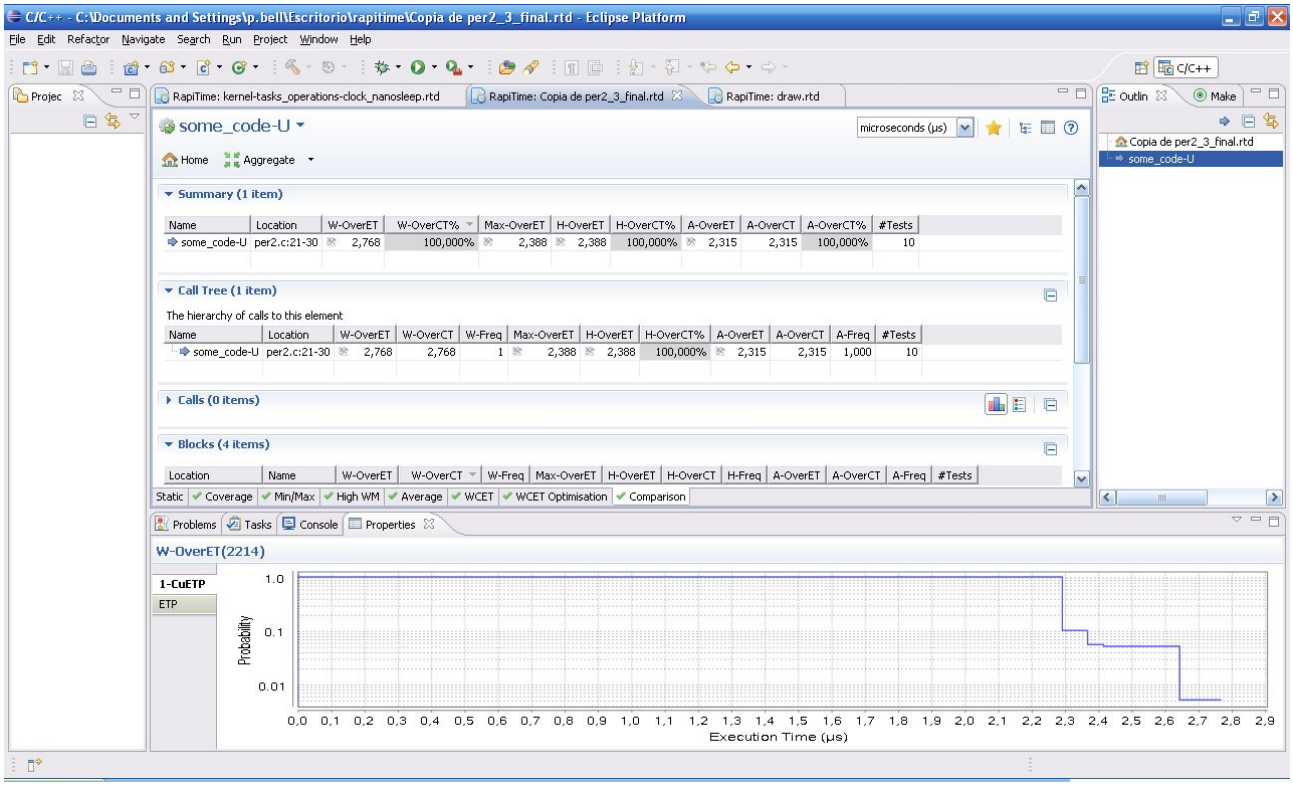

Figura 2.2. Reportviewer.

El visor *reportviewer* está construido sobre la arquitectura interna para *plug-ins* de la plataforma *Eclipse*. En este apartado nos concentraremos en los componentes de la GUI relevantes para el visionado del informe, para más información sobre *Eclipse* el lector deberá referirse a la web del proyecto *Eclipse* [ECL08].

Para invocar al visor, basta con hacer doble *click* en la base de datos *.rtd* o, simplemente, abrirla desde una instancia abierta de *Eclipse*. De esta forma se abrirá la interfaz del visor. El visor es una ventana multisección no editable que consta de los siguientes elementos:

- Una sección principal denominada *report view* en la que se muestran los informes generados con *RapiTime*. Cada vez que se abre un archivo se abre una nueva pestaña con su correspondiente informe, permitiendo así tener varios informes abiertos al mismo tiempo. Cada informe en la sección *report view* se divide en una serie de subsecciones accesibles por sus respectivas pestañas. De estas subsecciones una de las que más datos nos puede aportar es la denominada *Comparison*. En ella se nos presentan varios de los valores del informe relevantes para este trabajo, como son:

    - **W-OverET**: se trata del *WCET* **computado** para el elemento seleccionado. Este tiempo incluye todas las llamadas a subfunciones.

    - **Max-OverET**: es el máximo tiempo de ejecución **medido** para el elemento seleccionado a partir de los datos obtenidos de todas las trazas. Este tiempo incluye todas las llamadas a subfunciones.

    - **A-OverET**: representa el tiempo de ejecución medio **medido** para el elemento seleccionado según los datos de todas las trazas, incluyendo subfunciones.

  Otra subsección con datos útiles para este trabajo es la denominada *Min/Max*, donde podemos obtener el **Min-OverET**, correspondiente al mínimo tiempo de ejecución **medido** para el elemento seleccionado. Este dato, junto a los tiempos de ejecución de peor caso y medio antes citados, se corresponden, como veremos más tarde, con algunos de los datos relativos a tiempos de ejecución requeridos por *MAST*.

- Una sección llamada *outline*, que permite navegar por el árbol formado por todas las llamadas a funciones a partir de la función raíz del análisis.

- Una sección propiedades que o bien resume propiedades particulares de elementos seleccionados en la vista *report view*, o muestra el perfil de tiempo de ejecución (ETP) de un segmento de código en particular.

- Botones atrás y adelante que permiten la navegación por lugares previamente visitados, de forma similar a como lo haríamos con un navegador.

- Un botón *bookmarks* que permite marcar páginas concretas del informe.

- Un selector de unidad de tiempo que muestra la granularidad con la que la información temporal se muestra en el informe.

3 - Instrumentación de código a nivel de aplicación en plataforma de tiempo real

Uno de los objetivos de este trabajo es establecer una metodología para la instrumentación, mediante *RapiTime*, de código a nivel de aplicación ejecutándose sobre el sistema operativo *MaRTE OS*. Para que esto sea posible *RapiTime* necesita ser integrada con el sistema operativo, de forma que el código instrumentado con ella sea capaz de generar ficheros de trazas temporales de acuerdo a un formato legible por la herramienta, siendo por ello necesario crear una librería de instrumentación adaptada a nuestro sistema.

Otro requisito inicial es que los cambios de contexto y las interrupciones queden señalados en las trazas, de forma que *RapiTime* pueda así discernir qué tiempos corresponden a qué tareas. Esto se consigue posicionando llamadas a funciones específicas de la librería de instrumentación antes mencionada en puntos clave del código del sistema operativo.

También ha sido necesaria la creación de una serie de ficheros de configuración, denominados *filtros*, cuya función es indicar a algunas de las herramientas de *RapiTime* datos relevantes para el proceso de instrumentación y cálculo.

3.1 - Implementación de la librería de funciones de *RapiTime* para *MaRTE OS*

Cuando *RapiTime* instrumenta el código fuente de un programa, lo que hace es añadir ciertas llamadas a funciones cuyo objetivo es generar puntos de instrumentación, que más tarde indicarán a la herramienta la posición dentro del código y el tiempo en el que se pasó por ese punto. Además existen ciertas funciones que generan puntos de instrumentación con identificadores concretos que sirven para señalar cambios de contexto e interrupciones. Por último, es necesaria una función encargada de enviar las trazas temporales generadas en el sistema real para su posterior análisis.

Estas funciones deben ser implementadas para cada sistema concreto, siendo parte de este trabajo el desarrollo de un archivo C llamado *rpt.c* (ver Anexo 1), y su versión ADA, con la implementación de dichas funciones. A continuación se muestran las interfaces C y ADA correspondientes:

```
/* INTERFAZ C */

#ifndef __RPT_H
#define __RPT_H

  void RPT_Ipoint( unsigned long i );

  void RPT_Context_Switch_To( unsigned long i );

  void RPT_Intrpt_Entry( void );
```

```
  void RPT_Intrpt_Exit( void );
  void RPT_Output_Trace( void );
#endif
```

```
--  INTERFAZ ADA
with Interfaces;
package rptinstrlib is
   type RPT_Ipoint_ID_Type is new Interfaces.Unsigned_32;
   procedure RPT_Ipoint (i : RPT_Ipoint_ID_Type);
   pragma Import (C, RPT_Ipoint, "RPT_Ipoint");
   procedure RPT_Context_Switch_To (i : RPT_Ipoint_ID_Type);
   pragma Import (C, RPT_Context_Switch_To, "RPT_Context_Switch_To");
   procedure RPT_Intrpt_Entry;
   pragma Import (C, RPT_Intrpt_Entry, "RPT_Intrpt_Entry");
   procedure RPT_Intrpt_Exit;
   pragma Import (C, RPT_Intrpt_Exit, "RPT_Intrpt_Exit");
   procedure RPT_Output_Trace;
   pragma Import (C, RPT_Output_Trace, "RPT_Output_Trace");
end rptinstrlib;
```

Será necesario, por tanto, enlazar estos archivo con el código instrumentado en el momento de generación del ejecutable. Además, gracias a estas funciones, los datos generados se irán almacenando en memoria para, posteriormente, ser enviados por red a otro ordenador con entorno *Linux*, donde se guardarán en un fichero.

