Optimización y Paralelización con SMPSs de un código de detección de patrones

Proyecto Final de Carrera

Ingeniería de Informática

Autor: Jeremies Marín Armengod

Director: Jesús José Labarta Mancho
Codirectora: Judit Giménez Lucas

Volumen: 1/1

Fecha: 20 de Junio de 2011

Datos del Proyecto

Título: Optimización y Paralelización con SMPSs de un código de detección de patrones
Autor: Jeremies Marín Armengod
Fecha: 20 de Junio de 2011

Créditos: 37,5
Titulación: Ingeniería Informática
Centro: Facultad de Informática de Barcelona (FIB)
Universidad: Universidad Politécnica de Cataluña (UPC) BarcelonaTech

Director: Jesús José Labarta Mancho
Departamento del director: Arquitectura de Computadores (AC)

Codirectora: Judit Giménez Lucas
Departamento de la codirectora: Centro Europeo de Paralelismo de Barcelona

Miembros del Tribunal

Presidente: Eduardo Ayguadé Parra
Departamento del presidente: Arquitectura de Computadores (AC)

Vocal: Guillermo González Casado
Departamento del vocal: Matemática Aplicada II

Secretario: Jesús José Labarta Mancho
Departamento del Secretario: Arquitectura de Computadores (AC)

Calificación

Calificación numérica:

Calificación descriptiva:

Fecha:

Quiero agradecer a todos los que han hecho posible este proyecto, en especial a:
Jesús José Labarta, por haberme brindado la oportunidad de realizar este proyecto
Judit Giménez, por toda su dedicación en dirigir el proyecto.
German Llort Sanchez, en su gran ayuda para comprender el funcionamiento de la aplicación.
Alejandro Duran, por tomarse su tiempo en la instalación de OmpSs de mi PC.

Índice

Índice de figuras

Índice de códigos

1. Introducción

1.1. Presentación del proyecto y el documento

Este documento, contiene la memoria del Proyecto de Final de Carrera, que describe el trabajo que he realizado, al optimizar y paralelizar con SMPSs de un código de detección de patrones, por el estudiante de Ingeniería de Informática de la Facultad de Informática de Barcelona (FIB). El proyecto, fue propuesto por Jesús José Labarta Mancho y Judit Giménez Lucas. Acepté a realizarlo ya que me interesa la optimización de los códigos en la paralelización, para aprovechar al máximo el hardware de hoy en día, de los multicores y así ganar eficiencia en las aplicaciones.

En las herramientas de análisis de rendimiento **CEPBA-Tools**, centradas sobre la aplicación de **Paraver**, un analizador de trazas, se ha desarrollado un módulo para detectar la periodicidad de las trazas. Dicha aplicación, a partir de una traza en formato Paraver, realiza un análisis con wavelets de la señal, permitiéndonos así, detectar su zona periódica para poder realizar un corte en la región repetitiva con "calidad". La utilidad de la aplicación, es poder analizar trazas de un gran tamaño (unos cuantos GB) de una forma más cómoda, ya que la aplicación reduce considerablemente de tamaño la traza a analizar automáticamente. La alternativa, que tiene el usuario, es realizar este proceso de forma manual, se filtra la traza y acto seguido, se corta la zona periódica que nos interesa.

El objetivo principal del proyecto es optimizar el código de esta aplicación y paralelizar con SMPSs, de tal forma que su tiempo de respuesta sea menor al actual.

El documento se divide en las siguientes partes:

- **Estudio de la aplicación original**: se explica la instalación y el funcionamiento de la aplicación de su código original.
- **Optimización secuencial**: expone todas las optimizaciones que realicé a la aplicación antes de paralelizarla.
- **Optimización paralela**: contiene toda la información sobre la paralelización de la aplicación.
- **Tiempos de ejecución de las diferentes versiones**: recoge todos los tiempos detalladamente, de las versiones de las optimizaciones que se han tratado en los dos capítulos anteriores.
- **Versión final**: para el código final, se explica su instalación y utilización.
- **Planificación del proyecto**: antes de realizar el proyecto se elaboró una planificación, para poder realizarlo de forma correcta, y, se explica los cambios que surgieron de la planificación inicial.
- **Conclusiones**: explicación de las conclusiones a las que se ha llegado al finalizar el proyecto.

- **Anexo A**: recoge las características de mi ordenador, para poder comprender los datos obtenidos de las ejecuciones realizadas.
- **Anexo B**: Instalación y utilización de Extrae.
- **Anexo C**: Instalación de Paraver. Además, como se analizan trazas para esta herramienta, aquí se explicara su formato.
- **Anexo D**: uso de las dos versiones utilizadas StarSs, la SMPSs y la OmpSs.

1.2. Objetivos del proyecto

Tal como se ha explicado en la presentación del proyecto, el objetivo principal es rebajar el tiempo de ejecución de la aplicación. La utilidad de dicha aplicación, es realizar un trabajo automáticamente que el usuario lo realiza manualmente, pero no puede tardar tanto como lo hace ahora, la aplicación ha de ser más rápida, para que el usuario se decante a utilizarla. El problema reside, en que por ejemplo, para analizar una traza del tamaño de **1,1 GiB**, la aplicación tarda **71,82 segundos**. El usuario ejecuta esta aplicación de forma iterativa, si la aplicación para una traza de ese tamaño tarda ese tiempo, cuando analice otras más grandes, se pensará que la aplicación se ha bloqueado, o, ha tenido algún error y esta no responde.

Las trazas que analiza la aplicación, son archivos de texto que contienen la información sobre una ejecución de un programa. Estas trazas están creadas en formato para la herramienta de análisis de Paraver. Este, esta explicado detalladamente en el Anexo C. Como breve descripción, las trazas contienen toda la información referente al número de threads que se ejecutaron en el programa que se hizo el traceo, el tiempo de dicha ejecución e información detallada de los registros que se han generado. Los registros, son de tres tipos: estado, evento y de comunicación. Cada registro nos indica una información referente a la ejecución en un determinado momento.

La utilización de la aplicación o módulo que se tiene como objetivo optimizar, es para poder analizar trazas grandes con Paraver. Tal como se ha mencionado anteriormente, la aplicación analiza estas trazas para encontrar una zona periódica, de esta forma, se podrá cortar esta región repetitiva y reducir el tamaño de la traza, a la zona de interés para analizar cómodamente con Paraver. Las trazas contienen registros de diferentes tipos. Para encontrar la zona repetitiva, la aplicación genera una señal semántica a partir de un tipo en concreto de estos registros. Dependiendo la opción que introduce el usuario a la aplicación, generará diferentes señales semánticas, de sus respectivos registros, para así, poder ayudar en la búsqueda de la zona deseada.

El código original de la aplicación es heredado de GridSuperSalar, haciendo que la aplicación ejecute las diferentes etapas del análisis en binarios, y, su comunicación entre ellas, es mediante la lectura y escritura de ficheros intermedios. Las primeras mejoras serán sobre el acceso a los datos. Para reducir los tiempos, en vez de ir leyendo y escribiendo en ficheros en las diferentes etapas, se hará todos los cálculos en memoria. Se realizará otras optimizaciones secuenciales, como la fusión de funciones, para no tener que hacer dos veces el mismo trabajo o no tener que leer la señal más de una vez.

Una vez se tenga una versión secuencial optimizada, por último, se paralelizará con SMPSs de forma que se pueda ejecutarse eficientemente en máquinas de memoria compartida o multicores.

Otro de los objetivos, es de la utilización de la herramienta Paraver, para analizar la aplicación y así proceder correctamente en la optimización de dicha aplicación, tanto a nivel secuencial como con SMPSs.

1.3. Metodología

Para poder realizar una buena optimización sobre un código, se ha de seguir una cierta metodología. Para empezar, se tendrá que hacer un análisis del comportamiento de la aplicación, que tiempos tarda en hacer las ejecuciones de sus diferentes funciones o partes del código. Para este propósito, se usa herramientas de análisis como **gprof**, aunque en este caso, tal como se explica en el apartado anterior de los objetivos del PFC, se ha de utilizar **Paraver**.

Una vez sabemos que puntos se han de optimizar del código, la pregunta que uno se ha de hacer es, ¿vale la pena optimizar esa parte del código? Uno ha de tener en cuenta, la relación de tiempo necesario para realizar dicha optimización y la ganancia que se obtendrá de ella. Si se pierde mucho tiempo en una optimización, para que luego la ganancia obtenida sea muy pequeña, no vale la pena perder el tiempo en realizar dicha optimización. Otro ejemplo para elegir que optimización se ha de escoger a hacer, es que si tenemos dos trozos de código diferentes, uno tarda más en ejecutar que el otro, no se ha de perder mucho tiempo en intentar optimizar en el que menos tarda. Aunque se consiga una cierta ganancia, al ser el código que menos tarda, el código de mayor tardanza hace que se desprecie casi este logro, ya que, el tiempo donde más se pierde es en el más lento. En este caso, se ha de mirar antes y dedicarse más tiempo, en el trozo que tarda más en ejecución.

Sabiendo que se ha de optimizar esa parte del código, se procede a hacerlo. Una vez está hecho, se ha de mirar si realmente funciona el código como el original, es decir, si sus resultados son correctos. Si estos resultados son correctos, se ha de mirar si realmente se ha conseguido una ganancia o por lo contrario, se ha empeorado el rendimiento de la aplicación. Si se ha mejorado, el rendimiento del programa, esa optimización es válida, con lo que se procede a realizar otra optimización.

Para no perder tiempo en comprobar si los resultados son correctos de la ejecución del programa y si se ha mejorado el tiempo de respuesta, lo mejor, es realizar un script que nos automatice este trabajo. En el apartado "6. Versión final", se explica el script que realicé para este propósito.

2. Estudio de la aplicación original

El primer contacto con la aplicación es entender su funcionamiento. Para eso, lo que hice fue primero instalarla en un ordenador. Una vez la tenía instalada hice unas ejecuciones de prueba para saber su comportamiento. Al mismo tiempo iba mirando el código fuente para poder comprender su estructura.

Una vez estuve familiarizado con la aplicación, fue la hora de mirar que es lo que se tenía que optimizar. Para realizar esta labor, utilicé la herramienta de análisis de rendimiento **gprof**, para comprender el comportamiento del programa.

En los siguientes apartados se explica todo este procedimiento detalladamente.

2.1. Compilación de la aplicación

Para una mayor eficiencia, la aplicación corre sobre un sistema operativo basado en 64 bits. El sistema operativo que he utilizado en mi ordenador es un Ubuntu de 64 bits.

Antes de poder compilar la aplicación, era necesario instalar librerías externas que utilizaba la aplicación. Primero, se instala las librerías que están al alcance de todos, las dos que necesitamos son:

- **fftw**: poder hacer cálculos de las transformadas rápidas de Fourier.
- **zlib**: librería que implementa el método de descompresión deflate que se encuentra en gzip y PKZIP.

Los siguientes comandos son para poder instalar en Ubuntu:

```
$ sudo apt-get install libfftw3-dev libfftw3-doc
$ sudo apt-get install zlib1g zlib1g-dev zlib-bin zlibc
```

La aplicación también usa unas librerías externas desarrolladas en el BSC, que su conjunto lleva por nombre **TRACE2TRACE**. Las cuales, las compilaremos y colocaremos en una carpeta todos sus binarios.

Una vez tenemos todas las librerías necesarias se procede a compilar la aplicación. En el `Makefile`, la variable `ARCHITECTURE` se indicará, si se va a compilar en un ordenador personal (`laptop`) o en el MareNostrum (`MN`). Esto es, para poder encontrar las librerías necesarias según la arquitectura. En este caso, se pondrá:

Código 2.1: Makefile cambio Arquitectura

```
1 # ARCHITECTURE => MN , laptop
2 ARCHITECTURE = laptop
```

Con el siguiente comando, finalmente, se compila.

```
$ make
```

2.2. Ejecución de la aplicación

La aplicación acepta los siguientes parámetros:

```
./optim <primer archivo prv> <segundo archivo prv> ... <BW|MPIp2p|CPUBurst|CPUDurBurst|IPC>
```

En primer lugar se especifica el archivo (o los archivos) `.prv` , que corresponden a las trazas que se desean cortar. En segundo lugar se le pondrá la opción de corte, que método de búsqueda para la zona periódica deberá la aplicación utilizar. Las opciones que se pueden utilizar son:

- **BW**: buscará la zona periódica en los registros de las trazas de tipo comunicaciones.
- **MPIp2p**: en los registros de tipo evento de "MPIp2p".
- **CPUBurst**: con los eventos de "flushing".
- **CPUDurBurst**: generará la señal "wavelet", para la búsqueda de las zonas con los eventos de "flushing", como la opción `CPUBurst`. Pero, se basa en el análisis de la búsqueda con una señal de semántica de los eventos de "flusing" modificada, teniendo en cuenta los intervalos de tiempo, en el valor semántico.
- **IPC**: con los eventos de "IPC".

Antes de la ejecución de la aplicación, primero, se han de definir las siguientes variables globales:

- **SPECTRAL_HOME**: donde indicaremos el path de la aplicación.
- **FILTERS_HOME**: especificaremos el path a la carpeta de binarios, que se encuentran todas las librerías del conjunto **TRACE2TRACE**.
- **SPECTRAL_TRACE_SIZE**: indica el tamaño máximo que ha de tener la traza resultante, en bytes. Si no se define, como predeterminado la traza será como máximo 100.000.000 bytes.

Para ejecutar la aplicación a modo de prueba, escogeremos una traza de Paraver para poder analizar su zona periódica, y, en este caso utilizaremos la opción `CPUDurBurst`, al que le pasaremos como variables de entrada a la aplicación que recibe como nombre su binario optim. De modo ejemplo, para poder ejecutarla en mi ordenador, ejecuté los siguientes comandos con la traza de ejemplo `bt.C.64.prv`.

```
$ export SPECTRAL_HOME="./"
$ export FILTERS_HOME="/home/jere/Escritorio/TRACE2TRACE/bin"
$ ./optim /home/jere/Escritorio/bt.C.64/bt.C.64.prv CPUDurBurst
```

2.3. Tiempos de la ejecución

Una de las tareas para poder averiguar en qué partes de la aplicación se pueden optimizar para ganar velocidad en la ejecución, es decir, rebajar el tiempo de espera del usuario a que le muestre los resultados, es en mirar los tiempos de ejecución de las funciones que componen dicho programa. De esta manera, no solo se puede ver los puntos flacos de este, sino, que me sirvió para tener un primer contacto con el código para saber su estructura y su tiempo de respuesta.

Utilizando la herramienta de análisis de ejecución `gprof`, obtuve los siguientes resultados para la ejecución del binario de la aplicación `optim`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 39.67      0.23     0.23        1   230.07   230.07  GenerateEventSignal
 25.87      0.38     0.15                             qsort_cmp
 12.07      0.45     0.07        2    35.01    35.01  SignalDurRunning
  6.90      0.49     0.04        1    40.01    40.01  SignalRunning
  3.45      0.51     0.02        3     6.67     6.67  Cutter_signal
  3.45      0.53     0.02        2    10.00    10.00  Sampler
  3.45      0.55     0.02        1    20.01    20.01  Correction
  3.45      0.57     0.02                             frame_dummy
  1.72      0.58     0.01        1    10.00    10.00  GetTime
  0.00      0.58     0.00        9     0.00     0.00  Get_Binary_Dir
  0.00      0.58     0.00        2     0.00     0.00  Cutter3
  0.00      0.58     0.00        1     0.00    70.02  Analysis
  0.00      0.58     0.00        1     0.00     0.00  Crosscorrelation
  0.00      0.58     0.00        1     0.00     0.00  Cutter2
  0.00      0.58     0.00        1     0.00     0.00  Fft
  0.00      0.58     0.00        1     0.00     0.00  Fftprev
  0.00      0.58     0.00        1     0.00     0.00  FilterRunning
  0.00      0.58     0.00        1     0.00     0.00  Generatesinus
  0.00      0.58     0.00        1     0.00     0.00  GetBoundary
  0.00      0.58     0.00        1     0.00    40.01  GetPeriod
  0.00      0.58     0.00        1     0.00     0.00  Maximum
  0.00      0.58     0.00        1     0.00     0.00  Nthreads
  0.00      0.58     0.00        1     0.00     0.00  Sampler_wavelet
  0.00      0.58     0.00        1     0.00     0.00  SenyalD
  0.00      0.58     0.00        1     0.00     0.00  SenyalE
  0.00      0.58     0.00        1     0.00     0.00  Totaltime
  0.00      0.58     0.00        1     0.00     0.00  Wavelet
  0.00      0.58     0.00        1     0.00     0.00  getValues
  0.00      0.58     0.00        1     0.00     0.00  ini_rand
```

El tiempo de respuesta total de una ejecución con la traza de prueba `bt.C.64.prv` (de 1,1GiB), me dio una tardanza de 71,82 segundos.

Como se puede observar, al tener estos resultados, la primera impresión, es que se tendrá que mirar las primeras funciones que son las que tardan más, para mirar de optimizarlas: `GenerateEventSignal`, `qsort_cmp`, `SignalDurRunning`, `SignalRunning`, `Cutter_signal` y `Sampler`. El propósito de mirar las funciones que tardaban más, es para así saber cuáles se tenía que dedicar más tiempo en la optimización, ya que por la ley de Amdahl, por más que se optimice el trozo del código que menos tarda, se seguirá perdiendo más tiempo en ejecución en el que más tarda.

2.4. Diagrama de la ejecución

Gracias a los resultados que obtuve con `gprof`, generé un diagrama de la ejecución de las funciones, con el script de Marc Vertes[1]. El siguiente diagrama de la Figura 2.1, se pude observar que funciones se ejecutan y a partir de los colores puedes tener una idea de su tardanza. El rango de los colores van del rojo intenso (tarda el 100 % del tiempo en el la ejecución del programa), a violeta pálido (tarda el 0 % de tiempo en la ejecución), como si fuera un arcoíris.

De esta manera tan visual, ya tenemos una mejor idea de que se va ejecutando en el programa. En el siguiente apartado explicaré la secuencia en que se va ejecutando cada función. El diagrama, al generarse a partir de los resultados de `gprof`, se puede observar que sólo se destaca las funciones del principio, las demás, tal como muestran los resultados en el apartado anterior tienen tiempos parecidos, por eso sus colores también lo son.

Figura 2.1: *Diagrama de ejecución de `optim` con la traza `bt.C.64`. El rango de los colores va de rojo intenso (100 % de tardanza) a violeta pálido (0 %).*

[1]El script de Marc Vertes (http://mvertes.free.fr), sirve para generar diagramas a partir de los datos obtenidos de gprof. Se incluye el script en el fichero adjunto a la memoria, con el nombre `cgprof.sh`, dentro de la carpeta `Aplicación`.

2.5. Fases de la ejecución

Para una mejor compresión del código, lo que se ha de hacer, es mirar directamente en el código fuente. Teniendo en mente el diagrama de la Figura 2.1, Se explicará a continuación, en un modo muy resumido, del funcionamiento de las principales funciones en las diferentes fases.

La *primera fase*, es la de conseguir los datos de la traza que estamos analizando. Para esto, se encargan las funciones `Nthreads` (busca el número de threads que contiene la traza) y `Totaltime` (el tiempo de tardanza de la traza). En esta misma fase, se inicializan todas las estructuras necesarias para generar el análisis.

En la *segunda fase*, se genera las señales semánticas para poder hacer el análisis. Primero, se genera la señal semántica de los eventos de "Flushing" de disco, para poder después conseguir las zonas fronteras para el corte de la zona periódica. La función que se encarga de esta tarea es la `GenerateEventSignal` (en la Figura 2.2). Para diferencias las distintas señales que se explicaran a continuación, a esta la llamaremos *signal1*. Pasando la *signal1* a la función `GetBoundary`, se consigue las zonas de frontera que se estaban buscando.

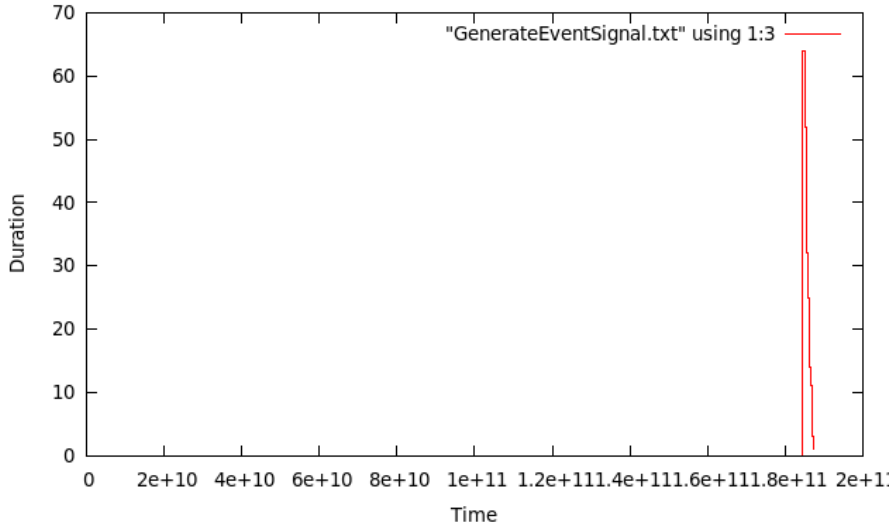

Figura 2.2: *La imagen muestra la señal de los eventos de "flushing" generada por `GenerateEventSignal`, de la traza `bt.C.64`.*

Después, se ha de crear una señal de "wavelet", para poder conjuntamente con las zonas de corte encontradas con la señal de "Flushing" *signal1*, refinar dichas zonas para posteriormente hacer el análisis de una forma más correcta. Para generar la señal de "wavelet" con la opción `CPUDurBurst`, que he utilizado como prueba, primero se filtra la traza original con `FilterRunning`. La función `FilterRunning`, hace una llamada a la herramienta externa de "trace_filter" de la librería TRACE2TRACE como paramentos de filtrado los eventos de "flushing". Acto seguido, se genera una señal de "burst" con la función `SignalRunning` (Figura 2.4) de la traza que se ha filtrado con `FilterRunning`, a esta la llamaremos *signal2*.

Con *signal2*, se encontrarán las zonas de corte con una señal de "wavelet". Para realizarlo, se tratará la señal *signal2* con la función `Sampler_wavelet`, con su resultado, finalmente, se podrá conseguir estas posibles zonas, con la función `Wavelet`.

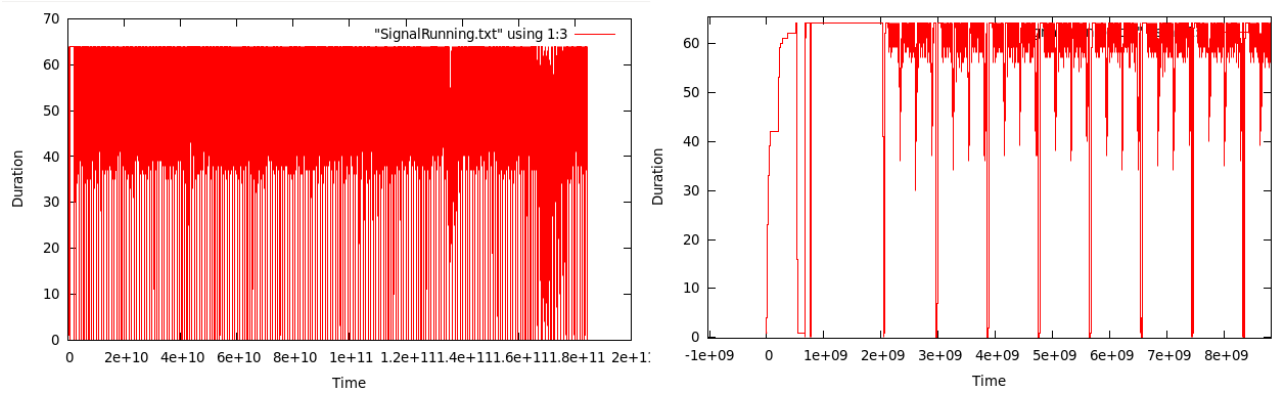

Figura 2.3: *Las dos imágenes son de la señal generada por* `SignalRunning` *de la* `traza bt.C.64`*. La de la izquierda, es la señal entera. La de la derecha, una ampliación de la zona del principio.*

En este punto, también se genera dos señales iguales de "burst" con `SignalDurRunning` de la traza que se ha filtrado con `FilterRunning`. Una se utilizará para el análisis como base para la búsqueda de la zona repetitiva, por utilizar la opción `CPUDurBurst`, le pondremos como nombre *signal3*. La segunda, se usará al final del análisis, para buscar el punto máximo de la zona de corte, esta se genera para todas las opciones, sin excepción, y, la llamaremos *signal4*.

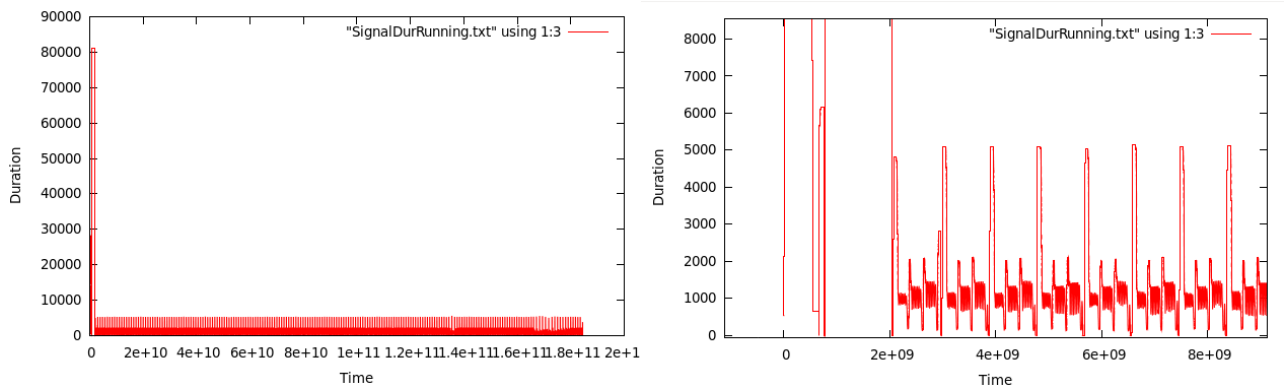

Figura 2.4: *Las dos imágenes son de la señal generada por* `SignalDurRunning` *de la* `traza bt.C.64`*. La de la izquierda, es la señal entera. La de la derecha, una ampliación de la zona del principio.*

Con las zonas encontradas con la"wavelet", se afinan con las de "Flushing" de la señal *signal1*. Las zonas de posible corte nuevas obtenidas, se analizarán cuáles de ellas son válidas con la función `Analysis`. Esta función, a partir de la señal *signal3* se analiza con `GetPeriod`, si esas zonas son correctas o no, para el corte de una iteración periódica y las acaba afinando más. La aplicación, para la primera zona que es correcta, buscará el número mínimo de iteraciones periódicas que debe cortar, para que el fichero de la traza resultante no tenga un tamaño mayor al que el usuario establece en la variable global `SPECTRAL_TRACE_SIZE`. Contra más iteraciones tenga el fichero resultante mejor, pero sin pasarse de ese tamaño establecido.

Se buscará de la zona inicial de corte válida, el punto máximo y mínimo en la señal, para así, a la hora de realizar el corte de la traza, no se cortará por la mitad de un evento. Para la búsqueda del punto mínimo, se generará dos señales nuevas. Una será del seno (`Generatesinus`) y la otra será tratando la señal original de *signal3* con `Sampler`. Utilizando las dos señales nuevas generadas, con `Crosscorrelation` obtenemos la zona minina de la señal. Después, para buscar la zona máxima utiliza `Maximum` con la señal *signal4*. Para acabar de afinar esa zona, para tener el fichero resultante con el tamaño ideal, con `Cutter3` llamará a la herramienta externa "Cutter" de la librería

TRACE2TRACE, para obtener dos trazas en dos zonas diferentes de la traza original. La primera zona contiene el periodo repetitivo encontrado, y, la segunda con el mismo tiempo de duración, pero con el periodo desplazado. En base a los tamaños de las trazas resultantes, se utilizará una zona o la otra para el corte final. Se buscan dos zonas, ya que en la traza original contiene todos los eventos, y, podría tener una zona más eventos que la otra, haciendo que el tamaño no cuadrase con el que desea el usuario.

Finalmente en la *tercera fase*, con `Cutter2`, llama a la herramienta externa "Cutter", se consigue la traza cortada en formato de Paraver. Y con `cutter_signal` se obtiene el trozo repetitivo de la señal semántica que se ha basado en la búsqueda de la zona periódica (Figura 2.5), en este caso de la señal *signal3*.

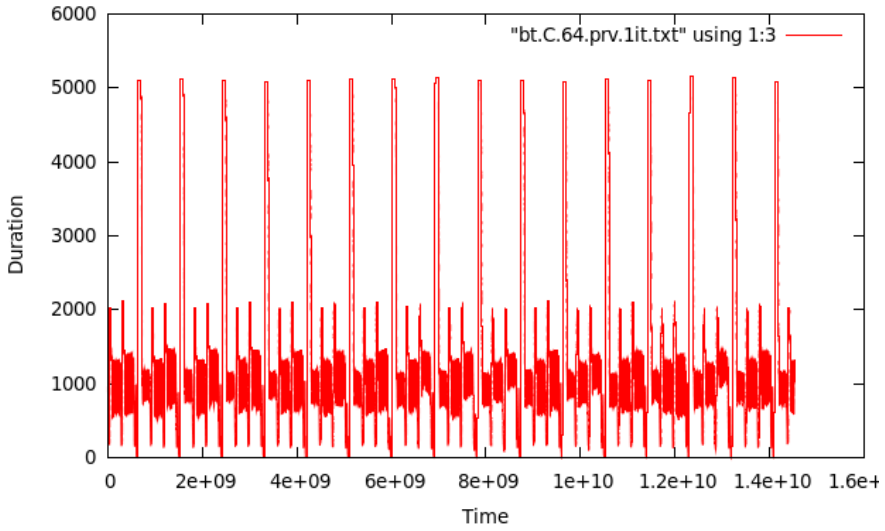

Figura 2.5: *La imagen muestra la zona periódica encontrada de la traza `bt.C.64` con la opción `CPUDurBurst`.*

3. Optimización secuencial

Para poder paralizar la aplicación, antes hay que tener una versión de esta bien optimizada. En este apartado, voy se expondrán las diferentes versiones significativas que se han hecho.

En el apartado anterior, realicé un traceo con `gprof` para tener una idea en que partes del programa se tenía que optimizar. El problema que me encontré, es que los tiempos no concordaban con la realidad. Es decir, las funciones que más tardaban como `Anlaysis` o la de `FilterRunning`, me daba como resultado unos milisegundos y en la realidad tardaban bastantes segundos. El problema resultó, que gprof no da los resultados de los tiempos de las llamadas a binarios, solo da el tiempo que tarda en ejecutar la función, pero no el tiempo que se tarda en la llamada. Como el código, la mayoría era a llamadas a binarios, no acumulaba los tiempos de tardanza de las llamadas a binarios. Para solucionar este problema, instrumenté la aplicación con **Extrae** para generar una traza de Paraver, para obtener los resultados reales. En el Anexo B, se explica cómo instalar esta librería e instrumentar un código. Para tener una mejor idea del comportamiento del programa, no solo he realizado las ejecuciones con una sola traza, sino con cuatro diferentes y con diferentes tamaños. De esta manera, es posible detectar mejor los puntos débiles de la aplicación y ver si realmente el cambio que he realizado en ese momento ha optimizado el código y he obtenido una ganancia. Esto se debe, a que cada traza puede llevar a un comportamiento diferente del programa. Las siguientes trazas son las que he utilizado:

Nombre Traza	Tamaño Traza	CPUS
ALYA.64	3,8 GiB	64
bt.C.64	1,1 GiB	64
VAC.128	212,7 MiB	128
WRF.64	5,1 GiB	6

3.1. Versión original en Binarios

La primera versión a mirar los tiempos de ejecución instrumentado con Extrae, es la versión original. Sus diferentes funciones lo que hace es llamar a unos binarios para realizar sus tareas. En el apartado de "5. Tiempos de ejecución de las diferentes versiones", están todos los resultados de las versiones significativas, él de esta versión viene como nombre "Binarios". Gracias a la instrumentación con Extrae, se puede observar que la función que más tarda es `FilterRunning` (Zona3) en la Figura 3.1. Tal como se ha explicado anteriormente, con gprof no me daba los resultados correctos, pero con esta traza de Paraver queda en constancia que será más útil utilizar un traceo con Paraver, que con gprof.

Para concordar la explicación de las fases de ejecución, explicadas en el apartado "2.5. Fases de la ejecución", con la Figura 3.1, la primera fase corresponde con la Zona 1. La segunda, con las Zonas

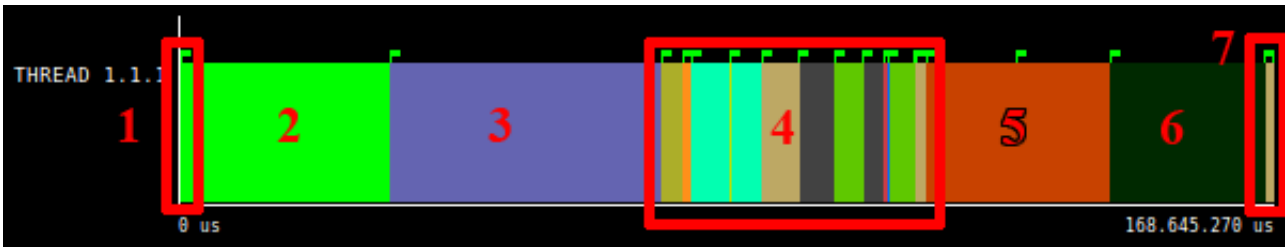

Figura 3.1: *Traza de aplicación versión con binarios con la ejecución de bt.C.64. La **Zona 1** contiene la ejecución de las funciones del programa principal, destacando Totaltime y Nthreads. **Zona 2** la función GenerateEventSignal. **Zona 3** FilterRunning. **Zona 4** las funciones de generación de las señales semánticas y ejecución del análisis hasta la función Cutter3. **Zona 5** dos ejecuciones de Cutter3. **Zona 6** Cutter2. **Zona 7** finalización del programa y como destacado la función Cutter_signal.*

2, 3, 4 y 5. La tercera con las Zonas 6 y 7.

Al mirar el código fuente, se puede observar que la primera mejora es quitar las llamadas a binarios. Tal como se explicó en el apartado "1.2. Objetivos del proyecto", esto se debe a que la aplicación es heredada de **GridSuperScalar**.

3.2. Versión sin binarios

Primero, antes de optimizar completamente la aplicación sin ficheros ni binarios por culpa de **GridSuperSalar**, en esta versión lo que hice fue sacar todos los binarios, pero dejando la comunicación mediante ficheros. De esta manera, es más fácil debugar y ver que mejoría hay respecto al código original sin tener que hacer las llamadas a binarios. Así que, saqué todas las llamadas a binarios y las convertí en llamadas a funciones.

Poniendo como ejemplo "Wavelet", esta función tiene un código fuente llamado `wavelet.c` con su respectivo `main` y este compilado aparte. Desde, la aplicación llama a una función con nombre "Wavelet", que lo que hace es ejecutar el binario generado aparte del código `wavelet.c`. En esta esta versión, solo quite los binarios, únicamente lo que hice fue borrar todas estas funciones que llaman a sus binarios, como esta de la "Wavelet":

Código 3.1: Función Wavelet en la aplicación