3.1.1 - Función *RPT_Ipoint*

Los puntos de instrumentación incluidos en el código se corresponden en las trazas temporales con un binomio *identificador-tiempo*. Cada vez que en el código aparece una llamada a la función *RTP_Ipoint* (añadida bien manualmente por el usuario o automáticamente por la herramienta) se genera uno de estos puntos, que sirven para relacionar una posición dentro del código con un instante temporal. Mientras que el identificador es un valor de entrada de la función, el instante

temporal es generado en tiempo de ejecución, en nuestro caso mediante instrucciones ensamblador que nos permiten acceder al valor del *tsc*, "Time Stamp Counter", un registro presente en todos los procesadores *x86* a partir del *Pentium*, que cuenta el número de *ticks* desde el arranque. Para que esta función sea segura en el caso de que lleguen interrupciones durante su ejecución se utiliza la pareja *cli/sti* de instrucciones ensamblador.

Un ejemplo, en hexadecimal, de la estructura básica generada por un punto de instrumentación sería el siguiente:

```
     ID          Timestamp

 [fbff 0000]   [a9ee fd59]
```

correspondiente a un punto con identificador con valor 65531 en decimal (*FBFF* en sistema hexadecimal y formato *little-endian*)

3.1.2 - Función *RPT_Context_Switch_To*

Para que la herramienta pueda seguir el flujo del programa y asignar correctamente los datos a las tareas correspondientes, es necesario indicar ciertos momentos clave en dicho flujo. Uno de ellos es el momento de cambio de contexto, en el cual es necesario almacenar el nuevo identificador de tarea. Para ello hay que colocar una llamada a la función *RPT_Context_Switch_To* en el punto preciso del código del sistema operativo donde ese cambio de contexto sucede. En *MaRTE OS* se ha estimado que el punto correcto para ello se encuentra en el paquete *Kernel.Scheduler*, al final de la función *Do_Scheduling*. El cambio de contexto se lleva a efecto en las funciones *Change_To_Context* o *Context_Switch*, dependiendo de si hay que guardar el contexto de la tarea actual o no. Por tanto, justo antes de ejecutarlas lo indicamos con un punto de instrumentación, como se puede ver indicado en negrita en el listado siguiente:

```
[...]

--  RapiTime awareness
with rptinstrlib;

package body Kernel.Scheduler is

 [...]

         ------------
         --  The context switch is actually carried out in
         --  'Change_To_Context' or 'Context_Switch'.
         ----
         Running_Task := Heir_Task;
         if Old_Task.Magic = TERMINATED then
            --  RapiTime begin
            rptinstrlib.RPT_Context_Switch_To (
  rptinstrlib.RPT_Ipoint_ID_Type (TCB_Ac (Running_Task).Id));
            --  RapiTime end
            HWI.Change_To_Context (New_Task => Heir_Task.Stack_Ptr'Address);
```

```
            Pool_TCBs.Release_TCB (Old_Task);

         else
            --  RapiTime begin
            rptinstrlib.RPT_Context_Switch_To (
   rptinstrlib.RPT_Ipoint_ID_Type (TCB_Ac (Running_Task).Id));
            --  RapiTime end
            HWI.Context_Switch (Old_Task => Old_Task.Stack_Ptr'Address,
                                New_Task => Heir_Task.Stack_Ptr'Address);
         end if;

      elsif After_Timed_Event_Expiration then

         After_Timed_Event_Expiration := False;
         TE_T.Reprogram_Timer_After_Expiration (Running_Task);

      end if;  --  Running_Task /= Heir_Task
   end Do_Scheduling;
   pragma Inline (Do_Scheduling);

[...]
```

Un ejemplo de la estructura básica generada por un punto de instrumentación correspondiente a un cambio de contexto sería el siguiente:

```
0300 0000 b53a 5059 0100 0000 0000 0000
```

donde los 4 primeros bytes se corresponden al identificador asignado a un cambio de contexto (3), los 4 siguientes la marca temporal, los 4 siguientes al identificador de tarea a comenzar (en este caso la 1) y los 4 restantes contienen ceros.

3.1.3 - Funciones *RPT_Intrpt_Entry* y *RPT_Intrpt_Exit*

Para indicar cuándo nos encontramos en un manejador de interrupción, de forma que la herramienta sepa discernir qué tiempo corresponde a una tarea y cuál no, debemos colocar al inicio y final del manejador llamadas a las funciones *RPT_Intrpt_Entry* y *RPT_Intrpt_Exit* respectivamente. Lo ideal sería encontrar un punto en el código por el que pasaran todos los manejadores de interrupción antes y después de ejecutar su código y colocar ahí estas funciones, de forma que no haya que incluirlos en cada manejador de interrupción, pero por ahora no ha sido posible encontrar los puntos que cumplan estas características.

3.1.4 - Función *RPT_Output_Trace*

Cuando se ejecuta esta función, los datos almacenados en memoria son enviados desde el ordenador donde se están ejecutando los tests hacia una máquina *GNU*/*Linux* en la que se almacenarán en forma de fichero binario y se analizarán con *RapiTime*. El array donde se almacenan los datos se va recorriendo y enviando los datos haciendo uso del *logger* de ejemplo que se proporciona con *MaRTE OS*, hasta que todos los datos almacenados han sido enviados. La llamada a esta función debe hacerse como paso final del test.

A continuación se muestra un extracto de un fichero de trazas generado tras usar esta función, visto en un editor hexadecimal:

```
00000000: 0300 0000 64a6 7296 0100 0000 0000 0000  ....d.r.........
00000010: 0100 0000 0bfd ec96 0200 0000 fd10 ed96  ................
00000020: 0300 0000 df14 ed96 0000 0000 0000 0000  ................
00000030: 0300 0000 2f91 ed96 0100 0000 0000 0000  ..../...........
00000040: 0100 0000 b1c3 ed96 0200 0000 89c4 ed96  ................
00000050: 0300 0000 7ec5 ed96 0000 0000 0000 0000  ....~...........
00000060: 0300 0000 7fcd ed96 0100 0000 0000 0000  ................
00000070: 0100 0000 1b00 ee96 0200 0000 cb00 ee96  ................
00000080: 0300 0000 9301 ee96 0000 0000 0000 0000  ................
00000090: fbff 0000 cf9a 2097 0a00 0000 1ad3 2097  ...... ....... .
000000a0: 0a00 0000 7fd3 2097 0a00 0000 d8d3 2097  ...... ....... .
000000b0: 0b00 0000 3bd4 2097 0d00 0000 c7d8 2097  ....;. ....... .
000000c0: 0e00 0000 8bda 2097 1000 0000 80f3 2097  ...... ....... .
000000d0: 0e00 0000 d2f3 2097 1000 0000 50fe 2097  ...... .....P.
```

3.2 - Filtros

Entre los parámetros de entrada que necesitan algunas de las herramientas de línea de comandos están los ficheros de filtros. Estos ficheros contienen instrucciones referentes a la forma de tratar la información de las trazas. En nuestro caso hemos tenido que crear tres ficheros de filtro: *scope.flt* para usar con las herramientas *xstutils* y *traceutils*; y *little_endian.flt* y *marte.flt* para usar con *traceutils.* A continuación se dará una breve descripción de estos filtros, pudiéndose acceder a su implementación final en el Anexo 2.

3.2.1 - scope.flt

Este filtro simplemente se asegura de que los identificadores de las trazas analizadas se encuentren comprendidas entre los valores válidos.

3.2.2 - little_endian.flt

Los datos de las trazas son almacenados en *little endian*, este filtro se encarga de decirle a la herramienta el orden en que debe leer los bytes de la traza para formar el binomio *identificador-tiempo*.

3.2.3 - marte.flt

Este filtro se encarga de separar los tiempos debidos a cada tarea, eliminando los tiempos debidos a interrupciones y diferenciando las tareas gracias al punto de instrumentación insertado en el momento de un cambio de contexto.

3.3 - Preprocesado de ficheros fuente

Como se vio en el Capítulo 2, las herramientas de instrumentación automática de código *cins* y *adains* necesitan que los ficheros con código fuente sean previamente tratados. En el caso de *cins*, los ficheros C deben ser preprocesados (código C puro con todas las *macros* e *includes* expandidos por el preprocesador C), lo que en el caso de código para la plataforma *MaRTE OS* se realiza como se muestra a continuación:

```
gcc -Wall -E -nostdinc -I/home/alvaro/bin/marte/include in.c
> out.c
```

La entrada de la herramienta de instrumentación *adains* debe ser un fichero *.adt,* que se obtiene a partir del fichero ADA de la siguiente forma:

```
mgcc -c -gnatc -gnatt  in.adb
```

3.4 - Configuración del navegador web *Firefox* en *GNU/Linux* para el correcto visionado de los informes

Durante la realización de esta Tesis se ha trabajado con diferentes versiones de *RapiTime*. Mientras que en las últimas versiones el informe final se muestra haciendo uso de plug-in para Eclipse, hasta la versión 2.0rc1 los informes se visionaban como documentos *HTML* en el navegador web, siendo necesario, para la correcta generación de las gráficas *ETP*, Execution Time Profile, de los informes configurar el navegador para que usara la herramienta *plotgen,* incluida en la carpeta *bin* de la instalación de *RapiTime*, para abrir los archivos *.rpp* (tipo *MIME application/rpp*).

Mientras que con el navegador *Opera* esto se conseguía fácilmente a través de sus opciones de configuración de tipos *MIME*, con *Firefox Rapita* sugería modificar una serie de archivos *xml*. El problema de este método es que resulta menos directo y no siempre funciona correctamente. Para solucionarlo, se ha encontrado una extensión para *Firefox* que facilita la configuración de los tipos *MIME*, llamado *MIME Edit* [MIM08]

Haciendo uso de esta extensión resulta muy sencillo incluir un nuevo tipo *MIME* con los siguientes datos:

- **MIME Type**: *application/rpp*
- **Description**: *rpp*
- **Extension**: *rpp*
- **Open with**: *path/to/plotgen*

4 - Instrumentación de código de sistema operativo de tiempo real

La caracterización temporal de los servicios de un sistema operativo de tiempo real es esencial a la hora de analizar un sistema de tiempo real que haga uso de él. Para el correcto análisis de dicho sistema, ya sea manualmente o mediante herramientas de análisis de tiempo real, es necesario conocer valores fiables de algunas características de tiempo real, como el tiempo que conlleva realizar un cambio de contexto o el tiempo de latencia por interrupciones. Por ello será necesario aplicar las técnicas vistas en el capítulo anterior para instrumentar las partes relevantes del código fuente del sistema operativo de forma que podamos obtener los valores de *WCET* correspondientes a estas características.

En este trabajo se ha tratado de establecer una metodología de uso de una herramienta de cálculo de tiempos de ejecución peor caso para la obtención de dichos tiempos, centrándonos en primera aproximación, como se verá en el siguiente capítulo, en la obtención del *WCET* para un cambio de contexto concreto.

4.1 - Instrumentación de código fuente de *MaRTE OS*

Para obtener valores de tiempos de ejecución de peor caso de las características antes mencionadas será necesario localizar el lugar en el código del sistema operativo donde estas toman efecto. *RapiTime* no proporciona una forma de obtener valores del *WCET* entre dos puntos de instrumentación arbitrarios, siendo la unidad de información la función. El hecho de que la herramienta no nos permita establecer el valor del *WCET* entre dos puntos cualesquiera del código, sino sólo de funciones y los bloques que las forman, nos supone una limitación a la hora de medir características temporales del sistema operativo.

En el caso concreto de los cambios de contexto, existen multitud de situaciones que conllevan uno, como pueden ser la suspensión de una tarea (*clock_nanosleep*, *delay*...), acceso bloqueado a un *mutex*, variables condicionales, semáforos, espera de señales, *pthread_join* (esperar a que termine otro *thread*), cambio de prioridad, *yield* (ceder CPU), fin de la rodaja temporal en *round-robin*... siendo necesario para cada caso estudiar la forma en que se ven reflejadas en el código del sistema, de manera que seamos capaces de calcular el *WCET* de cada una. La limitación de la herramienta antes mencionada conlleva que, en un principio, nos hayamos centrado en aquellas que se pueden medir instrumentando una función, como es el caso del *clock_nanosleep*, que se estudiará en el Capítulo 5. Una vez localizada la parte de código del sistema operativo que deseamos instrumentar, se usará la herramienta *adains*, con los correspondientes *pragmas* para indicar las opciones de instrumentación deseadas. El uso de *adains* se debe a que la mayor parte del código del sistema operativo está escrito en ADA. En caso de querer instrumentar alguna de las partes en C se hará uso de *cins*.

4.2 - Adaptación de un fichero instrumentado a las normas de estilo de compilación de *MaRTE OS*

El hecho de que *MaRTE OS* esté fuertemente ligado al compilador ADA *gnat* conlleva que su compilación siga unas normas de estilo bastante restrictivas. Esto supone un pequeño problema debido a que los archivos instrumentados por la herramienta *adains* de *RapiTime* no siguen dichas normas de estilo, y deben ser adaptados a ellas para poder compilar *MaRTE OS* correctamente.

Existe una herramienta incluida en el compilador *gnat* llamada “gnat pretty print”, *gnatpp*, que resuelve gran parte de las incorrecciones de estilo que pueden aparecen en nuestros archivos instrumentados, como por ejemplo los espacios innecesarios en las funciones insertadas por *adains*. A continuación se muestra un ejemplo de uso de *gnatpp*:

```
gnatpp -I./ -I/home/alvaro/bin/marte/kernel
-I/home/alvaro/bin/marte/arch/hwi
-I/home/alvaro/bin/marte/sll
-I/home/alvaro/bin/marte/libmgnat
-I/home/alvaro/bin/marte/misc
-I/home/alvaro/bin/marte/posix5
-of out.adb in.adb
```

Desafortunadamente, *gnatpp* no soluciona todos los errores de estilo que podemos sufrir, siendo necesaria una revisión manual del código tras su uso para, por ejemplo, asegurar que la longitud de las líneas no supere los 80 caracteres.

4.3 - Versiones recomendadas

Mientras que gran parte de los ejemplos de instrumentación de código de aplicación se han realizado con todas las versiones de la herramienta analizadas, durante el desarrollo de este trabajo se encontró un error en la versión 1.2 al intentar analizar el código del sistema operativo. Es por ello que es necesario utilizar la versión 2.0 o superior para esta tarea.

En el momento de escribir estas líneas, y tras la aparición de algunos errores de funcionamiento en el visor de informes, se nos ha facilitado una nueva versión de la herramienta, la 2.1a, que debería ser la utilizada a partir de ahora.

4.4 - Pragmas

RapiTime nos permite el uso de *pragmas* dentro del código. Los *pragmas* son directivas de compilación utilizadas en este contexto para configurar la forma en que las herramientas *adains* y *cins* instrumentan el código, de forma que podamos definir con precisión qué partes queremos instrumentar y cuáles no.

Por ejemplo, si deseamos instrumentar todo el código utilizaremos el siguiente pragma:

```
pragma RPT ("default_instrument(TRUE)");
```

En el caso que nos ocupa, lo normal es que queramos precisar más, por lo que podemos indicar funciones concretas a instrumentar. Si, por ejemplo, quisiéramos instrumentar la función *Do_Scheduling* del paquete *Kernel.Scheduler* de *MaRTE OS*, parte importante en el proceso de cambio de contexto, utilizaríamos lo siguiente:

```
pragma RPT (instrument("Kernel.Scheduler.Do_Scheduling","TRUE"));
```

Esto nos permite reducir el número de puntos de instrumentación, gracias a lo cual conseguimos ficheros de trazas más reducidos y, además que el informe sólo contenga datos relevantes para nuestro estudio.

5 - Casos de uso

Para estudiar las metodologías desarrolladas para la instrumentación a nivel de aplicación y sistema operativo y la capacidad de la herramienta de instrumentar código escrito tanto en C como en ADA, se ha utilizado un programa en C que ejecuta una serie de *threads* que hacen uso de la función *clock_nanosleep()* para suspenderse. De esta forma hemos sido capaces de utilizar la herramienta tanto con una aplicación corriendo sobre el sistema operativo, como con una parte del sistema operativo en sí. En este último caso, se ha instrumentado la función *Clock_Nanosleep* de *MaRTE OS*, con lo que se ha obtenido el correspondiente tiempo de cambio de contexto debido a su llamada.