```
1  void Wavelet(char * input, char * output)
2  {
3          #ifdef TRACE_MODE
4                  Extrae_user_function(1);
5          #endif
6          char aux[200];
7
8          sprintf(aux, "%s/wavelet %s %s", SPECTRAL_HOME("wavelet"), input, output);
9
10         GS_SYSTEM(aux);
11         #ifdef TRACE_MODE
12                 Extrae_user_function(0);
13         #endif
14 }
```

Acto seguido borrar el main(Código 3.2) y convertirlo como una función normal (Código 3.3) con su respectivo fichero header (Código 3.4):

Código 3.2: main del código fuente wavelet.c

```
int main (int argc, char *argv[])
{

        double *input;
        double *output, a, i, freq, MAXvalue, SUM, DEV;
        long long int n, m, k, j, MAX, MIN, MAX2, MIN2, MAX3, MIN3, tini;
        FILE *fp;
        block *result, *maximum;
        long long int results[5][82];

        if (argc!=3 && argc!=4)
        {
                printf("Usage: %s <input datafile> <output datafile> <frequency (optional)>\n", argv
                    [0]);
                exit (0);
        }
        if(argc==4)
        {
                freq=atof(argv[3]);
        }

        if((fp = fopen(argv[1], "r")) == NULL)
        {
                printf("\nWavelet: Can't open file %s !!!\n\n", argv[1]);
                exit(-1);
        }

        ....
}
```

Código 3.3: wavelet.c sin main, convertido en una función

```
void Wavelet_exec(char * inputFile, char * outputFile)
{
        /**
        *pre:   inputFile: input datafile
        *       outputFile: output datafile
        **/
        #ifdef TRACE_MODE
                Extrae_user_function(1);
        #endif
        double a, i, MAXvalue, SUM, DEV;
        long long int n, m, k, j, MAX, MIN, MAX2, MIN2, MAX3, MIN3, tini;
        FILE *fp;
        block *result, *maximum;
        long long int results[5][82];


        if((fp = fopen(inputFile, "r")) == NULL)
        {
                printf("\nWavelet: Can't open file %s !!!\n\n", inputFile);
                exit(-1);
        }

        ...
}
```

Código 3.4: Header wavelet.h

```
void Wavelet_exec(char * inputFile, char * outputFile);
```

En la aplicación, se llama como antes a la función, ya que lo único que he hice fue borrar una función intermedia que ejecutaba a un binario. Eso sí, solo se tendrá que incluir el nuevo header `wavelet.h` al código de la aplicación.

No solo cambié los binarios que forman parte del programa, sino, tuve que crear una librería para

que llamase a los programas externos de TRACE2TRACE en forma de funciones.

En el apartado "5. Tiempos de ejecución de las diferentes versiones" están todos los resultados de las ejecuciones de las 4 diferentes trazas, está como nombre **Sin Binarios**. En la Figura 3.2 muestra de una manera visual como en la segunda traza que es de la versión sin binarios acaba antes que la versión original con binarios. La optimización afecta en todas las zonas, pero donde más se nota la mejora, es en la zona remarcada en un cuadro, ya que es donde hay más llamadas a diferentes funciones.

Con esta versión, obtuve una mejora de un **2,59 %** respecto al código original.

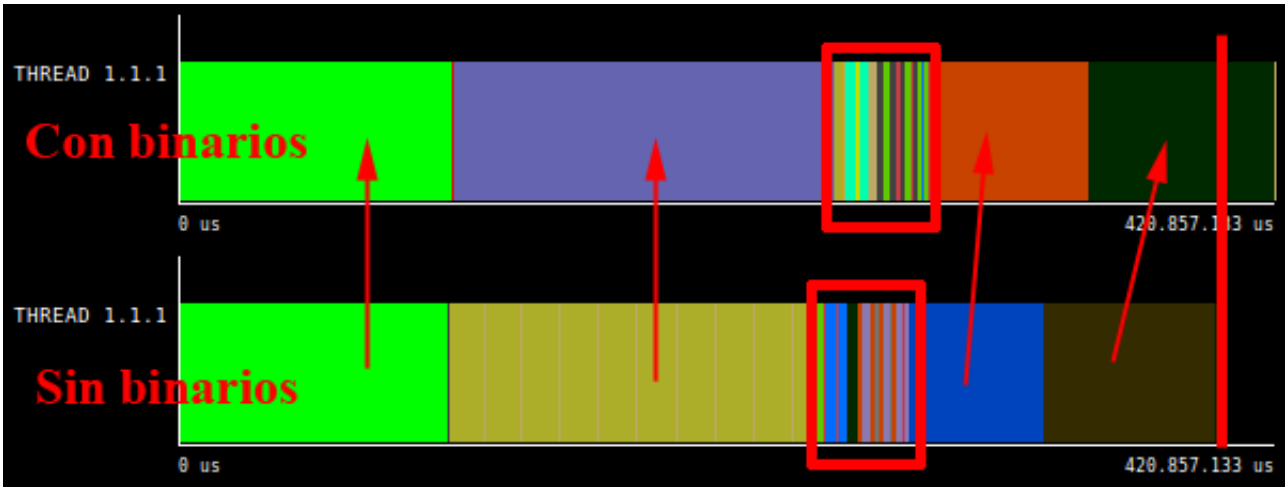

Figura 3.2: *Las dos trazas son de la aplicación con la ejecución de la traza ALYA.64. La superior es de la versión original con binarios y la inferior la de sin binarios. Las flechas indican que aunque sean de colores diferentes son las mismas funciones.*

En la Figura 3.3, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión anterior.

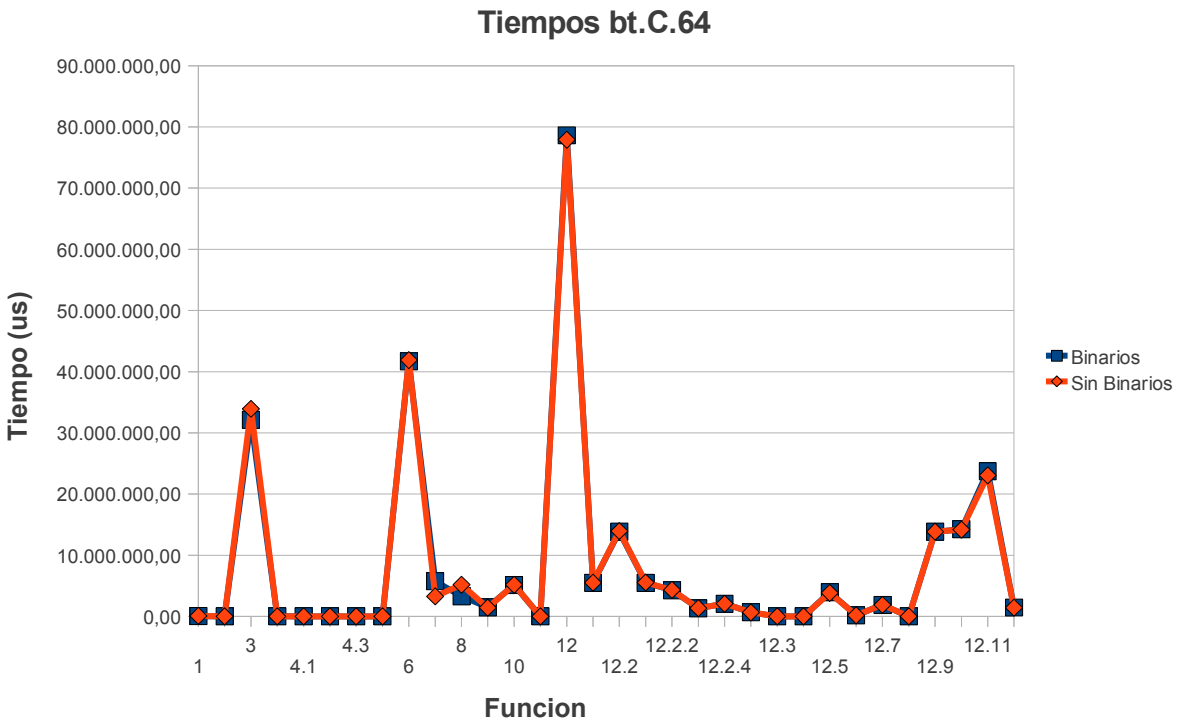

Figura 3.3: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones con binarios y sin binarios. La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

3.3. Versión sin ficheros

El siguiente proceso a optimizar fue sacar toda la lectura y escritura a ficheros, que se realizaban para poder hacer las comunicaciones en la versión de GridSuperScalar entre los diferentes binarios, ya que, ahora lo tendremos todo en memoria. Sin llamadas a binarios y la comunicación basada en memoria, se obtiene una buena ganancia de tiempo. A medida que se quitaban la lectura y escritura de ficheros, se realizaban otras optimizaciones secuenciales que se iban encontrando, tanto a la hora de reescribir código para hacer la comunicación en memoria, como aplicando otras series de optimizaciones. En el siguiente apartado, se explica más detalladamente las diferentes optimizaciones realizadas.

3.3.1. Optimizaciones realizadas

3.3.1.1. Estructuras en vez de ficheros de texto

Como hay que sacar la lectura y escritura en ficheros para la comunicación entre funciones y hacerlo todo en memoria, la forma que se hace, es mediante el paso de variables a las funciones.

Se procedió como se hace normalmente para pasar variables:

- Si se ha de pasar un número, este se le pasará a la función mediante el tipo que sea, ejemplo, entero o de como coma flotante. Parece una tontería pasar un simple número, pero es mucho más eficiente que antes, ya que, se tenía que crear un fichero para escribir un solo número para comunicárselo a otra función.

- Si son varios números seguidos, se hizo a partir de una tabla (un array) del tipo de valores que tocaba.

Pero, en los más complicados como las señales semánticas, estos se pasaron a partir de una tabla en que su tipo es complejo, este es una estructura. Es decir, se creó primero el tipo de estructura que tuviera los datos de la señal en una posición concreta. El Código 3.5 representa la estructura que se empleó para definir en una posición determinada de la señal.

Código 3.5: Estructura de la señal semántica

```
struct signal_info
{
        long long int t;//valor del tiempo
        long long int delta;//valor para delta
        double semanticvalue;//valor semantico
};
```

Los diferentes puntos de la señal, se guardan en una tabla del tipo de esta estructura, de esta manera se inicializa una señal:

Código 3.6: Inicializar una señal

```
struct signal_info *signal;//definir señal
signal = (struct signal_info *)malloc( SIZE_SIGNAL * sizeof(struct signal_info));//guardar espacio
    en memoria del tamaño de SIZE_SIGNAL
```

Y de la siguiente, se le asigna un valor a una determinada posición "i":

Código 3.7: Asigna un valor

```
signal[i].t=time_value;//ej: 246143
signal[i].delta=delta_value;//ej: 492071
signal[i].semanticvalue=semantic_value;//ej: 534.377786
```

Al ser tres valores para cada punto, es mucho más eficiente hacerlo de esta manera, de un array con una struct, que 3 arrays donde cada una es de un tipo de valor diferente (en la versión original utilizaba este método de las 3 tablas). Con la struct, aprovechamos la cache y se obtiene menos fallos en la memoria. De la forma de los 3 arrays, tendríamos muchos fallos de cache al tener que ir buscando en la posición determinada en las 3 diferentes tablas, ya que se podrían mappear en la misma línea, originando a remplazos constantes de estas. De esta forma, fusionando las 3 tablas en una estructura, se obtiene una mejora en la localidad de los datos en la memoria, disminuyendo los fallos a cache.

No solo se obtuvo ganancia de tiempo por no tener que trabajar con ficheros, sino, al tenerlo de esta forma con estructuras o tablas, se pudo aplicar algunas optimizaciones en el código. Explicaré tres mejoras:

La primera, es ahorrarnos de generar una señal con `SignalDurRunning`, es decir, con la opción `CPUDurBurst` se generaba una misma señal dos veces. Pero esta mejora no se aplicó en esta versión, sino, se realizó en la Versión sin repeticiones, donde se explica claramente en su subapartado "3.4.1.3. Hacer la señal 3". Esta versión, se dejó que siempre se generara las dos señales. Pero, gracias a tener todas las señales en memoria con esta estructura, fue posible en la siguiente versión aplicar la mejora.

La segunda mejora, es en la función `Correction`. En el código original, como entrada tiene una señal (está escrita en un fichero) y como salida el nombre de otra señal (otro fichero diferente). Su

tarea es copiar la señal de entrada y dependiendo de la situación, añadirá un punto más a la señal o no. Con el código original, se obligaba a crear otro fichero, pero en esta versión al ser todo en memoria, se le pude aplicar un pequeño truco. Este consiste en reservar en memoria para esta señal de entrada una posición más. Luego dentro de la función `Correction`, si ha de añadir una posición extra, solo tendrá que añadirla en la misma estructura de la señal de entrada y sumar por uno el contador del tamaño de la señal. Con esto se obtiene una gran ventaja, ya que `Correction`, se encuentra dentro de un bucle con el que tenía que ir generando nuevas señales, a partir de la misma señal de entrada, antes de entrar en el bucle y según las variables locales generar una nueva señal. Con la nueva versión, solo tiene que ir tocando una sola posición y el tamaño será según la situación el mismo o uno más, pero para nada perderá tiempo generado una señal nueva cada vez.

La tercera, va un poco ligado a la segunda mejora, en generar señales nuevas o solamente conocer el tamaño total de una. Con la versión original, para saber el tamaño de una señal lo que tenía que hacer es leerse un fichero entero para conocer el tamaño de esta (Código 3.8). Por el otro lado, en esta versión al tenerlo todo en memoria, se tiene una variable local guardando el tamaño de dicha señal. Conllevaba, que en el código original al tener en un fichero la señal, que luego con la misma se tenía que tratar en diferentes funciones, se tenía que leer entero el fichero en cada función para conocer su tamaño, después reservar memoria de ese tamaño encontrado y luego volver a leer entero para guardar la señal en memoria. Ahora al tenerlo todo en memoria, le podemos pasar a las diferentes funciones toda la estructura ya desde un principio y su tamaño como variable, sin tener que hacer dos lecturas entraras del fichero origen.

Código 3.8: Conocimiento del tamaño de una señal en el código original

```
long int a, b, size;
float c;
FILE *fp;
fp = fopen(filename, "r");
if (fp==NULL)
{
        printf("file not open\n");
        exit (77);
}
size=0;
while (fscanf(fp, "%ld %ld %f", &a, &b, &c)==3)
{
        size++;//tamaño señal
}
```

3.3.1.2. Limpieza del código

Para sacar tiempo extra, a medida que se reescribía el código, borré funciones, variables o instrucciones que no se hacían servir o que al ejecutar una segunda instrucción la primera ya quedaba invalidada. Ejemplo, en un lugar se asigna el valor a la variable 3 y más adelante 4, y, el valor que se acaba utilizando es el 4, con lo que borré la instrucción que asignaba 3. Es posible, que igual el compilador ya lo haga, pero siempre no cabe de más hacerlo, y, de esta manera se obtiene un código más limpio y se puede conseguir tiempo extra con menos código. Otro ejemplo, es que al sacar la utilización de binarios, todas estas funciones, variables de entorno y macros que se utilizaban para este propósito, ya no son servibles, como resultado se han borrado todo.

La aplicación era heredada de GridSuperScalar, en consecuencia había que eliminar o modificar toda instrucción de GridSuperScalar, para que sea el programa en lenguaje de C. El código original, estaba preparado para ejecutar en secuencial, para eso las sentencias de GridSuperScalar se ponían con una `macro`, que después según la compilación se pondrían las de GridSuperScalar o las de C.

Todo este código, se tenía que borrar, porque las `macros` introducían código extra, como la del Código 3.9, introduce un switch.

Código 3.9: macro GS_FOPEN con compatibilidad con GridSuperScalar y C

```
#if defined(GS_VERSION)
#define GS_FOPEN(res, a, b) res = GS_FOpen(a, b)
#else
#define GS_FOPEN(res, a, b) \
{ \
        switch(#b[0]) \
        { \
                case 'R': res = fopen(a, "r"); \
                break; \
                \
                case 'W': res = fopen(a, "w"); \
                break; \
                \
                case 'A': res = fopen(a, "a"); \
                break; \
                \
                default: \
                        res = NULL; \
                        break; \
                } \
        }
#endif
```

Las instrucciones que especifican la zona que se aplicará GridSuperScalar, la instrucción de inicio de esta zona `GS_ON()` y la instrucción de finalización de la zona `GS_OFF()`, estas dos solo había que borrarlas del código. En cambio, las siguientes solo tuve que sustituirlas por las que corresponden en C.

- `GS_FOPEN()` **->** `fopen()`.
- `GS_FCLOSE()` **->** `fclose()`.
- `GS_SYSTEM()` **->** `system()`.

Otra forma de reducir el número de variables, es reutilizarlas o escribir el código de otra manera para no tener que utilizarlas. Con el Código 3.10 como ejemplo de la función `Wavelet`, servirá para entenderlo mejor. En este caso, utiliza una estructura llamada `block` como tipo para una tabla con el nombre `result` para guardar la posición y el valor de la misma, sobre unos valores a tratar de la tabla `output`. Se hace de esta manera, ya que después de calcular los valores para la tabla `output` de un tamaño `n`, solo nos interesan los valores de `n/2` hasta `n`. La solución, es deshacerse de esta estructura, de esta forma liberas memoria al no necesitar la estructura y la tabla `result`. Tal como se muestra en el Código 3.11, únicamente nos ha de preocupar de hacer los cálculos en el bucle desde la posición `n/2`, de esta forma más adelante, solo se tendrá que jugar con la posición que indica el bucle en esa iteración sin necesidad de guardar la posición anterior. Simplemente que la posición que se guarda es la misma posición de la tabla. Se ha de añadir, que al jugar con una tabla no habrá posibilidad de conflictos en los accesos entre las dos tablas en la cache, aumentando el rendimiento de la utilización de la cache.

Código 3.10: Utilización de una estructura en wavelet para guardar la posición de la tabla y el valor

```
...
typedef struct
{
        long long int t;
        double val;
} block;
```

```
...

haartransform(input, output, n);//output como tabla de resultados de tamaño n

SUM=0;
//solo interesan los resultados desde n/2
for(k=0; k<n/2; k++)
{
        result[k].t=k;//guardamos la posición
        result[k].val=fabs(output[k+n/2]);//guardamos el valor calculado
        SUM=SUM+result[k].val;
}

...
```

Código 3.11: Sin la estructura block, solo jugando con los punteros

```
...
haartransform(inOut, output, n);

SUM=0;
m=n/2;
for(k=m; k<n; k++)
{
        output[k]=fabs(output[k]);//con k-m tenemos la posición de result[k].t del código anterior
        SUM=SUM+output[k];
}
...
```

Reduciendo el número de llamadas a otras librerías del sistema, también es una buena forma de reducir código e instrucciones ejecutadas. El caso más común que me encontré, fue con la escritura por pantalla. Para varios `printf` seguidos, los uní para que se ejecutasen menos.

3.3.1.3. Condicionales anidados

Los condicionales para los procesadores de hoy en día reducen el rendimiento, ya que rompen la ejecución secuencial, esto se debe a que los procesadores son pipelined y superescalars. Una forma de sacar condicionales habitual, es con la utilización de instrucciones aritméticas, memoization o bit hacks.

En el código, se pude suprimir o minimizar la cantidad de condicionales reescribiéndolo, para que no se tuvieran que hacer unas determinadas operaciones o del caso más común que me encontré, fue tener que anidar condicionales. En el código original del programa, ejecutaba todos los condicionales uno detrás del otro como este ejemplo:

Código 3.12: Condicionales uno detrás del otro

```
if(condición1)
{
        ....
}
if(condición2)
{
        ....
}
if(condición3)
{
        ....
}
```

Pero, en la realidad, el comportamiento que se esperaba era que si entraba por un condicional, no entrara por los siguientes. Una forma de optimizarlo, es la utilización de condicionales anidados con

la estructura "else if" en el Código 3.13. De este modo, si entra en una rama de los condicionales no ha de calcular o comprobar el resto, cosa que con la forma anterior hacía una comprobación de todos. Para una sola ejecución, no se gana mucho tiempo, aunque para ganar algo siempre ayuda, pero tal como ocurre en la aplicación, esto también sucede dentro de un bucle, dando como resultado una ganancia más considerable, utilizando la técnica de los condicionales anidados.

Código 3.13: Condicionales anidados

```
if(condición1)
{
        ....
}
else if(condición2)
{
        ....
}
else //condición3
{
        ....
}
```

3.3.1.4. Código hoisting

Tal como se ha explicado en el apartado anterior con los condicionales anidados, el propósito es siempre reducir las ejecuciones de condicionales. En este caso, es de reducirlos dentro de un bucle. Es decir, para el Código 3.14 ejecutaba todo el rato un condicional, que en esta ocasión se puede ejecutar sólo una vez. La técnica del *hoisting*, es sacar fuera del bucle el condicional, el resultado es que sólo se ejecutará una sola vez el condicional, tal y como se puede observar en el Código 3.15.

Código 3.14: Un bucle de cutter_signal

```
while([condición salida])
{
        //el condicional se comprueba todas las veces
        if([condición válida una vez])
        {
                ....
        }
}
```

Código 3.15: El bucle de cutter_signal optimizado

```
//condicional se ejecuta una sola vez, sin interferir en los cálculos del bucle
if([condición válida una vez])
{
        ...
}
while([condición salida])
{
        ...
}
```

Otra aplicación de esta técnica, es la de hacer cálculos fuera de un bucle. En el sentido, que si dentro del bucle se hacen unos cálculos que resultan ser los mismos para cada iteración, lo que hice fue, hacer esos cálculos fuera del bucle y guardarlos en una variable. De esta forma, se reducen el número de instrucciones a ejecutar en la vida de un bucle.

3.3.1.5. Fusión de bucles

La fusión de dos bucles es efectiva, si las iteraciones de los dos bucles se pueden solapar, de esta manera reduciremos el overhead de crear dos bucles, y, si en la misma iteración comparten los mismos datos (pero sin conflictos), podemos explotar los datos locales de la memoria mejorando el rendimiento. En este caso, he fusionado dos bucles de la función `Wavelet` (Código 3.16) y como resultado he obtenido uno solo (Código 3.17), en consecuencia, se ha reduciendo el overhead de tener dos bucles. La implementación de un bucle, es la comprobación de un condicional y si se hace el código dentro del bucle o no. A tal efecto de la optimización, se disminuye el número de ejecuciones de condicionales, reduciendo el overhead creado. Añadiendo, que en este código comparten datos de la tabla `output`, con lo que se explota la jerarquía de la memoria en los accesos de datos locales.

Código 3.16: Dos bucles separados

```
1  for(k=0; k<n; k++)
2  {
3
4          DEV=DEV+(SUM-output[k])*(SUM-output[k]);
5
6  }
7  for(k=0; k<n; k++)
8  {
9          if(output[k]>MAXvalue)
10         {
11                 MAXvalue=output[k];
12         }
13 }
```

Código 3.17: Fusión de dos bucles

```
1  for(k=0; k<n; k++)
2  {
3
4          DEV=DEV+(SUM-output[k])*(SUM-output[k]);
5
6          if(output[k]>MAXvalue)
7          {
8                  MAXvalue=output[k];
9          }
10
11 }
```

3.3.2. Ganancia obtenida

En el apartado "5. Tiempos de ejecución de las diferentes versiones", se encuentran los resultados de las ejecuciones de esta versión, con el nombre **sin ficheros**. Con todas las mejoras realizadas, explicadas en esta versión, obtuve una mejora de un **11,92 %** respecto a la Versión sin binarios (un **14,51 %** respecto al código original).

En la Figura 3.4, se puede apreciar visualmente la mejora respecto la versión sin binarios, a esta, sin ficheros y con otras optimizaciones, que al hacerlo todo en memoria es mucho más eficiente. Esto se ve claramente en el la imagen, en el recuadro rojo, este recoge todas las funciones de generación de las señales semánticas y unos cálculos del análisis (tal como se explicó en la Figura 3.1 de la Zona 4). Lo interesante de ese cuadro, es que hay bastantes funciones que se basaban su comunicación mediante ficheros y que al optimizarlo se ve una gran mejoría en el tiempo de respuesta.

En la Figura 3.5, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión anterior.

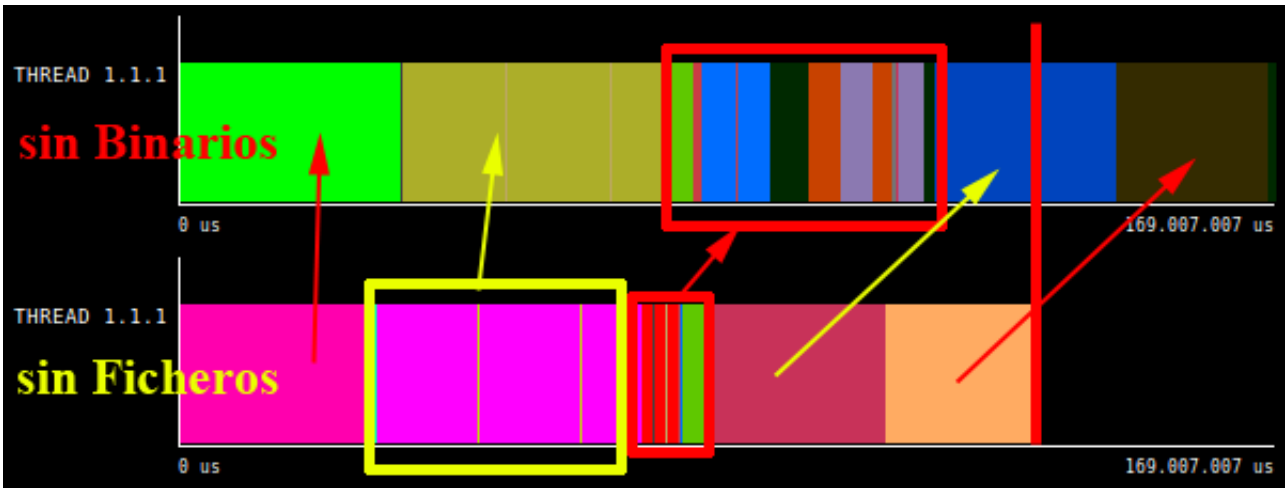

Figura 3.4: *Las dos trazas son de la aplicación con la ejecución de la traza bt.C.64. La superior es de la versión sin binarios y la inferior la de sin binarios. Las flechas indican que aunque sean de colores diferentes son las mismas funciones.*

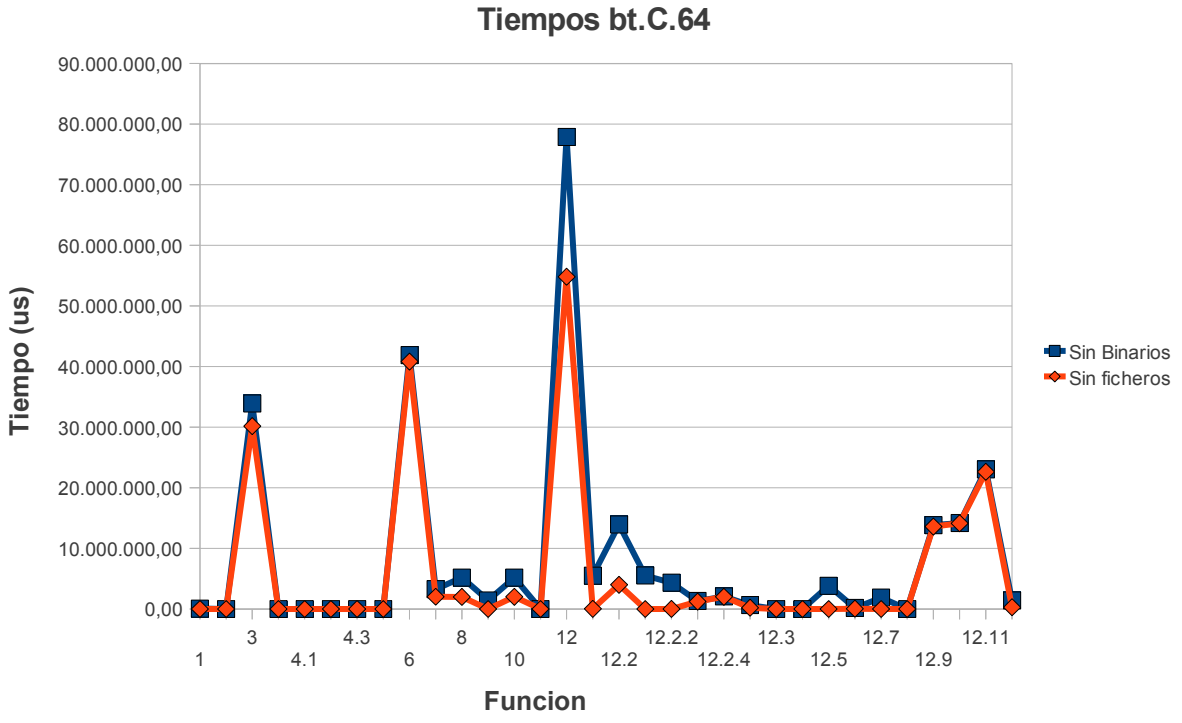

Figura 3.5: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones sin binarios y sin ficheros. La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

3.4. Versión sin repeticiones

Una vez tenía todo sin binarios y las optimizaciones que surgían al tenerlo todo en memoria (explicadas con detalle en el apartado anterior), era la hora de reducir las ejecuciones repetitivas, es decir, hay funciones que realizan el mismo trabajo o que un trozo de su código es idéntico. En consecuencia, esas funciones, hay que fusionarlas para no tener que repetir esos trozos de código, reduciendo el tiempo de ejecución y explotando los datos locales de la memoria en ese tiempo.

3.4.1. Funciones fusionadas

3.4.1.1. Nthreads y Totaltime

Las funciones `Nthreads` y `Totaltime` (Código 3.18), leen la primera línea de la traza para sacar la información que necesitan. `Nthreads`, el número de threads que se ejecutaron y `Totaltime`, el tiempo total que duró la ejecución del programa. En vez de tener que leer la primera línea dos veces, lo que se debe hacer es solo leerla una sola vez. En el Código 3.19, se lee la primera línea una vez, esta se guarda y luego, con esta se puede sacar las dos informaciones que necesitamos.

Código 3.18: Nthreads y Totaltime