5.1 - Entorno de pruebas

El entorno de desarrollo utilizado durante la realización de este trabajo, que sigue el modelo representado en la Figura 5.1, está compuesto por dos PCs: un ordenador *target* donde corre el programa compilado sobre *MaRTE OS* y un ordenador *host* con sistema operativo *GNU/Linux* donde se genera dicho programa, se reciben las trazas y se ejecuta el conjunto de herramientas de *RapiTime.*

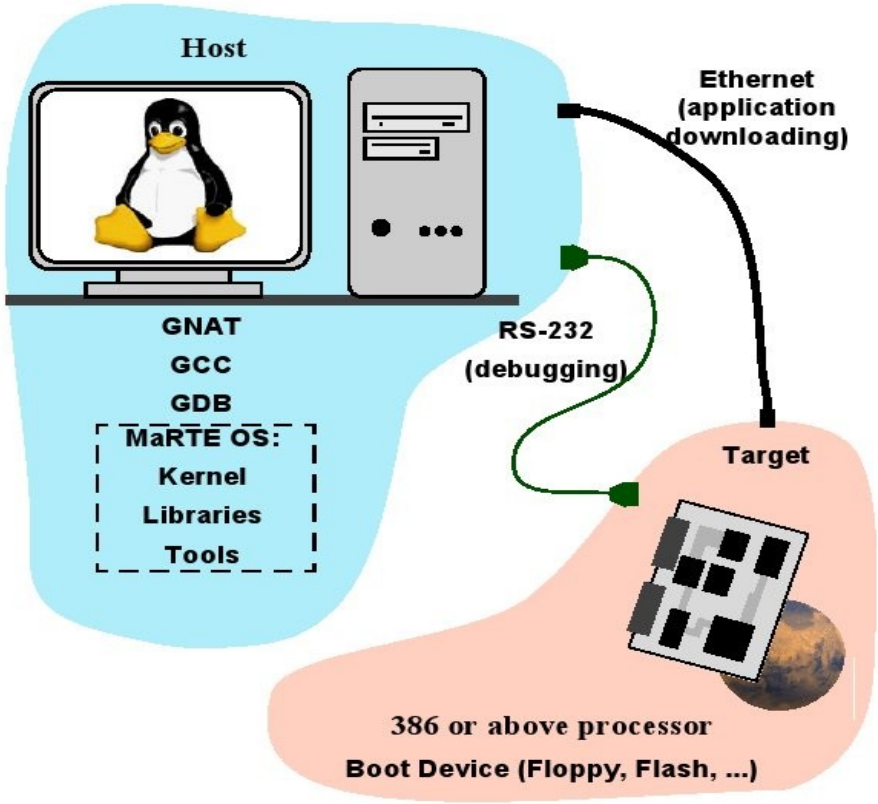

Figura 5.1. Entorno de pruebas.

Las versiones de *RapiTime* a partir de la 2.0 hacen uso de un *plug-in* para *Eclipse* que sólo funciona sobre la versión *Windows*, por lo que nuestro *host* también tiene instalado dicho sistema operativo.

El ordenador *target* usado en las siguientes pruebas tiene estas características: procesador Pentium III 800 Mhz y 256 MB de RAM.

5.2 - Instrumentación a nivel de aplicación

Tenemos tres *threads* C con periodos uno, dos y tres segundos respectivamente, ejecutándose durante diez segundos. Los tres threads ejecutan el mismo código, una función que hemos denominado *some_code* que simplemente ejecuta un bucle *for* que realiza unas iteraciones sumando números. La necesidad de tener una función dentro del cuerpo de los threads viene impuesta por *RapiTime*, que necesita una función raíz dentro del cuerpo del *thread* a la que hacer referencia en el análisis. Será pues esta función la que estudiaremos.

5.2.1 - Proceso de medida

Nuestra intención es generar un informe, Figura 5.2, para cada uno de los tres *threads*, de forma que podamos observar sus características individuales. El proceso para generar dicho informe para el primer *thread* sería el siguiente (ver. 2.0), siendo *per2.c* el fichero con nuestro código inicial.

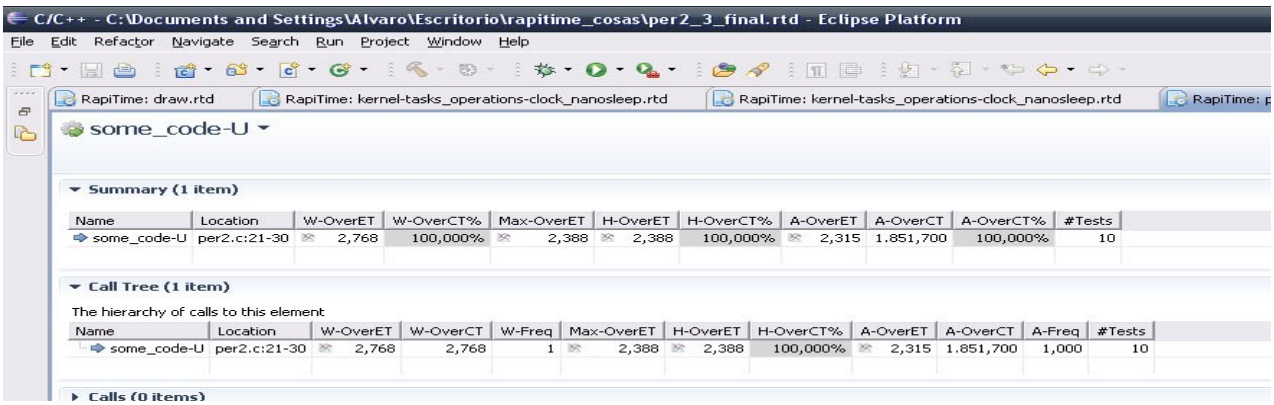

Figura 5.2. Informe para el thread 1 (RapiTime 2.0rc1)

El primer paso es generar el fichero preprocesado y añadir los puntos de instrumentación con la herramienta *cins*. Para ello hacemos uso del compilador *gcc* con la opción *-E*, como se muestra a continuación

```
gcc -Wall -E -nostdinc -I/home/alvaro/bin/marte/include per2.c > per2.i
cins --no-exf --cext gcc per2.i -o per2.i.c
```

Con esto estamos preparados para compilar nuestro programa, que ejecutaremos sobre nuestro ordenador *target* para generar el fichero de trazas, al que llamaremos *output_file.bin*. Esto se consigue ejecutando el compilador C para *MaRTE OS*, *mgcc*:

```
mgcc -c rpt.c -o rpt.o #librería rapitime
mgcc -c logger.c -o logger.o #librería envío de trazas por red
mgcc per2.i.c rpt.o logger.o
```

Ya tenemos todo lo necesario para generar la base de datos. Como en nuestro caso estamos interesados en evaluar la función *some_code,* le indicamos a la herramienta *xstutils*, encargada de generar el árbol de representación del programa, que esa será nuestra función raíz (*root*). Por otro lado, *traceutils* nos generará un fichero *rpz* por cada *thread,* siendo el que nos interesa en este caso el llamado *per2_3.rpz* y, por tanto, el que pasaremos a la función *traceparser*. A continuación se muestra la secuencia de comandos correspondiente, donde también se puede observar cómo las herramientas *xstutils* y *traceutils* hacen uso de los filtros comentados en el Capítulo 3.

```
cp per2.i.xsc per2.xsc
xstutils -r some_code per2.xsc --scope-filter scope.flt
traceutils --outname-template per2.rpz -f little_endian.flt -f marte.flt
-f scope.flt output_file.bin
cp per2.rtd per2_3.rtd
traceparser per2_3.rtd per2_3.rpz
```

y para finalizar calculamos los tiempos de peor caso, utilizando la herramienta *wcalc* a la que le pasaremos como parámetro de entrada la base de datos *rtd*:

```
wcalc per2_3.rtd
```

Con esto hemos generado un informe para el primer thread, que contendrá los datos de tiempo de ejecución. Se hace lo mismo para los otros dos threads.

5.2.2 - Análisis de los resultados

RapiTime genera gráficas *end-to-end* de la ejecución de las funciones, Figura 5.3, que nos muestran el número de ejecuciones y sus tiempos observados, con lo que podemos comprobar que el comportamiento de nuestros *threads* ha sido el esperado.

Se puede observar cómo el primer *thread* se ha ejecutado 10 veces, el segundo 5 y el tercero 4, lo que se corresponde con lo esperado para sus periodos (1, 2 y 3 segundos respectivamente) y tiempo de ejecución del programa (10 segundos).