```
1  int Nthreads(file dades)
2  {
3          #ifdef TRACE_MODE
4                          Extrae_event (1000, 7);
5          #endif
6          FILE *fp;
7          char s[10000];
8          int p;
9
10         fp=fopen(dades, "r");
11
12         fgets(s, 10000, fp);
13
14         sscanf(s, "%*s %*[^:]:%*[^:]:%*[^:]:%*[^:]:%*[^:]:%d", &p);
15
16         fclose(fp);
17         #ifdef TRACE_MODE
18                         Extrae_event (1000, 0);
19         #endif
20         return p;
21 }
22
23
24 long long int Totaltime(file dades)
25 {
26         #ifdef TRACE_MODE
27                         Extrae_event (1000, 6);
28         #endif
29
30         FILE *fp;
31         char s[100];
32         long long int t;
33
34         fp=fopen(dades, "r");
35
36         if(fp==NULL)
37         {
38                 printf("Error loading file\n");
39                 exit(-2);
40         }
41
42         fgets(s, 100, fp);
43
44         sscanf(s, "%*s %*[^:]:%*[^:]:%lld", &t);
45
```

```
46          fclose(fp);
47          #ifdef TRACE_MODE
48                          Extrae_event (1000, 0);
49          #endif
50          return t;
51  }
```

Código 3.19: Nthreads y Totaltime fusionadas

```
1  kp=fopen(argv[trace_num],"r");
2  /*Extraction of the execution time and the number of threads*/
3  fgets(s, 10000, kp);
4
5  //Nthreads
6  sscanf(s, "%*s  %*[^:]:%*[^:]:%*[^:]:%*[^:]:%*[^:]:%d", &p[trace_num]);
7
8  //Totaltime
9  sscanf(s, "%*s  %*[^:]:%*[^:]:%lld", &totaltime[trace_num]);
```

3.4.1.2. signalRunning y signalDurRunning

Las funciones `signalRunning` y `signalDurRunning`, leen del fichero de la misma traza filtrada por `FilterRunning`, y además, guardan su información en una misma estructura (Código 3.20) `burst_times`, y, con el mismo tamaño. Leer una misma traza dos veces es una gran penalización de tiempo, lo ideal es leerla una sola vez y sacar toda la información que se necesite en ese momento. Para solucionarlo, se ha fusionado las dos funciones en el Código 3.21, de tal manera, que al ir leyendo la traza para cada evento que les corresponde a los dos, los resultados para generar la señal, se guarda en dos tablas diferentes, una para `signalRunning` y la otra para `signalDurRunning`. Así, se evita la penalización de leer dos veces un mismo fichero.

Código 3.20: signalRunning y signalDurRunning

```
1  int SignalRunning(char * input, struct signal_info *signal)
2  {
3          ....
4          /* Parsing trace */
5          while(fgets(buffer, READ_BUFFER_SIZE, fileIn) != NULL)
6          {
7                  if(buffer[0] == '1')
8                  {
9                          sscanf(buffer, "%*d:%*d:%*d:%*d:%*d:%lld:%lld:%d\n", &time_1, &time_2, &
                               state);
10                         if(state == 1)
11                         {
12                                 burst_times[num_times].time = time_1;
13                                 burst_times[num_times].value = 1;
14                                 burst_times[num_times+1].time = time_2;
15                                 burst_times[num_times+1].value = -1;
16                                 num_times+=2;
17                         }
18                 }
19         }
20
21         .....
22         //generar señal con burst_times
23 }
24
25 int SignalDurRunning(char * input, struct signal_info *signal)
26 {
27         .....
28         /* Parsing trace */
29         while(fgets(buffer, READ_BUFFER_SIZE, fileIn) != NULL)
30         {
31                 if(buffer[0] == '1')
32                 {
```

```
33                          sscanf(buffer, "%*d:%*d:%*d:%*d:%*d:%lld:%lld:%d\n", &time_1, &time_2, &
                                state);
34                          if(state == 1)
35                          {
36                                  burst_times[num_times].time = time_1;
37                                  burst_times[num_times].value = time_2 - time_1;
38                                  burst_times[num_times+1].time = time_2;
39                                  burst_times[num_times+1].value = -(time_2 - time_1);
40                                  num_times+=2;
41                          }
42                  }
43          }
44          ....
45          //generar señal con burst_times
46  }
```

Código 3.21: signalRunning y signalDurRunning fusionados

```
1   void SignalRunning_DurRunning(char * input, struct signal_info *signal,int *size,struct signal_info
        *signal2,int *size2)
2   {
3           .....
4           /* Parsing trace */
5           while(fgets(buffer, READ_BUFFER_SIZE, fileIn) != NULL)
6           {
7                   if(buffer[0] == '1')
8                   {
9                           sscanf(buffer, "%*d:%*d:%*d:%*d:%*d:%lld:%lld:%d\n", &time_1, &time_2, &
                                state);
10                          if(state == 1)
11                          {
12                                  //signalRunning
13                                  burst_times[num_times].time = time_1;
14                                  burst_times[num_times].value = 1;
15                                  burst_times[num_times+1].time = time_2;
16                                  burst_times[num_times+1].value = -1;
17
18                                  //signalDurRunning
19                                  burst_times2[num_times].time = time_1;
20                                  burst_times2[num_times].value = time_2 - time_1;
21                                  burst_times2[num_times+1].time = time_2;
22                                  burst_times2[num_times+1].value = -(time_2 - time_1);
23                                  num_times+=2;
24                          }
25                  }
26          }
27
28          .....
29          //generar señales signalRunning con burst_times y signalDurRunning con burst_times2
30  }
```

3.4.1.3. Hacer solo una señal con signalDurRunning

Más que fusionar funciones, es ahorrase una de ellas, no repetir el mismo trabajo. Tal como se mencionó en el apartado "3.3.1.1. Estructuras en vez de ficheros de texto", en la Versión de sin ficheros, gracias a tener las señales en una estructura, es posible aplicar un truco para generar sólo una vez esa señal.

La situación, es que para poder afinar las zonas de corte, tal como se explicó en el apartado "2.5. Lectura del código fuente", se utilizaba una señal en concreto para poder examinar los máximos, la que generaba `SignalDurRunning`, y, además con la opción `CPUDurBurst`, también generaba esta señal. Al ser una estructura en memoria, no es necesario generarla dos veces, lo que hacemos es apuntar el puntero de una con la otra. En el Código 3.22, podemos representar `signal` como la señal generada en `CPUDurBurst` y `signal3` representa la señal a generar para la búsqueda de los

máximos, sólo se ha de generar una vez en `signal` y luego copiar su puntero a `signal3`. De esta manera, solamente generaremos una vez esa señal, y, si más adelante hay un cambio en alguna de las dos señales, se tiene en cuenta, que al cambiar una no modifique la otra (Código 3.23).

Código 3.22: Generar la señal una sola vez

```
signal3=signal;//misma señal
sizeSig3=sizeSig;//tamaño es el mismo
```

Código 3.23: Teniendo en cuenta el cambio de una señal

```
//si cambia signal, no borrar signal3
if(signal3 == signal)
{
        //signal3 == signal, entonces signal necesita una @ nueva
        signal = (struct signal_info *)malloc( SIZE_SIGNAL* sizeof(struct signal_info));
}
```

3.4.1.4. cutter_signal y Maximum

Después de hacer el análisis de las zonas de corte, hay que afinarlas y que al cortar la traza original su tamaño resultante sea igual al que exige el usuario. Para poder hacerlo, es necesario primero cortar en la zona que nos ha dado el análisis la señal semántica y luego buscar el máximo. Originalmente, para cortar la señal se utilizaba la función `cutter_signal`, que nos corta la señal en una zona que le determinamos devolviéndonos la señal resultante. Acto seguido, con la señal nueva deseada, se ponía a buscar con la función `Maximum` (Código 3.24) y nos devolvía el punto máximo de esta.

Código 3.24: cutter_signal y Maximum

```
int Cutter_signal(struct signal_info *signal, int sizeSig,struct signalFloat_info *signal2, long
    long int t0, long long int t1)
{
        #ifdef TRACE_MODE
                Extrae_user_function(1);
        #endif
        int i,size=0;

        /* Cutting signal */
        i=0;
        while(i<sizeSig && ((signal[i].t + signal[i].delta) < t0))
        {
                i++;
        }

        if(i<sizeSig && ((signal[i].t + signal[i].delta) >= t0))
        {
                signal2[size].t=0;
                signal2[size].delta=signal[i].t + signal[i].delta - t0;
                signal2[size].semanticvalue=(float)signal[i].semanticvalue;
                size++;
                i++;
        }

        while(i<sizeSig && ((signal[i].t + signal[i].delta) <= t1))
        {
                signal2[size].t=signal[i].t - t0;
                signal2[size].delta=signal[i].delta;
                signal2[size].semanticvalue=(float)signal[i].semanticvalue;
                size++;
                i++;
        }

        if(i<sizeSig && ((signal[i].t + signal[i].delta) > t1))
```

```
34          {
35                  signal2[size].t=signal[i].t - t0;
36                  signal2[size].delta=t1 - signal[i].t;
37                  signal2[size].semanticvalue=(float)signal[i].semanticvalue;
38                  size++;
39          }
40          #ifdef TRACE_MODE
41                  Extrae_user_function(0);
42          #endif
43          return size;
44  }
45
46  long long int Maximum(struct signalFloat_info *signal, int sizeSig)
47  {
48          #ifdef TRACE_MODE
49                  Extrae_user_function(1);
50          #endif
51
52          long long int max, maxt, maxdelta, t, delta;
53          double value;
54          int i;
55          max=0;
56          maxt=0;
57          maxdelta=0;
58          //while(fscanf(fp, "%lld %lld %lf", &t, &delta, &value)==3)
59          for(i=0;i<sizeSig;i++)
60          {
61                  if(signal[i].semanticvalue>max)
62                  {
63                          max=signal[i].semanticvalue;
64                          maxt=signal[i].t;
65                          maxdelta=signal[i].delta;
66                  }
67          }
68
69          #ifdef TRACE_MODE
70                  Extrae_user_function(0);
71          #endif
72          return (maxt+maxdelta/2);
73  }
```

La ineficiencia que se detecta, en todo este proceso es, que primero corta una señal y luego de esta busca su máximo, en este punto ya se ha leído dos veces la señal, además después esta señal cortada no se vuelve a utilizar. No hace falta crear una señal nueva, que no se va utilizar más que para buscar un máximo, es mejor, sólo leer la señal y mientras buscar el máximo.

Tal como se muestra en el Código 3.25, se lee la señal y en vez de crear una nueva señal cortada en la zona que se indica como parámetro, esta no hace el código de generarla, sino ejecuta el código de la búsqueda del máximo. De esta manera, no solo se ha fusionado las dos funciones, haciendo que solo se hiciera una sola lectura de la señal, sino que también se ha evitado generar una señal inservible.

Código 3.25: Cutter_signal y Maximum fusionadas sin la generación de la señal

```
1   long long int Cutter_signal_Maximum(struct signal_info *signal, int sizeSig,long long int t0, long
        long int t1)
2   {
3           long long int max, maxt, maxdelta;
4           int i;
5           max=0;
6           maxt=0;
7           maxdelta=0;
8
9           /* Cutting signal */
10          i=0;
11          while(i<sizeSig && ((signal[i].t + signal[i].delta) < t0))
12          {
13                  i++;
```

```
14          }
15
16          if(i<sizeSig && ((signal[i].t + signal[i].delta) >= t0) && (float)signal[i].semanticvalue>
                max)
17          {
18                  max=(float)signal[i].semanticvalue;
19                  maxt=0;
20                  maxdelta=signal[i].t + signal[i].delta - t0;
21                  i++;
22          }
23
24          while(i<sizeSig && ((signal[i].t + signal[i].delta) <= t1))
25          {
26                  if((float)signal[i].semanticvalue>max)
27                  {
28                          max=(float)signal[i].semanticvalue;
29                          maxt=signal[i].t - t0;
30                          maxdelta=signal[i].delta;
31                  }
32                  i++;
33          }
34
35          if(i<sizeSig && ((signal[i].t + signal[i].delta) > t1) && (float)signal[i].semanticvalue>max
                )
36          {
37                  max=(float)signal[i].semanticvalue;
38                  maxt=signal[i].t - t0;
39                  maxdelta=t1 - signal[i].t;
40          }
41          return (maxt+maxdelta/2);
42  }
```

3.4.2. Ganancia obtenida

En el apartado "5. Tiempos de ejecución de las diferentes versiones", se encuentran los resultados de las ejecuciones de esta versión, con el nombre **sin repeticiones**. En las tablas, se puede observar que para las funciones que se han optimizado, dan mejor rendimiento, estas son según los apartados explicados anteriormente:

- `Totaltime` **->** Fila 1
- `Nthreads` **->** Fila 2
- `signalRunning` y `signalDurRunning` **->** las celdas unidas de las filas 7 y 8
- Hacer solo una señal `signalDurRunning` **->** Fila 10
- `Cutter_signal` y `Maximum` **->** las celdas unidas de las filas 12.7 y 12.8

Con todas las mejoras realizadas, explicadas en esta versión, obtuve una mejora de un **9,84 %** respecto a la Versión sin ficheros (un **24,35 %** respecto al código original).

En la Figura 3.6, se pude ver que la ganancia obtenida mayor, es gracias a no tener que leer dos veces la traza para `signalRunning` y `signalDurRunning`, con lo que en el cuadrado rojo, se encuentran, no sólo estas funciones sino otras más, es en el lugar donde se mejora más la eficiencia.

En la Figura 3.7, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión anterior.

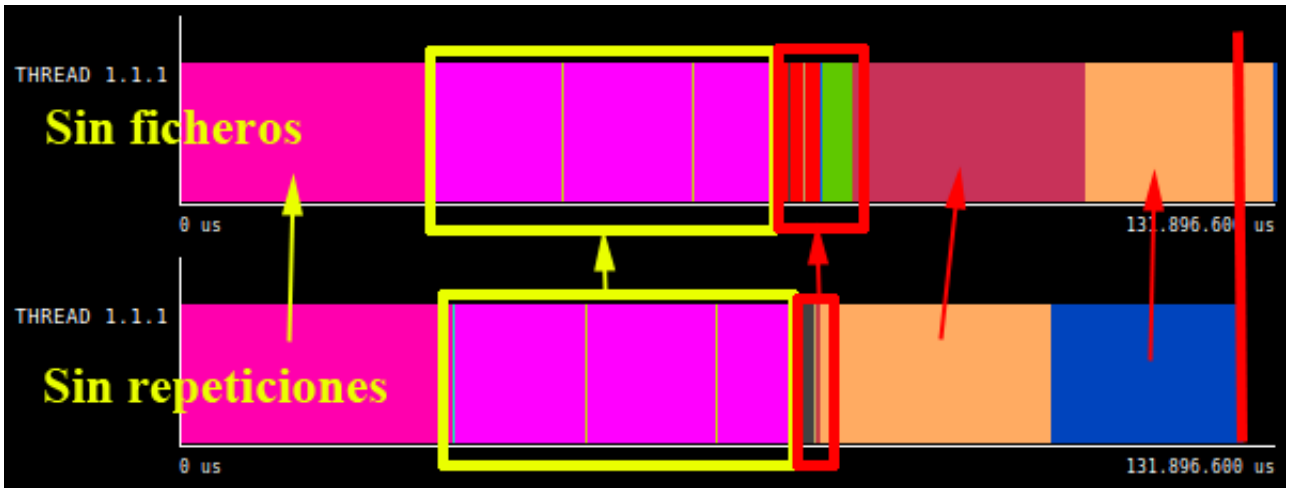

Figura 3.6: *Las dos trazas son de la aplicación con la ejecución de la traza bt.C.64. La superior es de la versión sin ficheros y la inferior la de sin repeticiones. Las flechas indican que aunque sean de colores diferentes son las mismas funciones.*

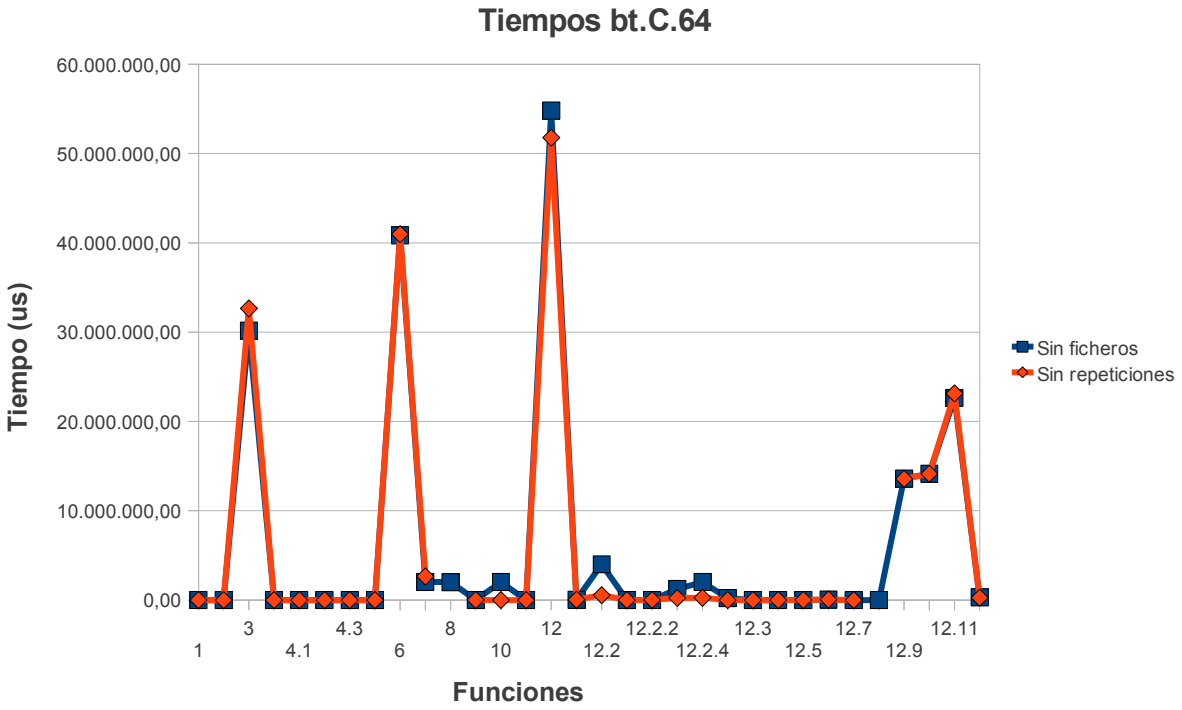

Figura 3.7: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones sin ficheros y sin repeticiones. La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

3.5. Versión reestructuración

En la anterior versión, constaba en quitar todas las repeticiones, pero tal como estaba estructurada la aplicación seguía habiendo una repetición más, nada menos la que más penaliza de todas, leer dos veces entera la traza original. Dependiendo de la traza, el tamaño puede ser considerable, con lo que no es nada factible tener que leer la traza original más de una vez. En el siguiente apartado, se explicará cómo se solucionó el problema.

También se detectó, que se utilizaba en dos ocasiones una herramienta externa de la librería TRACE2TRACE, la aplicación **Cutter**, para realizar unos cálculos para el análisis. Esta herramienta, sólo se debe utilizar al final, para cortar la traza la zona periódica, y, no utilizarla para realizar cálculos para el análisis.

3.5.1. Optimizaciones realizadas

3.5.1.1. Cambiar secuencia de ejecución

El problema de leer más de dos veces la traza original, se origina, en que primero se lee la traza para generar una señal semántica para el análisis, la de "flushing". Después, dependiendo que opción se escoge en la aplicación, esta utilizará la librería externa de TRACE2TRACE, la **trace_filter**, para filtrar la traza, los eventos que realmente se requieren, para poder generar las otras señales semánticas que se necesitan para el análisis (como la que genera `signalRunning`).

La primera señal de "flushing", para generala lo que realmente hace, es obtener de la traza original sólo los eventos de "flushing" y luego generar la señal. Si cambiamos el orden en que se ejecutan las funciones, podremos evitar hacer dos lecturas de la traza original. Para esto, primero se filtrará la traza con el **trace_filter**, con las opciones de filtrado según la opción que el usuario pasa a la aplicación, y se le añadirá, que filtre también los eventos de "flushing". De esta forma, primero se filtra la traza original, luego se genera las señales semánticas como antes (las de la opción pasada por el usuario), y además, se generará la señal semántica de "flusing" a partir de la traza filtrada. Causando, a fusionar la función que generaba `signalRunning` y `signalDurRunning`, con la de "flushing", creando una función que lee la traza filtrada y genera 3 señales semánticas a la vez.

3.5.1.2. Sacar dos ejecuciones de Cutter

Al finalizar el análisis de las zonas de corte de una iteración repetitiva, si estas son válidas, se ha de mirar el número de iteraciones que la aplicación cortará, que como resultado al cortar la traza, no supere el tamaño establecido por el usurario. El método que utiliza el código original, es buscar dos zonas de corte diferentes y cortar con la aplicación **Cutter**. Al tener las dos trazas diferentes, se comparan el tamaño final de las trazas y se queda con la que mejor se le acerque al tamaño deseado. Después, se continúa con unos cálculos para afinarlo mejor.

No se puede utilizar la herramienta Cutter, para realizar cálculos de análisis, es una gran penalización en el tiempo de ejecución de la aplicación, ya que, lee la traza original entera y escribe la nueva traza cortada deseada. No solo se ejecuta una vez, sino dos veces para realizar este análisis, añadiendo, que después esas dos trazas no serán válidas, ya que más adelante se afinará el cálculo de la zona a cortar.

Con la utilización de un heurístico, se puede sustituir las dos ejecuciones de Cutter. El problema, es buscar un heurístico que de buenos resultados, para este propósito, se hicieron pruebas en las que se destacan tres heurísticos: heurístico uniforme, heurístico del caso pero y el heurístico del promedio.

Para la implementación de los siguientes heurísticos, se necesitó unos datos que el análisis ya nos proporcionaba, entre paréntesis, el alias que le he dado, para luego hacer las fórmulas de los heurísticos:

- El tamaño total de la traza original (SizeTraza).
- El tiempo total de ejecución de la traza original (TimeTraza).
- El tiempo del período que se ha encontrado de la zona de corte (periodo).
- El número de iteraciones encontradas (Niters).

La implementación de los tres heurísticos:

Heurístico uniforme

El heurístico uniforme, es el que se estima que los eventos se distribuyen de forma uniforme, es decir, que el tamaño de las iteraciones están distribuidas de una forma uniforme. El porcentaje, del período encontrado, en relación del tiempo total de la duración de la traza, se sabe el porcentaje que ocupan las iteraciones en esa traza. De este modo, al estar distribuido de una forma uniforme, si multiplicamos el porcentaje de ocupación por el tamaño total de la traza, nos da el tamaño estimado de una iteración. Las siguientes ecuaciones, son las fórmulas que se ha explicado, el de la derecha del corche es la fórmula simplificada. Como resultado, nos da un tamaño estimado **SizeEstimado1**.

$$\left.\begin{aligned} porcentaje &= \frac{(periodo \cdot 100)}{TimeTraza} \\ SizeEstimado1 &= \frac{(porcentaje \cdot SizeTraza)}{100} \end{aligned}\right\} SizeEstimado1 = \frac{(periodo \cdot SizeTraza)}{TimeTraza}$$

Heurístico del caso peor

Como caso peor, nos ponemos en la situación en que el número de iteraciones del período ocupan casi toda la traza, no está distribuido de una forma uniforme. Simplemente, se ha de dividir el tamaño de la traza original, entre el número de iteraciones encontradas. **SizeEstimado2**, es el tamaño estimado de una iteración en la fórmula inferior.

$$SizeEstimado2 = \frac{SizeTraza}{Niters}$$

Heurístico del promedio

SizeEstimado3, es el tamaño estimado resultante de hacer el promedio de los dos heurísticos anteriores.

$$SizeEstimado3 = \frac{(SizeEstimado1 + SizeEstimado2)}{2}$$

A continuación, se muestran los resultados obtenidos de los tres heurísticos, más la ejecución del original (Cutter3), que utilizaba dos veces el Cutter. Lo interesante, es mirar que los resultados del tamaño de la traza cortada, sea parecido al que desea el usuario. Para estas ejecuciones, se especificó como tamaño resultante 100.000.000 bytes.

Los datos representados en la tabla, son los necesarios para poder aplicar las fórmulas explicadas anteriormente. También, se ha añadido el número de iteraciones que contiene la traza cortada. Se ha de añadir, en caso de que el tamaño de la traza resultante sea superior al deseado, la aplicación como mínimo dará dos iteraciones. Este es el caso, para la traza ALYA.64.

ALYA.64

	Tamaños (bytes)			Tiempos (ms)		Número de iteraciones	
Heurístico	Traza Original	Traza cortada	Estimación de una iteración	Total traza original	Periodo	Traza Original	Traza cortada
Cutter3 (original)	4.027.242.440	415.233.340	206.970.742	278.173	11.008	20,851796	2
Uniforme		415.233.340	159.368.036				2
Caso peor		415.233.340	193.136.464				2
Promedio		415.233.340	176.252.256				2

bt.C.64

	Tamaños (bytes)			Tiempos (ms)		Número de iteraciones	
Heurístico	Traza Original	Traza cortada	Estimación de una iteración	Total traza original	Periodo	Traza Original	Traza cortada
Cutter3 (original)	1.162.746.394	91.987.140	9.105.166	187.475	1.453	128,918473	10
Uniforme		101.310.552	9.011.710				11
Caso peor		101.310.552	9.019.238				11
Promedio		101.310.552	9.015.474				11

VAC.128

	Tamaños (bytes)			Tiempos (ms)		Número de iteraciones	
Heurístico	Traza Original	Traza cortada	Estimación de una iteración	Total traza original	Periodo	Traza Original	Traza cortada
Cutter3 (original)	223.047.187	97.839.091	2.144.670	503.649	4.645	106,098601	46
Uniforme		102.271.034	2.057.095				48
Caso peor		100.055.261	2.102.263				47
Promedio		102.271.034	2.079.679				48

WRF.64

	Tamaños (bytes)			Tiempos (ms)		Número de iteraciones	
Heurístico	Traza Original	Traza cortada	Estimación de una iteración	Total traza original	Periodo	Traza Original	Traza cortada
Cutter3 (original)	5.429.010.365	100.297.067	8.813.636	1.196.304	1.510	571,394818	11
Uniforme		127.609.771	6.852.610				14
Caso peor		91.219.223	9.501.329				10
Promedio		109.214.793	8.176.969				12

Los resultados revelan, que **el mejor heurístico es el del caso peor**.

Con un buen heurístico, puede darnos más o menos iteraciones, pero tal como se ha explicado, lo importante, es que el tamaño de la traza final sea el más parecido al que desea el usuario. La explicación se puede demostrar viéndolo en la Figura 3.8, donde la imagen de la izquierda, es una iteración de la traza `bt.C.64`, de su señal semántica realizada por `signalDurRunning`, y, la de la derecha, sus N iteraciones que se cortan para la traza final. Como se puede observar, si al finalizar tuviera una iteración más o una menos, la información que realmente necesita el usuario no la pierde, y, además el fichero final tendría su tamaño óptimo.

En el caso contrario, con la señal `ALYA.64` (Figura 3.9), se ha de cortar como mínimo dos iteraciones de esta. No tendrá un tamaño final ideal, pero, con dos iteraciones el usuario obtiene toda la información de la zona periódica.

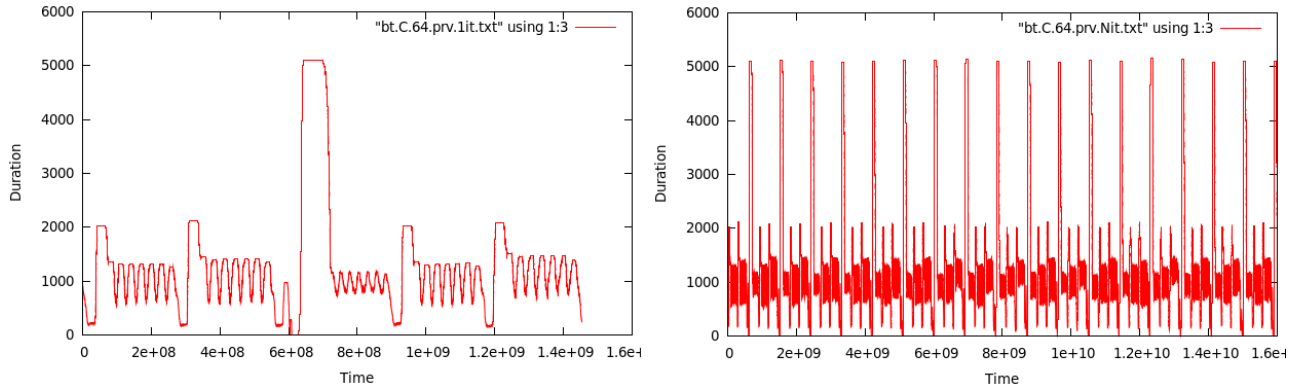

Figura 3.8: *La imagen de la izquierda es una iteración de la traza `bt.C.64`, de su señal semántica realizada por `signalDurRunning`, y, la de la derecha sus N iteraciones que se cortan para la traza final.*

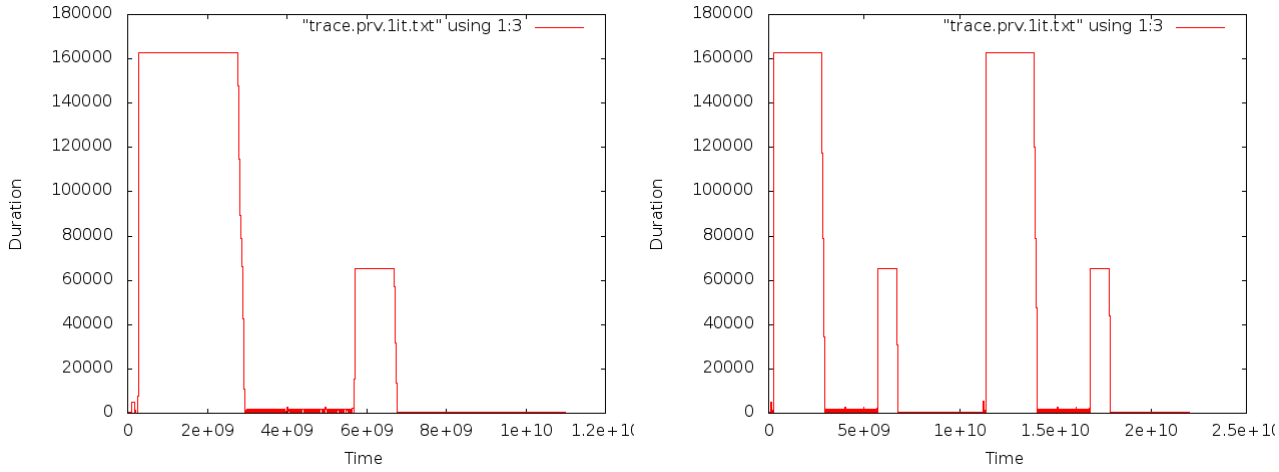

Figura 3.9: *La imagen de la izquierda es una iteración de la traza `ALYA.64`, de su señal semántica realizada por `signalDurRunning`, y, la de la derecha sus N iteraciones que se cortan para la traza final.*

3.5.2. Ganancia obtenida

En el apartado "5. Tiempos de ejecución de las diferentes versiones", se encuentran los resultados de las ejecuciones de esta versión, con el nombre **Reestructuración**. En dichas tablas, se puede observar, que para la fila 3 con esta versión no hay tiempo, esto se debe a que al reestructurar la ejecución de la aplicación, se ha creado una nueva función que genera 3 señales semánticas (`signalRunning`, `signalDurRunning` y la de "flushing"), al mismo tiempo. El tiempo para esta función, está en la fila 7 y 8 que están unidas. Resultando, una buena mejora en los tiempos de respuesta.

Hay que destacar también, que al sacar la ejecución de los dos `Cutter3` por un heurístico, dando a lugar, un tiempo de ganancia excepcional. Nos lo encontramos en la fila 12.8 y 12.9.

Con todas las mejoras realizadas, explicadas en esta versión, obtuve una mejora de un **24,99 %** respecto a la Versión sin repeticiones (un **49,34 %** respecto al código original).

En la Figura 3.10, se demuestra lo efectivo que ha sido en la reestructuración, hacer fusionar una función que genere las 3 señales, haciendo que solo se leyese una sola vez la traza original, y, eliminando dos funciones que utilizasen la aplicación "Cutter", de la librería TRACE2TRACE.

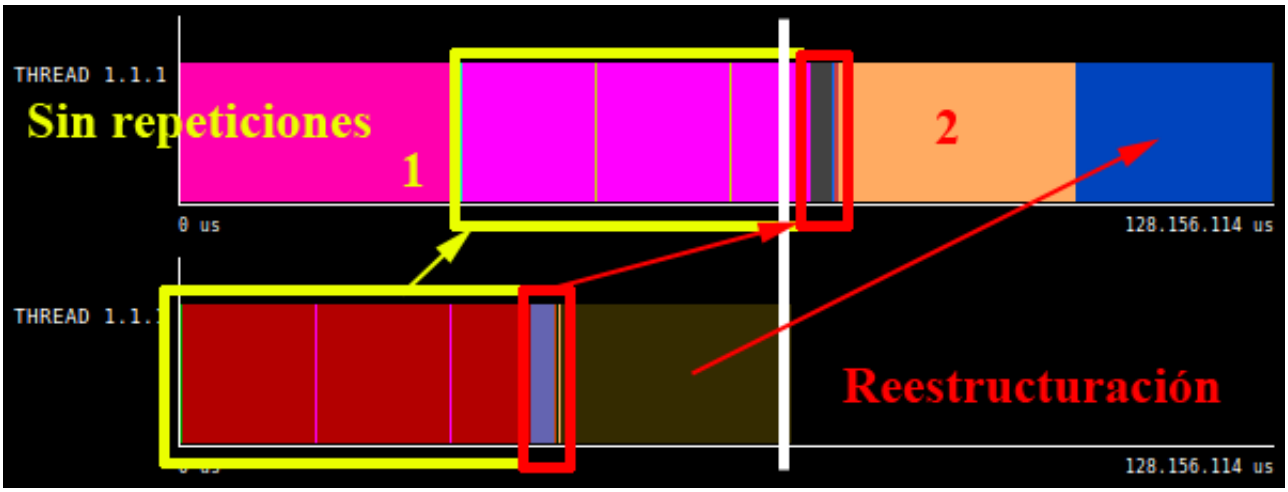

Figura 3.10: *Las dos trazas son de la aplicación con la ejecución de la traza bt.C.64. La superior, es de la versión sin repeticiones y la inferior, la de reestructuración. Las flechas indican que aunque sean de colores diferentes son las mismas funciones. Los números 1 y 2 representan a funciones que en la versión de reestructuración desaparecen.*

Con la Figura 3.11, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión anterior.

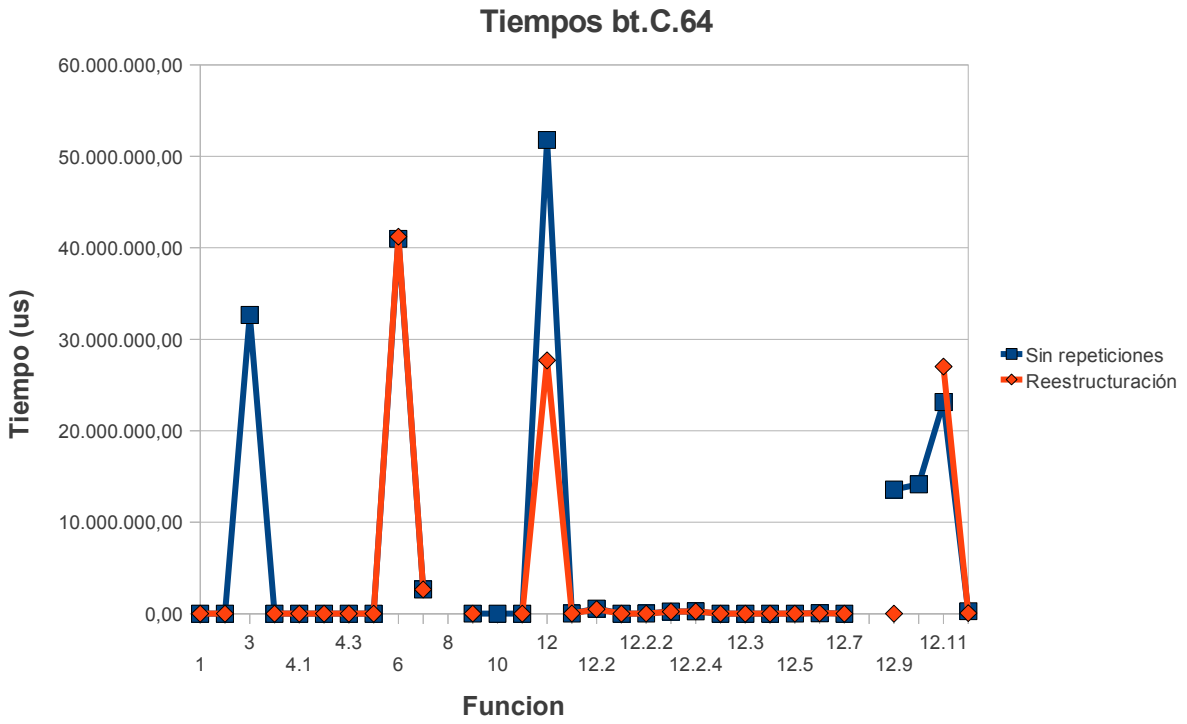

Figura 3.11: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones sin repeticiones y reestructuración. La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

4. Optimización paralela

Una vez, tenía la versión secuencial optimizada, era la hora de paralelizar la aplicación, con **StarSs**. La idea, es que se puedea ejecutar eficientemente, en máquinas de memoria compartida o multicores. Se pretende conseguir con esto, dividir el trabajo de la aplicación en los diferentes cores y que se ejecute este en paralelo, para poder acabar antes.

La versión de StarSs, que utilicé, es la de **SMPSs** (Anexo D.1), aunque más adelante, casi al finalizar el proyecto, se optó por la versión nueva **OmpSs** (Anexo D.2). Se utilizó la nueva, para corregir unos errores de ejecución daba la SMPSs (fallos de segmentación), poder paralelizar directamente en el código sin tener que crear funciones nuevas, y, el uso de una API de instrumentación con Extrae.

La información, que se ha de tener en cuenta a la hora de paralelizar, es que se han de definir, en todas las funciones, bien sus dependencias de unas a otras para una correcta ejecución, en el Anexo D hay unos ejemplos explicativos. Otro detalle importante, es que la paralelización se hace mediante tasks, como si fueran threads. Por eso, se ha de tener en cuenta que para poder crear los threads o los tasks, hay un overhead existente tanto para poder crearlos, como manejarlos. Así, que para saber si es viable poder paralelizar un trozo de código, hay que tener en cuenta la granularidad de los tasks, es de 500 μs. Teniendo en mente este dato, no valdrá la pena paralizar trozos de código que su ejecución sea menor a 500 μs.

4.1. Versión ejecución paralela

La primera versión, fue cambiar el `Makefile` y el script, que se utiliza para ejecutar las pruebas para optimizar correctamente, así para poder compilar y ejecutarlo con SMPSs. En el Anexo D.1, hay una pequeña documentación de como instalar, compilar con SMPSs y unos ejemplos del lenguaje.

Esta versión consta, en realizar la paralelización en *grano grueso*, es decir, paralelizar solo las funciones que se pueden ejecutar a la vez independientemente. Ejecutando al mismo tiempo, diferentes funciones, ganaremos tiempo de respuesta, ya que, cuando se hacía en secuencial, hasta que no acababa una, no se podía ejecutar otra. Pero de esta forma, se pude realizar diferentes trabajos a la vez. En la siguiente versión, se mirará de paralelizar en *grano fino*, en este caso, es paralelizar internamente, todas esas funciones que se ejecutan independientemente u otros trozos de código.

4.1.1. Optimizaciones realizadas

4.1.1.1. Especialización de funciones genéricas

En la versión anterior, Versión reestructuración, se utilizaba una función genérica, para crear todas las señales semánticas a partir de la traza filtrada por `FilterRunning`, según la opción que pasaba el usuario para el análisis. En esta versión, se ha aplicado la optimización de crear funciones especializadas y no genéricas. Es decir, para la función `GenerateSemanticsSignals`, que generaba todas las funciones semánticas según la situación, se han creado diferentes funciones especializadas para estas situaciones. En el caso que el usuario pasa como variable `CPUDurBurst`, se ha creado la función específica para esta situación, `Generate_Event_Running_DurRunning`, que genera sólo las señales semánticas para esta opción.

De esta forma, se obtiene funciones con menos condicionales, que como ya se ha explicado antes, es uno de los objetivos a optimizar, ya que, así no se rompe la secuencia de la ejecución, y, para los procesadores de hoy en día va mejor contra menos condicionales hayan. Se ha de añadir, que especializando esta función, tanto para esta versión, como las posteriores, será muy positivo, para crear las funciones paralelas necesarias. En consecuencia, se obtendrá un código más rápido y simple, con el que además, se podrán aplicar otras optimizaciones, según la función especializada.

4.1.1.2. Paralelización de funciones

Las siguientes funciones, fueron las que se detectarón para prodceder a convertir en tasks (para la opción `CPUDurBurst`, para las otras opciones, se han de paralelizar sus funciones específicas de generar las señales semánticas):

- **Generate_Event_Running_DurRunning**: Función que genera, la señal de "Flushing" y las señales de "burst" para las señales "DurRunning" y la "Running".
- **signalDurRunning_out**: genera la señal de "DurRunning", a partir de una señal de "burst", generada por `Generate_Event_Running_DurRunning`.
- **signalRunning_out**: genera la señal de "Running", a partir de una señal de "burst", generada por `Generate_Event_Running_DurRunning`.
- **signalChange**: el trozo de Código 3.22, explicado en el apartado "3.4.1.3. Hacer solo una señal con signalDurRunning", de la mejora para no repetir una misma ejecución de una función.
- **GetBoundary**: encuentra las zonas de corte, a partir de la seña de "flushing", generada por `Generate_Event_Running_DurRunning`.
- **Sampler_wavelet**: prepara la señal, para generar la "Wavelet", a partir de la señal de "Running", generada por `signalDurRunning_out`.
- **Wavelet_exec**: genera la señal de "Wavelet", a partir de la señal generada de `Sampler_wavelet`.
- **Sampler_double**: prepara una señal para `Crosscorrelation`, a partir de la señal principal del análisis, generada por `signalDurRunning_out` y luego cortada en el análisis, en la zona de corte a analizar.
- **Generatesinus**: genera la señal de seno, para el período de la zona de análisis encontrada para refinar.

- **Crosscorrelation**: búsqueda del mínimo, con las señales generadas por `Generatesinus` y `Sampler_double`.
- **Cutter2**: corta la traza original, en la zona periódica encontrada.
- **Cutter_signal_OutFile**: copia la señal semántica de la zona periódica encontrada, en un fichero.

El resto de funciones, no se podían ejecutar de forma paralela, o, no cumplía la granularidad de SMPSs, para que sea efectivo crear tasks. Tal como se explicó anteriormente, la granularidad de SMPSs es de unos 500 μs, en consecuencia, todas esas funciones que su ejecución sea inferior a unos 500 μs, no valen la pena paralelizarlas. Esto también ocurre con trozos de código, el ejemplo más común de paralelizar, son los bucles con iteraciones independientes. Si su ejecución tarada menos de 500 μs, es muy ineficiente hacerlo paralelo, ya que, tiene un overhead añadido a la hora de crear los tasks o enviar a los diferentes procesadores a realizar el trabajo.

En la Figura 4.1, se puede observar como se ejecuta en paralelo las diferentes funciones. `Generate_Event_Running_DurRunning` genera 3 señales. Estas pueden ser tratadas con diferentes funciones, por eso se pueden ejecutar estas en paralelo, realizando diferentes trabajos al mismo tiempo. Gracias a esto, en el caso de la función `signalRunning_out`, al terminar antes que `signalDurRunning_out`, puede ejecutarse dos funciones antes de que esta acabe, con lo que se aumenta la eficiencia del uso de los diferentes procesadores y se disminuye el tiempo de respuesta de la aplicación.

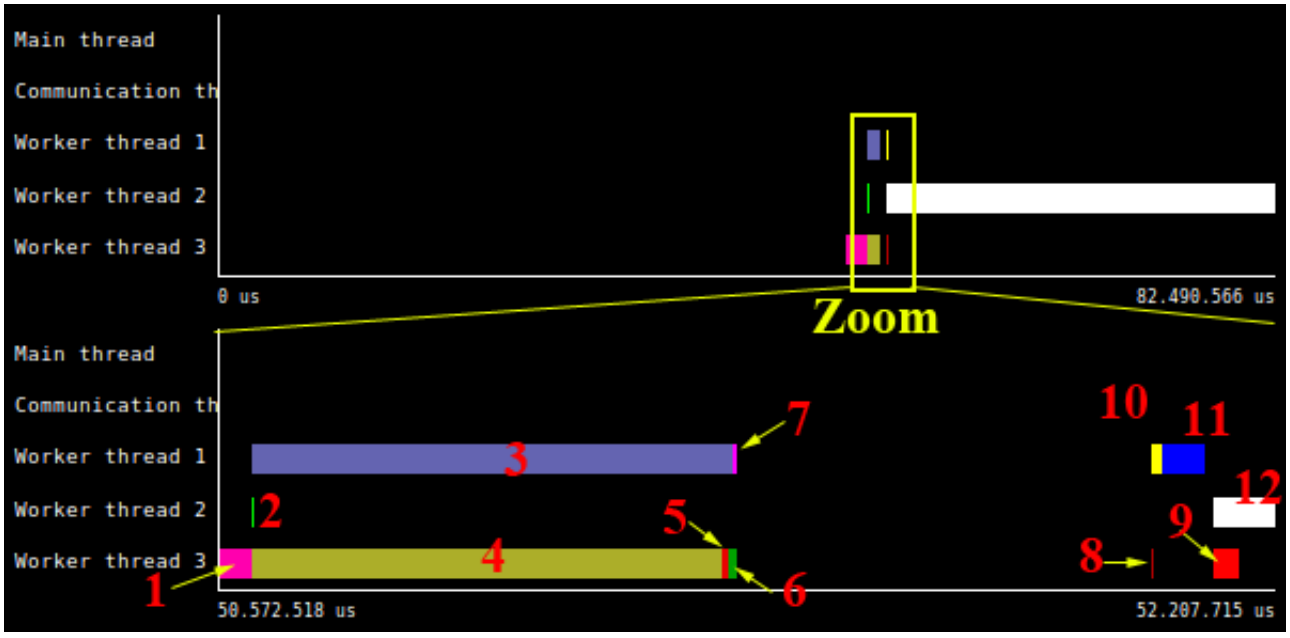

Figura 4.1: *Las dos trazas son de la aplicación con la ejecución de la traza bt.C.64 con SMPSs, la inferior es un zoom de la zona marcada con un rectángulo amarillo de la traza superior. Los números simbolizan las funciones que se han paralelizado, en el orden de 1 al 12 son las siguientes: 1. Generate_Event_Running_DurRunning, 2. GetBoundary, 3. signalDurRunning_out, 4. signalRunning_out, 5. Sampler_wavelet, 6. Wavelet_exec, 7. signalChange, 8. Generatesinus, 9. Cutter_signal_OutFile, 10. Sampler_double, 11. Crosscorrelation y 12. Cutter2.*

El código de la paralelización, se ha realizado según los ejemplos que se explica en el apartado del "Anexo D.1.2. Utilización" de SMPSs. Como es una aplicación de varios archivos con diferentes funciones, la forma de tenerlo ordenado y que funcione, es colocar todos los `pragmas` de la paralelización de las funciones paralelas, en los headers (los ficheros `.h`). El siguiente código, es del header de la función `Wavelet_exec`.

Código 4.1: wavelet.h en SMPSs