Asimismo, la gráfica del primer thread nos permite observar cómo su primera ejecución conlleva un tiempo bastante superior a las siguientes, lo que podemos atribuir a efectos de la caché.

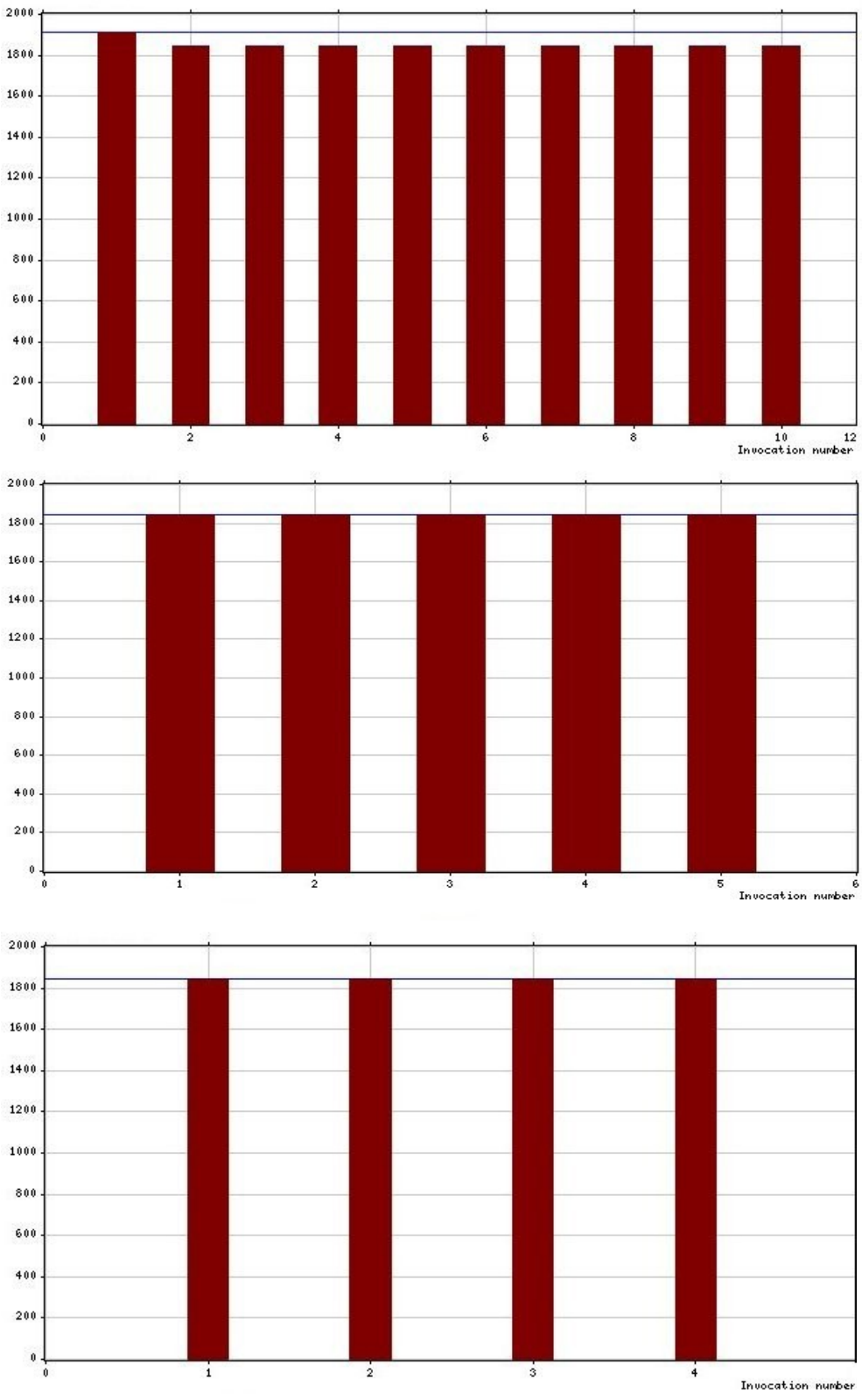

Figura 5.3. Gráficas de tiempo de ejecución - número de invocación.

Esto nos lleva a los datos principales del análisis, los tiempos de ejecución de peor caso observados y calculados. Los tiempos observados se obtienen directamente de las trazas, mientras que los calculados son generados por la herramienta *wcalc* de *RapiTime* tras analizar el código y los resultados medidos. Los resultados se presentan en la Tabla 5.1.

	W-OverET (µs)	***Max-OverET (µs)***
Thread 1	2,768	2,388
Thread 2	2,054	1,845
Thread 3	2,054	1,845

Tabla 5.1. Tiempos de ejecución de peor caso calculados (W-OverET) y medidos (Max-OverET).

En este caso, se puede ver cómo los tiempos calculados son mayores que los medidos. También vemos que el primer *thread* tiene un tiempo máximo superior a los otros dos, a pesar de ejecutar un código idéntico, lo que concuerda con lo observado en las gráficas *end-to-end.*

5.3 - Tiempo de cambio de contexto

Como ya se ha mencionado, la herramienta *RapiTime* tiene como unidad principal de análisis la función, no siendo posible obtener el tiempo de ejecución de peor caso de dos puntos arbitrarios cualesquiera dentro del código. Esto impone una limitación en el desarrollo de metodologías para el cálculo de características del sistema operativo como, en este caso, los cambios de contexto. Por esta limitación se ha tratado de buscar un caso en que el tiempo debido al cambio de contexto pudiera ser medido a través de la instrumentación de una función. Tras estudiar el código del sistema operativo y consultar a su principal desarrollador, se ha decidido instrumentar la función *Clock_Nanosleep* de *MaRTE OS*, a la que en nuestro caso de estudio llaman los *threads* haciendo uso de la función C *clock_nanosleep()*. Por tanto, en este caso de uso trataremos de obtener un *WCET* para el tiempo de cambio de contexto que conlleva la función *clock_nanosleep* en nuestras tareas y representado en la Figura 5.4.

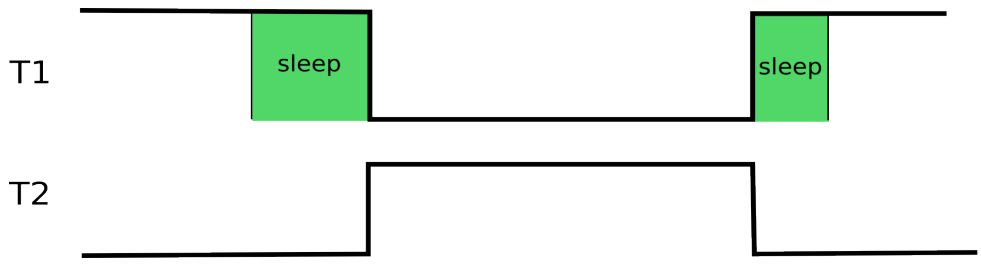

Figura 5.4. Ejemplo de cambio de contexto por clock_nanosleep.

5.3.1 - Proceso de medida

El código correspondiente a esta función se encuentra en la carpeta *kernel* de la instalación de *MaRTE OS*, en el fichero *kernel-tasks_operations-clock_nanosleep.adb*. Este es, por tanto, el fichero que deseamos instrumentar, y más concretamente la función *Clock_Nanosleep* que, dado que ha sido exportada, se corresponde al código que se llama desde una aplicación *C Posix* que ejecute la función *clock_nanosleep()*. Para instrumentar solamente esa función en concreto dentro del fichero arriba mencionado, incluimos el siguiente *pragma* en él:

```
pragma
RPT(instrument("Kernel.Tasks_Operations.Clock_Nanosleep.Clock_Nanosl
eep","TRUE"));
```

Una vez hecho ésto, dado que el código está escrito en ADA, debemos generar un fichero *.adt* e instrumentarlo con *adains*:

```
mgcc -c -gnatc -gnatt kernel-tasks_operations-clock_nanosleep.adb
rm -f kernel-tasks_operations-clock_nanosleep.ali

adains --no-exf kernel-tasks_operations-clock_nanosleep.adb

cp kernel-tasks_operations-clock_nanosleep.adbi kernel-
tasks_operations-clock_nanosleep.adb
```

Con esto tenemos un fichero instrumentado pero, como vimos en el capítulo anterior, será necesario formatearlo para que cumpla las normas de estilo de compilación de *MaRTE OS*:

```
gnatpp -I./
-I/home/alvaro/bin/marte/kernel
-I/home/alvaro/bin/marte/arch/hwi
-I/home/alvaro/bin/marte/sll
-I/home/alvaro/bin/marte/libmgnat
-I/home/alvaro/bin/marte/misc
-I/home/alvaro/bin/marte/posix5
-of salida.txt kernel-tasks_operations-clock_nanosleep.adb
```

Una vez formateado, ya podemos compilar nuestro ejemplo C junto con el código instrumentado del sistema operativo, siendo esta parte del proceso similar a lo explicado en el punto anterior, por lo que no se incidirá en ello.

El resto del proceso es similar a lo visto para el nivel de aplicación, excepto porque en esta ocasión deseamos generar un informe conjunto en el que se tengan en cuenta los datos generados por todos los *threads*. Para ello debemos indicar a *traceparser* que debe unir todos los datos de la siguiente manera:

```
 traceparser kernel-tasks_operations-clock_nanosleep.rtd trace_3.rpz
--clock-hz 800_000_000 --time-unit microseconds
 traceparser kernel-tasks_operations-clock_nanosleep.rtd --merge-
results trace_4.rpz --clock-hz 800_000_000 --time-unit microseconds
 traceparser kernel-tasks_operations-clock_nanosleep.rtd --merge-
results trace_5.rpz --clock-hz 800_000_000 --time-unit microseconds
```

Tras estos pasos, y utilizar *wcalc* como ya vimos anteriormente, se ha generado un informe de resultados.

5.3.2 - Análisis de los resultados

En la pestaña *Comparison* del informe podemos ver rápidamente los principales valores buscados, mostrándose dichos valores en la Tabla 5.2.

W-OverET (µs)	*Max-OverET (µs)*	*Min-OverET (µs)*	*A-OverET (µs)*
5,280	5,280	1,755	2,142

Tabla 5.2. Resultados temporales de tiempo de ejecución para la función clock_nanosleep.

RapiTime en esta ocasión nos da unos valores medidos y calculados iguales, 5,28 microsegundos. También se nos proporcionan, en la versión 2.0, los valores medidos mínimo y medio, siendo el valor medio medido aproximadamente la mitad del *WCET.*

De nuevo, la gráfica *end-to-end* de tiempos de ejecución medidos, Figura 5.5, nos sirve para analizar este hecho. En ella vemos como el valor máximo se corresponde al de la primera ejecución, siendo esta mucho más larga que las siguientes.

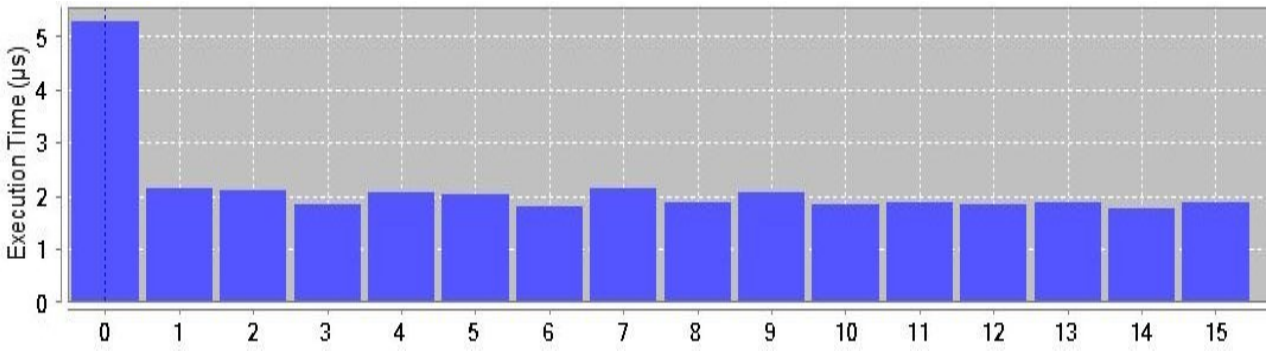

Figura 5.5. Tiempos de ejecución.

La herramienta también nos ofrece gráficas de probabilidad de tiempo de ejecución (ETP). Como la mostrada en la Figura 5.6. En esta gráfica podemos ver la probabilidad de que dado un margen temporal (eje x) el tiempo de ejecución supere ese margen cuando se elige el camino correspondiente al peor caso, comenzando, en este ejemplo, a bajar de 1 a partir del mínimo observado de 1,755.

Figura 5.6. 1-CumETP.

6 - Estudio de la integración del cálculo de *WCETs* con las herramientas *MAST*

MAST es un conjunto de herramientas de modelado y análisis de tiempo real desarrollado en el grupo de *Computadores y Tiempo Real* de la *Universidad de Cantabria* cuyo objetivo es facilitar al usuario el análisis de planificabilidad de sus aplicaciones de tiempo real [DRA08]. El modelo de la aplicación realizado con *MAST* y los resultados finales se especifican en una descripción *ASCII* que sirve de entrada y salida de las herramientas de análisis, existiendo dos formatos *ASCII* definidos para ello: un formato específico de la herramienta y un formato basado en *XML*.

Entre los resultados ofrecidos por *RapiTime* tenemos valores para los tiempos de peor caso, mejor y promedio, valores utilizados por *MAST* para la definición de algunos de los elementos de su modelo, por lo que parece interesante intentar desarrollar herramientas automáticas capaces de integrar dichos resultados con *MAST*. Para esto, es necesario antes que nada localizar aquellos parámetros del modelo *MAST* relacionados con el cálculo de tiempos y analizar si es posible utilizar *RapiTime* para su obtención.

6.1 - El modelo MAST

En *MAST*, un escenario de tiempo real se modela como un conjunto de transacciones concurrentes que compiten por los recursos ofrecidos por la plataforma. Cada transacción se activa con uno o más eventos externos, y representa un conjunto de actividades que se ejecutan en el sistema. Estas actividades generan eventos internos a la transacción, que pueden activar otras actividades [GON01].

En la Figura 6.1 se pueden ver los elementos que definen una actividad en el modelo *MAST* [DRA08]. Se puede observar que cada actividad se inicia con la llegada de un evento de entrada, y genera un evento de salida cuando finaliza. Cada actividad ejecuta una Operación (*Operation*), que representa un fragmento de código (que se ejecutará en un procesador), o un mensaje (a enviar por una red). Una operación puede tener una lista de Recursos Compartidos (*Shared Resources*) a los que necesite tener acceso mutuamente exclusivo. La actividad es ejecutada en un Servidor de Planificación (*Scheduling Server*), que representa una entidad planificable dentro del Planificador (*Scheduler*) al que ha sido asignada. Este Planificador pertenece a un Procesador (*Processor*) o a una Red (*Network*).

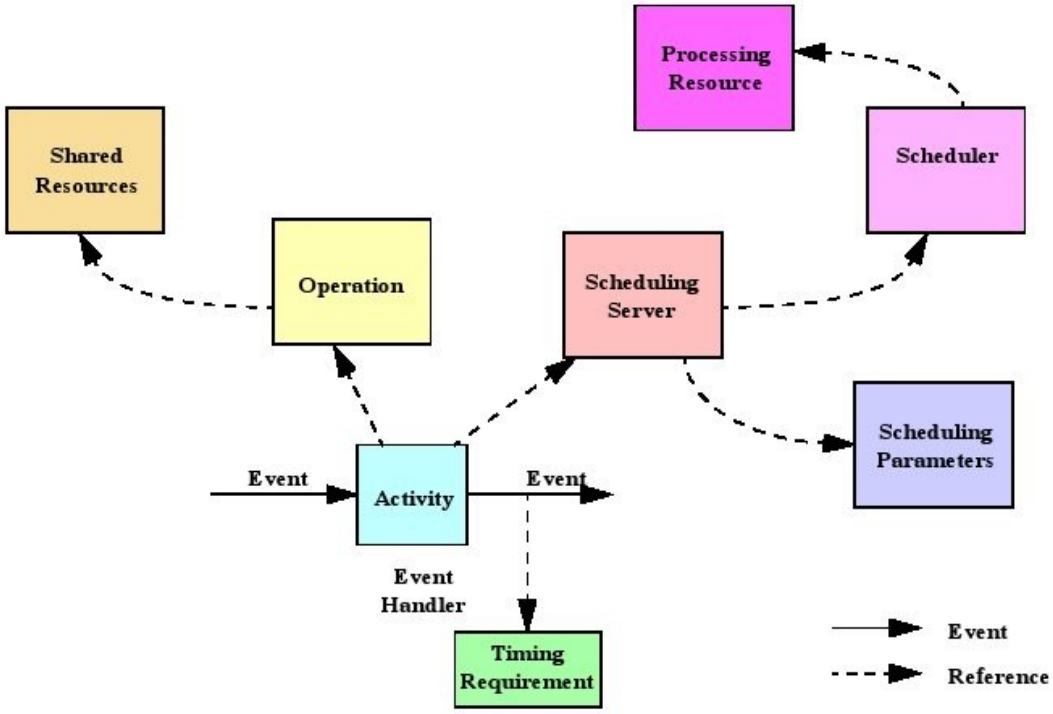

Figura 6.1. Elementos que definen una actividad en MAST.