```
#ifndef _WAVELET_H
#define _WAVELET_H

#pragma css task input(n) inout(inOut[n]) output(size[1])
int Wavelet_exec(double *inOut,int n,int *size);

#endif /* _WAVELET_H */
```

Se ha de tener en cuenta siempre, en como pasar cada variable a las funciones paralelas. Con los `pragmas input`, `output` o `input`, se crea las dependencias que hay entre las diferentes funciones, pero se ha de tener en mente, que con esto no es suficiente para una correcta ejecución. No sólo se pasará como puntero las estructuras, sino, también las variables simples que se modifican de una función a otra. Con el Código 4.2, se entenderá mejor. Con la variable simple `sizeSig3`, representa el tamaño de la señal `signal3`, se ha de pasar como puntero a la función `Sampler_wavelet`, ya que, se modifica en la función que genera la señal en `signalRunning_out`. Con el `pragma input`, mirar Código 4.3, se especifica la dependencia, pero, en la hora de la ejecución, su valor no será válido. Se debe, a que se le pasará como copia a la función su valor inicial, y, no el modificado, ya que, no se le ha pasado como puntero, para poder modificarlo cuando la función `signalRunning_out` haya terminado su ejecución. En la otra situación, la variable `number_of_points`, se le pasa como variable simple, puesto que, el valor no cambia en ningún momento, así que, se le hará una copia de este y la ejecución será correcta. Por eso, las variables simples que se pasan entre diferentes funciones y que se modifican, se han de pasar como puntero, para poder tener su valor correcto en todo momento.

Código 4.2: Llamadas a funciones paralelas

```
int number_of_points=4096,sizeSig3=0;

...

signalRunning_out(burst_times[0],&num_times[0],signal3,&sizeSig3);
Sampler_wavelet(signal3,&sizeSig3,waveletSignal, number_of_points);
```

Código 4.3: Header de las funciones paralelas

```
#pragma omp task input(signal[*size],size[1],number_of_points) output(waveletSignal[number_of_points
    ])
void Sampler_wavelet(struct signal_info *signal,int *size,double *waveletSignal, int
    number_of_points);

#pragma omp task input(burst_times[*num_times],num_times[1]) output(signal[*size],size[1])
void signalRunning_out(struct burst_info *burst_times,unsigned long long *num_times,struct
    signal_info *signal,int *size);
```

4.1.2. Ganancia obtenida

El primer paso, fue mirar los tiempos de ejecución de la Versión reestructuración, compilada con el compilador de SMPSs. Antes de paralizar nada, lo que se pretendía era analizar la diferencia de tiempos en ejecución de una misma versión, pero con compiladores diferentes. Los tiempos se encuentran en el apartado de "5. Tiempos de ejecución de las diferentes versiones". Bajo el nombre de **Reestructuración**, tenemos los tiempos de la versión reestructuración, y, con el nombre de **SMPSs secuencial**, la misma versión pero compilada con el compilador de SMPSs. Viendo los resultados, se ve claramente que el compilador de SMPSs, no realiza algunas optimizaciones que el compilador de `gcc` si las hace, ya que, se pierde algo más de tiempo en las diferentes funciones. A demás, en la ejecución final de la aplicación, se ha de añadir el overhead del tratamiento de tasks, aunque no lo utilice en este caso, pero es código que se ejecuta. Compilando la versión reestructuración con SMPSs, se obtiene una pérdida de 3,82 %.

Como la aplicación se ejecuta un 3,82 % más lento que la versión reestructuración, sólo con utilizar un compilador diferente, en este momento no es muy preocupante esta pérdida, ya que, se ha de aprovechar todo el potencial de realizar trabajo en paralelo, y, este overhead se verá despreciado.

Los resultados en el apartado "5. Tiempos de ejecución de las diferentes versiones" de la versión ya paralelizada, lleva como nombre **SMPSs (smpss)**. Se obtiene una **pérdida de 3,2 %** respecto a la versión anterior, la Versión reestructuración, pero una mejora de un 0,62 % respecto a la versión de reestructuración compilada con SMPSs. Pero mirando la Figura 4.2, se pude ver que para el cuadrado rojo, esa paralización con respecto al código entero es despreciable la mejora, pero para ese trozo, se ha mejorado un **71,52 %**.

Se ha mejorado, paralelizando respecto a la versión de reestructuración compilada con SMPSs, pero al no superar realmente a la versión secuencial con `gcc`, no vale la pena paralelizarlo. Si miramos en la Figura 4.2, se ve claramente porque la optimización de esta paralelización no da tan buenos resultados. En el recuadro rojo, las únicas funciones que realmente hacen que la optimización salga bien, son las de `signalDurRunning_out` y `signalRunning_out`. El resto, está bien que se inicien rápidamente, pero no se ve una gran mejoría, porque su tiempo de ejecución es muy pequeño. Pasa lo mismo, con las funciones del recuadro amarillo, si la de `Cutter_signal_OutFile` (color rojo) su tiempo fuera mayor, al ejecutar de forma paralela con la función `Cutter2`(color blanco), que tarda bastante, el tiempo de mejoría se habría notado suficiente.

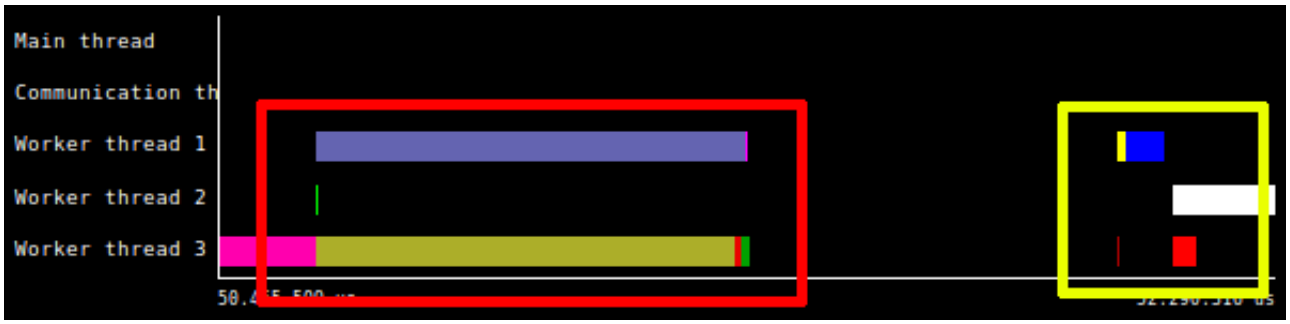

Figura 4.2: *La imagen muestra un trozo de la traza de la Figura 4.1 (en su explicación viene que funciones son), remarcando dos zonas de paralelización separadas.*

El objetivo es utilizar al máximo la paralización, viendo que esta versión no funciona, hay que mirar otros métodos. Con los resultados obtenidos se llega a la conclusión, que no solo se ha de paralelizar en *grano grueso*, ejecutando en paralelo las distintas funciones, sino, lo que se ha de intentar

en la siguiente versión es ir al *grano fino*, ejecutar en paralelo el código interno de estas funciones. De esta forma, a ver si se puede compensar el overhead de crear y manipular los tasks, que en esta versión no ha sido capaz de superar.

En la Figura 4.3, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión anterior.

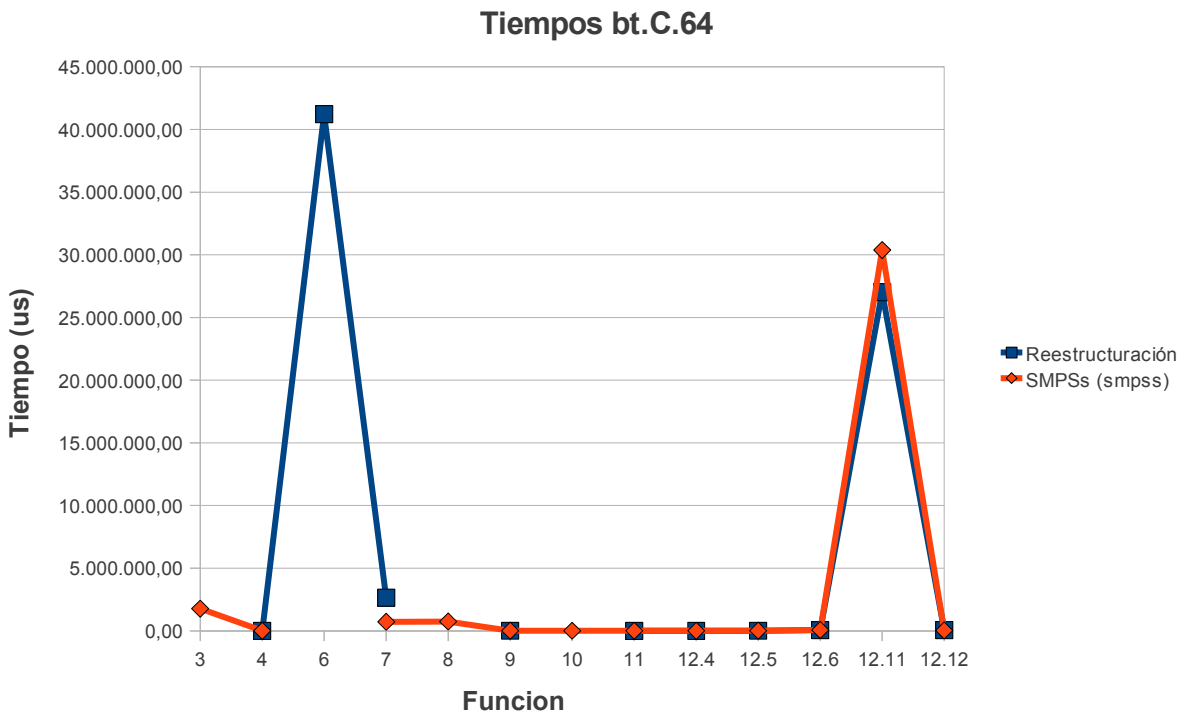

Figura 4.3: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones reestructuración y SMPSs (smpss). La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

4.2. Versión paralelización optimizada

Con la versión anterior, se hizo un gran paso detectando todas las funciones a paralelizar, pero al tener funciones con ejecuciones pequeñas, el resultado final, es que no se llega a aprovechar bien de la paralelización. Tal como se ve en la Figura 4.1, hay muchos huecos libres, es decir, hay muchos threads que su procesador está libre. Para mejorar, hay que rellenar todos estos huecos y la solución es paralelizar en *grano fino*, paralelizando las funciones su código interno u otros trozos del código suelto.

Lo que se hará, es paralelizar el código interno de estas funciones ya paralelas, dado que los trozos de código sultos con la versión SMPSs, se ha creado funciones específicas para los trozos paralelos como por ejemplo, se ha creado la función `signalChange`, para el trozo de Código 3.22.

4.2.1. Optimizaciones realizadas

La paralelización en *grano fino*, se paralelizará el código interno de las funciones paralelas. Con la versión **SMPSs**, se ha de crear nuevas funciones. Pero, para no tener que hacerlo, se optó en cambiar a la nueva versión **OmpSs**, que se puede paralelizar trozos de código al estilo de **OpenMP**, sin necesidad de crear funciones nuevas.

El primer paso, fue modificar toda la aplicación para adaptarla a OmpSs, para esto, está explicado en el Anexo D.2 con detalle. Las modificaciones importantes que se aplicaron fueron las siguientes:

- Se modificó el `Makefile` y el script, para compilar con el nuevo compilador.
- Se cambió todos los `pragmas` de SMPSs para la nueva versión OmpSs, como `css` por `omp` o `barrier` por `taskwait`.
- Los `pragmas` de inicio (`#pragma css start`) y fin (`#pragma css finish`) de la zona paralela se quitaron, ya que, en esta versión no son necesarios.

Con la nueva versión, no sólo me permitió poder paralelizar trozos de código suelto, sino que, se consiguió solucionar algunos problemas de ejecución que tenía la versión antigua (como fallos de segmentación) y las funciones se ejecutaban unos microsegundos más rápidas (una mejora de unos "μs" es poco tiempo, pero siempre viene bien tiempo extra).

Para saber que funciones se han de paralelizar su código interno para rellenar los huecos, se seguirá la mima regla que siempre, toda función que dure más de 500 μsson candidatas a paralelizar. En este caso nos encontramos con:

- **Generate_Event_Running_DurRunning**: es la función que lee la traza filtrada de `Filter-Running`, su optimización se explica en el apartado "4.2.1.1. Blocking de las funciones para generar las señales semánticas", esto va para todas las funciones que se especializaron para generar las señales semánticas.
- **signalDurRunning_out** y **signalRunning_out**: no se puede paralelizar más, ya que el 91 % del tiempo de ejecución de la función es sobre un QuickSort. Se intentó cambiar el algoritmo, para sacar el QuickSort, pero lo tiempos que se conseguían eran peores que utilizándolo.
- **Cutter2**: se usa una herramienta externa, la "Cutter" de TRACE2TRACE, no se puede paralelizar.
- **FilterRunning**: no es una función paralela, esta función llama a la herramienta externa "trace_filter" de la librería TRACE2TRACE. Pero, es la función que más tarda de la aplicación, así que, ya que se dispone del código fuente, se paralelizará. Se explica con detalle en el apartado "4.2.1.2. Blocking herramienta externa trace_filter".

4.2.1.1. Blocking de las funciones para generar las señales semánticas

Esta optimización, se aplicó para todas las funciones que generan las señales semánticas. En este caso, se expondrá el de la `Generate_Event_Running_DurRunning`. La función, lo que hace, es leer una traza línea a línea. La forma de optimizar este proceso, es dividir el trabajo entre los diferentes cores. La idea es utilizar la técnica del *blocking*, se divide la traza en un número determinado de bloques, en este caso, es igual al número de threads ejecutados. Cada thread analizará su bloque, su trozo del fichero de la traza, para generar la señal.

En este caso, tenemos 3 señales, la de "flushing" y dos de "burst". La idea, será tener 2 funciones separadas, una que analice el bloque actual para la señal de "flushing" y la otra para generar las dos de "burts". El Código 4.4, se demuestra que la función `get_Burst_Running_DurRunning` analizará el bloque para las dos señales de "burst" y `get_FlushingSignal` la señal de "flushing".

Código 4.4: Gestión del blocking de las señales semánticas

```
1  struct burst_info *burst_times[NUM_SIGNALS+1],*burst_times2[NUM_SIGNALS+1];
2  unsigned long long num_times[NUM_SIGNALS+1],num_times2[NUM_SIGNALS+1];
3
4  ...
5
6  //Inicialización  de las señales intermedias
7  for(j=0;j<NUM_SIGNALS+1;j++)
8  {
9          burst_times[j] = (struct burst_info *)malloc( SIZE_SIGNAL* sizeof(struct burst_info));
10         burst_times2[j] = (struct burst_info *)malloc(SIZE_SIGNAL * sizeof(struct burst_info));
11         num_times[j]=0;
12 }
13
14 ...
15
16 /* Parsing trace */
17 //Tamaño fichero
18 fseek(kp, 0L, SEEK_END);
19 total_size=ftello64(kp);
20 fclose(kp);
21
22 offset=total_size/NUM_SIGNALS;//Tamaño del bloque, NUM_SIGNALS es el número de threads
23 k=1;
24
25 //el primer bloque no necesita merge_burst
26 get_Burst_Running_DurRunning(filename,offset,0,burst_times[0],burst_times2[0],&num_times[0]);
27 get_FlushingSignal(filename,offset,j,signal2,&sizeSig2,&last_time,&signalValue);
28 //los siguientes bloques
29 for(j=offset;j<total_size-offset+1;j+=offset)
30 {
31         get_Burst_Running_DurRunning(filename,offset,j,burst_times[k],burst_times2[k],&num_times[k])
               ;
32         merge_burst(burst_times[0],burst_times2[0],&num_times[0],burst_times[k],burst_times2[k],&
               num_times[k]);
33         get_FlushingSignal(filename,offset,j,signal2,&sizeSig2,&last_time,&signalValue);
34         k++;
35 }
36
37 //trozo que queda si total_size mod(NUM_SIGNALS) != 0
38 if(offset*NUM_SIGNALS != total_size)
39 {
40         get_Burst_Running_DurRunning(filename,offset,offset*4,burst_times[k],burst_times2[k],&
               num_times[k]);
41         merge_burst(burst_times[0],burst_times2[0],&num_times[0],burst_times[k],burst_times2[k],&
               num_times[k]);
42         get_FlushingSignal(filename,offset,offset*4,signal2,&sizeSig2,&last_time,&signalValue);
43
44 }
```

La finalidad de la técnica de *blocking*, es crear bloques donde después se analizarán cada uno y se pondrá el resultado en la posición que toca. El problema, en el caso de las señales de "burst", es

que no sabemos cuántos eventos de "burst" nos encontraremos. En consecuencia, no se puede saber desde que posición inicial y final de la señal resultante, toca para cada bloque. El truco que se ha realizado, es inicializar para cada thread una señal de "burts" resultante, y, después con la función `merge_burst` las unirá.

Para el caso de la generación de "flushing", no hará falta el truco de la de "burst", puesto que, tiene una dependencia, en que para ir generando la función necesita un valor anterior, con lo que, ya se concatenara ella sola.

Código 4.5: Header de las funciones para blocking

```
1 #pragma omp task input(burst_times3[*num_times2],burst_times4[*num_times2],num_times2[1]) inout(
      burst_times[*num_times+*num_times2],burst_times2[*num_times+*num_times2],num_times[1])
2 void merge_burst(struct burst_info *burst_times,struct burst_info *burst_times2,unsigned long long *
      num_times,struct burst_info *burst_times3,struct burst_info *burst_times4,unsigned long long *
      num_times2);
3
4 #pragma omp task input(input[1],total_size,read_point) inout(signal2[40000000],size2[1],last_time
      [1],signalValue[1])
5 void get_FlushingSignal(char * input,int total_size,int read_point ,struct signal_info *signal2,int
      *size2,long long int *last_time,long long int *signalValue);
6
7 #pragma omp task input(input[1],total_size,read_point) output(burst_times[40000000],burst_times2
      [40000000],num_times[1])
8 void get_Burst_Running_DurRunning(char * input,int total_size,int read_point ,struct burst_info *
      burst_times,struct burst_info *burst_times2,unsigned long long *num_times);
```

En la Figura 4.4, nos muestra la ejecución paralela de estas funciones de blocking. La función `Generate_Event_Running_DurRunning` de la versión anterior con SMPSs para la traza `bt.C.64`, se puede ver los resultados en el apartado "5. Tiempos de ejecución de las diferentes versiones" con el nombre de **SMPSs (smpss)**, tardaba **1.771.337,28** μs. Ahora, la función con la técnica de blocking que se ha aplicado, tarda unos **581.051,11** μs. Tarda menos, porque se ha dividido el trabajo entre los diferentes cores rellenando esos huecos.

Se explicó que, con la función `get_FlushingSignal`, por culpa de las dependencias que tiene, al final, se concatenaría la señal sin necesidad de una función para unir los trozos, y, tal como muestra la imagen, el scheduling de OmpSs ha sido inteligente y le ha dado al mismo core a todas esas ejecuciones, en vez de partirlas, ya que la ejecución de una de estas funciones necesita el valor de la anterior.

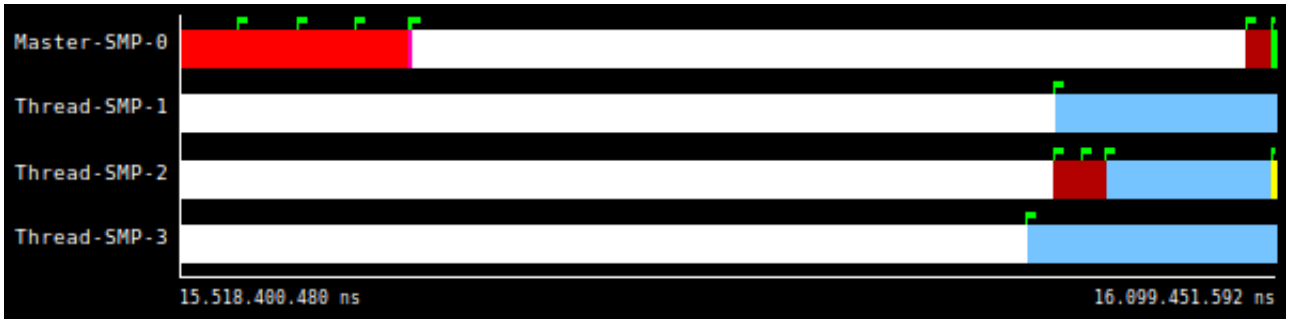

Figura 4.4: *La imagen muestra la ejecución en OmpSs de la traza bt.C.64. Sólo se muestra la parte de la ejecución de blocking de la función Generate_Event_Running_DurRunning de la versión de SMPSs. Los colores de los trozos simbolizan las funciones de blocking: los rojos son las funciones get_FlushingSignal, blancos get_Burst_Running_DurRunning, marrón merge_burst y el azul son zonas inactivas.*

4.2.1.2. Blocking herramienta externa trace_filter

La función `FilterRunning`, es la que más tarda de toda la aplicación, así que, para poder reducir el tiempo de ejecución, esta se ha de optimizar. La función, llama a la herramienta externa "trace_filter", de la librería TRACE2TRACE. El proyecto, no tiene como objetivo optimizar las herramientas externas, pero ya que disponemos del código, para reducir los tiempos, se ha de hacer algo. Como este se compila con OmpSs, no se pierde nada paralelizar su código.

La funcionalidad de "trace_filter", es parecida a la función de `Generate_Event_Running_DurRunning`, de la versión anterior de SMPSs. Lee la traza línea a línea y dependiendo de la situación, genera una traza nueva. La idea, es utilizar la misma tenica de blocking, que se utilizó para `Generate_Event_Running_DurRunning`, en el apartado anterior, generar un número determinado de bloques y que cada thread analizase su bloque.

El código fuente tenía ya hecha una paralización similar que la de blocking, pero escrita para **OpenMP**. Así que, se procedió a añadir compatibilidad con **OmpSs**. El Código 4.6, muestra una simplificación de como estaba paralelizado para OpenMP. Este, lo que hace es, para cada thread leer un trozo de la traza entrante, analizarlo línea a línea, y, generar en un fichero temporal cada thread su trozo de traza nueva. Al finalizar todos los threads, se unen todos los ficheros temporales en uno solo, dando como resultado la traza final.

Código 4.6: trace_filter en OpenMP

```
...

unsigned long long num_records = 0;//número de registros traza resultante

...

#pragma omp parallel reduction(+ : num_records) default(shared) private([lista de variables])
{
        //abrir fichero temporal de la traza para este thread
        ...
        //búsqueda de la posición inicial en la traza entrante a empezar a leer para el bloque de
            este thread
        ...
        while([leer línea a línea bloque de la traza a analizar])
        {
                ...

                if([línea actual es la que se busca])
                {
                        //tratamiento de la línea actual
                        ...
                        //imprimir línea tratada al fichero temporal del thread
                        ...

                        //aumentar contador de líneas global
                        num_records++;
                }

                ...
        }

        //cerrar fichero temporal
        ...
}

//unir en un fichero todas las trazas temporales realizadas por los threads
//num_records es el número de líneas que tendrá la traza final

...
```

Como se está utilizando la versión OmpSs, se puede paralelizar al estilo de OpenMP, sin necesi-

dad de crear funciones nuevas como en SMPSs. El Código 4.7, se muestra los cambios significativos que se han de hacer para pasar los programas de OpenMP a OmpSs. En el Anexo D.2.2, hay ejemplos de la utilización de OmpSs. OmpSs se trata con tasks, con lo que, se cambia el `pragma parallel`, por `task`, y, como no se hacen automáticamente, con un bucle (remarcado como BUCLE1), se genera todos los tasks. En OpenMP, se ha definido la variable `num_records` como `reduction`, en OmpSs se define como una variable `concurrent`, y, después para tratarla se usará el `pragma atomic`. Como se está utilizando tasks, antes de juntar todos los ficheros en uno, se tendrá que esperar a que todos los threads realicen su trabajo, para eso, se esperará con el `pragma taskwait`. Las funciones, como `omp_get_num_threads`, para saber el número de threads que hay, en OmpSs son como OpenMP (se explica en el Anexo D.2.2).

Código 4.7: trace_filter en OmpSs

```
...

unsigned long long num_records = 0;//número de registros traza resultante

...

int iGen;
for(iGen=0;iGen<omp_get_num_threads();iGen++)//para generar los tasks
{
        #pragma omp task concurrent(num_records) default(shared) private([lista de variables])
        {
                //abrir fichero temporal de la traza para este thread
                ...
                //búsqueda de la posición inicial en la traza entrante a empezar a leer para el
                    bloque de este thread
                ...
                while([leer línea a línea bloque de la traza a analizar])
                {
                        ...

                        if([línea actual es la que se busca])
                        {
                                //tratamiento de la línea actual
                                ...
                                //imprimir línea tratada al fichero temporal del thread
                                ...

                                //aumentar contador de líneas global
                                #pragma omp atomic
                                num_records++;
                        }

                        ...
                }

                //cerrar fichero temporal
                ...
        }
}
// Esperar que acaben todos los tasks
#pragma omp taskwait

//unir en un fichero todas las trazas temporales realizadas por los threads
//num_records es el número de líneas que tendrá la traza final