6.1.1 - Elementos del modelo *MAST* con tiempos de ejecución

Dentro de los diferentes elementos que conforman el modelo *MAST* existen algunos para los que es necesario asignar valores de tiempo de ejecución a algunos de sus parámetros para su correcta definición. En la Tabla 6.1 se muestran estos parámetros y el estado de su cálculo haciendo uso de *RapiTime* siguiendo las metodologías desarrolladas en este trabajo. Hemos dividido el estado del cálculo en tres grupos: *Hecho*, correspondiente a los casos que concuerdan exactamente con las metodologías vistas; *Pendiente*, para aquellos casos donde aún queda pendiente analizar la posibilidad de usar las metodologías ya vistas para su cálculo; y *A estudiar,* que se corresponde con aquellos casos para los que inicialmente no se ha encontrado una forma de usar la herramienta para su cálculo y necesitan ser estudiados en mayor profundidad.

SISTEMA OPERATIVO / RED		
Elemento	***Parámetro***	***Estado del cálculo con RapiTime***
Processing resources	*Worst_ISR_Switch*	A estudiar
	Avg_ISR_Switch	A estudiar
	Best_ISR_Switch	A estudiar
System Timers	*Worst_Overhead*	Pendiente
	Avg_Overhead	Pendiente
	Best_Overhead	Pendiente
Network Drivers:	*Character_Transmission_Time*	Pendiente
Scheduling Policies:	*Worst_Context_Switch*	Hecho
	Avg_Context_Switch	Hecho
	Best_Context_Switch	Hecho
	Packet_Worst_Overhead	Pendiente
	Packet_Avg_Overhead	Pendiente
	Packet_Best_Overhead	Pendiente
Scheduling_Parameters:	*Polling_Worst_Overhead*	Pendiente
	Polling_Avg_Overhead	Pendiente
	Polling_Best_Overhead	Pendiente
APLICACIÓN		
Elemento	***Parámetro***	***Estado del cálculo con RapiTime***
Operations:	*Worst_Case_Execution_Time*	Hecho
	Avg_Case_Execution_Time	Hecho
	Best_Case_Execution_Time	Hecho

Tabla 6.1. Obtención de tiempos de ejecución para su integración en MAST.

6.2 - Desarrollo de herramientas para la integración automática de resultados con *MAST*

El siguiente paso, una vez se hayan obtenido valores para los parámetros vistos en la sección anterior, será el desarrollo de herramientas para la integración automática de estos valores en el modelo *MAST*. *RapiTime* almacena los resultados de cada análisis en un fichero correspondiente a una base de datos en formato *SQLite*, a la que se puede acceder de forma programática mediante lenguajes como *PHP* o *C*. Solamente será necesario conocer la estructura de las tablas de la base de datos y la localización de los datos deseados dentro de ella para tener acceso a los valores deseados mediante una aplicación desarrollada en alguno de los lenguajes antes mencionados. Por otro lado, como ya se ha visto, el modelo y los resultados obtenidos con *MAST* se guardan en ficheros *ASCII*, en formato exclusivo o en *XML*. De nuevo, lenguajes como *PHP* y *C* nos permiten la generación y manipulación de este tipo de ficheros, por lo que parece clara la posibilidad de desarrollar una aplicación que tome los resultados de *RapiTime* y genere o modifique los ficheros *ASCII* correspondientes al modelo *MAST*, pudiendo incluso utilizar los esquemas *XML* proporcionados por *MAST* [MAS08] para validar los archivos *XML* generados, en caso de escoger dicho formato de salida.

7 - Conclusiones y líneas futuras

El trabajo aquí presentado tenía como principales áreas de estudio la instrumentación de código para el cálculo de tiempos de ejecución y respuesta en sistemas de tiempo real, concretamente la aplicación de herramientas de cálculo de tiempos de ejecución de peor caso a la instrumentación de código de tiempo real tanto a nivel de aplicación como de sistema operativo, así como a la integración de los datos obtenidos con herramientas de análisis de tiempo real.

En concreto, durante el presente trabajo se han abordado los siguientes puntos:

- Estudio del funcionamiento e integración de una herramienta de cálculo de tiempos de ejecución de peor caso *RapiTime* con una plataforma de tiempo real basada en el sistema operativo de tiempo real *MaRTE OS* y para lenguajes de programación ADA y C.

- Creación de una librería de instrumentación adaptada al sistema usado.

- Desarrollo de una metodología para la instrumentación de código a nivel de aplicación.

- Desarrollo de una metodología para la instrumentación de código a nivel de sistema operativo.

- Estudio de la integración de las medidas de *WCET* con las herramientas de análisis *MAST*.

Esto comprende los pasos iniciales para la obtención de un entorno funcional con el objetivo de instrumentar y obtener tiempos de peor caso para cualquier aplicación de tiempo real, además de la caracterización de los principales parámetros del sistema operativo.

La integración y evaluación realizada durante este trabajo nos ha permitido también apreciar ciertas limitaciones de la herramienta, que influyen, en un principio, en, por ejemplo, el desarrollo de métodos para la obtención de tiempos de peor caso de las características temporales del sistema operativo, siendo la principal el hecho de que la herramienta tenga como unidad básica la función, no permitiéndonos obtener el *WCET* entre dos puntos cualesquiera del código. Otra dificultad relacionada con la herramienta ha sido la necesidad de ir cambiando de versión de *RapiTime* o alguno de sus componentes debido a la detección de errores en la misma que impedían la realización de algunas tareas.

En cuanto a las líneas futuras, y como consecuencia de lo anteriormente dicho, quedan por completar las medidas exhaustivas de todos los servicios del sistema operativo *MaRTE OS*, siendo necesario establecer la forma de obtener, con la herramienta analizada, las características temporales del sistema operativo aún no obtenidas (vistas en la Tabla 6.1) como, por ejemplo, el tiempo de

cambio debido a rutinas de servicio de interrupción, así como para todos los tipos de cambio de contexto, listados en el Capítulo 4.

Una vez que seamos capaces de obtener estos datos se podrán desarrollar las herramientas que integren automáticamente estos resultados de tiempos de peor caso con *MAST*, accediendo directamente a las bases de datos *SQLite* generadas durante el proceso y exportando los datos a ficheros *XML* que sigan el formato utilizado por *MAST*.