...
```

La Figura 4.5, muestra la paralización de "trace_filter", como se ejecuta en bloques según el número de threads. El tiempo que tarda se calcula, casi desde el principio de los trozos azules oscuros hasta la barrera que forman los trozos blancos (`get_Burst_Running_DurRunning`) y los rojos (`get_FlushingSignal`). Los trozos azules oscuros, son los tasks generados, y, el tiempo de la zona no paralela (trozo azul claro), es cuando hace la unión de los ficheros temporales y la preparación para las siguientes tareas. En la versión anterior con SMPSs, se puede ver los resultados en el apartado

"5. Tiempos de ejecución de las diferentes versiones", con el nombre de **SMPSs (extrae)**, tarda unos **50.325.150,48** μs y ahora nos tarda unos **14.467.036,74** μs. Viendo los resultados, esta optimización ha sido muy beneficiosa, ya que reduce el tiempo, a la función que más tarda de la aplicación.

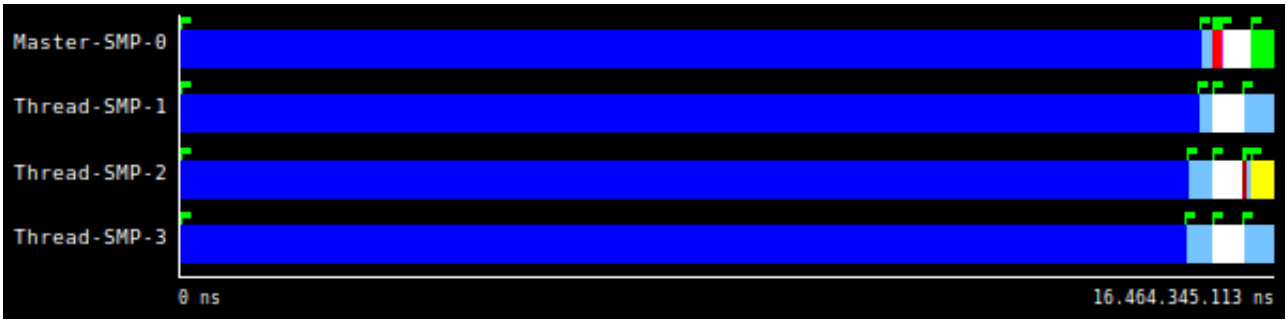

Figura 4.5: *La imagen muestra un trozo de la ejecución de la aplicación con la traza bt.C.64. Los trozos azules oscuros, corresponden a los tasks de la herramienta "trace_filter".*

4.2.2. Ganancia obtenida

Antes de buscar los tiempos de ejecución para esta versión, se ha de encontrar el número de threads que se utilizarán para ejecutar la aplicación. En mi PC, se encontró que el número óptimo eran **4 threads**. Se llegó a la conclusión, que cuanto más threads se le asignaba, peor era la ejecución. En la Figura 4.6, se ve claramente, que contra más threads, la ejecución de la función `Cutter2` va a peor, aunque, para el blocking de "trace_filter", se obtiene alguna mejora. En el gráfico de la Figura 4.7, se puede observar que con más de 4 threads, la aplicación va peor, porque el tiempo de ejecución del "Cutter" aumenta, aunque el de "trace_filter" disminuya.

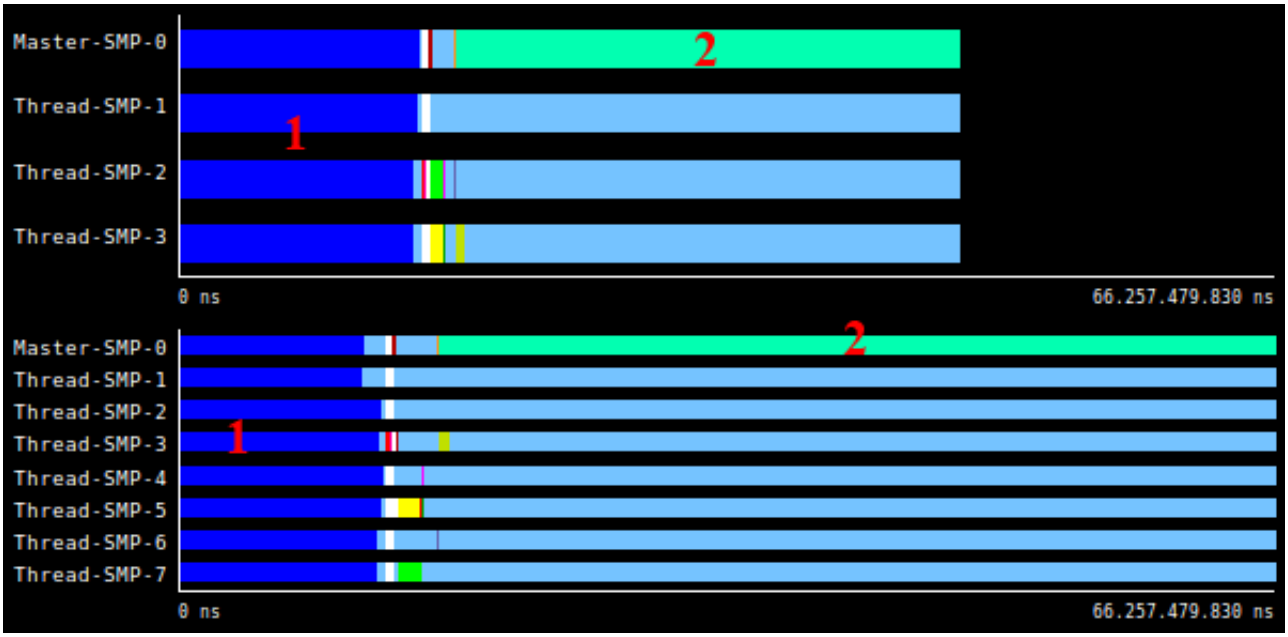

Figura 4.6: *La imagen muestra dos ejecuciones de la aplicación con OmpSs con la traza bt.C.64, la superior se le asignó 4 threads y a la inferior 8. El número 1 (los trozos azules oscuros) simboliza los tasks del blocking de "trace_filter", y, el número 2 (el trozo de verde claro) la función Cutter2.*

En la búsqueda de la cantidad de threads que se le asigna a la aplicación, se ha de tener en cuenta que dependiendo el número, la carga de la aplicación no será balanceada. En la Figura 4.8, se muestra

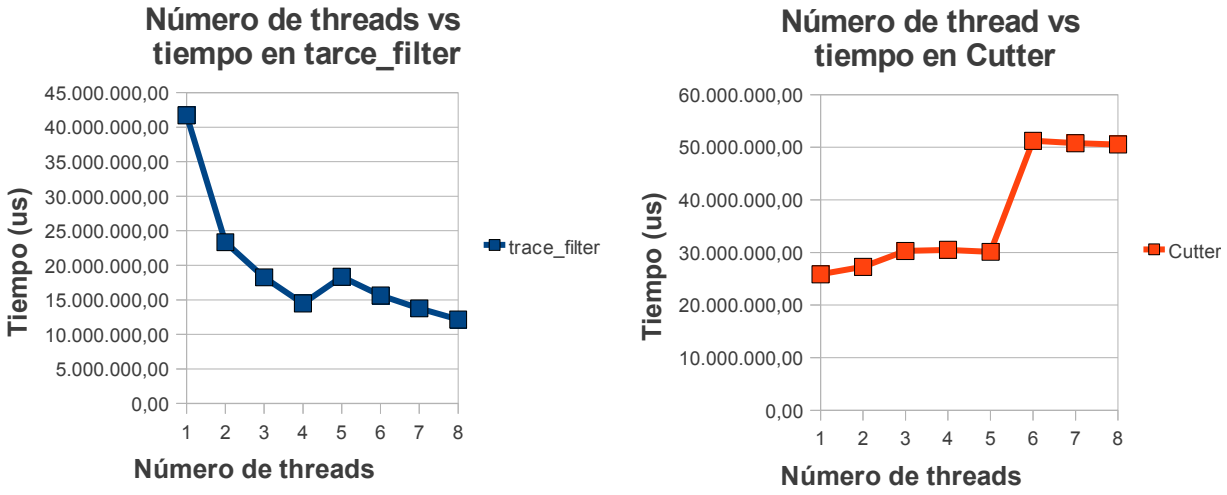

Figura 4.7: *El grafico de la derecha, muestra que con más de 4 threads, "Cutter" tarda más. En cambio, en el gráfico de la izquierda, "trace_filter" con más threads se obtiene una mejora.*

que con 5 threads, la ejecución del blocking de "tarace_filter" no esta tan balanceada, como con 4 threads.

En el apartado "5. Tiempos de ejecución de las diferentes versiones", se encuentran los resultados de las ejecuciones de esta versión, con el nombre **OmpSs**. En dichas tablas, se puede observar que la optimización del blocking de "trace_filter", los tiempos se encuentran en la fila 6, es la que ha dado mejor resultado, rebajando considerablemente el tiempo de respuesta de la aplicación.

Con todas las mejoras realizadas explicadas en esta versión, más la anterior de SMPSs, obtuve una mejora de un **29,44 %**, respecto a la Versión sin reestructuración (un **78,78 %** respecto al código original). La Figura 4.9, muestra el progreso de las optimizaciones realizadas en la versión OmpSs, con respecto a la versión de reestructuración.

En la Figura 4.10, se muestra los tiempos de todas las funciones del apartado "5. Tiempos de ejecución de las diferentes versiones", en relación con la versión de reestructuración.

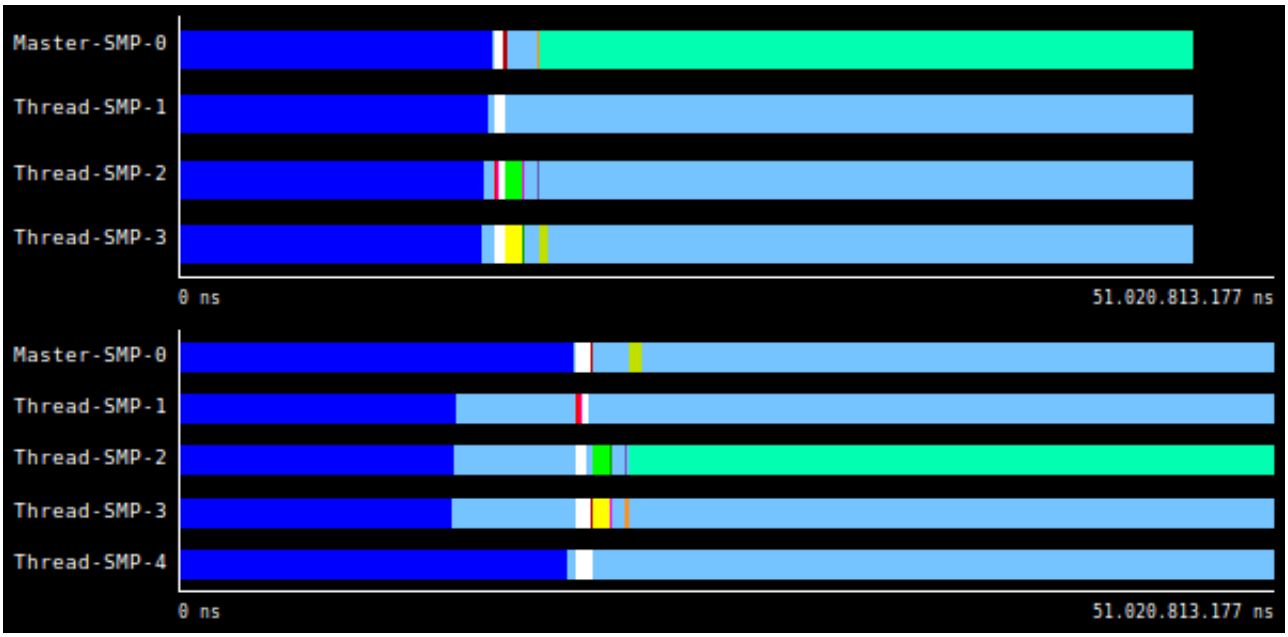

Figura 4.8: *La imagen muestra dos ejecuciones de la aplicación con OmpSs con la traza bt.C.64, la superior se le asignó 4 threads y a la inferior 5. Los trozos de color azul oscuro, son los tasks del blocking de "trace_filter". Con 4 threads tiene una carga mejor balanceada que con 5.*

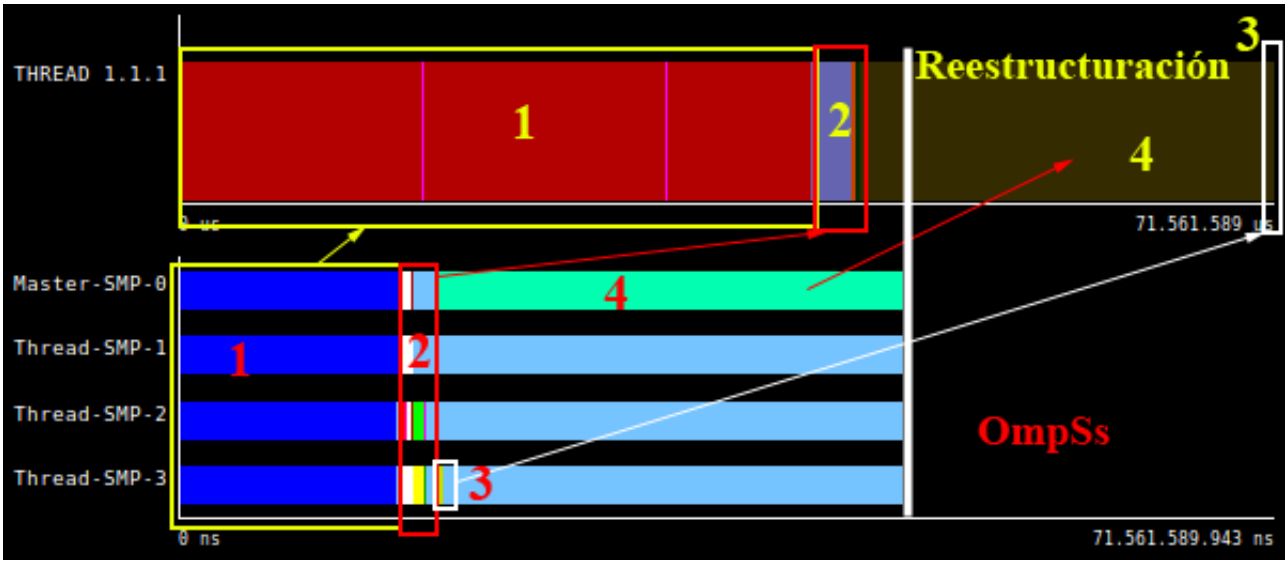

Figura 4.9: *Las dos trazas, son de la aplicación con la ejecución de la traza bt.C.64. La superior es de la versión reestructuración y la inferior de OmpSs con 4 threads. Los números simbolizan zonas de ejecución que se concuerdan de una versión a la otra. La zona 1 remarcada, es la función FilterRunning. La zona 2 son las ejecuciones de las funciones que generan las señales semánticas y el análisis de las zonas de corte. La zona 3 es la función Cutter_signal_OutFile. La última zona, la número 4, concuerda con la función Cutter2.*

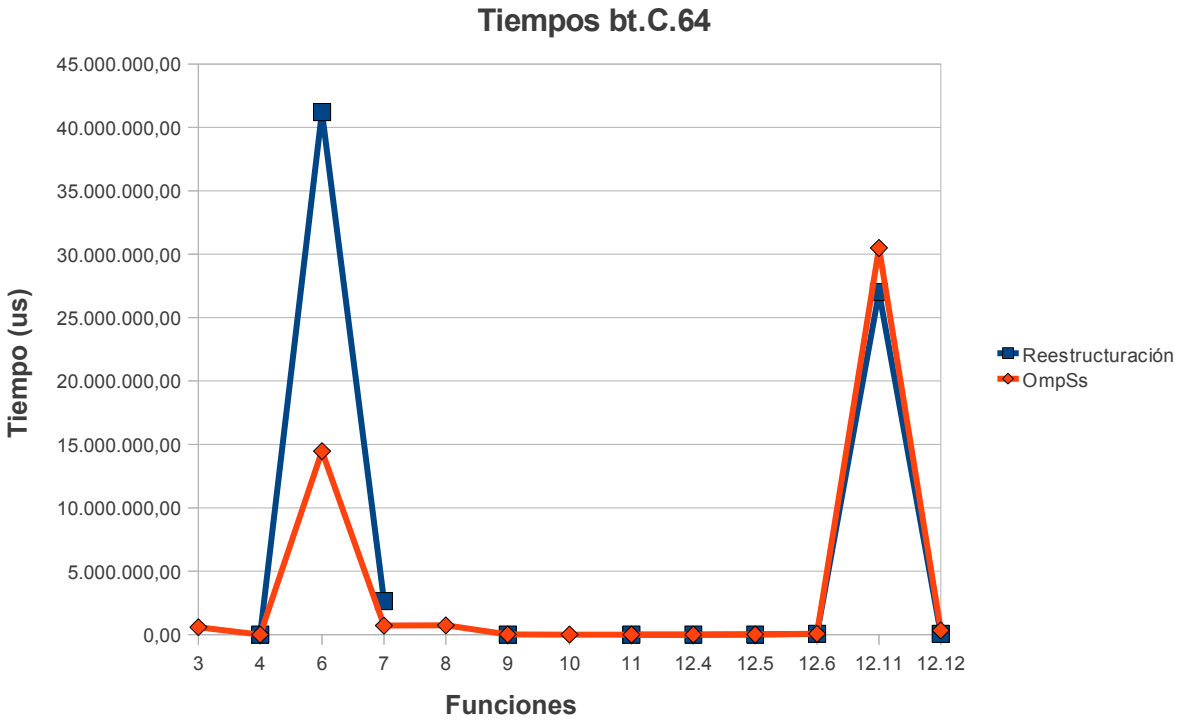

Figura 4.10: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en las versiones reestructuración y OmpSs. La leyenda de las funciones, es el mismo que en el de la tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

5. Tiempos de ejecución de las diferentes versiones

5.1. Tiempos de las funciones ejecutadas

Para interpretar cada traza su respectiva tabla, es necesario mirar en la siguiente, el significado de la columna "Secuencia". La tabla, es una leyenda para entender las otras.

	Eventos (funciones o trozos de código)		
Secuencia	**Función principal**	**SubFunción**	**SubFunción2**
1	Totaltime		
2	Nthreads		
3	GenerateEventSignal		
4	GetBoundary		
4.1		SenyalD	
4.2		SenyalE	
4.3		getValues	
5	LoopGetBoundary		
6	FilterRunning		
7	SignalRunning		
8	SignalDurRunning		
9	Sampler_wavelet		
10	SignalDurRunning (signal3)		
11	Wavelet		
12	Analysis		
12.1		Cutter_signal (signal)	
12.2		GetPeriod	
12.2.1			Correction
12.2.2			Sampler
12.2.3			Fftprev
12.2.4			Fft
12.2.5			GetTime
12.3		GetPeriodResults	
12.4		Generatesinus	
12.5		Sampler	
12.6		Crosscorrelation	
12.7		Cutter_signal (signal3)	
12.8		Maximum	
12.9		Cutter3	
12.10		Cutter3	
12.11		Cutter2	
12.12		Cutter_signal (signal)	

La "Secuencia", es la posición en que una función es ejecutada respecto al tiempo, es decir, la número 1 es la primera y la 2 la segunda. Si dentro de una función se ejecuta otra función, esta tiene su nombre en la columna de subfunción o en subfunción2. Como excepción, en la columna de las tablas de las trazas de la versión de "reestructuración" y en las de SMPSs, estas no se ejecutarán por este orden. Pero, para simplificar la lectura de los resultados, las tablas, más que representar en que orden se ejecutaron, estas representarán el tiempo de su tardanza.

Las filas que se han unido, representan que entre dos funciones estas se han fusionado. La explicación de dicha fusión, está explicada detalladamente en el apartado de la versión que pertenece.

Las filas que tengan un guión, esa función no se ha ejecutado para esa versión o es que se ha fusionado con otra y su tiempo está en otra fila.

Todas las versiones, se han recogido sus valores gracias a los eventos de Extrae, excepto en la versión de "SMPSs (smpss)" y "OmpSs", que generaron su traza automáticamente, y, sólo se recogieron los datos de las funciones paralelizadas. Con este motivo, en su columna las demás funciones que no son paralelas, no tienen ningún valor.

Hay que tener en cuenta, que todos los tiempos, han sido recogidos al ejecutar la aplicación en un mismo ordenador. Para tener en mente los tiempos que han surgido, hay que tener en cuenta, las características del ordenador, estas se encuentran en el Anexo A.

Tiempos ALYA.64

	Tiempos ALYA.64 (us)								
Secuencia	**Binarios**	**Sin Binarios**	**Sin ficheros**	**Sin repeticiones**	**Reestructuración**	**SMPSs secuencial**	**SMPSs (extrae)**	**SMPSs (smpss)**	**OmpSs**
1	49.525,43	86,45	43,93	1,55	1,54	1,92	1,92		
2	111,03	59,59	20,19	9,96	8,68	10,94	14,74		
3	104.602.643,85	102.641.393,07	104.913.594,38	101.833.798,34	-	-	1.658.846,33	1.504.221,10	600.093,07
4	403.773,92	3.834,10	1.895,42	1.935,90	1.538,85	1.818,46	657.000,60	3.381,99	1.763,46
4.1	386.467,83	1.319,07	778,27	807,79	444,06	530,22	640.240,04		
4.2	14.858,61	1.516,30	591,97	595,13	581,41	690,08	705,65		
4.3	904,41	308,79	14,94	17,50	14,71	14,87	17,06		
5	89,32	28,85	0,65	2,77	3,59	5,03	0,66		
6	146.677.908,76	141.875.627,11	141.150.954,99	139.258.283,92	139.793.522,00	166.525.249,83	173.421.372,66		133.671.847,91
7	2.728.095,96	2.611.440,77	2.049.426,40	2.347.552,37	2.282.229,15	2.591.113,68	1.724,19	541.548,79	558.815,05
8	4.224.260,10	4.193.536,58	1.640.620,78				173,97	580.360,78	569.404,30
9	1.164.669,02	1.176.479,62	6.777,07	6.839,48	6.905,70	9.114,39	11.224,77	10.095,18	14.073,36
10	4.219.831,99	4.184.605,10	1.648.964,51	0,17	-	-	1.502,11	4.451,22	5.587,43
11	258.098,80	3.317,47	239,34	211,47	206,21	235,36	253,78	255,19	5.957,00
12	155.995.731,60	140.939.780,78	122.303.162,67	117.251.838,99	80.561.330,85	75.465.921,19	85.919.841,36		
12.1	2.784.760,55	2.759.481,26	18.930,90	17.217,76	18.115,98	21.774,42	22.804,57		
12.2	8.729.649,00	8.538.380,00	3.306.078,00	454.685,00	435.108,41	487.073,78	492.935,62		
12.2.1	2.839.455,23	2.749.720,58	0,17	0,17	0,92	1,17	1,54		
12.2.2	2.288.019,03	2.317.113,88	6.384,47	4.468,33	18.116,44	20.348,87	20.347,74		
12.2.3	1.046.621,19	1.034.109,40	1.030.893,39	185.958,57	191.272,85	246.780,21	209.917,20		
12.2.4	1.768.169,40	1.671.662,51	1.649.398,12	182.919,33	183.590,86	215.348,35	215.067,21		
12.2.5	786.629,47	765.259,35	246.742,35	969,39	971,96	1.120,11	1.127,47		
12.3	41,15	38,21	40,40	23,31	59,17	61,80	61,57		
12.4	21.642,66	50.587,75	867,69	879,88	868,60	1.019,78	7.259,67	1.019,14	1.020,50
12.5	1.951.766,60	2.016.862,62	4.265,22	4.189,70	4.184,27	5.865,71	725,03	6.592,35	5.484,52
12.6	151.884,80	151.184,20	39.082,15	52.402,96	38.637,34	45.155,43	69.440,08	59.554,91	50.961,22
12.7	394.947,73	408.968,44	1.693,75	1.174,67	1.183,11	1.639,37	1.686,01		
12.8	80.161,58	87.961,54	192,35						
12.9	25.995.915,70	18.852.711,79	20.873.384,02	20.655.616,79	0,42	0,37	0,33		
12.10	34.978.900,70	32.599.026,04	32.400.043,14	32.392.225,43					
12.11	70.819.477,28	65.599.105,43	64.097.633,40	63.383.118,69	79.888.556,95	74.734.089,58	85.320.061,95	88.805.962,96	74.725.062,98
12.12	608.245,58	582.870,31	143.306,69	238.645,98	169.982,54	164.122,86	204.855,61	159.910,82	317.893,62

Tiempos bt.C.64

	Tiempos bt.C.64 (us)								
Secuencia	**Binarios**	**Sin Binarios**	**Sin ficheros**	**Sin repeticiones**	**Reestructuración**	**SMPSs secuencial**	**SMPSs (extrae)**	**SMPSs (smpss)**	**OmpSs**
1	55.431,74	86.689,96	93,78	4,54	1,75	1,80	2,04		
2	97,97	48,73	56,52	29,60	8,29	11,04	11,42		
3	32.138.330,93	33.931.383,71	30.174.444,90	32.659.147,41	-	-	1.749.867,49	1.771.337,28	581.051,11
4	27.700,65	2.788,30	1.337,36	1.364,31	1.061,81	1.273,66	710.711,25	1.337,69	886,24
4.1	12.958,37	882,03	479,18	487,00	222,87	265,28	275,25		
4.2	12.777,52	990,02	328,75	340,27	327,69	400,90	397,81		
4.3	601,24	202,17	13,98	14,44	13,92	14,45	14,13		
5	71,17	23,03	0,53	3,09	0,49	4,69	0,43		
6	41.706.393,62	41.895.354,61	40.828.616,02	40.976.942,70	41.210.198,60	49.382.676,22	50.325.150,48		14.467.036,74
7	5.760.090,72	3.265.943,12	2.028.877,71	2.644.291,38	2.641.986,42	3.134.682,23	733.890,15	724.437,20	714.186,80
8	3.295.613,90	5.187.922,57	2.001.467,72				909,59	741.288,47	728.807,03
9	1.475.145,99	1.433.006,17	8.095,18	8.058,44	7.963,79	11.100,66	12.944,42	12.849,20	12.425,28
10	5.137.420,74	5.185.543,31	2.016.079,63	0,21	-	-	5.677,98	6.301,54	4.948,54
11	11.382,60	3.727,95	213,81	210,90	197,78	233,28	255,26	10.687,11	5.356,15
12	78.623.675,03	77.901.610,13	54.815.861,91	51.782.766,17	27.683.784,48	30.808.990,23	30.585.968,32		
12.1	5.483.719,68	5.522.786,99	35.528,75	34.861,66	34.657,20	41.827,40	43.293,24		
12.2	13.842.656,66	13.953.402,71	4.002.303,91	557.290,02	495.554,16	585.758,92	595.919,28		
12.2.1	5.485.465,25	5.544.831,97	0,15	0,11	0,15	0,28	0,29		
12.2.2	4.269.985,76	4.335.480,84	14.103,45	17.490,52	14.520,92	17.835,87	17.861,94		
12.2.3	1.336.782,04	1.337.809,17	1.232.008,28	234.136,08	224.608,27	264.559,58	266.385,38		
12.2.4	2.070.672,40	2.083.219,70	2.015.819,49	253.697,31	252.303,11	298.294,22	298.571,61		
12.2.5	678.862,31	651.537,56	220.043,50	1.097,93	1.091,05	1.241,93	1.249,49		
12.3	41,49	42,84	23,46	25,48	22,41	24,66	24,25		
12.4	4.020,05	4.016,84	218,61	219,04	230,45	249,37	246,94	277,56	287,36
12.5	3.936.829,67	3.847.049,73	8.267,75	8.506,01	8.551,39	11.910,33	11.895,05	11.869,50	11.143,10
12.6	195.547,41	186.432,49	55.467,95	55.425,46	55.549,98	65.746,42	65.767,70	65.532,63	68.244,25
12.7	1.877.377,67	1.886.393,33	7.520,27	7.438,55	7.427,90	10.525,24	10.650,61		
12.8	2.212,40	2.214,59	20,05						
12.9	13.856.129,99	13.853.497,00	13.615.505,21	13.571.369,30	0,25	0,29	0,32		
12.10	14.240.618,93	14.178.232,94	14.145.677,89	14.138.642,12					
12.11	23.730.468,06	23.050.005,16	22.635.016,31	23.132.147,51	27.013.829,79	30.046.642,71	29.850.683,40	30.374.490,02	30.490.977,05
12.12	1.451.760,24	1.416.025,59	303.407,29	270.239,60	61.181,07	39.411,82	36.497,90	36.996,87	334.384,48

Tiempos VAC.128

	Tiempos VAC.128 (us)								
Secuencia	**Binarios**	**Sin Binarios**	**Sin ficheros**	**Sin repeticiones**	**Reestructuración**	**SMPSs secuencial**	**SMPSs (extrae)**	**SMPSs (smpss)**	**OmpSs**
1	52.264,62	58,12	43,81	4,81	1,53	2,11	4,37		
2	111,55	37,95	19,97	28,42	8,87	12,89	25,76		
3	5.758.352,11	5.530.336,40	5.483.954,93	5.921.321,20	-	-	363.626,54	363.811,24	126.343,94
4	22.903,80	5.150,89	2.909,91	8.240,47	2.491,95	2.935,22	137.448,65	3.323,76	2.496,34
4.1	9.039,63	2.087,07	1.396,71	3.919,81	997,71	1.180,88	2.831,20		
4.2	8.526,79	2.057,88	987,96	2.845,02	979,76	1.154,21	1.172,22		
4.3	1.128,65	322,27	14,52	38,77	14,29	14,90	15,20		
5	80,58	23,62	0,53	8,18	3,28	0,85	0,41		
6	7.753.210,68	7.707.713,05	7.713.681,36	7.643.807,42	7.877.818,68	9.117.886,25	9.627.923,18		9.171.256,84
7	675.410,24	670.659,81	413.210,76	526.833,07	534.177,48	632.405,64	143.679,57	138.285,05	135.801,13
8	1.105.346,62	1.134.257,72	406.771,49				887,23	141.791,59	139.225,91
9	293.534,58	291.384,28	2.065,45	2.409,99	2.351,02	2.996,20	3.337,95	4.092,71	3.924,65
10	1.112.214,92	1.086.967,69	410.192,17	0,14	-	-	1.123,81	1.194,68	1.023,80
11	20.441,37	3.457,43	234,66	207,28	193,93	230,06	259,21	958,51	1.391,01
12	34.488.334,79	34.426.323,53	29.872.026,64	18.760.651,54	17.055.835,03	21.222.075,05	20.799.776,57		
12.1	1.170.295,55	1.174.289,87	6.948,48	6.411,27	6.763,18	8.496,26	8.831,39		
12.2	14.845.471,00	14.814.005,88	12.613.263,13	2.353.736,71	2.453.636,23	2.818.887,76	2.863.052,69		
12.2.1	1.150.954,10	1.160.002,82	0,15	0,13	0,23	0,27	0,26		
12.2.2	2.238.615,57	2.269.181,95	7.579,01	8.241,82	8.333,59	9.607,43	9.667,45		
12.2.3	3.536.901,33	3.512.278,34	4.091.701,76	1.102.088,68	1.149.448,51	1.317.175,08	1.325.576,29		
12.2.4	5.998.763,88	5.955.595,22	6.401.548,72	1.213.074,40	1.263.983,07	1.453.540,26	1.451.973,96		
12.2.5	1.919.498,09	1.916.432,23	654.051,42	2.879,24	2.884,13	3.348,34	3.412,49		
12.3	41,76	38,66	16,21	23,95	31,70	36,47	25,09		
12.4	10.716,30	10.662,23	707,34	531,42	523,22	611,00	351,45	626,44	664,41
12.5	930.217,19	925.990,83	3.041,10	2.411,03	2.443,18	3.119,53	3.446,62	3.039,74	3.734,85
12.6	10.716,30	512.306,85	120.094,74	119.606,18	119.559,52	140.601,59	145.471,65	203.664,50	148.004,91
12.7	284.422,81	282.908,79	1.894,37	1.584,34	1.619,64	2.117,24	1.942,66		
12.8	3.666,94	3.724,08	23,05						
12.9	1.986.959,38	1.976.899,72	1.960.745,33	2.007.199,05	0,35	0,38	0,49		
12.10	2.041.242,48	2.060.573,56	2.166.490,48	2.048.967,78					
12.11	12.022.681,74	12.020.565,62	12.291.255,54	11.916.468,85	14.459.249,40	18.235.524,41	17.773.308,78	17.230.807,47	18.086.772,83
12.12	676.692,47	642.881,60	704.221,95	300.669,78	9.130,55	9.193,30	40.014,84	11.059,16	408.486,13

Tiempos WRF.64

	Tiempos WRF.64 (us)								
Secuencia	**Binarios**	**Sin Binarios**	**Sin ficheros**	**Sin repeticiones**	**Reestructuración**	**SMPSs secuencial**	**SMPSs (extrae)**	**SMPSs (smpss)**	**OmpSs**
1	34.927,05	72.906,27	36.154,16	5,00	1,73	1,88	2,02		
2	99,34	110,58	75,72	34,84	8,36	10,70	10,88		
3	143.831.510,71	149.464.512,93	143.644.338,62	142.809.536,51	-	-	1.072.564,23	1.077.617,32	408.862,52
4	267.166,68	13.267,83	8.243,06	8.308,92	7.428,29	8.530,74	369.862,82	8.332,87	8.166,87
4.1	249.137,68	5.386,12	3.947,68	3.941,27	3.016,74	3.487,62	8.481,09		
4.2	14.147,28	6.345,51	3.755,97	3.820,08	3.878,25	4.418,42	4.476,01		
4.3	2.437,42	836,91	16,75	16,99	17,38	18,11	19,64		
5	93,11	34,45	0,60	3,39	0,56	4,81	0,88		
6	199.981.668,41	205.160.656,97	203.819.721,85	203.982.983,12	202.935.846,90	239.895.022,79	246.337.132,34		293.563.588,96
7	1.810.184,28	1.878.315,70	1.128.475,33	1.653.601,63	1.518.582,96	1.799.671,74	380.060,09	375.040,97	374.412,26
8	2.885.208,33	2.910.621,14	1.082.106,18				3.167,27	387.521,75	380.629,52
9	790.305,50	788.792,62	4.967,55	4.794,28	4.911,97	6.525,93	7.540,29	7.550,66	8.801,36
10	2.917.799,79	2.920.068,05	1.087.281,40	0,16	-	-	2.643,02	3.521,74	3.664,79
11	275.110,02	3.364,45	248,60	210,06	200,89	233,17	244,02	263,14	4.036,55
12	344.660.114,17	348.869.325,76	338.171.493,18	336.179.006,48	121.941.776,97	138.257.475,23	135.604.712,54		
12.1	1.406.208,55	1.385.517,07	7.469,75	6.376,94	6.487,96	8.424,14	8.494,67		
12.2	8.369.643,45	7.968.051,40	3.627.765,42	417.748,82	422.307,49	491.750,79	549.529,25		
12.2.1	2.368.537,25	2.316.107,58	1,20	1,10	1,19	1,15	1,28		
12.2.2	2.078.075,89	2.053.305,57	6.287,71	6.620,35	6.680,58	7.890,40	7.939,14		
12.2.3	1.281.910,25	1.148.716,98	1.084.892,65	187.966,24	189.551,59	220.877,92	236.386,84		
12.2.4	2.040.145,75	1.844.132,74	1.866.067,69	217.857,83	220.291,15	256.072,40	257.773,74		
12.2.5	599.735,68	604.984,13	196.166,58	945,50	946,54	1.055,24	1.080,87		
12.3	383,43	169,12	51,62	51,00	49,72	57,58	58,64		
12.4	10.135,53	3.493,04	202,52	223,96	204,83	235,73	238,71	234,04	227,92
12.5	455.366,33	457.276,35	1.143,02	1.127,94	1.132,33	1.329,79	1.462,84	1.313,46	1.405,59
12.6	169.061,20	166.337,36	47.723,35	47.423,67	47.720,88	56.186,21	56.379,38	58.334,41	58.096,27
12.7	900.499,16	873.555,57	3.825,03	3.656,16	3.682,95	5.162,09	5.223,79		
12.8	1.822,47	1.815,67	20,59						
12.9	107.203.115,95	108.465.313,05	107.755.354,65	107.528.424,07	0,29	0,31			
12.10	106.604.602,17	108.247.782,84	107.070.398,25	107.934.580,89					
12.11	118.635.921,52	120.211.545,21	119.282.360,48	120.016.324,06	121.447.180,31	137.682.148,92	134.981.087,31	135.624.402,27	134.600.251,35
12.12	900.904,19	1.086.986,08	372.380,49	219.818,46	10.159,75	9.187,46	8.102,23	8.339,17	68.644,82

5.2. Gráficos de los resultados de los tiempos

A continuación, se muestran las gráficas de los resultados de las tablas anteriores.

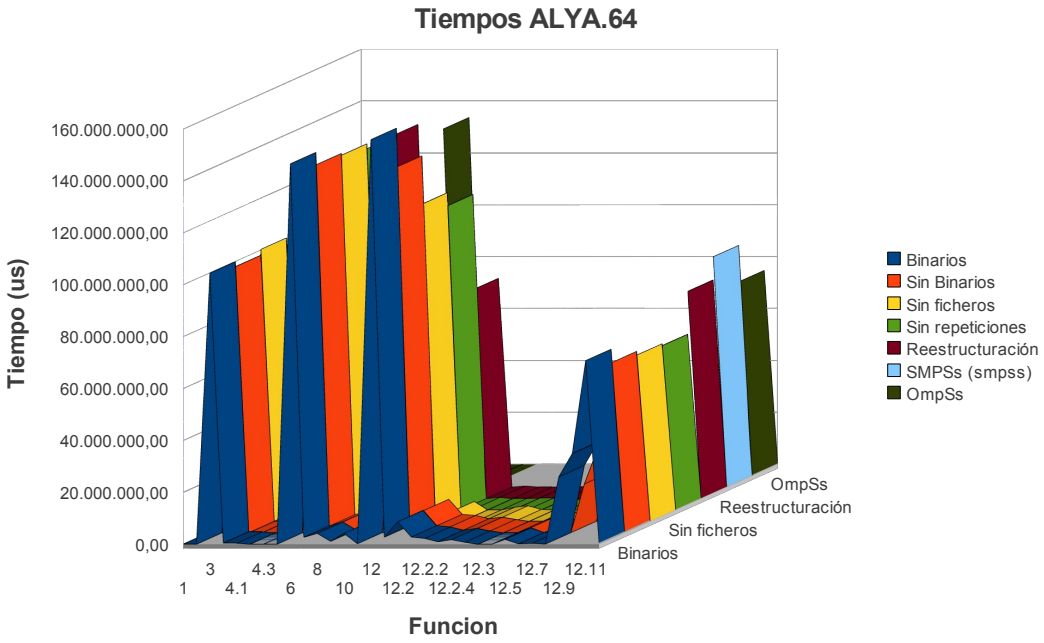

Figura 5.1: *Gráfico de los tiempos de las funciones de la traza ALYA.64, en todas las versiones importantes realizadas del código. La leyenda de las funciones, en la misma tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

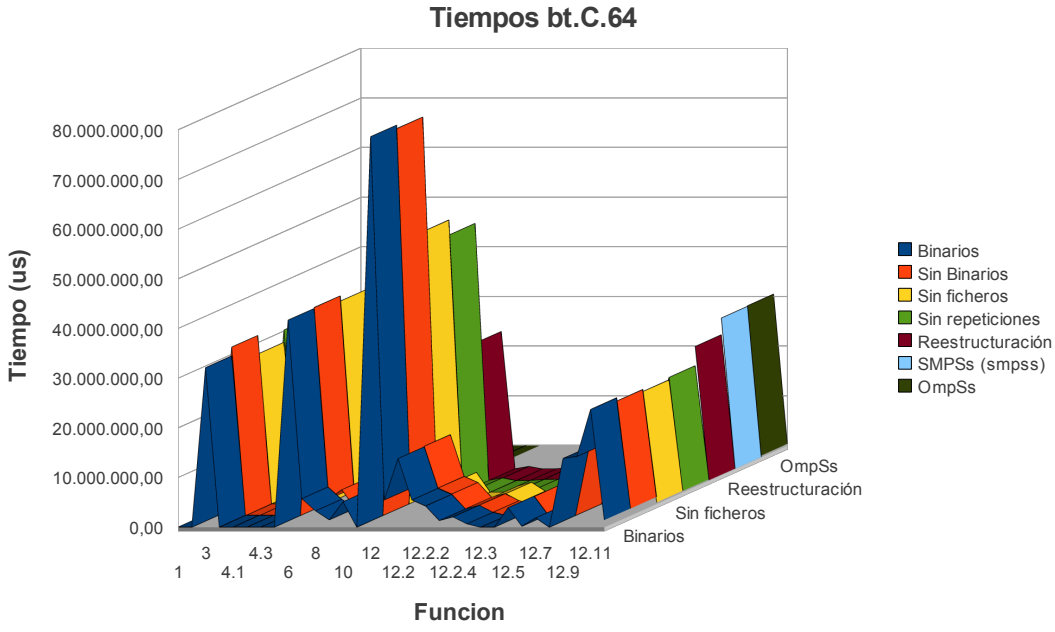

Figura 5.2: *Gráfico de los tiempos de las funciones de la traza bt.C.64, en todas las versiones importantes realizadas del código. La leyenda de las funciones, en la misma tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

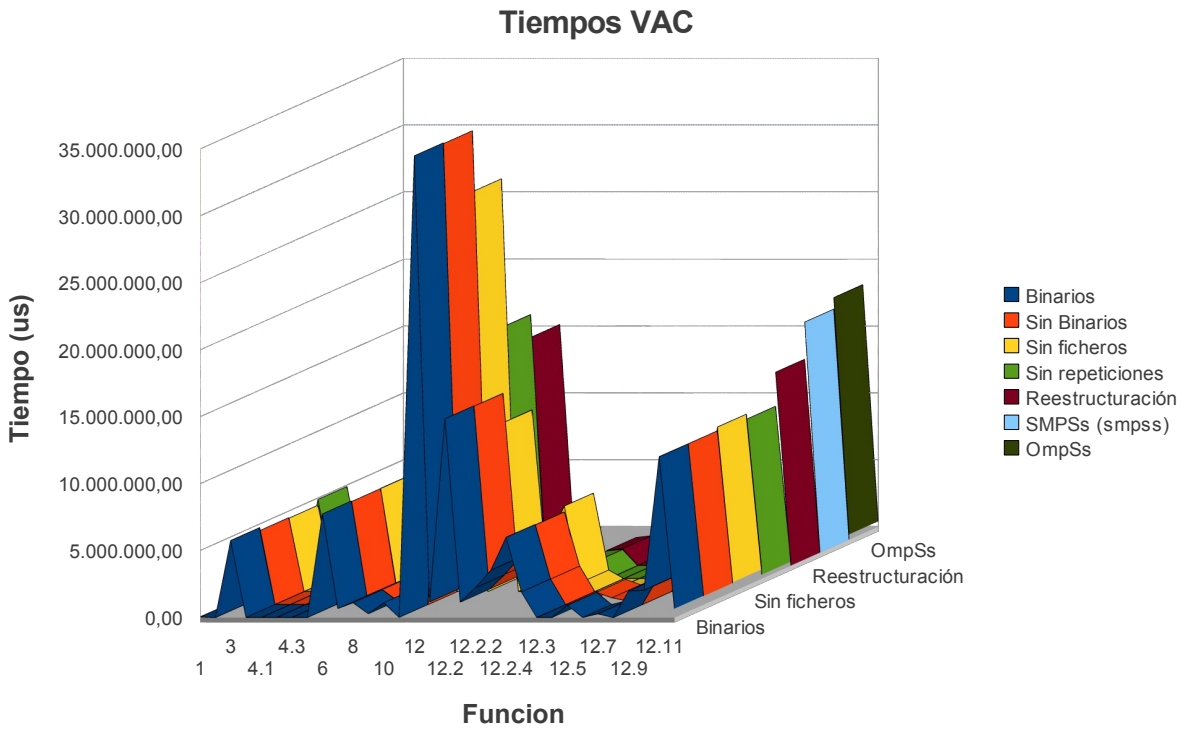

Figura 5.3: *Gráfico de los tiempos de las funciones de la traza VAC.128, en todas las versiones importantes realizadas del código. La leyenda de las funciones, en la misma tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

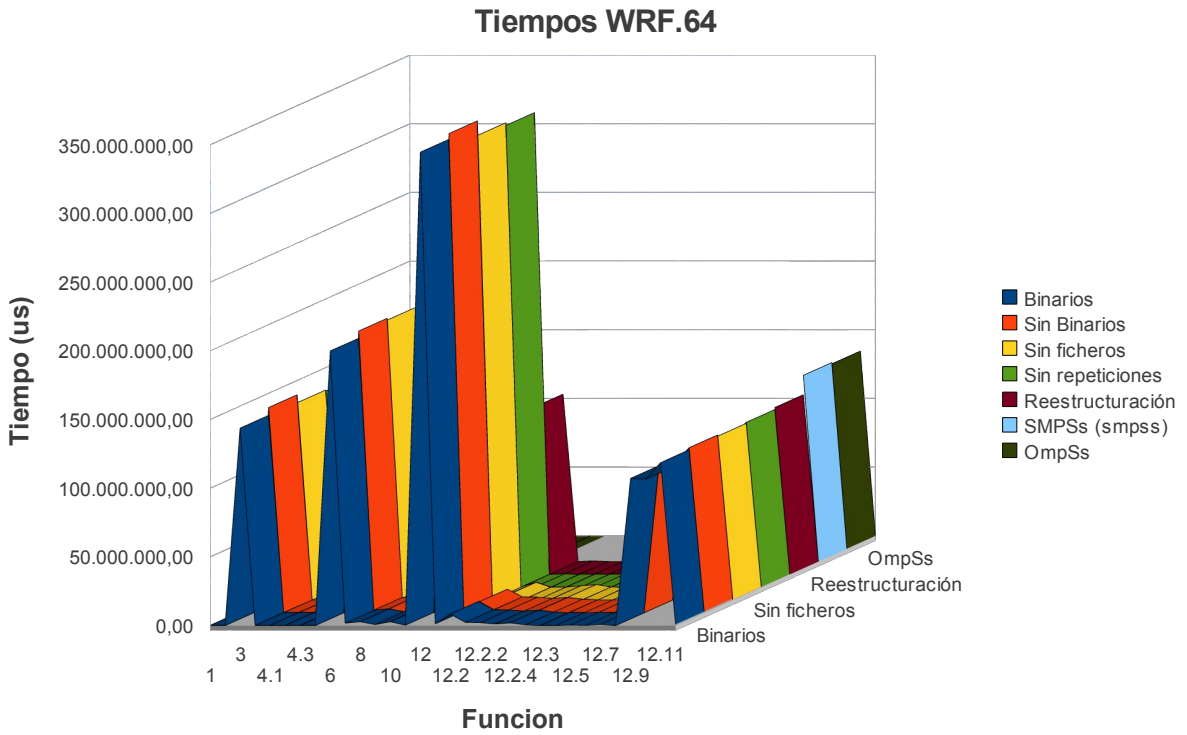

Figura 5.4: *Gráfico de los tiempos de las funciones de la traza WRF.64, en todas las versiones importantes realizadas del código. La leyenda de las funciones, en la misma tabla del apartado "5.1. Tiempos de las funciones ejecutadas".*

5.3. Tiempos de ejecución de la aplicación

La siguiente, tabla recoge los resultados de las ejecuciones normales de la aplicación, para cada versión que se ha ido explicando. En la Figura 5.5, muestra el gráfico de los resultados de la tabla para las versiones que se han optimizado. Por eso la versión **SMPSs (extrae)** no sale, porque teóricamente es la misma que **SMPSs (smpss)**, y, **SMPSs secuencial**, tampoco, porque esta es solo de prueba, de la versión **reestructuración** compilada con SMPSs.

	Tiempos(s)			
Código	**ALYA.64**	**bt.C.64**	**VAC.128**	**WRF.64**
Binarios	175,40	71,82	22,32	302,72
Sin Binarios	169,12	70,32	21,85	299,98
Sin ficheros	160,34	56,67	19,23	296,18
Sin lecturas repetitivas	154,96	54,84	13,89	291,76
Reestructuración	103,09	31,36	11,06	139,58
SMPSs (smpss)	103,98	34,14	12,22	159,94
OmpSs	83,86	20,08	9,63	135,45

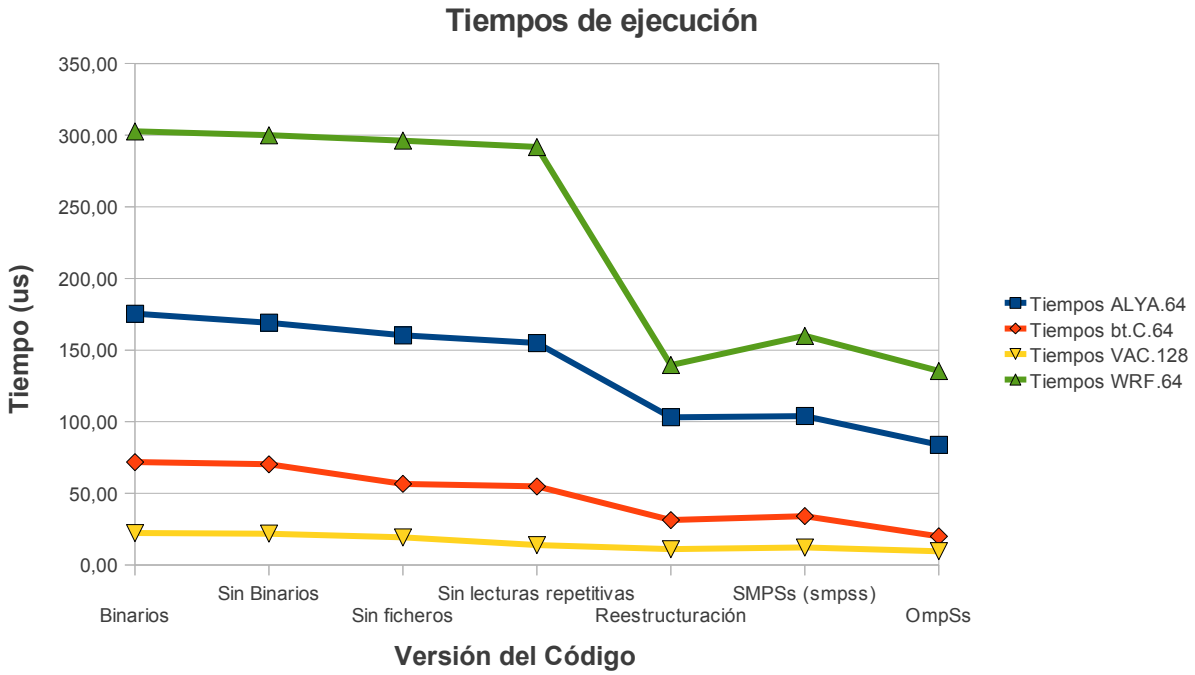

Figura 5.5: *Gráfico de los tiempos de las diferentes ejecuciones de la aplicación, de las diferentes versiones, que ha sufrido el código.*

Los porcentajes de las mejoras, que se han ido dando a lo largo de cada versión, estos están calculados a partir de una media de entre las cuatro diferentes trazas. Se ha hecho una media entre las cuatro, porque dependiendo de que traza, una optimización concreta puede ser mejor o no, pero se ha de mirar en conjunto y no en una sola traza. En consecuencia, se puede saber que para otras trazas, esas optimizaciones realmente han mejorado la aplicación en su tiempo de respuesta.

6. Versión final

La versión final conseguida, se ha desarrollado en StarSs con la versión OmpSs. Tal como se ha ido explicando en los apartados anteriores, la aplicación, ha ido evolucionando a través de una serie de etapas. Ha pasado de ser una aplicación con llamadas a binarios, a ser una aplicación secuencial optimizada, con todos los cálculos en memoria. Como último paso, se paralelizó a OmpSs.

En este apartado, se explicará la instalación de la aplicación para la versión final. Como ayuda, se generó un script como comodidad para ejecutar la aplicación, y, poder optimizar de una forma correcta. La forma correcta, es para cada optimización, recoger los tiempos de respuesta de unas ciertas ejecuciones y mirar que los resultados son correctos. Hacer este proceso a mano es engorroso, por eso, se generó este script para poder realizar toda esta labor de forma automática.

6.1. Instalación

La aplicación, hace uso de unas herramientas externas desarrolladas por el BSC. En una primera instancia, las usaba a través de una llamada para ejecutar sus binarios. Como era una forma poco eficiente, se cogió el código fuente, y, se compiló al estilo de una librería, a la cual, se le llamaba a través de una función normal. En la última etapa de la optimización, se adaptó estas al lenguaje de OmpSs.

La librería, se encuentra en una carpeta con el nombre "TRACE2TRACE", de los archivos proporcionados del proyecto. Dentro de su carpeta, incluye un `Makefile` para poder compilarlo. Como la última versión es necesario compilarlo con OmpSs, se tendrá que especificar las siguientes variables globales, como se indican en el apartado de instalación del Anexo D.2.

- `PATH`: la ubicación de la instalación del compilador OmpSs.
- `NX_INSTRUMENTATION`: especificar la instrumentación con la librería de Extraer.

Los siguientes comandos, son los necesarios para poder realizar la compilación deseada.

```
:~/TARCE2TRACE$ export PATH=$PATH:/opt/OmpSs/bin
:~/TARCE2TRACE$ export NX_INSTRUMENTATION=extrae
:~/TARCE2TRACE$ make all-OmpSs
```

En la carpeta "Aplicación", se encuentra el programa. En el interior de esta, se encuentra su `Makefile`, que se tendrá que modificar los paths para su correcta instalación. Se modificará la variable `TRACE2TRACELIBPATH` con el path de la librería TRACE2TRACE, que acabamos de compilar. Después de compilar la librería, los archivos necesarios, incluyendo los headers (archivos .h), se encuentran en la subcarpeta con el nombre `lib`. Como ejemplo, podemos tenerlo de la siguiente forma:

Código 6.1: Makefile de la aplicación, path a TRACE2TRACE

```
#Path lib trace2trace
TRACE2TRACELIBPATH=/home/usuario/TRACE2TRACE/lib
```

En la variable ARCHITECTURE se indicará, si se va a compilar en un ordenador personal (laptop) o en el MareNostrum (MN). Esto es, para poder encontrar las librerías necesarias según la arquitectura. En este caso, se pondrá:

Código 6.2: Makefile de la aplicación, Arquitectura

```
# ARCHITECTURE => MN , laptop
ARCHITECTURE = laptop
```

Como se dispone de un script para ejecutar la aplicación más cómodamente, cambiaremos los paths del script, tal como se indica en el siguiente apartado. Para poder hacer una ejecución de prueba utilizaremos el siguiente comando, donde /path/trace.prv, es el path a la traza a analizar.

```
:~/Aplicacion$ ./test.sh -exec -ompss -traceName /path/trace.prv
```

6.2. script para automatizar el trabajo

El script test.sh, se encuentra en la carpeta "Aplicación", es aquel que se ha ido utilizando para automatizar el trabajo de optimización y para ejecutar cómodamente la aplicación. Se repasará las variables globales, que se deberá modificar para la versión OmpSs.

De los argumentos predefinidos, la que se podrá cambiar es la variable N. Esta indica el número de ejecuciones, que se hará de la aplicación, para saber el tiempo medio que tarda en ejecutar.

La parte de los paths, se cambiará TRACE_ORIGINAL, donde se encuentra todos los archivos de las ejecuciones realizadas correctamente. En el ejemplo, se especifica en /home/usuario/Original/. Dentro de la carpeta Original, se encontrará los ficheros de las ejecuciones correctas en subcarpetas, con el nombre de las trazas que se han pasado como parámetro (ejemplo bt.C.64). Esta variable, sólo se cambiará, si queremos utilizar el script, para hacer las comprobaciones de si una ejecución ha sido correcta, para sólo ejecutar la aplicación no hará falta. El resto, son paths de las ubicaciones de trazas, que no hace faltar cambiarlas, ya que se dispone de un parámetro a la hora de ejecutar el script, para indicar que traza se quiere analizar.

De las variables de Extrae y OmpSs, sólo cambiaremos las de OmpSs, ya que este lleva una API integrada para Extrae, y, no nos hace falta la instalación de la librería como para la versión secuencial. Las que debemos cambiar son:

- LD_LIBRARY_PATH: ubicación de la librería del compilador de OmpSs.
- PATH: dónde se encuentra el binario del compilador.
- OMP_NUM_THREADS: el número de threads, que se le asignará a la aplicación como default, a través de un parámetro a la hora de ejecutar el script, se le puede definir la cantidad que nosotros deseamos.
- NX_INSTRUMENTATION: se definirá Extrae, como la librería de instrumentalización.

Las siguientes, son las que se han definido como ejemplo.

Código 6.3: test.sh, paths y variables globales

```
1 LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/OmpSs/lib
2 PATH=$PATH:/opt/OmpSs/bin
3 OMP_NUM_THREADS=8
4 NX_INSTRUMENTATION=extrae
```

La siguiente tabla, lista los argumentos que se puede pasar al script `test.sh`.

```
Usage: ./test.sh OPTIONS
      OPTIONS: las siguientes opciones se pueden ejecutar de manera desordenada.
               NOTA: si no se especifica -option, -traceName y -time se ejecutará el programa como
                     default la opción CPUDurBurst, la traza b.C.64 y el tiempo 71.82 de b.C.64.
                     Y del resto de opciones como default se ejecutará -test.

             -test: ejecución del programa y comparación si esta es correcta. Se calcula el tiempo
                    de mejora.
             -gprof [static/nostatic/2static/]: gprof de optim.
                    También nos genera el grafo de la ejecución.
                    Se compila el código en estático (static), no estático (nostatic), y,
                    tanto estático y no estático(2static). Como default 2static(./test.sh -gprof).
             -exec: ejecución normal del programa.
             -trace [extrae/ompss/]: ejecución normal del programa generando una traza (optim.prv)
                    para Paraver.
             -debug: ejecución del programa y comparación si esta es correcta con la compilación
                    en modo debug, genera todos los archivos intermedios y comprueba que se ha
                    ejecutado correctamente.
             -gdb: ejecución del programa en gdb para debugar el programa.
             -traceName [path/16/64]: introducir el path de la traza (/path/ALYA/trace.prv), ha de
                    tener el nombre de la traza en su carpeta. Son posibles los valores 64 y 16
                    como alias de las trazas bt.B.16.prv y bt.C.64 respectivamente.
             -option [BW/IPC/MPIp2p/CPUBurst/CPUDurBurst]: introduce las opciones BW, IPC, MPIp2p,
                    CPUBurst o CPUDurBurstal al programa.
             -time segs: para -test especifica que ha de comprobar la mejora a partir de
                    "segs" segundos.
             -ompss [NUM cpus/]: compilar con OmpSs. Se puede especificar el número de CPUs
                    (./test.sh -ompss 8), sino se especifica (./test.sh -ompss) se utilizará
                    como default 4 CPUs.
             -help: salida de este USAGE.

       Ejemplos:
             ./test.sh -trace -traceName 16 --> genera la traza de ejecución a la traza
                    bt.B.16.prv
             ./test.sh -time 104.01 -traceName /tmp/ALYA/trace.prv -test --> hace el test del
                    programa con la traza /tmp/ALYA/trace.prv y comprueba la mejora con el
                    tiempo 104.01
```

El script tiene bastantes opciones, ya que se han ido quedando a largo del desarrollo del proyecto, pero la que nos interesa ahora para ejecutar la aplicación es la siguiente, donde `/path/tarce.prv` es el path a la traza a analizar con la aplicación.

```
:~/Aplicacion$ ./test.sh -exec -ompss -traceName /path/trace.prv
```

Del modo anterior, ejecutará la aplicación asignándole el número de threads que tiene por defecto el script en la variable `OMP_NUM_THREADS`, en este caso 8. Pero si le queremos asignar otros, la forma de hacerlo, es pasándolo a la opción de compilación "-ompss" el número de threads que deseamos, ejemplo 6.

```
:~/Aplicacion$ ./test.sh -exec -ompss 6 -traceName /path/trace.prv
```

7. Planificación del proyecto

El número de créditos, del proyecto por ser de la Ingeniería de Informática, es de 37,5. Haciendo cálculos, salen, que su duración es de un cuatrimestre, unas 15 semanas a tiempo completo (unas 40 horas por semana).

Sabiendo que dura un cuatrimestre, la fecha de inicio del proyecto es el **14 de Febrero del 2011**, y, la final el **10 de Junio del 2011**.

En la Figura 7.1, se muestra la planificación inicial en un diagrama de **Gantt**, que se pensó seguir. Tal como se muestra en el diagrama, el proyecto se dividió su planificación en 4 partes:

- Estudio
- Versión secuencial
- Versión SMPSs
- Documentación

7.1. Estudio

Es la parte inicial, se tuvo que estudiar el código de la aplicación en la que se tenía que trabajar. Todo este proceso, está bien explicado en el aprtado "2. Estudio de la aplicación original".

7.2. Versión secuencial

Una vez acabado el estudio de la aplicación, como primer objetivo, es optimizar la aplicación de forma secuencial. En el apartado "3. Optimización secuencial", se explica detalladamente todo el proceso y las diferentes versiones que se llegó a realizar.

En el diagrama, se indica las tareas de implementación del código y de tracear la aplicación. Estas dos tareas, vinieron muy juntas al optimizar una versión, se le realizaba un traceo a la ejecución de la aplicación, para poder comprobar que realmente esa optimización era correcta. En consecuencia, el tiempo de inicio de la tarea del traceo indicado en el Gantt no fue el correcto, ya que, esta tarea se realizó bastantes veces y antes.

Al encontrarse con un código nuevo, instalaciones de herramientas desconocidas (como Extrae) y realizar más optimizaciones, de las previstas en la versión secuencial, se alargó el tiempo de finalización de esta etapa unos 10 dias más de lo previsto.

7.3. Versión SMPSs

Con una buena aplicación optimizada de forma secuencial, se realizó la etapa a paralelizar la aplicación con SMPSs. En el apartado "4. Optimización paralela", se encuentra todo lo relaciónado con esta tarea.

Esta etapa también se atrasó, debido a que no se imaginó que darían tan malos resultados con la versión SMPSs, tal como se ha explicado anteriormente, la solución fue cambiarlo a OmpSs. Por esto, la siguiente etapa de la elaboración de la documentación se atrasó. Esta etapa se alargó unos 10 días más de lo previsto, pero para poder tener la documentación a tiempo, no se esperó a su finalización. Tal como se muestra en el diagrama de Gantt de la Figura 7.2, esta se hizo paralela a la implementación del código.

7.4. Documentación

Por último, tocó hacer la documentación del proyecto, la memoria, donde se ha reflejado todo el trabajo realizado y los resultados obtenidos de reducción de tiempos, en las diferentes versiones expuestas anteriormente. Al mismo tiempo, se comentó el código y se limpió para que este fuese presentable.

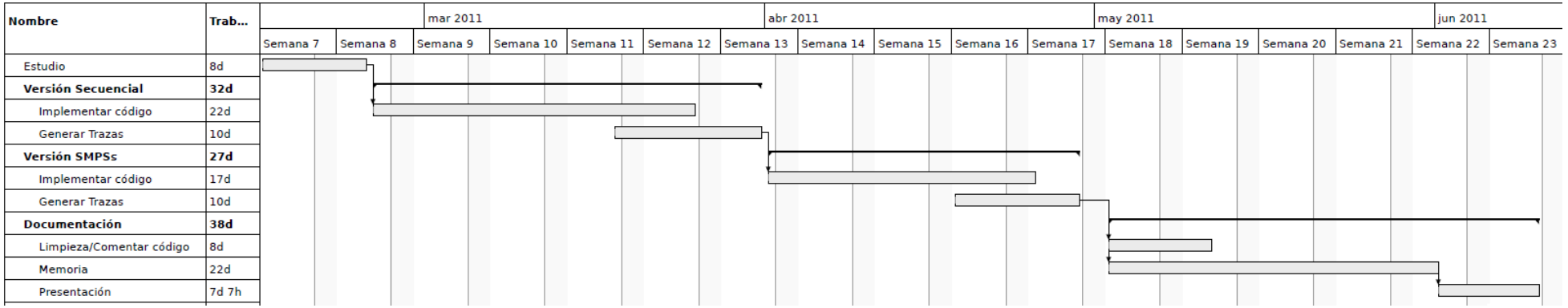

Figura 7.1: *La imagen muestra la planificación inicial del proyecto, plasmado en un diagrama de Gantt. Su fecha de inicio es el 14 de Febrero del 2011, y, como final el 10 de Junio del 2011.*

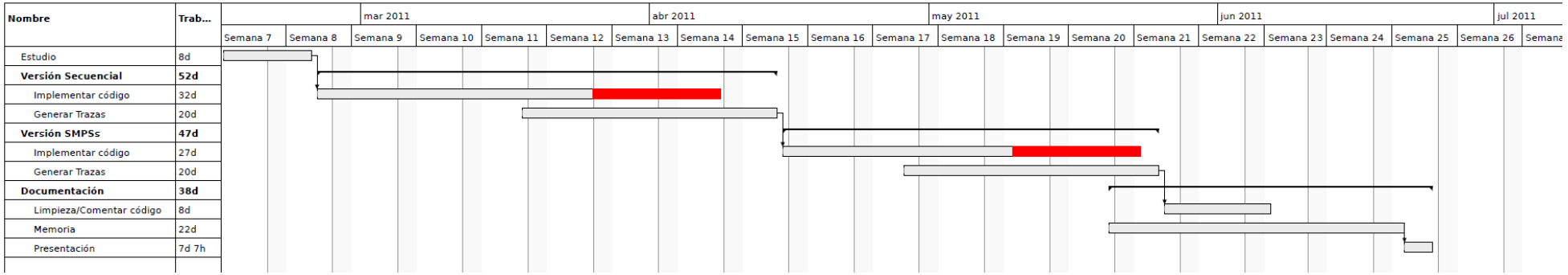

Figura 7.2: *La imagen muestra la planificación real del proyecto, plasmado en un diagrama de Gantt. Su fecha de inicio es el 14 de Febrero del 2011, y, como final el 24 de Junio del 2011.*

8. Conclusiones

El objetivo principal del proyecto, era optimizar una aplicación y luego paralelizarla con SMPSs. Este objetivo se ha cumplido satisfactoriamente, se ha obtenido **una mejora de un 78,78 %**, respecto al código original, y, se ha conseguido un **speedup de 3,41x**.

El **speedup**, es un cálculo de cuanto es de rápido un algoritmo paralelo a uno secuencial. La siguiente fórmula, representa como se calcula el speedup, donde $Tsec$ representa el tiempo de ejecución del código secuencial original, y, $Tpar$ el tiempo de ejecución del algoritmo paralelo.

$$SpeedUp = \frac{T_{sec}}{T_{par}}$$

Se ha logrado alcanzar el objetivo, gracias al seguimiento del método de optimización explicado en el apartado "1.3. Metodología a seguir para optimizar". Para cada optimización que se realizaba, se efectuaba una serie de ejecuciones al programa, en que, si los resultados de dichas ejecuciones eran válidos (una optimización puede dar a resultados erróneos) y se obtenía una ganancia (se disminuía el tiempo de respuesta de la aplicación) con la optimización, se consideraba aquella optimización válida. Como para realizar esta tarea es muy similar y se efectúa bastantes veces, lo ideal es automatizarla. Por eso, se creó un script, explicado en el apartado "6.2. script para automatizar el trabajo", que ejecutaba, de forma automática, una serie de ejecuciones del programa, comprobaba que los resultados eran correctos, y finalmente, mostraba si esas ejecuciones disminuían el tiempo de respuesta, en comparación al tiempo anterior.

La etapa de la optimización en secuencial de la aplicación, es la que ha dado más mejora. Esto se debe, a que, tal como se esperaba, el código original utilizaba llamadas a binarios, para realizar las diferentes etapas de cálculos, y, utilizaba como método de comunicación la lectura y escritura en ficheros, era una forma ineficiente de realizar este trabajo. Resultando que, al pasar todo este procedimiento a memoria, realizar todos los cálculos y comunicaciones de datos en memoria, se mejoraría considerablemente el tiempo de respuesta de programa. Añadiendo que, al utilizar este algoritmo, se conseguía otras optimizaciones al tener todos los datos en memoria.

Paralelizando la aplicación, se consiguió también mejorar bastante la aplicación. El punto fuerte, es que, al hacer división de trabajo y realizarlo de forma paralela, se aprovechaba del uso eficiente de las máquinas de memoria compartida o multicores. Realizando el trabajo en paralelo, se conseguía, que las funciones no tuvieran que esperar a que su antecesora acabase su tarea, si esta se podía ejecutar independientemente, al realizar la tarea en paralelo, se ganaba tiempo de respuesta. Además, al dividir el trabajo entre los multicores, con la técnica de blocking que se explicó, este aún ganaba más tiempo extra. Esto se debía, a que, con la técnica del blocking dividía una tarea en bloques entre los diferentes cores. De esta forma, cada core de forma individual, sacaban partido al uso de la memoria local para ese bloque.

Se ha demostrado, que el uso de las herramientas de traceo, en este caso Extrae (para tracear una ejecución) y Paraver (para visualizar las trazas), son de una gran ayuda, para poder optimizar de forma eficiente. Con estas herramientas, se podía detectar las zonas de los algoritmos a tener en cuenta para optimizar, y, si una optimización realizada, realmente mejoraba la aplicación.

Para acabar, el futuro está en la realización de algoritmos paralelos, tal como se ha demostrado dando ganancia a la aplicación, ya que, estamos en una era de la utilización de los multicores, de alguna forma se ha de beneficiar de ellos. Tal como se ha visto, con solo optimizaciones secuenciales no es suficiente.

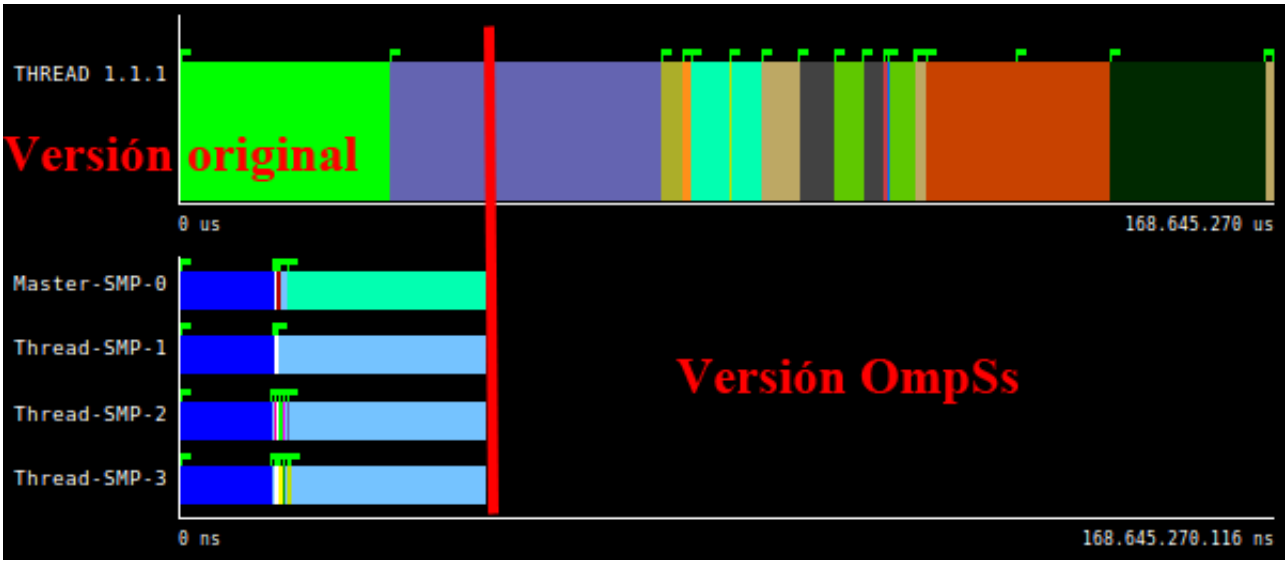

Figura 8.1: *La imagen muestra dos ejecuciones de la aplicación con la traza bt.C.64, la superior es del código original y la inferior es la versión final de OmpSs. Se puede apreciar la ganancia obtenida de todas las optimizaciones realizadas, dando lugar la última versión.*

9. Bibliografía y Webgrafía

Generales

- http://www.google.es/
- http://es.wikipedia.org/wiki/Wikipedia:Portada

Apuntes y Documentación Asignaturas FIB

- **MP**: todo lo referente a paralización, en especial con StarSs.
- **PCA**: técnicas de optimización.

Herramientas de Barcelona Supercomputing Center

- **GridSuperScalar**: http://www.bsc.es/plantillaH.php?cat_id=156
- **Paraver**: http://www.bsc.es/plantillaA.php?cat_id=493
- **Extrae**: http://www.bsc.es/plantillaA.php?cat_id=493
- **SMPSs**: http://www.bsc.es/media/3833.pdf
- **OmpSs**: documentación proporcionada por Alejandro Duran.

Librería FFTW

- http://www.fftw.org/
- http://stackoverflow.com/questions/4430172/fftw-signal-consists-of-noise-after-ifft
- http://www.mpmlab.com.ar/dm/node/18

Librería PAPI

- http://icl.cs.utk.edu/papi/
- http://spiral.cs.drexel.edu/wiki/Papi370Ubuntu910

Anexo A: Características del PC

A continuación, se muestran las características del PC que se han realizado todas las pruebas de las ejecuciones.

uname -a

```
Linux PCname 2.6.35-28-generic #50-Ubuntu SMP Fri Mar 18 18:42:20 UTC 2011 x86_64 GNU/Linux
```

gcc –version

```
gcc (Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5
Copyright (C) 2010 Free Software Foundation, Inc.
```

lshw

```
PCname
    description: Desktop Computer
    product: System Product Name
    vendor: System manufacturer
    version: System Version
    serial: System Serial Number
    width: 64 bits
    capabilities: smbios-2.6 dmi-2.6 vsyscall64 vsyscall32
    configuration: boot=normal chassis=desktop uuid=4060001E-8C00-00A6-C7D4-90E6BA21C89A
  *-core
       description: Motherboard
       product: P7P55D PRO
       vendor: ASUSTeK Computer INC.
       physical id: 0
       version: Rev 1.xx
       serial: 101507660002729
       slot: To Be Filled By O.E.M.
     *-firmware
          description: BIOS
          vendor: American Megatrends Inc.
          physical id: 0
          version: 0501 (08/21/2009)
          size: 64KiB
          capacity: 1984KiB
          capabilities: isa pci pnp apm upgrade shadowing escd cdboot bootselect socketedrom edd
              int13floppy1200 int13floppy720 int13floppy2880 int5printscreen int9keyboard
              int14serial int17printer int10video acpi usb ls120boot zipboot
              biosbootspecification
     *-cpu
          description: CPU
          product: Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz
          vendor: Intel Corp.
          physical id: 4
          bus info: cpu@0
          version: Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz
          serial: To Be Filled By O.E.M.
          slot: LGA1156
          size: 1200MHz
          capacity: 3800MHz
          width: 64 bits
          clock: 133MHz
```

```
          capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
              cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp
              x86-64 constant_tsc arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf
               pni dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1 sse4_2
              popcnt lahf_lm ida tpr_shadow vnmi flexpriority ept vpid cpufreq
        *-cache:0
             description: L1 cache
             physical id: 5
             slot: L1-Cache
             size: 128KiB
             capacity: 128KiB
             capabilities: internal write-through data
        *-cache:1
             description: L2 cache
             physical id: 6
             slot: L2-Cache
             size: 1MiB
             capacity: 1MiB
             capabilities: internal write-through unified
        *-cache:2
             description: L3 cache
             physical id: 7
             slot: L3-Cache
             size: 8MiB
             capacity: 8MiB
             capabilities: internal write-back unified
     *-memory
          description: System Memory
          physical id: 35
          slot: System board or motherboard
          size: 4GiB
        *-bank:0
             description: DIMM DDR Synchronous 1333 MHz (0.8 ns)
             product: PartNum0
             vendor: Manufacturer0
             physical id: 0
             serial: SerNum0
             slot: DIMM0
             size: 2GiB
             width: 64 bits
             clock: 1333MHz (0.8ns)
        *-bank:1
             description: DIMM [empty]
             product: PartNum1
             vendor: Manufacturer1
             physical id: 1
             serial: SerNum1
             slot: DIMM1
        *-bank:2
             description: DIMM DDR Synchronous 1333 MHz (0.8 ns)
             product: PartNum2
             vendor: Manufacturer2
             physical id: 2
             serial: SerNum2
             slot: DIMM2
             size: 2GiB
             width: 64 bits
             clock: 1333MHz (0.8ns)
        *-bank:3
             description: DIMM [empty]
             product: PartNum3
             vendor: Manufacturer3
             physical id: 3
             serial: SerNum3
             slot: DIMM3
     *-pci
          description: Host bridge
          product: Core Processor DMI
          vendor: Intel Corporation
          physical id: 100
          bus info: pci@0000:00:00.0
          version: 11
          width: 32 bits
          clock: 33MHz
        *-pci:0
```

```
             description: PCI bridge
             product: Core Processor PCI Express Root Port 1
             vendor: Intel Corporation
             physical id: 3
             bus info: pci@0000:00:03.0
             version: 11
             width: 32 bits
             clock: 33MHz
             capabilities: pci msi pciexpress pm normal_decode bus_master cap_list
             configuration: driver=pcieport
             resources: irq:45 ioport:b000(size=4096) memory:fa000000-fbbfffff ioport:ce000000(
                 size=301989888)
           *-display
                description: VGA compatible controller
                product: GT215 [GeForce GT 240]
                vendor: nVidia Corporation
                physical id: 0
                bus info: pci@0000:01:00.0
                version: a2
                width: 64 bits
                clock: 33MHz
                capabilities: pm msi pciexpress vga_controller bus_master cap_list rom
                configuration: driver=nvidia latency=0
                resources: irq:16 memory:fa000000-faffffff memory:d0000000-dfffffff memory:
                    ce000000-cfffffff ioport:bc00(size=128) memory:fbb00000-fbb7ffff
           *-multimedia
                description: Audio device
                product: High Definition Audio Controller
                vendor: nVidia Corporation
                physical id: 0.1
                bus info: pci@0000:01:00.1
                version: a1
                width: 32 bits
                clock: 33MHz
                capabilities: pm msi pciexpress bus_master cap_list
                configuration: driver=HDA Intel latency=0
                resources: irq:17 memory:fbbfc000-fbbfffff
      *-generic:0 UNCLAIMED
           description: System peripheral
           product: Core Processor System Management Registers
           vendor: Intel Corporation
           physical id: 8
           bus info: pci@0000:00:08.0
           version: 11
           width: 32 bits
           clock: 33MHz
           capabilities: pciexpress cap_list
           configuration: latency=0
      *-generic:1 UNCLAIMED
           description: System peripheral
           product: Core Processor Semaphore and Scratchpad Registers
           vendor: Intel Corporation
           physical id: 8.1
           bus info: pci@0000:00:08.1
           version: 11
           width: 32 bits
           clock: 33MHz
           capabilities: pciexpress cap_list
           configuration: latency=0
      *-generic:2 UNCLAIMED
           description: System peripheral
           product: Core Processor System Control and Status Registers
           vendor: Intel Corporation
           physical id: 8.2
           bus info: pci@0000:00:08.2
           version: 11
           width: 32 bits
           clock: 33MHz
           capabilities: pciexpress cap_list
           configuration: latency=0
      *-generic:3 UNCLAIMED
           description: System peripheral
           product: Core Processor Miscellaneous Registers
           vendor: Intel Corporation
           physical id: 8.3
```

```
             bus info: pci@0000:00:08.3
             version: 11
             width: 32 bits
             clock: 33MHz
             configuration: latency=0
        *-generic:4 UNCLAIMED
             description: System peripheral
             product: Core Processor QPI Link
             vendor: Intel Corporation
             physical id: 10
             bus info: pci@0000:00:10.0
             version: 11
             width: 32 bits
             clock: 33MHz
             configuration: latency=0
        *-generic:5 UNCLAIMED
             description: System peripheral
             product: Core Processor QPI Routing and Protocol Registers
             vendor: Intel Corporation
             physical id: 10.1
             bus info: pci@0000:00:10.1
             version: 11
             width: 32 bits
             clock: 33MHz
             configuration: latency=0
        *-usb:0
             description: USB Controller
             product: 5 Series/3400 Series Chipset USB2 Enhanced Host Controller
             vendor: Intel Corporation
             physical id: 1a
             bus info: pci@0000:00:1a.0
             version: 05
             width: 32 bits
             clock: 33MHz
             capabilities: pm debug ehci bus_master cap_list
             configuration: driver=ehci_hcd latency=0
             resources: irq:16 memory:f9ffe000-f9ffe3ff
        *-multimedia
             description: Audio device
             product: 5 Series/3400 Series Chipset High Definition Audio
             vendor: Intel Corporation
             physical id: 1b
             bus info: pci@0000:00:1b.0
             version: 05
             width: 64 bits
             clock: 33MHz
             capabilities: pm msi pciexpress bus_master cap_list
             configuration: driver=HDA Intel latency=0
             resources: irq:52 memory:f9ff8000-f9ffbfff
        *-pci:1
             description: PCI bridge
             product: 5 Series/3400 Series Chipset PCI Express Root Port 1
             vendor: Intel Corporation
             physical id: 1c
             bus info: pci@0000:00:1c.0
             version: 05
             width: 32 bits
             clock: 33MHz
             capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
             configuration: driver=pcieport
             resources: irq:46 ioport:1000(size=4096) memory:c0000000-c01fffff ioport:c0200000(
                 size=2097152)
        *-pci:2
             description: PCI bridge
             product: 5 Series/3400 Series Chipset PCI Express Root Port 5
             vendor: Intel Corporation
             physical id: 1c.4
             bus info: pci@0000:00:1c.4
             version: 05
             width: 32 bits
             clock: 33MHz
             capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
             configuration: driver=pcieport
             resources: irq:47 ioport:2000(size=4096) memory:c0400000-c05fffff ioport:c0600000(
                 size=2097152)
```

```
          *-pci:3
               description: PCI bridge
               product: 5 Series/3400 Series Chipset PCI Express Root Port 6
               vendor: Intel Corporation
               physical id: 1c.5
               bus info: pci@0000:00:1c.5
               version: 05
               width: 32 bits
               clock: 33MHz
               capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
               configuration: driver=pcieport
               resources: irq:48 ioport:3000(size=4096) memory:c0800000-c09fffff ioport:c0a00000(
                   size=2097152)
          *-pci:4
               description: PCI bridge
               product: 5 Series/3400 Series Chipset PCI Express Root Port 7
               vendor: Intel Corporation
               physical id: 1c.6
               bus info: pci@0000:00:1c.6
               version: 05
               width: 32 bits
               clock: 33MHz
               capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
               configuration: driver=pcieport
               resources: irq:49 ioport:d000(size=4096) memory:fbd00000-fbdfffff ioport:c0c00000(
                   size=2097152)
             *-storage
                  description: SATA controller
                  product: JMB362/JMB363 Serial ATA Controller
                  vendor: JMicron Technology Corp.
                  physical id: 0
                  bus info: pci@0000:03:00.0
                  version: 03
                  width: 32 bits
                  clock: 33MHz
                  capabilities: storage pm pciexpress ahci_1.0 bus_master cap_list
                  configuration: driver=ahci latency=0
                  resources: irq:18 memory:fbdfe000-fbdfffff
             *-ide
                  description: IDE interface
                  product: JMB362/JMB363 Serial ATA Controller
                  vendor: JMicron Technology Corp.
                  physical id: 0.1
                  bus info: pci@0000:03:00.1
                  version: 03
                  width: 32 bits
                  clock: 33MHz
                  capabilities: ide pm bus_master cap_list
                  configuration: driver=pata_jmicron latency=0
                  resources: irq:19 ioport:dc00(size=8) ioport:d880(size=4) ioport:d800(size=8)
                      ioport:d480(size=4) ioport:d400(size=16)
          *-pci:5
               description: PCI bridge
               product: 5 Series/3400 Series Chipset PCI Express Root Port 8
               vendor: Intel Corporation
               physical id: 1c.7
               bus info: pci@0000:00:1c.7
               version: 05
               width: 32 bits
               clock: 33MHz
               capabilities: pci pciexpress msi pm normal_decode bus_master cap_list
               configuration: driver=pcieport
               resources: irq:50 ioport:c000(size=4096) memory:fbc00000-fbcfffff ioport:f8f00000(
                   size=1048576)
             *-network
                  description: Ethernet interface
                  product: RTL8111/8168B PCI Express Gigabit Ethernet controller
                  vendor: Realtek Semiconductor Co., Ltd.
                  physical id: 0
                  bus info: pci@0000:02:00.0
                  logical name: eth0
                  version: 03
                  serial: 90:e6:ba:21:c8:9a
                  size: 100MB/s
                  capacity: 1GB/s
```

```
          width: 64 bits
          clock: 33MHz
          capabilities: pm msi pciexpress msix vpd bus_master cap_list rom ethernet physical
               tp mii 10bt 10bt-fd 100bt 100bt-fd 1000bt 1000bt-fd autonegotiation
          configuration: autonegotiation=on broadcast=yes driver=r8169 driverversion=2.3LK-
              NAPI duplex=full ip=172.16.0.10 latency=0 link=yes multicast=yes port=MII
              speed=100MB/s
          resources: irq:51 ioport:c800(size=256) memory:f8fff000-f8ffffff memory:f8ff8000-
              f8ffbfff memory:fbcf0000-fbcfffff
     *-usb:1
          description: USB Controller
          product: 5 Series/3400 Series Chipset USB2 Enhanced Host Controller
          vendor: Intel Corporation
          physical id: 1d
          bus info: pci@0000:00:1d.0
          version: 05
          width: 32 bits
          clock: 33MHz
          capabilities: pm debug ehci bus_master cap_list
          configuration: driver=ehci_hcd latency=0
          resources: irq:23 memory:f9ffd000-f9ffd3ff
     *-pci:6
          description: PCI bridge
          product: 82801 PCI Bridge
          vendor: Intel Corporation
          physical id: 1e
          bus info: pci@0000:00:1e.0
          version: a5
          width: 32 bits
          clock: 33MHz
          capabilities: pci subtractive_decode bus_master cap_list
          resources: ioport:e000(size=4096) memory:fbe00000-fbefffff
        *-firewire
             description: FireWire (IEEE 1394)
             product: VT6306/7/8 [Fire II(M)] IEEE 1394 OHCI Controller
             vendor: VIA Technologies, Inc.
             physical id: 3
             bus info: pci@0000:07:03.0
             version: c0
             width: 32 bits
             clock: 33MHz
             capabilities: pm ohci bus_master cap_list
             configuration: driver=firewire_ohci latency=64 maxlatency=32
             resources: irq:18 memory:fbeff800-fbefffff ioport:ec00(size=128)
     *-isa
          description: ISA bridge
          product: 5 Series Chipset LPC Interface Controller
          vendor: Intel Corporation
          physical id: 1f
          bus info: pci@0000:00:1f.0
          version: 05
          width: 32 bits
          clock: 33MHz
          capabilities: isa bus_master cap_list
          configuration: latency=0
     *-ide:0
          description: IDE interface
          product: 5 Series/3400 Series Chipset 4 port SATA IDE Controller
          vendor: Intel Corporation
          physical id: 1f.2
          bus info: pci@0000:00:1f.2
          logical name: scsi0
          logical name: scsi1
          version: 05
          width: 32 bits
          clock: 66MHz
          capabilities: ide pm bus_master cap_list emulated
          configuration: driver=ata_piix latency=0
          resources: irq:21 ioport:9c00(size=8) ioport:9880(size=4) ioport:9800(size=8) ioport
              :9480(size=4) ioport:9400(size=16) ioport:9080(size=16)
        *-disk:0
             description: ATA Disk
             product: ST3500418AS
             vendor: Seagate
             physical id: 0
```

```
          bus info: scsi@0:0.0.0
          logical name: /dev/sda
          version: CC38
          serial: 9VMANAQL
          size: 465GiB (500GB)
          capabilities: partitioned partitioned:dos
          configuration: ansiversion=5 signature=36c7ddce
        *-volume:0
             description: Windows NTFS volume
             physical id: 1
             bus info: scsi@0:0.0.0,1
             logical name: /dev/sda1
             version: 3.1
             serial: 6a29c14c-41b3-f945-aaa2-ce601ca14b55
             size: 39GiB
             capacity: 39GiB
             capabilities: primary bootable ntfs initialized
             configuration: clustersize=4096 created=2010-03-11 22:19:50 filesystem=ntfs
                 label=WINXP state=clean
        *-volume:1
             description: Windows NTFS volume
             physical id: 2
             bus info: scsi@0:0.0.0,2
             logical name: /dev/sda2
             version: 3.1
             serial: 92a31e16-c131-b94c-96e1-db469233de9f
             size: 426GiB
             capacity: 426GiB
             capabilities: primary ntfs initialized
             configuration: clustersize=4096 created=2010-03-11 23:32:55 filesystem=ntfs
                 label=WIN7 state=clean
     *-cdrom
          description: DVD-RAM writer
          product: DVDRAM GH22NS40
          vendor: HL-DT-ST
          physical id: 0.1.0
          bus info: scsi@0:0.1.0
          logical name: /dev/cdrom
          logical name: /dev/cdrw
          logical name: /dev/dvd
          logical name: /dev/dvdrw
          logical name: /dev/scd0
          logical name: /dev/sr0
          version: NL01
          capabilities: removable audio cd-r cd-rw dvd dvd-r dvd-ram
          configuration: ansiversion=5 status=nodisc
     *-disk:1
          description: ATA Disk
          product: ST3500418AS
          vendor: Seagate
          physical id: 1
          bus info: scsi@1:0.0.0
          logical name: /dev/sdb
          version: CC38
          serial: 9VMAN41N
          size: 465GiB (500GB)
          capabilities: partitioned partitioned:dos
          configuration: ansiversion=5 signature=1f3b1f5e
        *-volume:0
             description: Linux swap volume
             physical id: 1
             bus info: scsi@1:0.0.0,1
             logical name: /dev/sdb1
             version: 1
             serial: 6f3c6a42-5f33-44b3-8a4d-0d4aa4134c9e
             size: 7631MiB
             capacity: 7631MiB
             capabilities: primary nofs swap initialized
             configuration: filesystem=swap pagesize=4096
        *-volume:1
             description: Extended partition
             physical id: 2
             bus info: scsi@1:0.0.0,2
             logical name: /dev/sdb2
             size: 458GiB
```

```
                  capacity: 458GiB
                  capabilities: primary extended partitioned partitioned:extended
               *-logicalvolume:0
                    description: Linux filesystem partition
                    physical id: 5
                    logical name: /dev/sdb5
                    logical name: /
                    capacity: 279GiB
                    configuration: mount.fstype=ext4 mount.options=rw,relatime,errors=remount-ro
                        ,barrier=1,data=ordered state=mounted
               *-logicalvolume:1
                    description: Linux filesystem partition
                    physical id: 6
                    logical name: /dev/sdb6
                    logical name: /media/72140e5f-896a-4b5e-9afd-63d3742cedd3
                    capacity: 159GiB
                    configuration: mount.fstype=ext4 mount.options=rw,nosuid,nodev,relatime,
                        barrier=1,data=ordered state=mounted
               *-logicalvolume:2
                    description: Linux filesystem partition
                    physical id: 7
                    logical name: /dev/sdb7
                    capacity: 19GiB
          *-serial UNCLAIMED
               description: SMBus
               product: 5 Series/3400 Series Chipset SMBus Controller
               vendor: Intel Corporation
               physical id: 1f.3
               bus info: pci@0000:00:1f.3
               version: 05
               width: 64 bits
               clock: 33MHz
               configuration: latency=0
               resources: memory:f9ffc000-f9ffc0ff ioport:400(size=32)
          *-ide:1
               description: IDE interface
               product: 5 Series/3400 Series Chipset 2 port SATA IDE Controller
               vendor: Intel Corporation
               physical id: 1f.5
               bus info: pci@0000:00:1f.5
               version: 05
               width: 32 bits
               clock: 66MHz
               capabilities: ide pm bus_master cap_list
               configuration: driver=ata_piix latency=0
               resources: irq:21 ioport:ac00(size=8) ioport:a880(size=4) ioport:a800(size=8) ioport:
                   a480(size=4) ioport:a400(size=16) ioport:a080(size=16)
     *-network DISABLED
          description: Ethernet interface
          physical id: 1
          logical name: vboxnet0
          serial: 0a:00:27:00:00:00
          capabilities: ethernet physical
          configuration: broadcast=yes multicast=yes
```

Anexo B: Extrae

B.1. Instalación

Nos bajamos de http://www.bsc.es/plantillaC.php?cat_id=625 los archivos fuente, en este caso vamos a compilar la librería a nuestro gusto, también están disponibles en la página web los binarios.

Primero, vamos a instalar las librerías externas. Las primeras a instalar son: **bfd**, **iberty** y **xml2**. Al tener como sistema operativo Ubuntu, se ejecutará el siguiente comando para instalarlas.