8 - Anexos

8.1 - Anexo 1: Librería de instrumentación C (código fuente)

```
/*==================================================================
 * Rapitime
 * Instrumentation library header
 * (c) 2007 Rapita Systems Ltd.  All rights reserved
 *==================================================================*/

/*==================================================================
 * 2007-2008: Modified by Alvaro
 * Instrumentation functions adapted to MaRTE OS
 * RPT_Output_Trace: now uses marte logger code to send traces through
 * ethernet
 *==================================================================
 */

#pragma RPT instrument( "RPT_Output_Trace", "FALSE" );
#pragma RPT instrument( "RPT_Ipoint", "FALSE" );

#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <assert.h>
#include <drivers/console_switcher.h>
#include "logger.h"

#if 1
#define DEBUG(x,args...) printf("%s: " x, __func__ , ##args)
#else
#define DEBUG(x,args...)
#endif

#define LOG_DEVICE LOG_ETHERNET // LOG_CONSOLE

typedef struct {

#ifndef RPT_COMPRESS

   unsigned long id;

   unsigned long time;
#else
```

```
   unsigned long combined;
#endif
} RPT_Ipoint_Type;


#define RPT_ARRAY_BYTES 100*1024*1024
#define RPT_ARRAY_SIZE (RPT_ARRAY_BYTES / sizeof(RPT_Ipoint_Type))

RPT_Ipoint_Type RPT_array[ RPT_ARRAY_SIZE ];
RPT_Ipoint_Type *RPT_array_ptr = RPT_array;

#pragma RPT instrument( "RPT_Ipoint", "FALSE" );

void RPT_Ipoint( unsigned long i )
{
   unsigned long long tsc;
   asm volatile( "cli" );

#ifndef RPT_COMPRESS

   RPT_array_ptr->id = i;
   asm volatile( "rdtsc" :"=A" (tsc) );
   RPT_array_ptr->time = tsc;
   ++RPT_array_ptr;

#else

   asm volatile( "rdtsc" :"=A" (tsc) );
   /* 20 bits of time, 12 bits of ID */
   RPT_array_ptr->combined = tsc << 12 | i;
   ++RPT_array_ptr;

#endif

   asm volatile( "sti" );

}


void RPT_Context_Switch_To( unsigned long i )
{
   unsigned long long tsc;

#ifndef RPT_COMPRESS

   RPT_array_ptr->id = 3;
   asm volatile( "rdtsc" :"=A" (tsc) );

   RPT_array_ptr->time = tsc;
   ++RPT_array_ptr;
   RPT_array_ptr->id = i;
```

```
   RPT_array_ptr->time = 0;
   ++RPT_array_ptr;

#else

   asm volatile( "rdtsc" :"=A" (tsc) );
   /* 20 bits of time, 12 bits of ID */
   RPT_array_ptr->combined = tsc << 12 | 3;
   ++RPT_array_ptr;
   RPT_array_ptr->combined = i & ((1 << 12)-1);
   ++RPT_array_ptr;

#endif

}


void RPT_Intrpt_Entry( void )
{
   unsigned long long tsc;

#ifndef RPT_COMPRESS

   RPT_array_ptr->id = 1;
   asm volatile( "rdtsc" :"=A" (tsc) );
   RPT_array_ptr->time = tsc;
   ++RPT_array_ptr;

#else

   asm volatile( "rdtsc" :"=A" (tsc) );
   /* 20 bits of time, 12 bits of ID */
   RPT_array_ptr->combined = tsc << 12 | 1;
   ++RPT_array_ptr;

#endif

}


void RPT_Intrpt_Exit( void )
{
   unsigned long long tsc;

#ifndef RPT_COMPRESS

   RPT_array_ptr->id = 2;

   asm volatile( "rdtsc" :"=A" (tsc) );
   RPT_array_ptr->time = tsc;
   ++RPT_array_ptr;
```

```
#else

   asm volatile( "rdtsc" :"=A" (tsc) );
   /* 20 bits of time, 12 bits of ID */
   RPT_array_ptr->combined = tsc << 12 | 2;
   ++RPT_array_ptr;

#endif

}


void RPT_Output_Trace( void )
{
   RPT_Ipoint_Type *RPT_array_ptr2 = RPT_array;
   RPT_Ipoint_Type *RPT_array_ptr3 = RPT_array_ptr;

   int err, fd, count;
   struct timespec logger_period = {4, 0};
   DEBUG("inicio envio de traza\n");
   fd = open(MEMBUFFER_PATH, O_WRONLY);
   err = logger_init(LOG_DEVICE);
   assert(err == 0);
   int counter=0;



   while( RPT_array_ptr2 != RPT_array_ptr3 )
   {
      count = write(fd, &RPT_array_ptr2->id,4);
      count = write(fd, &RPT_array_ptr2->time,4);
      ++RPT_array_ptr2;
      counter++;
      if(counter>=64){
            logger_manual_call();
            counter=0;
      }

   }

   if(counter!=0)
   DEBUG("mando el ultimo\n");
   logger_manual_call();
   DEBUG("ending output\n");
 }
```

8.2 - Anexo 2: Implementación de los filtros

scope.flt:

```
replace( (65535, t), ( 999998, t) (65535, t) );
replace( (65534, t), ( 65534, t) (999999, t) );
keep_sequence( 999998, 999999 );
remove( 999998, 999999 );
```

little_endian.flt:

```
byte_reader( B3 B2 B1 B0, B7 B6 B5 B4 );
wrap( 32 );
```

marte.flt:

```
escape( 3 );
# Prefix escape ipoint 3 by ipoint 6
replace( (3, t, i, 0), (6, t) (3, t, i, 0) );
demux( I1 );
# Remove interrupt time not executing thread
replace( (1, .) .* (2, .), , adjust_times);
# Remove time not executing thread
replace( (6, .) (3, ., ., .), , adjust_times );
```

9 - Bibliografía

- [BUR97] Alan Burns y Andy Wellings.. *Real Time Systems and Programming Languages (Second Edition)*, Addison Wesley, 1997.
- [PAL99] José Carlos Palencia y Michael González (director). *Análisis de planificabilidad de sistemas distribuidos de tiempo real basados en prioridades fijas*, Tesis Doctoral, Universidad de Cantabria, 1999.
- [WIL07] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström. *The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools*, ACM Transactions on Embedded Computing Systems (TECS), 2007.
- [HEI07] Gernot Heiser, Stefan M. Petters Patryk Zadarnowski. *Measurements or Static Analysis or Both?*, 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.
- [BER03] Guillem Bernat, Antoine Colin, Stefan Petters. *pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems*, Jun 2003 Technical Report YCS-2003-353, Department of Computer Science, University of York, UK, Febrero 2003.
- [BER02] Guillem Bernat, Antoine, Colin Stefan, M. Petters. *WCET Analysis of Probabilistic Hard Real-Time Systems*, Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), 2002.
- [BER05] Guillem Bernat, Alan Burns, Martin Newby. *Probabilistic timing analysis: An approach using copulas*, Journal of Embedded Computing, 2005.
- [BUR03] A. Burns, G. Bernat y I. Broster. *A Probabilistic Framework for Schedulability Analysis*, Lecture Notes in Computer Science, ISSU 2855, pages 1-15, 2003.
- [QUE03] José Juan Quesada-Molina. *What are copulas?*, Universidad de Zaragoza: Prensas Universitarias de Zaragoza, 2003.
- [PET07] Stefan M. Petters. *Execution-Time Profiles*, Technical Report , NICTA, 2007.
- [COL02] Antoine Colin, Guillem Bernat. *Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis*, ecrts,pp.50, 14 th Euromicro Conference on Real-Time Systems (ECRTS'02), 2002.
- [DAN04] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, y Björn Lisper. *Static Timing Analysis of Real-Time Operating System Code,* Leveraging applications of formal methods (First international symposium, ISoLA 2004, Paphos, Cyprus, October 30-November 2, 2004), 2004.
- [ALO05] Marta Alonso, Mario Aldea y Michael González. *Caracterización temporal de los servicios del sistema operativo de tiempo real MaRTE OS*, I Simposio de Sistemas de Tiempo Real, en CEDI 2005, Granada, Spain, Septiembre 2005.
- [RAP08a] *Rapita Systems Ltd. website*, www.rapitasystems.com
- [RAP08b] *RapiTime White Paper*, Jun 2008.

- [RAP08c] *RapiTime User Guide 2.0* & *RapiTime User Guide 1.0*, 2007-2008.
- [PER08] Héctor Pérez Tijero y J. Javier Gutiérrez (director). *Adaptación y optimización para una plataforma distribuida de tiempo real de un middleware basado en los estándares RT-CORBA y ADA*, Tesis de Máster, Universidad de Cantabria, 2008.
- [GON01] M. González Harbour, J.J. Gutiérrez, J.C. Palencia y J.M. Drake. *MAST: Modeling and Analysis Suite for Real-Time Applications*. Proceedings of the EuromicroConference on Real-Time Systems, Delft, The Netherlands, June 2001.
- [DRA08] J.M. Drake, M González, J.J Gutiérrez, P. López, J.L. Medina y J.C. Palencia. *Modeling and Analysis Suite for Real Time Applications (MAST 1.3.7)*. 2008. http://mast.unican.es/mast_description.pdf
- [MIM08] *Mime Edit page*, https://addons.mozilla.org/es-ES/firefox/addon/4498
- [MAR08] *MaRTE OS website*, http://marte.unican.es
- [POS03] The Institute of Electrical and Electronics Engineers. *IEEE Std. 1003.13-2003. Information Technology - Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP)*. 2003.
- [MAS08] *MAST website*, http://mast.unican.es
- [SYM08] *SymTA/S webpage*, http://www.symtavision.com/symtas.html
- [ECL08] *Eclipse webpage*, http://www.eclipse.org
- [SWE08] *SWEET webpage*, http://www.mrtc.mdh.se/projects/wcet/sweet.html
- [CHA08] *University of Chalmers - Timing analysis methods for high-performance real-time systems' webpage*, http://www.ce.chalmers.se/research/group/hpcag/project/wcet.html
- [AIT08] *aiT webpage*, http://www.absint.com/ait/
- [BOU08] *Bound-T webpage*, http://www.tidorum.fi/bound-t/
- [TUV08] *TU Viena Real-Time Systems Group webpage*, http://www.vmars.tuwien.ac.at/
- [FSU08] *Predicting Execution Time at FSU Project webpage*,http://www.cs.fsu.edu/~whalley/predict.html
- [HEP08] *Heptane webpage*, http://www.irisa.fr/aces/work/heptane-demo/heptane.html
- [NSU08] *National Singapore University - Compiler and Architecture for Worst case Program Execution's webpage*, http://www.comp.nus.edu.sg/~tulika/cawp.htm