```
$ sudo apt-get install libxml2 libxml2-dev binutils-dev
```

Buscamos el código fuente para las siguientes librerías:

- **unwind**: http://download.savannah.gnu.org/releases/libunwind/libunwind-0.99.tar.gz
- **PAPI**: http://icl.cs.utk.edu/projects/papi/downloads/papi-3.7.2.tar.gz

Para instalar la **unwind**, después de descomprimir el código, especificaremos el directorio donde la instalaremos, en este caso será en `/opt/unwind`. Para finalizar, tendremos que ejecutar estos comandos en el directorio del código fuente de la librería, para así compilarla e instalarla en nuestro PC.

```
:~/unwind$ CFLAGS=-U_FORTIFY_SOURCE ./configure
:~/unwind$ make prefix=/opt/unwind
:~/unwind$ sudo make install prefix=/opt/unwind
```

Antes de comenzar la instalación de la librería **PAPI**, instalaremos todos los paquetes necesarios para poder compilar su código fuente.

```
$ sudo apt-get install build-essential gfortran linux-headers-generic libncurses5-dev
```

Extraeremos el código y especificaremos el path de donde se encuentra en la variable global `PAPI_SRC`. Después, decidiremos donde instalaremos la librería, en este caso en `/usr/local/ /papi-3.7.2`. Una vez tenemos decidido donde la instalaremos, ejecutaremos los siguientes comandos para compilarla.

```
:~$ PAPI_SRC=/home/user/papi-3.7.2/
:~$ cd $PAPI_SRC/src
:~/papi-3.7.2/src$ ./configure --with-pcl \
--with-pcl-incdir=/usr/src/linux-headers-2.6.35-24/include/linux --prefix=/usr/local/papi-3.7.2
:~/papi-3.7.2/src$ make
```

Ejecutando el siguiente comando, en la carpeta con todo el código compilado, nos hará un test, para ver si se instalará correctamente en nuestro PC.

```
:~/papi-3.7.2/src$ make test
```

Si el test ha sido satisfactorio, procederemos a instalarla.

```
:~/papi-3.7.2/src$ make fulltest
:~/papi-3.7.2/src$ sudo make install-all
:~/papi-3.7.2/src$ sudo make install-tests
:~/papi-3.7.2/src$ sudo cp run_tests_exclude.txt run_tests.sh /usr/local/papi-3.7.2/share/papi/
```

Para finalizar, con el siguiente comando podremos averiguar que eventos tendemos disponibles con nuestra instalación.

```
:~/papi-3.7.2/src$ /usr/local/papi-3.7.2/bin/papi_avail
```

Extraeremos el código fuente de **Extrae**. En mi caso, la instalé en `/opt/extrae`. Para instalarla, ejecutaremos los siguientes comandos, especificando el path de su instalación y el lugar donde se encuentra las librerías PAPI y unwind.

```
:~/extrae$ ./configure --enable-merge-in-trace --enable-sampling --enable-smpss \
--prefix=/opt/extrae --with-unwind=/opt/unwind --with-papi=/usr/local/papi-3.7.2
:~/extrae$ make
:~/extrae$ sudo make install
```

B.2. Utilización

B.2.1. Especificar zona de traceo

Para tracear un código con Extrae, el primer paso, es definir la zona de traceo, para eso tenemos las siguientes funciones de inicio y fin.

Código B.1: Funciones inicio y fin traceo con Extrae

```
1 ...
2 Extrae_init();//inicio traceo extrae
3
4 //CÓDIGO a tracear
5
6 Extrae_fini();//fin traceo extrae
7 ...
```

B.2.2. Eventos de funciones

Dentro de la zona de traceo, se definirán los eventos que se deseará que salgan en la traza de la aplicación, para este propósito hay unas funciones especializadas que contiene la librería. Para las funciones del código del programa, se utilizará la función `Extrae_user_function(valor)`. Pasándole como variable el número 1, indicaremos que empieza la función a tracear. Con 0, le estamos indicando que se termina la función. Hay que vigilar, que el final de traceo de una función se ha de declarar antes del return.

Código B.2: Traceo de una función con Extrae

```
1 void funcion1()
2 {
```

```
3           Extrae_user_function(1);//inicio traceo de la función
4
5           //código de la función
6
7           Extrae_user_function(0);//evento de finalización del traceo de la función
8   }
9
10  int funcion2()
11  {
12          Extrae_user_function(1);
13          int finalValue=0;
14
15          //código de la función
16
17          Extrae_user_function(0);
18
19          return finalValue;
20  }
```

B.2.3. Eventos personalizados

Si lo que queremos es tracear un trozo de código, para esto deberemos generar unos registros de tipo evento, con la función `Extrae_event(tipo,valor)`. En `tipo`, se especifica el tipo de evento que queremos generar y en `valor`, su valor del evento. En el valor, indicaremos un número natural diferente de cero, para indicar que comienza ese registro en concreto y con un cero que acaba. En el siguiente ejemplo, tenemos para un mismo tipo de evento, en varios lugares del código, se ha establecido diferentes valores.

Código B.3: Utilización de eventos personalizados

```
1   //BUCLE 1
2   Extrae_event(1000, 1);//evento 1000, con valor 1 indicando el tiempo de tardanza del bucle1
3   for(i=0;i<N;i++)
4   {
5           //código
6   }
7   Extrae_event(1000, 0);//fin traceo bucle1
8
9   //BUCLE 2
10  Extrae_event(1000, 2);//evento 1000, con valor 2 indicando el tiempo de tardanza del bucle2
11  for(i=0;i<M;i++)
12  {
13          //código
14
15          Extrae_event(1000, 3);//traceo tardanza trozo código interno del bucle
16          //código
17          Extrae_event(1000, 0);//fin traceo tardanza trozo código interno del bucle
18
19          //código
20  }
21  Extrae_event(1000, 0);//fin traceo bucle2
```

Para poder indicar en el archivo `.pcf`, el significado de cada evento con sus respectivos valores, este lo indicaremos de esta forma, siguiendo el ejemplo anterior del Código B.3. Para este caso, le ponderemos a este fichero como nombre `labels.data`, que contendrá la especificación de todos los eventos personalizados.

Código B.4: Fichero descripción de eventos `labels.data`

```
1   EVENT_TYPE
2   0    1000    Eventos trozos de codigo
3   VALUES
4   0    End
5   1    Bucle1
```

```
2   Bucle2
3   codigoInternoBucle2
```

B.2.4. Fichero de configuración

Necesitaremos un archivo XML, para definir la configuración de Extrae. A continuación, se muestra un ejemplo para que funcione con la instalación que se ha explicado anteriormente. En este ejemplo, se define donde se encuentra la librería, que en este caso es en `/opt/extrae`. Se activa algunos eventos de la librería PAPI, que genere sus registros. Al final del todo, se le da el nombre al fichero de traceo, en este caso se llama `AplicacionPrueba.prv`.

Código B.5: Fichero de configuración `extrae.xml`

```
<?xml version='1.0'?>

<trace enabled="yes"
      home="/opt/extrae/"
      initial-mode="detail"
      type="paraver"
      xml-parser-id="Id: xml-parse.c 494 2010-11-10 09:44:07Z harald $"
>
      <counters enabled="yes">
            <cpu enabled="yes" starting-set-distribution="1">
                  <set enabled="yes" domain="all" changeat-globalops="5">
                        PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
                        <sampling enabled="no" frequency="100000000">PAPI_TOT_CYC</sampling>
                  </set>
                  <set enabled="yes" domain="user" changeat-globalops="5">
                        PAPI_TOT_INS,PAPI_FP_INS,PAPI_TOT_CYC
                  </set>
            </cpu>
            <network enabled="no" />
            <resource-usage enabled="no" />
            <memory-usage enabled="no" />
      </counters>

      <storage enabled="no">
            <trace-prefix enabled="yes">TRACE</trace-prefix>
            <size enabled="no">5</size>
            <temporal-directory enabled="yes">/scratch</temporal-directory>
            <final-directory enabled="yes">/gpfs/scratch/bsc41/bsc41273</final-directory>
            <gather-mpits enabled="no" />
      </storage>

      <buffer enabled="yes">
            <size enabled="yes">150000</size>
            <circular enabled="no" />
      </buffer>

      <trace-control enabled="yes">
            <file enabled="no" frequency="5M">/gpfs/scratch/bsc41/bsc41273/control</file>
            <global-ops enabled="no"></global-ops>
            <remote-control enabled="no">
                  <signal enabled="no" which="USR1"/>
            </remote-control>
      </trace-control>

      <others enabled="yes">
            <minimum-time enabled="no">10M</minimum-time>
      </others>

      <merge enabled="yes"
            synchronization="default"
            binary="optim"
            tree-fan-out="16"
            max-memory="512"
            joint-states="yes"
            keep-mpits="yes"
```

```
56              sort-addresses="no"
57        >
58              AplicacionPrueba.prv
59        </merge>
60
61  </trace>
```

B.3. Compilar y ejecutar un programa para tracear

Antes de compilar, se deberá definir las siguientes variables globales:

- EXTRAE_CONFIG_FILE: El nombre del fichero de configuración XML.
- EXTRAE_HOME: path a la librería Extrae.
- UNWIND_HOME: path a la librería unwind.
- EXTRAE_LABELS: nombre del fichero con los labels de los eventos personalizados.
- LD_LIBRARY_PATH: añadir el path de la librerías Extrae, PAPI y unwind.

Con los siguientes comandos, se exportarán estas variables globales y se compilará la aplicación, linkando las librería de Extrae.

```
$ export EXTRAE_CONFIG_FILE=extrae.xml
$ export EXTRAE_HOME=/opt/extrae
$ export UNWIND_HOME=/opt/unwind
$ export EXTRAE_LABELS=$PWD/labels.data
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${EXTRAE_HOME}/lib/:/usr/local/papi-3.7.2/lib/:${
    UNWIND_HOME}/lib/
$ gcc -O3 -g AplicacionPrueba.c -o AplicacionPrueba -L ${EXTRAE_HOME}/lib -lseqtrace -lxml2 -L/usr
    /lib64 -lbfd -L/usr/lib64 -liberty
```

El último paso que queda, es ejecutar la aplicación y la traza se generará automáticamente.

Anexo C: Paraver

C.1. Instalación de la aplicación

Para instalar la aplicación de Paraver, primero nos bajamos el binario de la página web http://www.bsc.es/plantillaC.php?cat_id=625.

Una vez tengamos los binarios del programa, sólo hemos de colocarlo en un directorio que nosotros deseamos. En este caso, voy a ponerlo en `/opt`, y, la carpeta que contendrá Paraver, tendrá como nombre `wxparaver64`.

Al tenerlo en `/opt`, le cambiamos los permisos a la carpeta.

```
$ sudo chmod o+wrx /opt/wxparaver64/ -R
```

Acto seguido, se editará el fichero `/opt/wxparaver64/bin/wxparaver`, y, se modificará, la variable global del fichero que contiene el directorio donde se encuentra la aplicación, en este caso se escribirá:

```
PARAVER_HOME=/opt/wxparaver64
```

Ya sólo, se tendrá que ejecutar la aplicación de esta forma:

```
$ /opt/wxparaver64/bin/wxparaver
```

C.2. Formato de una traza

La aplicación lee directamente de las trazas de Paraver, en consecuencia, en este apartado del anexo, se explicará la estructura básica de una traza, en concreto con los documentos `.prv`.

En la Figura C.1, se muestra la estructura básica de un fichero `.prv`. Se empieza con la cabecera del documento, donde se encuentra la información de la traza, como el tiempo de ejecución o el número de threads que se han ejecutado. Cada línea, representa un registro en el traceo de una aplicación, estos se especifican en diferentes tipos. El tipo puede ser de estados, eventos o de comunicación. Al mismo tiempo, se ha de guardar en que CPU se ejecutó, a que aplicación pertenece, el proceso y el thread. El resto de especificaciones, dependen del tipo de registro, en la Figura C.1, se ve un ejemplo de un documento con los tres diferentes registros, y, sus especificaciones.

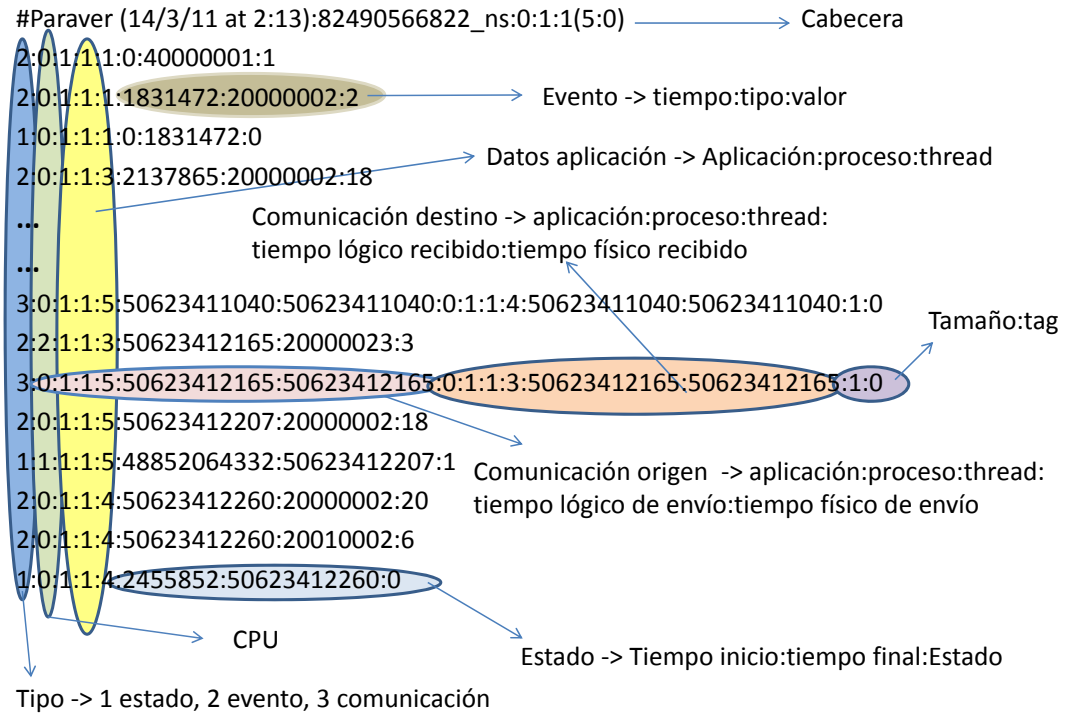

Figura C.1: *La imagen muestra la estructura básica de los ficheros* `.prv` *de Paraver.*

Anexo D: StarSs

El objetivo del proyecto, es paralelizar la aplicación con **StarSs**. En un primer momento, se especificó de hacerlo con la versión SMPSs, pero por necesidades y porque la nueva versión era estable, se decidió cambiarla a la OmpSs. En este anexo, se explicará como instalar sus respectivos compiladores y las instrucciones básicas para poder utilizar este lenguaje.

D.1. Versión SMPSs

D.1.1. Instalación

Primero, nos bajamos de http://bscgrid01.bsc.es/~jorgee/index.php?req=smp23 el compilador de SMPSs y descomprimimos el contenido del fichero.

En la carpeta extraída con el compilador, se procederá a crear el fichero de `Makefile` con el script de configuración `configure`. Se ha de decir donde se instalará el compilador, en mi caso, he decidido en `/opt/SMPSs`. Además, como en el Anexo B instalamos la librería **PAPI**, en la configuración le especificaremos donde se encuentra instalada, en este caso en `/usr/local/papi-3.7.2`. Ejecutaremos los siguientes comandos, para compilar e instalar el nuevo compilador.

```
:~/smpss$ ./configure --prefix=/opt/SMPSs --enable-papi --with-papi=/usr/local/papi-3.7.2
:~/smpss$ make
:~/smpss$ sudo make install
```

D.1.2. Utilización

D.1.2.1. Especificar zona de paralización

Para especificar la zona donde se va a paralelizar con SMPSs, se ha de definir con el `pragma` de inicio `css start` y el cierre de la zona con el `pragma css finish`. Dentro de esta zona, ya se podrán definir todos los `pragmas`, que se explican en los siguientes apartados.

Código D.1: Zona SMPSs

```
1 ...
2 #pragma css start
3
4 //Código con pragmas para SMPSs
5
6 #pragma css finish
7 ...
```

D.1.2.2. Creación de tasks

SMPSs se basa en la creación de tasks para threads. La definición de tasks, se hará sobre la declaración de una función, esta será la función paralela.

```
#pragma css task [input(<parametros>)] \
                 [inout(<parametros>)] \
                 [output(<parametros>)] \
                 [reduction(<parametros>)] \
                 [target device([cell, smp, cuda])] \
                 [highpriority]
<declaración o definición de una función>
```

La definición de parámetros, se efectuará de la siguiente forma, y, si la arquitectura acepta rangos, se hará como se indica a continuación.

```
parametros: parametro [,parametro]*
parametro: nombre_variable {[dimensiones]}* rangos*
rangos: {} | {index} | {start..end} | {start:length}
```

Se usaran las siguientes cláusulas para saber que dependencia hay entre funciones con sus datos:

- **input**: las variables como parámetro input, se trataran como datos de entrada para su lectura.
- **output**: las variables declaradas con esta cláusula, se trataran como datos de salida, como variables de escritura.
- **inout**: se tratan sus variables, como lectura y escritura para este task.
- **reduction**: como en OpenMP, esta variable será para operaciones de reducciones, ejemplo, las variables que llevan la suma total de algún cálculo.
- **highpriority**: especifica que este task en el scheduler, se le dará más prioridad que las demás.

En el Código D.2, se ve el ejemplo que para la `funcion2`, se ha de esperar a que se complete la `funcion1`, porque `funcion1` escribe en la variable `B` antes. En cambio, `funcion3` se puede ejecutar en cualquier momento, antes que `funcion1` y `funcion2`, ya que, no son dependientes, pero `funcion4`, sí que depende de los resultados de `funcion3`.

Código D.2: Ejemplo task en funciones SMPSs

```
1  #pragma css task input(A[N]) output(B[N])
2  void funcion1(int *A,int *B) {...}
3
4  #pragma css task input(B[N])
5  void funcion2(int *B) {...}
6
7  #pragma css task output(C[N])
8  void funcion3(int *C) {...}
9
10 #pragma css task inout(C[N])
11 void funcion4(int *C) {...}
12
13 void main()
14 {
15         ...
16
17         #pragma css start
18
19         funcion1(A,B);
20         funcion2(B);
21         funcion3(C);
22         funcion4(C);
```

```
23
24          #pragma css finish
25
26          ...
27
28  }
```

Hay dos formas de declarar las dimensiones de los datos, una es como en Código D.2 y la otra como en Código D.6.

Código D.3: Definición de task alternativa

```
1   #pragma css task input(A) output(B)
2   void funcion1(int A[N],int B[N]) {...}
3
4   #pragma css task input(B)
5   void funcion2(int B[N]) {...}
6
7   #pragma css task output(C)
8   void funcion3(int C[N]) {...}
9
10  #pragma css task inout(C)
11  void funcion4(int C[N]) {...}
```

En la modificación de una variable declarada como `reduction`, dentro de una función paralela, se deberá incluir la cláusula `mutex`, para poder modificarla.

Código D.4: Variable reduction

```
1   #pragma css task input(A) inout(sum) reduction(sum)
2   void accum (int A[BS], int *sum)
3   {
4           ...
5           //cálculo local de local_accum
6           ...
7
8           #pragma css mutex lock(sum)
9                   *sum = *sum + local_accum;
10          #pragma css mutex unlock(sum)
11
12          ...
13  }
14
15  void main()
16  {
17          ...
18
19          #pragma css start
20
21          ...
22
23          for (i=0; i<N; i+=BS)
24          {
25                  accum (&A[i], &sum);
26          }
27
28          ...
29
30          #pragma css finish
31
32          ...
33
34  }
```

D.1.2.3. Espera de datos o tasks

Con el `pragma barrier`, se espera a que todos los tasks hayan acabado, así, en el ejemplo solo tendrá que imprimir por pantalla una sola vez.

Código D.5: Espera de todas las tasks

```
...

for(int i=0;i<N;i++)
{
        funcion1(..);//función paralela
}

#pragma css barrier
printf("Acabado\n");

...
```

Al contrario, con `wait on(direccion_dato)` solo se espera a que una cierta variable tenga los valores listos.

Código D.6: Espera de los datos de una variable

```
...

funcion(&data);//función paralela con output(data)

...

#pragma css wait on(&data)
//se espera hasta tener los datos de data

...
```

D.1.3. Compilar y ejecutar un programa con SMPSs

Para poder ejecutar un programa con SMPSs, deberemos exportar las siguientes variables globales, antes, de poder compilar o ejecutar dicho programa.

- `PATH`: se especificará, el path de donde se encuentra el binario del compilador.
- `LD_LIBRARY_PATH`: path a la carpeta de la ubicación de la librería de SMPSs.
- `CSS_NUM_CPUS`: se especificará, el número de CPUS que se le asignará a la ejecución del programa.

Con los siguientes comandos a modo de ejemplo, se exportará las variables que se han expuesto anteriormente, se compilará el programa y se ejecutará.

```
$ export PATH=$PATH:/opt/SMPSs/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SMPSs/lib
$ smpss-cc -O3 -o programa programa.c
$ export CSS_NUM_CPUS=8
$ ./programa
```

D.1.4. Compilar y tracear un programa con SMPSs

Si lo que queremos, es que nos genere una traza de Paraver, a la hora de compilar el programa, le pasaremos como parámetro "-t" al compilador.

```
$ export PATH=$PATH:/opt/SMPSs/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SMPSs/lib
$ smpss-cc -t -O3 -o programa_trace programa.c
$ export CSS_NUM_CPUS=8
$ ./programa_trace
```

D.2. Versión OmpSs

D.2.1. Instalación

Antes de instalar nada, hemos de tener el sistema los programas necesarios para poder compilar los archivos. El siguiente comando, instala todas las herramientas que necesitaremos y que normalmente no vienen instaladas por defecto, en este caso utilizando el sistema operativo Ubuntu.

```
$ sudo apt-get install automake1.9 libtool gperf bison flex git
```

La versión, de StarSs OmpSs utiliza el compilador Mercurium. Así que, nos bajaremos del repositorio el compilador y la librería OmpSs:

- **Librería OmpSs (nanos++)**: http://pm.bsc.es/git/nanox.git
- **Mercurium**: http://pm.bsc.es/git/mcxx.git

Se decidirá donde instalarlo, en este caso se decidió en `/opt/OmpSs`. Primero, instalaremos la librería de OmpSs de esta forma:

```
:~$ export TARGET=/opt/OmpSs
:~$ git clone http://pm.bsc.es/git/nanox.git
:~$ cd nanox
:~/nanox$ autoreconf -f -i
:~/nanox$ ./configure --prefix=$TARGET
:~/nanox$ sudo make install
```

Acto seguido instalamos compilador Mercurium con compatibilidad OmpSs.

```
:~$ git clone http://pm.bsc.es/git/mcxx.git
:~$ cd mcxx
:~/mcxx$ autoreconf -f -i
:~/mcxx$ ./configure --prefix=$TARGET --enable-ompss
:~/mcxx$ sudo make install
```

D.2.2. Utilización

A continuación, se explicará la nueva versión de OmpSs, pero haciendo una comparativa con la versión anterior de SMPSs.

D.2.2.1. Especificar zona de paralización

Con esta versión, no hará falta declarar la zona de paralización como se hacía con SMPSs, en el Código D.1 con los `pragmas start` y `finish`.

D.2.2.2. Creación de tasks en funciones

Para crear tasks en las funciones es muy parecido que en el apartado "D.1.2.2. Creación de tasks" de la versión anterior.

```
#pragma omp task [input(<parametros>)] \
                 [inout(<parametros>)] \
                 [output(<parametros>)] \
                 [concurrent(<parametros>)] \
                 [target (<device[cell, smp, cuda],copy_in,copy_out,copy_inout,copy_deps,
                     implements>)] \
                 [highpriority]
<declaración o definición de una función>
```

Los ejemplos, para la dependencia entre tasks utilizando `input`, `output` y `inout`, son los mismos, que en el apartado "D.1.2.2. Creación de tasks". La diferencia visible es, que la cláusula ahora es `omp`, en vez de `css`. Añadiendo, que la cláusula `reduction` no existe, ahora se utilizará `concurrent`. Esta cláusula, es como la `inout`, pero menos restrictiva, en el sentido que no hay un orden en la dependencia para su ejecución. Se ve claramente en el ejemplo, que para hacer una suma de una acumulación, no necesariamente ha de tener un orden en que se hace la suma. En el Código D.7, se muestra un ejemplo de su utilización, además, ahora para hacer una modificación de dicha variable, se utiliza la cláusula `atomic`.

Código D.7: Variable concurrent (como reduction)

```
1  #pragma omp task input(A[BS]) concurrent(*sum)
2  void accum (int *A, int *sum)
3  {
4          ...
5          //cálculo local de local_accum
6          ...
7
8          #pragma omp atomic
9                  *sum = *sum + local_accum;
10         ...
11 }
12
13 void main()
14 {
15         ...
16
17         for (i=0; i<N; i+=BS)
18         {
19                 accum (&A[i], &sum);
20         }
21
22         ...
23 }
```

D.2.2.3. Creación de tasks como OpenMP

D.2.2.4. Espera de datos o tasks

Para esperar todos los tasks, en vez de la cláusula `barrier` como en Código D.5, ahora es un `taskwait`.

Código D.8: Espera de todas las tasks

```
...

for(int i=0;i<N;i++)
{
        funcion1(..);//función paralela
}

#pragma omp taskwait
printf("Acabado\n");

...
```

Con el `pragma` de `wait on`, ahora, es `taskwait on(variable)`.

Código D.9: Espera de los datos de una variable

```
...

funcion(&data);//función paralela con output(data)

...

#pragma omp taskwait on(data)
//se espera hasta tener los datos de data

...
```

D.2.3. Compilar y ejecutar un programa con OmpSs

Para poder compilar, se ha de definir en `PATH`, la ruta donde se halla el compilador. Una vez se ha compilado el programa, para ejecutarlo, se le ha de definir a la variable global `OMP_NUM_THREADS` el número de threads que se le desea asignar a la ejeucion. A modo de ejemplo, los siguientes comandos son los que se utiliza para compilar y ejecutar un programa con OmpSs.

```
$ export PATH=$PATH:/opt/OmpSs/bin
$ mcc --ompss -O3 -o programa programa.c
$ export OMP_NUM_THREADS=8
$ ./programa
```

D.2.4. Compilar y tracear un programa con OmpSs

Que después de ejecutarse un `programa` de OmpSs, se obtenga una traza de la ejecución de dicho `programa` en formato de Paraver, a la hora de compilar, en este caso se le pasará como parámetro "--instrument", y, se le definirá a la variable global `NX_INSTRUMENTATION` la palabra "extrae", como librería a tracear.

```
$ export PATH=$PATH:/opt/OmpSs/bin
$ export NX_INSTRUMENTATION=extrae
$ mcc --ompss --instrument -O3 -o programa_trace programa.c
$ export OMP_NUM_THREADS=8
$ ./programa_trace
```

Esta versión, contiene una API para poder definir tus propios eventos al estilo de Extrae, que se explica en el Anexo de Extrae, en el apartado "B.2.3. Eventos personalizados". Para esto, se pondrá el `pragma instrument event(tipo, valor)`. En `tipo`, se especifica el tipo de evento que queremos generar y en `valor`, su valor del evento.

Código D.10: Utilización de eventos personalizados

```
1  //BUCLE 1
2  #pragma instrument event(1000, 1);//evento 1000, con valor 1 indicando el tiempo de tardanza del
       bucle1
3  for(i=0;i<N;i++)
4  {
5          //código
6  }
7  #pragma instrument event(1000, 0);//fin traceo bucle1
8
9  //BUCLE 2
10 #pragma instrument event(1000, 2);//evento 1000, con valor 2 indicando el tiempo de tardanza del
       bucle2
11 for(i=0;i<M;i++)
12 {
13         //código
14
15         #pragma instrument event(1000, 3);//traceo tardanza trozo código interno del bucle
16         //código
17         #pragma instrument event(1000, 0);//fin traceo tardanza trozo código interno del bucle
18
19         //código
20 }
21 #pragma instrument event(1000, 0);//fin traceo bucle2
```

Tal como se explica en el mismo apartado "B.2.3. Eventos personalizados", se hará lo mismo que en el Código B.4, para definir los valores de los eventos personalizados, para añadir al archivo `.prv`. Además, también, se definirá la variable global `EXTRAE_LABELS`, para indicar la ubicación de estas descripciones (tal como se hace con Extrae).

Los siguientes comandos reflejan un ejemplo de su compilación y ejecución.

```
$ export PATH=$PATH:/opt/OmpSs/bin
$ export NX_INSTRUMENTATION=extrae
$ export EXTRAE_LABELS=$PWD/labels.data
$ mcc --ompss --instrument -O3 -o programa_trace_eventos programa.c
$ export OMP_NUM_THREADS=8
$ ./programa_trace_eventos
```